THE UNIVERSITY OF CALGARY

AN EXPERIMENTAL PROGRAM ANIMATION SYSTEM

by

KENNETH J. LUTERBACH

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

MARCH, 1992

ISBN 0-315-75275-0

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "An Experimental Program Animation System" submitted by Kenneth Luterbach in partial fulfillment of the requirements for the degree of Master of Science.

Kenneth Loose

Supervisor, Dr. Kenneth Loose
Department of Computer Science

J.R.B. Cockett

Dr. Robin Cockett
Department of Computer Science

[illegible]

Dr. Michael Williams
Department of Computer Science

[illegible]

Dr. William J. Hunter
Department of Teacher Education and Supervision

Date 92-03-05

Abstract

A system for animating the execution of eleven specific Pascal programs was developed to help students acquire programming knowledge. The system, called EPAS (Experimental Program Animation System), contains an animation module for depicting run-time stack activity. In addition EPAS contains a tutorial module for instructional purposes. Compilation modules were also incorporated into EPAS because it is a prototype for a complete program animator that could accept any Pascal program as input.

The animation module depicts data manipulations within a run-time stack. Data are manipulated in accordance with the statements in an executing program. The stack is represented by a consecutive set of rectangles. Each rectangular cell represents one variable in which a name and a value appear. Cells are highlighted to depict memory read and write operations. Arrows are displayed on a temporary basis to depict value parameter passing. In contrast, arrows showing the links between formal and actual variable parameters remain on the screen until the links are broken by subroutine termination. Stack events are executed only at the user's request. In addition, the user can adjust the pace at which cells are accessed.

The tutorial module presents instruction, provides opportunities for practice and then tests the learner. All of these tutorial features are based on program animations that manipulate arrays, value parameters, or variable parameters. As an example program is animated, stack cells of particular importance are identified. During practice and testing program animations, the user is directed to identify specific cells. Feedback is provided after all attempts to identify cells.

A scanner, a parser and symbol table generation routines comprise the compilation modules. Thus, only type checking and code generation routines would have to be added to EPAS to make it a complete compiler. The compilation modules were incorporated into EPAS for enhancement purposes only. EPAS could bypass the compilation modules and still animate the eleven input programs.

To test the viability of EPAS, a small study was conducted. Eight students with little computing and programming experience first viewed two introductory program animations. Then the students advanced through three sets of program animations. The program animation sets pertained to arrays, value parameters and variable parameters. Each set contained an example, practice and quiz program animation. During the practice and testing animations, tallies were kept to monitor the number of correctly identified stack cells.

The results of the study showed that the students, on average, correctly identified stack cells more than seventy-five percent of the time on the three quiz programs. For the purposes of this study, that level of achievement was viewed as mastery. Data pertaining to the reverse execution feature in EPAS were also collected and analyzed. Given the encouraging results, it was suggested that EPAS be enhanced to the point of a complete program animator. Specific techniques for achieving that objective are discussed.

Acknowledgements

I wish to thank my supervisor, Dr. K. D. Loose especially. I also want to thank my friends and the students who participated in the study. Lastly, I thank the University of Calgary for supporting this work with a Thesis Research Grant.

Dedication

To my Parents, Sisters and Brothers

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background to the Problem

This study pertains to the acquisition and application of computer programming skills. Of particular concern is a system that could help students gain specific programming knowledge. Analyzing how that system could be enhanced to benefit experienced programmers is also important. Before examining these issues, which pertain to contemporary approaches to writing software, it will be useful to gain an appreciation for the techniques developed in the past. Thus a discussion about the history of software development follows.

Beginning about 1834 Charles Babbage worked at developing a new type of calculating machine. The new machine, called the analytical engine, was designed to perform operations specified in an external program. According to Williams (1985), calculating machines were programmed using some semblance of instructions punched on a paper tape or by intricate hardwired connections. The instructions for the analytical engine were stored on paper cards. With respect to programming those types of machines, Knuth and Pardo (1980, p. 200) noted: "Example programs written for early computing devices, such as those for Babbage's calculating machine, were naturally presented in *machine language* rather than in a true programming language." Example programs for Babbage's machine can be found in Knuth and Pardo (1980, p. 200) and Williams (1985, pp. 189-190).

The realization of the stored program concept (i.e., the storage of instructions in the memory of a computer) in the early part of 1944 altered the development of software considerably. In particular, the use of stored program computers increased the pace with which software was developed. The mathematician, John von Neumann is often credited with advancing the notion of the stored program concept. However, the identity of the true developer is vigorously contested. In fact, Williams (1985, p. 298) stated: "The question of who actually invented the concept of the stored program has caused more controversy than perhaps any other in the history of computing science."

It is still possible to program modern computers in machine language. When doing so a programmer specifies instructions that directly manipulate hardware devices. For example, the devices altered could be a register in the central processing unit or a byte of random access memory. Presently though, machine code is rarely written because that style of programming requires knowledge about a computer's architecture and the necessity to memorize or look-up the numeric codes that correspond to its instructions, among other skills.

Various modifications were made to machine languages in an attempt to ease the programming burden. For example, Eckhouse and Morris (1979, p. 17) noted: "To simplify the process of writing or reading a [machine language] program, each instruction is often represented by a simple two- to five-letter mnemonic symbol." In keeping with this approach, the mnemonic symbol LDA might be used in place of a number to indicate an instruction that loads a value into a register.

Given this enhancement a programmer can use letters and numbers to specify instructions that indicate the type of operation to be performed. Writing these types of instructions and using techniques such as *labeling*, for the automatic calculation of offsets, is called *assembly language programming.* Writing assembly language instructions relieves the programmer from the burden of memorizing or looking up codes. However the instructions must still manipulate hardware devices directly. Consequently, assembly language code is also rarely written today. Note that machine and assembly languages are often referred to as *low-level* languages, or ones that operate on a concrete level, since they directly alter hardware devices (see for example, Lafore, 1984, pp. 14-15).

The next step in software development decreased the extent to which a computer's architecture had to be learned. In 1951 Arthur W. Burks showed how machine code could be represented at a higher level of abstraction. For example, he introduced the notion of an assignment statement in which a variable, not a particular register or byte of memory, is given a value. Thus some computations could be specified without reference to hardware devices. Knuth and Pardo (1980) referred to this algorithm specification notation as an *intermediate* programming language.

Subsequently, the machine independence goal was fully realized when compilers for *high-level* languages were developed. According to Knuth and Pardo (1980, p. 242), "[The 'Formula Translator' (Fortran) compiler] is the earliest high-level language that is still in use." It was first released in April, 1957. (Incidentally, like some contemporary software, it was delivered later than expected and did not work particularly well.) The first Fortran compiler did not actually achieve full machine independence. In fact, Knuth and Pardo (1980) stated that the developers of Fortran did not perceive that as a primary goal, although they certainly recognized its importance.

Today, fully machine independent programming languages are available. Programs written in Basic, Pascal, PL/1 and other high-level languages seldom reference hardware components. With respect to existing high-level language translators, Fischer and LeBlanc (1988) stated: "A compiler allows most (indeed, virtually all) computer users to ignore the machine-dependant details of machine language." Consequently, rather than controlling the flow of data through hardware devices, programmers of this day use abstract processes to manipulate data in entities such as *arrays* or *records* or even more complex structures.

1.2 Statement of the Problem

The development of high-level programming languages is valuable, especially given that the variety of computers is growing rapidly. However, the realization of machine independent languages has merely shifted the programming burden. Rather than attending to details about a particular machine, programmers must be concerned about manipulating abstract data entities.

Such manipulations can be perplexing. For instance, Brad A. Myers (1989, p. 3) stated: "It is well known that conventional programming languages are difficult to learn and use, requiring skills that many people do not have." Fourteen years earlier Harlan D. Mills (1975, p. 43) expressed the difficulty of programming as follows: "Computer programming as a practical human activity is some 25 years old. Yet computer programming has already posed the greatest intellectual challenge that mankind has faced in pure logic and complexity."

In addition to opinions, an indicator of programming difficulty is the rate at which instructions are written. According to Boehm (1981), one can expect 376 deliverable source code instructions per person-month for a project of intermediate size (approximately 8000 instructions). Assuming twenty working days in a month, it follows that programmers can only be expected to write nineteen deliverable instructions per day!

In attempts to make programs more understandable in learning and debugging (i.e., error correction) settings, researchers are developing *program visualization* systems. According to Baecker (1988, p. 356), "Computer program visualization is the use of graphics to enhance the art of program presentation and thereby to facilitate the visualization, understanding, and effective use of computer programs by people."

For this study a new program visualization system was developed by the researcher. Since it is a prototype for a more complete system it is entitled: An Experimental Program Animation System (EPAS). It simulates and displays run-time stack activity.

A run-time stack is a block of memory that contains the data in a program. When a program is executing, the run-time stack is updated as data are manipulated. For example when a variable is assigned a value, the new value is written to the run-time stack cell assigned to the variable. (For later reference, note that memory cells to which values are written are called *targets* whereas cells from which values are read are called *sources.*)

Other run-time stack manipulations occur when subroutines are called and exited. For instance when a subroutine is activated, the variables local to it are assigned memory locations on the run-time stack. Conversely, local variables are removed from the stack when the subroutine is terminated. Those activities, and parameter passing, are depicted graphically in EPAS.

EPAS differs from other program visualization systems in two fundamental ways. First, EPAS pays particular attention to run-time stack activity. In contrast, displaying changes to data structures is the primary focus of other systems. (A review of program visualization systems appears in Chapter 2). The second significant difference pertains to the manner in which users view programs. All program visualization systems, including EPAS, allow users to proceed through a program one statement at a time. However EPAS, unlike the other systems, also allows users to go backward through a program. In this way statements may be repeated to review their consequences.

Presently, students often learn about the dynamic activities of executing programs by viewing static diagrams in textbooks. Contrastingly, EPAS simulates those dynamic activities by modifying the run-time stack display. In addition, the run-time stack modeling feature allows the user to view an abstract data structure, in particular the one dimensional array, in a concrete manner. This is done by depicting how it is stored in memory. These advantages led the researcher to advance the following hypothesis.

> After using EPAS, students will be able to identify the target and source memory cells referenced in statements that manipulate the following programming features:
>
> 1. one-dimensional arrays
> 2. value parameters
> 3. variable parameters

For the purposes of this study, students were expected to correctly identify target and source run-time stack cells seventy-five percent of the time. This percentage of accuracy is referred to subsequently as the *mastery level.*

To test that hypothesis a small study was conducted. Since EPAS is in a formative stage, only eight students participated in the study. Each student used EPAS to interact with eleven Pascal program execution simulations. The first two programs served to introduce the system. The remaining nine programs were divided into three sections, one for each of the programming features cited above. For each section the students first viewed an example program. Then they worked at identifying source and target cells in a practice program. Lastly, they completed a quiz program. When the students finished running EPAS, they completed a questionnaire.

1.3 Importance of the Study

The claim that programming languages are difficult to learn and use was supported at the start of the previous section. A goal of this research is to reduce that degree of difficulty. The goal would be reached if it could be shown that EPAS enables students to identify source and target cells of arrays, value parameters and variable parameters.

In addition, if the prototype system is beneficial, pursuing the development of an entire program visualization system, complete with compiler and run-time stack modeler, would be worthwhile. For instance, if modifications to EPAS could help students visualize their programs, tedious debugging tasks may be simplified. In any case, such a system would directly address the following concern raised by Plattner and Nievergelt (1981, p. 91):

> Monitoring program execution takes up a considerable fraction of a programmer's time; it is unlikely to decrease in importance, since the insight it can provide is complementary to, and cannot be replaced by, that obtained from static analysis of program texts. Hence it is surprising that today's commercial software rarely supports execution monitoring well, and that no advances comparable to those in the fields of programming languages have occurred over the past two decades.

In the ten years since that comment was made, improvements in program execution monitoring have certainly been implemented, as will be shown in Chapter 2. However, still more advances are sought. Evidence for this exists in the current interest in program visualization systems. Indeed Myers (1989, p. 3) observed: "There has been a great interest recently in systems that use graphics to aid in the programming, debugging, and understanding of computer programs." The future viability of EPAS is a major concern because an enhanced system may help students in these three areas. Technical details describing how EPAS could be enhanced are discussed at length in Chapter 5.

1.4 Scope and Limitations

The most restrictive aspect of EPAS is that it simulates the execution of precisely eleven specific Pascal programs. However, the system is a prototype for one that, if completed, could display the run-time stack of any Pascal program. A Pascal program would be submitted to the complete system and it would compile the program. During compilation it would add run-time stack display code to the machine code generated for the submitted program. In fact, the complete system would be a combination compiler and run-time stack modeler. In keeping with that approach, EPAS also contains compiling and run-time stack modeling components. EPAS is described fully in Chapter 3.

Introducing students to one dimensional arrays, as EPAS does, substantially increases their potential to manipulate data in a program. For instance by utilizing an array, related data can be manipulated in a single structure instead of as distinct entities. Further, EPAS instructs students in value and variable parameter manipulations.

By using parameters, programmers can control the flow of data within subroutines (i.e., procedures and functions). In addition, complex programs can be developed in a modular manner by writing self-contained subroutines. Procedures and functions that do not reference global variables are said to be self-contained. With respect to such procedures, Savitch (1991, p. 146) noted:

> Procedures separate a program into smaller, and hence more manageable, pieces. In order to get the full benefit of this decomposition, the procedures must be self-contained units that are meaningful outside the context of any particular program.

According to Niklaus Wirth, algorithms and data structures are the two fundamental aspects of programming. Evidence for this exists in the title of Wirth's (1976) book, *Algorithms + Data Structures = Programs.* Since EPAS may enhance a student's knowledge in both of those two vital aspects of programming, it could lead to a significant improvement in the quality of programs written by them.

1.5 Summary

A need to help students acquire programming skills was identified. To address this need an experimental program animation system (EPAS) was developed to show that run-time stack modeling can help students acquire specific programming knowledge. If the system is beneficial, students could use it to understand how one-dimensional arrays and value and variable parameters are manipulated. Further, it was noted that the prototype could be enhanced for use in both learning and simple debugging settings. The potential to develop EPAS to the point of a complete program animation system was described as a major concern.

Chapter 2

Review of the Literature

This review is divided into three sections. First, program visualization systems are examined. Second, an analysis of the debuggers from four distinct systems is presented. Last, the foundation of the tutorial module in EPAS, a specific instructional design theory, is discussed.

2.1 Program Visualization

There are two types of program visualization systems, *program animators* and *algorithm animators.* Ross (1991) described program animators as software systems that dynamically display their actions during execution. These animators typically highlight the statement in a program that is being executed and display the values of variables. Note that the system developed for this study is a program animator that simulates run-time stack activity.

After describing program animators, Ross (1991, p. 36) stated: "An algorithm animator is a software system similar to a program animator except that the underlying algorithm, rather than the program itself, is the subject of the animation." Consequently algorithm animators operate at a more abstract level than program animators. For example, an algorithm animator might depict the actions of the bubble sort routine by comparing and swapping bars of varying heights (representing random values) until the bars are in ascending order. Alternatively a program animator would highlight lines of code and show how each statement compares or swaps the actual values in an array. When the execution of all statements had been depicted, the values in the array would be in ascending order.

Before examining existing program visualization systems, it will be worthwhile to distinguish between program visualization and visual programming. Myers (1989, p. 4) stated: "Visual programming (VP) refers to any system that allows the user to specify a program in a two (or more) dimensional fashion." For example, flowcharts (Shu, 1988) or Nassi-Schneiderman diagrams (Nassi & Schneiderman, 1973) could be used when engaging in visual programming. Alternatively one could use a visual programming software system such as PECAN (Reiss, 1985) or MONDRIAN (Protsko, Sorenson, Tremblay & Schaefer, 1991). While enhancing the comprehensibility of software is the goal of both visual programming and program visualization, these fields differ in that the former is concerned exclusively with generating static representations of program flow. In contrast, program visualization systems depict program flow dynamically. Only program visualization systems for block structured languages are considered beyond this point.

2.1.1 Algorithm Animators

A discussion of program visualization systems would be incomplete if either algorithm or program animators were excluded. However, algorithm animators are discussed only briefly here because they depict programs at a high level of abstraction. In contrast, since EPAS is a program animator it depicts algorithms concretely.

To teach students nine different sorting algorithms, Baecker (1981) developed the 16 mm film, *Sorting Out Sorting*. One method used to depict the dynamic nature of the algorithms involves highlighting and moving rectangular bars. The bars vary in size to represent random values, as noted above. Verbal descriptions of the algorithms accompany the animations. The use of 16 mm film or video is a legitimate approach to algorithm animation. However, this approach is restrictive because the animation sequences are suitable for depicting only specific algorithms.

An instructional software system to simulate the functioning of a microprocessor was developed by Gurwitz, Fleming and Van Dam (1981). Specifically, the software depicts changes to internal registers, control lines, buses, memory buffers and peripherals. In this way students can learn algorithmic processes such as *handshaking*, or phrased more technically, asynchronous communication. Again, in this restricted approach the animation sequences can only be used to depict specific activities.

All programmers could animate their code by inserting additional statements into their source files. Alternatively the animation sequences could be written in an additional file, compiled separately and called when necessary. Krishnamoorthy and Swaminathan (1989) took such an approach. They wrote code for a limited number of animation primitives such as 'blink an object' and 'move an object horizontally.' The primitives were compiled separately in a Turbo Pascal (Borland, 1988) unit file. Calls to the primitive routines were then placed in existing source files. This approach is not as restrictive as the previous two methods because the primitives can be used for any algorithm. However, the original source code must be altered extensively by calls to the primitive routines. Also, it is unlikely that one set of animation primitives could be developed to the satisfaction of all programmers.

One of the most sophisticated algorithm animators is BALSA- II (Brown, 1988). It was derived directly from the original BALSA (Brown & Sedgewick, 1984). Unfortunately a considerable programming effort is still required to create BALSA animations. However, they allow the user substantial flexibility. Also, the original algorithm is largely unchanged. Only a few calls to the animation routines, at so-called *interesting events*, are necessary. For example, an interesting event could be a swap in a sorting routine or a node insertion in a tree balancing algorithm. It is up to the developer to decide what constitutes an interesting event. Lastly, note that BALSA is a flexible system because virtually any algorithm can be animated.

Two other systems of note, an unnamed system by London and Duisberg (1985) and TANGO (Stasko, 1990) also utilize the interesting event approach. Both of these systems contain modules that communicate by *message passing.* For example, the program module may send a message to the animator module indicating that an interesting event has occurred in a tree. In turn, the animator module may send a message to the display module to remove a tree node. Again, a significant programming effort is required to create animations with these systems. However, as in BALSA, these systems can be used to animate diverse algorithms. Table 2.1 contains a summary of the algorithm animators cited above.

2.1.2 Program Monitors and Animators

As defined in Section 1.2, visualization systems, such as program animators, must use graphics to depict execution. However some researchers have developed strictly textual displays for viewing program execution, as discussed below. The term, *program monitor* will be used to describe a software tool that uses this approach.

Two of the earliest program monitors were developed by Baecker (1975) to view Logo and PL/1 code. The displays generated by the monitors were actually used

Table 2.1: Algorithm Animators

Identification	Scope	Approach	Purpose	Year
Sorting out Sorting	Restricted	16 mm film	Instruction	1981
MIDAS	Restricted	Software	Instruction	1981
BALSA I	Open	Software	Instruction	1984
London and Duisberg	Open	Software	Design Development Testing	1985
BALSA II	Open	Software	Instruction	1989
Primitives	Open	Software	Instruction	1989
TANGO	Open	Software	Instruction	1990

to facilitate the development of instructional films. The work involved enhancing interpreters for subsets of Logo and PL/1 so that they also translated an animation control language. Displays were generated as the programs, supplemented with statements to control monitoring, were interpreted. It should be noted that while the mini-PL/1 interpreter generated text predominately, it could also display a circle and a rectangle. Nevertheless it generated rather rudimentary displays. In part, this was due to the state of computing at that time.

All programs could be monitored by inserting output statements after each line of code. Each output statement would contain the previous source statement and all the variables that change as a result of executing it. Hille and Higginbottom (1983) wrote a UNIX shell program to automate that process for Pascal code. To describe how their system functions, the developers (pp. 76-77) stated: "Source statements are displayed by inserting them together with their line numbers as strings into writeln statements. Other *writeln* statements for additional information are assembled in similar fashion. Breakpoints are set by inserting *readln* statements." This system exemplifies an uncomplicated approach to program monitoring. Indeed the developers (p. 76) noted: "It took one of us two weeks to design, implement and debug the entire system."

Amenda (1990), as part of an undergraduate research project, also developed a pre-processor system for monitoring Pascal program execution. Consequently this system, like the previous one, augments an input program file by adding more Pascal code. In this case, though, the additional statements are calls to subroutines that display data in a window environment.

In particular, each window is divided into two parts. The top section contains the variables local to a subroutine or in the main program. The lower section contains the line of code presently being executed. Each time a subroutine is called, a window is created. The variables are updated when parameters are passed and when an assignment statement is executed.

This system was included among the program monitors even though it is called the Graphical Pascal Execution Model. Rectangular boxes and arrows are used to create windows with scroll bars. However, the windows are the only graphical component of the system. All data appearing in the windows are composed of characters.

DYNAMOD (Ross, 1991) is a program animator that was designed to assist introductory Pascal programming students. The system contains a library of animated programs. All of the programs display code, values of variables, output and a running count of the statements executed. As users step through an animated program, the values of all active variables and data structures are displayed on the screen.

While users of this system must work within a simple interface and are limited to monitoring only the programs in its library, most of them believe it is valuable. Ross (1991, p. 39) reported the results of a survey to support that contention. For example, 92% of the respondents replied affirmatively (8% negatively) to the question: "Has DYNAMOD helped you understand better how to visualize program execution dynamics?"

The PV prototype system (Brown, Carling, Herot, Kramlich & Souza, 1985) and FIELD (Reiss, 1990a) also support the use of graphics to view data structures during program execution. Users of both systems can step through the statements of C programs. The code and data are displayed in separate windows.

Unlike many of the previous systems, graphical statements are not inserted into programs to view execution in the PV system. Instead a technique called, *binding* is utilized to map diagrams to variables. For example a programmer could create a rectangle and divide it into various sections by inserting lines. Then, by making appropriate menu selections, the programmer could map that diagram to a C structure variable. Each section of the diagram would correspond to one field in the structure.

The binding of diagrams to variables may be established before or after the code to be animated is compiled. This is considered a key feature of the PV system. Unfortunately Brown et. al. (1985) did not describe how it is accomplished.

FIELD is a sophisticated program animation system that integrates UNIX editing, debugging and compiling facilities with tools for program and data visualization. The modules in the system communicate by passing messages according to a mechanism Reiss (1990a, p. 89) called, *selective broadcasting.* Briefly, in this communication mechanism all messages are sent to a controller routine that selectively redirects them to other modules. This is possible since each module, at start-up, registers message patterns that it should receive with the controller. (Precise details concerning message passing in FIELD are described in Reiss, 1990b.)

When using FIELD to debug code that manipulates a data structure, the programmer normally makes requests to update the displays. Alternatively, though, the programmer can force the data structure displays to be updated at particular prespecified statements. Program execution can be monitored graphically in this way. The values of variables can also be displayed in a window. When this window is visible, variables local to the currently active routine are displayed first, followed by global variables. A summary of the program visualization systems discussed above appears in Table 2.2.

Table 2.2: Program Monitors and Animators

Identification	**Classification**	**Purpose**	**Year**
Logo Subset	Monitor	Making Instructional Films	1975
Pl/1 Subset	Monitor	Making Instructional Films	1975
Unix Shell Program	Monitor	Instruction	1983
PV System	Animator	Debugging and Instruction	1985
GPEM	Monitor	Instruction	1990
FIELD	Animator	Debugging	1990
DYNAMOD	Animator	Instruction	1991

2.2 Debuggers

In the previous section, program monitors were defined as software tools that generate textual displays to depict program execution. Since debuggers function in that manner they are program monitors. However unlike systems described previously, debuggers are designed exclusively to help programmers find and correct run-time errors. To convey a sense for how programmers utilize them, typical debugging features are described next. Then the debugger, Dbxtool (Sun Microsystems, 1990a) and the debuggers in Turbo Pascal (Borland, 1988), Dr. Pascal (Visible Software, 1989) and MagPie (Delisle, Menicosy & Schwartz, 1984) are discussed.

Debugging tools provide the following facilities:

- displays for the values of variables
- execution suspension/resumption
- code stepping
- a subroutine instantiation display

As a program runs, the values of unstructured and structured variables are typically displayed in a clearly marked section of the monitor. The values are displayed beside or below their variable names. Users may view the values of variables indefinitely by suspending program execution. A suspended program may be restarted or resumed from the point at which it was halted. Some debuggers can suspend execution when a particular key is pressed. However inserting special statement markers, called breakpoints, into a program before execution begins is the usual suspension technique.

Code stepping permits the programmer to execute source statements one at a time. It can be implemented by inserting a breakpoint at each statement. Some debuggers support the following enhancement to single stepping: if the statement being executed calls a subroutine, the user can step through the code in the routine or resume single stepping when it returns. Lastly, debuggers generally provide a facility through which programmers can determine the order in which subroutines were called.

For each of the three commercially developed debuggers below, the user-interface is described. Also, the displays for unstructured data types and arrays are cited. Further, the technique for monitoring subroutine instantiation, including parameter passing and the scoping of variables, is noted. Magpie, the last system discussed in this section is treated uniquely because it was developed by researchers in an academic setting.

2.2.1 Dbxtool

Programmers working in the SunView Window Environment (Sun Microsystems, 1990b) can use the generic debugger, Dbxtool. It can be used to debug programs written in C, Fortran, Pascal and Modula-2 or a combination of those languages. The Dbxtool window consists of five subwindows. The status subwindow displays information such as the active file name and the line number range of the code displayed in the source subwindow. The buttons subwindow contains the commands that can be selected by positioning and clicking a mouse. Alternatively, debugging commands can be typed in the command subwindow. This window also contains output generated by the program. The display subwindow can be used to monitor how the values of variables and results of expressions change.

Values are displayed beside their variable names. Only one variable is displayed on each line. Consequently they quickly scroll out of view; however, even though it is tedious, they can be scrolled back into view. Arrays are displayed as lists. Appropriately, two-dimensional arrays are displayed in rows and columns.

In Dbxtool it is difficult to determine when a Pascal subroutine is called because the source code subwindow does not change. Also there is no attempt to depict parameter passing. Consequently the programmer, without assistance, must notice that the formal parameters become equal to the actual parameters when a subroutine is called. To determine the flow of control through subroutines, the programmer can only get a listing of the order in which subroutines were called. Lastly, the scope of a local variable is indicated by preceding its name with the name of the subroutine that contains it.

2.2.2 Turbo Pascal

The debugging environment in Turbo Pascal consists of three windows. The edit and watch windows are displayed initially. The edit window contains source code and status information similar to that displayed by Dbxtool. The watch window contains variables and their values. Commands are selected by key strokes, either directly or from pull-down menus. The edit and watch windows are displayed simultaneously although each window can be enlarged to cover the full screen. The user must switch to the output window to view displays generated by the program.

Again here, values are displayed beside their variables names and only one variable appears on each line. Text that scrolls out of view can be brought back into sight. The values in one-dimensional arrays are displayed suitably in a list. Two-dimensional arrays are displayed in nested lists. The linear nature of the nested list display is an awkward way to depict multi-dimensional arrays.

In contrast to Dbxtool, subroutine calls are depicted appropriately by displaying the code that will be executed next in the edit window. Unfortunately parameter passing and subroutine flow are treated in the same manner as they are in Dbxtool. With respect to scope, the watch window can contain a variable from anywhere in the program. As a result, accessible local and global variables are indistinguishable. However, the words, "unknown identifier" do appear beside variable names that are not within the scope of the currently executing routine.

2.2.3 Dr. Pascal

The debugging or visibility screen in Dr. Pascal is divided into four sections. The top line shows which function keys need to be pressed when initiating debugging commands. The second line, called the *procedure line*, displays a list of subroutine names that indicates the order in which subroutines were activated. It is updated every time a subroutine is called or exited.

Below the procedure line is the output area. In this area, program source code and the values of variables are displayed. Output from the program is displayed in the fourth and lowest section of the screen. Generally, the quality of the displays in Dr. Pascal are below the standard of the day. For example, the screen can only be configured in two colour combinations. Further, one of those combinations obscures certain characters to the point where they cannot be read.

Once again, values are displayed beside their variable names. In this system, though, more than one variable appears on a single line. In addition, unlike the two previous debuggers, the programmer does not need to specify which variables should be monitored. The display areas in Dr. Pascal contain as many variables as can be shown.

Multiple display areas are created during program execution. The main program and each subroutine call generate a distinct display area when instantiated. These display areas, complete with source code, scroll on and off the screen. To the detriment of the programmer, when all of the variables in a routine cannot be displayed, there is no way to scroll the hidden variables into view at run-time. After terminating execution, the programmer can move the declaration of the hidden variables ahead of declarations of variables that were visible. However when the program is run again, previously visible variables will be hidden. While that course of action may seem nearly futile, it is all that the programmer can do.

Arrays are displayed suitably. One-dimensional arrays are displayed on a single line. Two-dimensional arrays are displayed in rows and columns.

The activation of subroutines is dramatically displayed by the appearance of display areas. Also, the procedure line is updated as noted earlier. The display areas are also useful for depicting local variables. The code in which local variables may be accessed appears directly beside or above, depending on the programmer's preference, the display areas containing them. Global variables are contained in other display areas that are evident if they have not scrolled off the screen. Unfortunately, parameter passing is treated in the same subtle way as it is in the two previous debuggers.

2.2.4 Magpie

All aspects of the Magpie programming environment are integrated. In this environment the Pascal programmer accesses both source code and information about its execution state through windows, called browsers. The windows are divided into sections, referred to as panes. Commands are selected from pop-up menus via a mouse. Each type of pane has a unique menu.

One way to view the execution state of a program is to display a set of code and stack browsers. A code browser contains the executable statements for one scope level (i.e., a subroutine or the main program). A stack browser consists of three panes. The leftmost pane contains a list of names that depicts the order in which subroutines were activated. When the programmer selects one of those activations, the variables local to it are placed in the middle pane. Also, the values of those variables are displayed in the last pane. Note also that one variable is displayed per line and one-dimensional arrays are represented linearly. This monitoring method can be used to review completed actions. The user must write event monitoring code to single step through a program, as discussed below.

According to Delisle, Menicosy and Schwartz (1984, p. 55), "Execution and debugging functions are implemented by instrumenting the actual code for a procedure with debugging code." (The reader may find the previous statement more illuminating by substituting, "supplementing" for "instrumenting.") The debugging code is written in Pascal but it does not become part of the program source code. Instead, the code is entered into an event monitoring browser.

Since debugging code is written in Pascal the potential for execution monitoring is greater in Magpie than in all three of the previous debuggers, at least for experienced programmers. In contrast, inexperienced programmers are restricted to the execution reviewing technique described earlier.

2.3 The Instructional Foundation of EPAS

In addition to computer science, this research extends to fields such as educational psychology and instructional design. A separate examination of both of those interdisciplinary fields is beyond the scope of this work. The reader interested in an extensive discussion on the contribution of those fields to the development of instructional software environments is directed to van Berkum and de Jong (1991). For alternative viewpoints, the reader is directed to Hannafin and Reiber (1989) and Merrill (1987). For the purposes of this research, discussion of computer-based instructional simulations will be restricted to the instructional design theory upon which the tutorial module in EPAS is based.

The tutorial module within EPAS is based on the instructional theory for the design of computer-based simulations developed by Reigeluth and Schwartz (1989). Before describing that theory it will be worthwhile to gain insight into the extent to which Reigeluth and Schwartz perceive the viability of computer-based simulation systems, such as EPAS. The following two quotations are provided for that purpose.

> The advent of the computer has made possible a new and exciting form of learning environment, the simulation. We now have the technology for a powerful form of instruction that is both dynamic and interactive and that can provide considerable variety within a simulated environment. Even a personal tutor is incapable of such versatility (Reigeluth and Schwartz, 1989, p. 1).
>
> The dynamic and interactive nature of computer-based simulations provides an ideal medium for teaching students content that involves change (Reigeluth and Schwartz, 1989, p. 2).

When briefly discussing an instructional design theory, as is the case here, it is possible to focus exclusively on the prescriptive aspects of the theory. This is in keeping with the Reigeluth's (1983, p. 4) assertion: "[Instructional design theories are] concerned primarily with prescribing optimal methods of instruction to bring about desired changes in student knowledge and skills." The Reigeluth and Schwartz instructional theory for the design of computer-based simulations consists of the four phases, *introduction*, *acquisition*, *application* and *assessment.*

The introductory phase should contain a sample simulation with simultaneous descriptions. In this way the learners should become aware of how the simulation will proceed. Also the learners should understand how to use the program. For example, they should know how to select items from menus to initiate the learner control features.

Following the introduction is the acquisition phase. During this stage the learner should acquire knowledge about a principle or the steps of a procedure. The developer may select an *expository* or *discovery* approach in this phase. For example, if the expository approach is taken to teach a principle, the principle would simply be stated. Alternatively, the discovery approach makes the learner formulate the principle by attending to simulations in which the principle is applied.

Throughout the application phase the learner should develop the ability to use the principle or procedure introduced in the previous stage. The most pertinent aspect of this phase is the opportunity for divergent practice. It is essential to include the following two features in the practice exercises. First, the practice items must force a user response. Second, the response must be accompanied by feedback.

The purpose of the assessment phase is to determine the extent to which the student mastered the content. The test items should be based on the objective. Further, they should be new to the learner and vary in difficulty and divergence.

The theory also provides guidelines for variations on the general model. The guidelines allow the developer to amend the model to accommodate a particular target audience or set of objectives, for instance. The precise manner in which EPAS incorporates the prescriptive elements of this theory is detailed in Chapter 3.

2.4 Summary

The two basic types of program visualization systems, algorithm animators and program animators, were discussed. The algorithm animators were defined as systems that depict programs in an abstract manner. These systems were described briefly because they are quite unlike EPAS. In contrast, program animators received greater attention because, in keeping with EPAS, they depict program execution concretely. Specifically, three program animators were discussed. The use of graphics to depict data structures was cited as the primary function of those systems.

Further, it was shown that *program monitors*, which are similar to program animators, have also been developed to help programmers visualize execution. Program monitoring systems depict execution concretely, but do not use graphics for that purpose. Much attention was directed to the special type of program monitor, the debugger. Debuggers provide facilities through which programmers can attempt to locate and correct errors. However, using them to gain insight into parameter passing mechanisms or to determine the scope of a variable can be difficult. The complexity of the program visualizations systems reviewed varies markedly. However, even the most sophisticated ones do not depict run-time stack activity graphically, nor do they permit reverse execution.

The last part of the review describes the prescriptive elements of an instructional design theory for the development of computer-based educational simulations. The theory presents techniques for teaching procedures and principles. According to the theory, the four phases, introduction, acquisition, application and assessment, should be incorporated into instructional simulation software. For the purposes of this summary, the first and last phases are self-explanatory. The key aspect of the second phase is the presentation of instruction. The application stage provides the learner with opportunities for divergent practice. Learner responses in this stage must be accompanied by feedback. Lastly, it should be noted that the theory is rather new and consequently, largely untested.

Chapter 3

Methodology

The tools and techniques used to test the hypothesis stated in Chapter 1 are described in this chapter. The most important tool used in this research is EPAS. Consequently, the modules in it are examined first. The compilation modules, namely the scanner, parser and symbol table generator are described briefly. In contrast, the animation and tutorial modules are described in detail. Lastly, the small study that was conducted to test EPAS is described.

3.1 The Compilation Modules

Before a compiler can be written, a grammar must be defined precisely. An extended Backus-Naur Form (BNF) description of the Pascal grammar compiled by EPAS appears in Appendix A. Technically, since the BNF description is free of left-recursion and left common factors it defines an LL(1) grammar. For clarity of expression, the BNF for EPAS was derived from Pascal BNF descriptions found in Cherry (1980) and Sun Microsystems (1989).

In practice, EPAS treats identifiers, unsigned integers, unsigned reals and character strings as terminals in the grammar. Thus for EPAS, the grammar rules effectively end with the definition of the *expression list tail.* (A line appears in Appendix A at that point.)

The compilation modules in EPAS are described briefly for two reasons. First, compilation modules are present in EPAS because they would exist in a complete program animation system. Including them in the prototype is necessary to gain insight into how they should be enhanced to create a complete system. EPAS would still fulfill its restricted program animation function if all of the compilation modules were replaced by one short routine that simply read the characters from an input program file and placed them into an array. Second, techniques for developing compilation modules are well documented in textbooks (e.g., Aho, Sethi & Ullman, 1988 and Fischer & Leblanc, 1988).

3.1.1 The Scanner

Scanning in EPAS begins by reading the first three characters in the input file. If the input file contains fewer than three characters, scanning is terminated and an error message is displayed. Each time a character is processed, a new character is read.

To determine what token is presently being scanned, the first character is examined. If it is a letter, the token to be returned will be that of the identifier or a reserved word. Consequently, characters from the input file are read and collected until a character other than a letter, a digit or the underscore is encountered. The collected characters form an identifier or a reserved word.

Next, the minimal-perfect hashing technique described in Chicelli (1980) is used to distinguish between identifiers and reserved words. Chicelli's hashing technique ensures that a comparison at only one array location is needed to make the distinction. Just one test is necessary because the hash value of each of the thirty-seven reserved words is unique.

When the characters of an identifier or reserved word have been collected, a hash value is calculated. Then the collected characters are compared against those in the reserved word array at the one location denoted by the hash value. If all characters match, the appropriate reserved word token is returned, otherwise the identifier token is returned.

If the first character is not a letter, it may be a digit. If so, the token to be returned will be either the integer or real constant. After storing the first digit, contiguous digits, if any, are read and collected. Additionally a decimal point, an exponent indicator (i.e., **e** or **E**) and a plus or minus sign for an exponent may be read. If a decimal point or an exponent indicator is present, the real constant token is returned. Alternatively the integer constant token is returned if those characters are absent.

If those first two cases are not pertinent, the first character could be an apostrophe. The apostrophe uniquely defines the presence of a string constant. In this case, characters are read until the next single apostrophe or newline character is encountered. If a newline character is encountered before a terminating apostrophe, an error message is generated. When two consecutive apostrophes are present, a single apostrophe is appended to the string.

If the current character is not a letter, a digit or an apostrophe, it could be the first character of a two-character token (e.g., >= or <>). To determine if it is, the first character and the one following it are tested to determine if one of the five two-character tokens is present. If one of them is present, it is returned.

If the four previous cases do not apply, the first character could be a one-character token (e.g., the semicolon, the colon or the comma). When this is so, the token denoting the single character is returned.

If all five of the previous cases have been considered and a token has not been returned, an invalid character is present. In this event, an error message is generated.

Also note that source code comments, initiated by { or (*, and white-space characters such as newline, tab and spacebar are disregarded by the scanner. This is the usual practice because white-space characters do not form any part of a token. Furthermore, characters in comments are not part of the code of a program.

3.1.2 The Parser

A *top-down recursive descent* parser was incorporated into EPAS. Since EPAS parses an LL(1) grammar, each rule was written as a subroutine. For example, the rule:

$< program >::\Rightarrow< program\ heading >< declarations >< block > .$

was written in C as:

```
program()
{
   program_heading();
   declarations();
   block();
   match_tokens(PERIOD_TOKEN);
}
```

The angle brackets were stripped from the nonterminal, $< program >$ to form the subroutine name. The remaining part of the rule determined the contents of the subroutine. In particular, each nonterminal became a subroutine call and the one terminal forced a call to the *match_tokens* routine.

The match_tokens routine determines if the currently scanned token is the same as the expected one. The expected token is specified as the argument in the call to match_tokens. In the example, the PERIOD_TOKEN is expected.

In a syntactically valid program, all scanned tokens match the expected tokens. The parser does not attempt to recover from syntax errors. However, it does generate an error message that identifies the mismatched tokens.

Lastly, to increase processing speed, tail-recursive subroutines were written iteratively. For example, the rule:

$< \textit{label list tail} > ::\Rightarrow\ , < \textit{label} >< \textit{label list tail} > \mid < \textit{empty} >$

was written in C as:

```
label_list_tail()
{
   while (token = get_next_token() == COMMA_TOKEN) {
      match_tokens(COMMA_TOKEN);
      label();
   }
}
```

3.1.3 Symbol Table Generation

While parsing, one symbol table is created for each new block level. When the end of a level is detected, its symbol table is rescinded. In this way, memory assigned to dynamically allocated structures is recovered.

Figure 3.1 contains a simplified representation of the symbol table structures that would be generated if the program below were parsed. It is included only to exemplify a limited number of the structures related to symbol table generation. A complete analysis would cover several pages.

```
program  sample;
const
   BBB = 4;
```

```
var
   BAC : integer;

   procedure   level1(a : integer; var b : integer);
   var
      c : 'a'..'z';
   begin
   end;

begin   { Main }
end.
```

The structure at the top of Figure 3.1 is the only structure related to symbol table generation that is fixed at run-time. It is a thirty-two element array of pointers to symbol table structures. The symbol tables contain 211 pointers to symbol table entry structures.

There are eight symbol table entries in the figure. Seven of them correspond to the seven identifiers in the program. The other symbol table entry is the system type declaration, *integer*. Every symbol table entry contains an identifier. Identifiers are passed through a hashing function to determine symbol table entry positions. The hashing function computes the sum of the characters in an identifier and returns that sum modulo 211. Identifiers that hash to the same position are resolved by chaining their entires in a linked list. This is shown for the identifiers, *BBB* and *BAC*.

The figure also contains two parameter entry structures attached to the subroutine declaration entry, *level1*. As shown, parameter entries form a linked list. Each entry defines a *value*, *variable* or *subroutine* parameter. In addition, each parameter entry points to a data type.

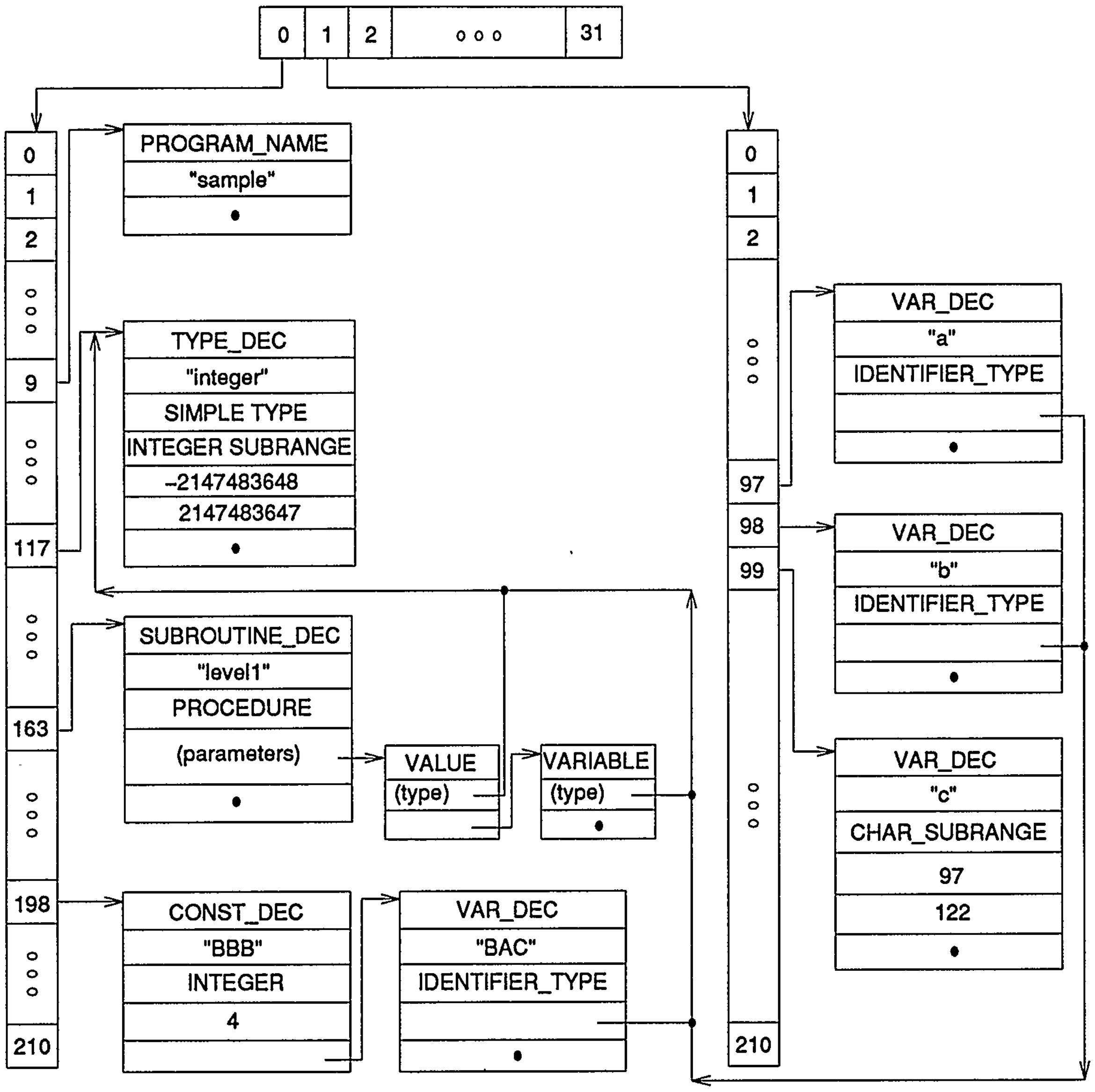


Figure 3.1: Symbol Table Structures

3.2 The Animation Module

This module was designed to depict run-time stack activity for eleven specific Pascal programs. The animation module, like all others in EPAS, was written in C and runs on a color SPARC station. In contrast to the other modules, this one calls X-Window library routines. Generally, X-Window library routines are useful for developing user-interfaces. For example, the routines can be used to create, manipulate and destroy windows.

Details concerning this module are presented in three parts. First, the statement execution and pace adjustment features of the animation module are noted. Also, the programming techniques used to incorporate them are documented. Second, statement execution examples are presented. The examples depict all of the run-time stack activities implemented in EPAS. In the final part, the reverse execution feature is discussed.

3.2.1 Statement Execution and Pace Adjustment

The animator performs one of three functions. The user indicates which function will be performed by using a mouse to select a menu item. The three menu items are *Next*, *Adjust Pace* and *Previous.*

Next is selected to execute program statements one at a time. The user can view run-time stack activities as a statement is executed. In EPAS, a stack of rectangles is used to depict the consecutive set of memory cells in a run-time stack. All run-time stack activities access the stack of rectangles. Table 3.1 contains the five run-time stack activities implemented in EPAS and indicates how each one is depicted.

Table 3.1: Run-time Stack Activities in EPAS

Activity	Change to the Stack
Declaration	a memory cell is assigned to a specific variable
Output	memory cells may be read to access variables
Procedure Call	memory cells are assigned to parameters
Module End	local variables are removed from memory cells
Memory Read/Write	values are read from and written to memory cells

The program execution simulations are controlled by the values in a data file. Each line of the data file contains twenty-six values and defines one stack activity. Data file entries are described briefly below. Specific details concerning stack event entries are presented in the next two sections.

The first seven entries identify one or two program segments that will be highlighted. Highlighted program segments include variable declarations and executable statements. A program segment is highlighted when it is the next statement to be executed. Usually only one segment is highlighted. However in the case of a procedure call, both the calling statement and the corresponding procedure heading are highlighted.

The eighth entry identifies one of the five stack activities. The interpretation of the remaining entries depends on the identity of the stack activity. The information required by the animator for each of the stack activities is listed in Table 3.2.

With respect to Table 3.2, the following points are important.

- Each source and target stack cell is uniquely identified by one integer.
- The number of parameters in Procedure Calls is limited to three.
- Three source and two target cells can be referenced by one Memory Read/Write activity, at most.

Table 3.2: Data Requirements for Stack Activities

Stack Activity	**Data Requirements**
Declaration	variable name number of indices (array declarations only)
Output	output string position (row & column) of the string in the Output Window source cells
Procedure Call	procedure name number of parameters formal parameter names formal parameter types constant values or stack cell identities
Module End	module name number of local variables
Memory Read/Write	number of source cells the source cells number of target cells the target cells the value for the target cells previous stack events that modified the target cells

These limitations apply only to the prototype, EPAS. A complete system should not be constrained in the same manner. The limitations were imposed after analysis of the eleven input programs revealed that they were sufficient for animation purposes.

The user can control the speed of the animator by selecting *Adjust Pace.* To adjust animator speed, the mouse is used to select one of the bars in the Delay Window. A picture of the Delay Window appears in Figure 3.2. A direct relationship exists between the length of the bars and the length of time it takes the animator to complete operations.

The speed of the animator is most evident when a stack cell is highlighted. That is, when the background color of a cell is gray instead of white. The amount of time

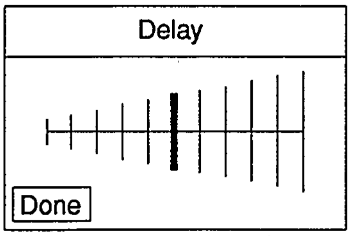
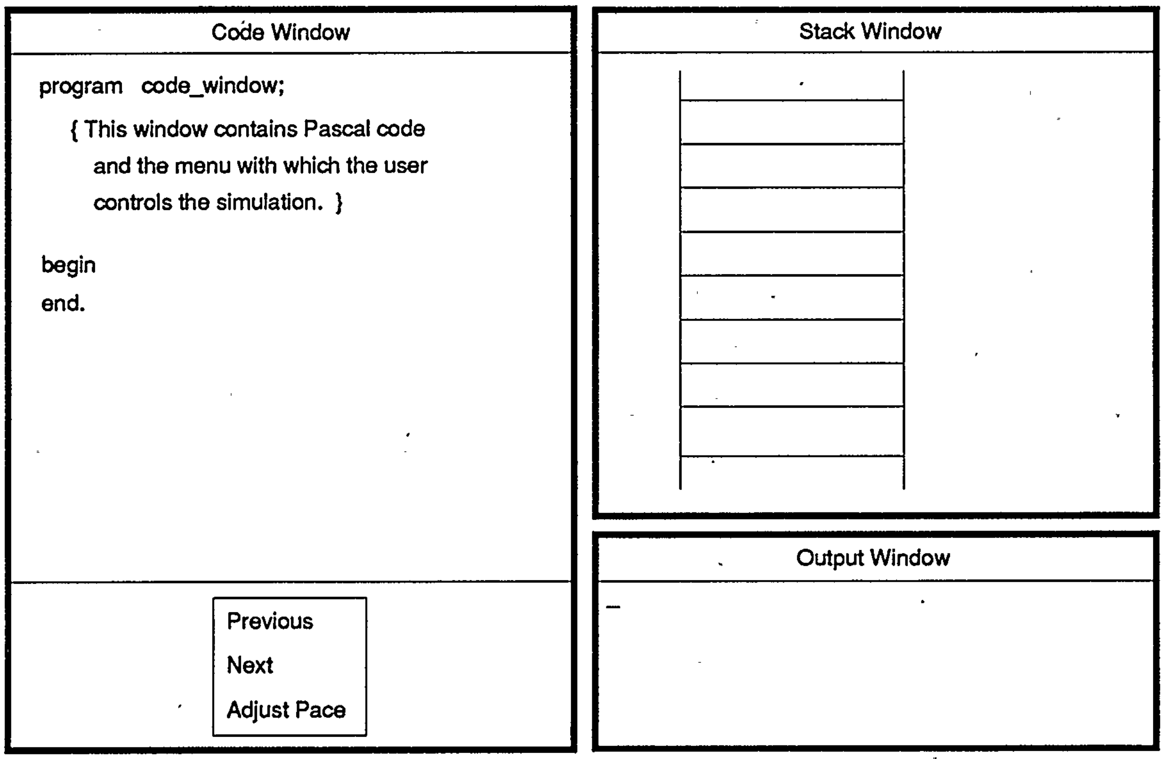


Figure 3.2: The Delay Window

that a cell remains highlighted can vary from zero to twelve seconds.

3.2.2 Statement Execution Examples

The general layout of the simulation screen in EPAS is shown in Figure 3.3. Both



```
program   code_window;
   { This window contains Pascal code
     and the menu with which the user
     controls the simulation. }

begin
end.
```

Figure 3.3: General Layout of the Simulation Screen

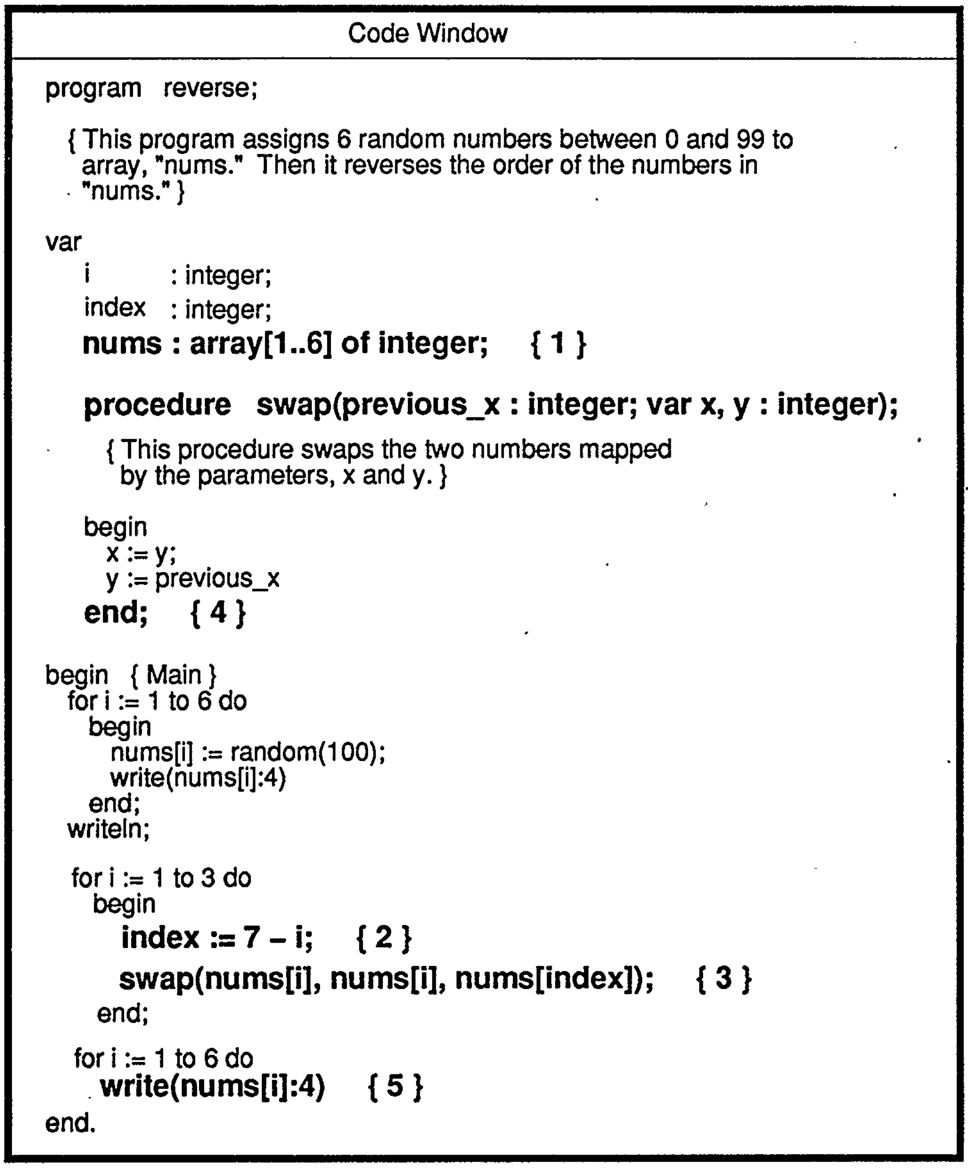
the length and the width of the windows were reduced by approximately fifty-three percent to display them on the page. The actual width of each window is sixteen centimeters. The actual lengths of the code, stack and output windows are twenty-two, fifteen and one half and six centimeters, respectively. Also note that there are actually twenty-one stack cells instead of the eight shown in the figure.

For each simulation, a complete Pascal program is displayed in the Code window. The next statement to be executed appears red. All other program statements are black. (In the figures of this document, the next executable statement is written in boldface type.) The menu in the lower portion of the Code window is used to control the simulation. The Stack window contains the run-time stack in which variables and their values appear. Output generated by *write* and *writeln* statements is displayed in the Output window.

The Code Window in Figure 3.4 contains the program that is used for all of the following five execution examples. When EPAS is running, each executable statement is highlighted individually. However, rather than highlighting each statement separately and displaying the Code window five times, the pertinent statements are highlighted simultaneously. The highlighted statements are numbered on the basis of execution order.

Figure 3.5 shows the stack window before and after executing the first highlighted statement. Declarations allocate space on the stack for variables. They do not assign values to variables. Consequently after declarations, the values of variables are undefined. This is depicted in the figure by the question marks.

The second highlighted statement is a Memory Read/Write event. The state of the stack before, during and after it is executed is shown in Figure 3.6. Before execution, only *index* is undefined. Notice that one variable is allocated by the system


Code Window

```
program   reverse;

  { This program assigns 6 random numbers between 0 and 99 to
    array, "nums."  Then it reverses the order of the numbers in
    "nums." }

var
   i        : integer;
   index    : integer;
   nums : array[1..6] of integer;     { 1 }

   procedure   swap(previous_x : integer; var x, y : integer);

     { This procedure swaps the two numbers mapped
       by the parameters, x and y. }

   begin
     x := y;
     y := previous_x
   end;    { 4 }

begin  { Main }
  for i := 1 to 6 do
    begin
      nums[i] := random(100);
      write(nums[i]:4)
    end;
  writeln;

  for i := 1 to 3 do
    begin
      index := 7 – i;     { 2 }
      swap(nums[i], nums[i], nums[index]);     { 3 }
    end;

  for i := 1 to 6 do
    write(nums[i]:4)    { 5 }
end.
```

Figure 3.4: Sample Program Code Window

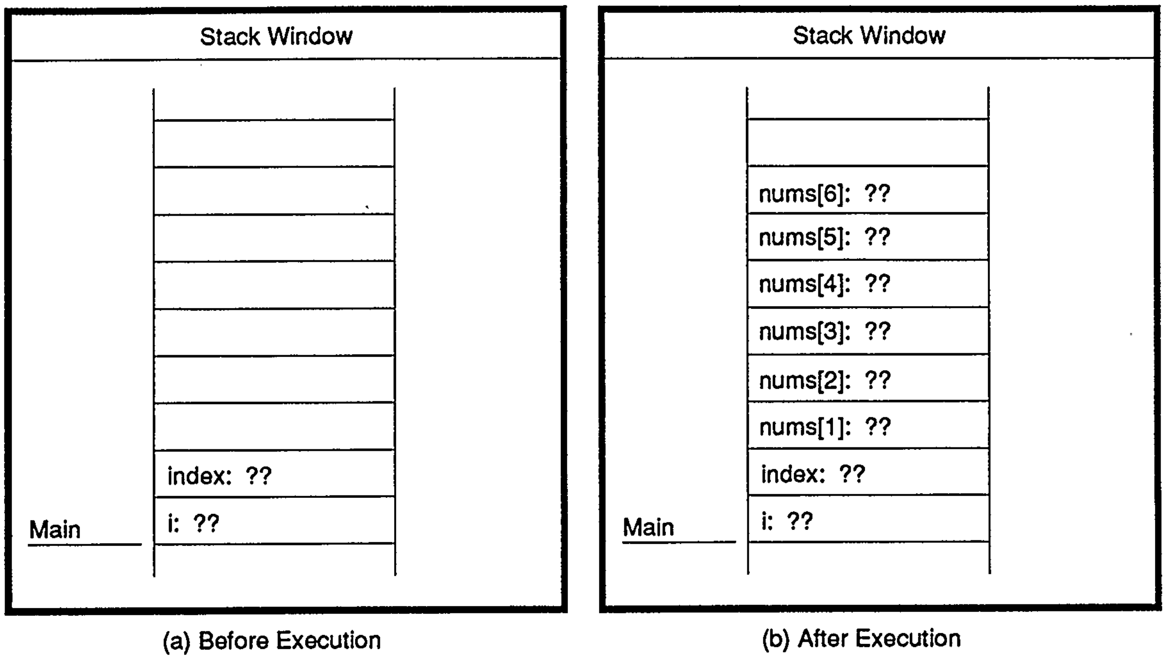


Figure 3.5: Depicting a Declaration

for holding results generated when evaluating expressions. To simplify matters for the learner, one system variable is allocated for the main program and for each procedure. Variable names allocated by the system always begin with the characters *%exp*. The system variable in Figure 3.6 is *%exp1*.

The first task in processing the highlighted assignment statement is computing the expression $7 - i$. Execution Step 1 shows the variable i against a light gray background to indicate a memory read operation. The black background filling the system expression cell depicts an impending write operation. (In EPAS, cells are always highlighted with a light gray background against red text for values of source cells and blue text in the case of target cells. Also the background fills the entire cell, not just the area of a cell outside the rectangular region containing a variable and its value.) Note that the value 7 was written to *%exp1* during previous processing;

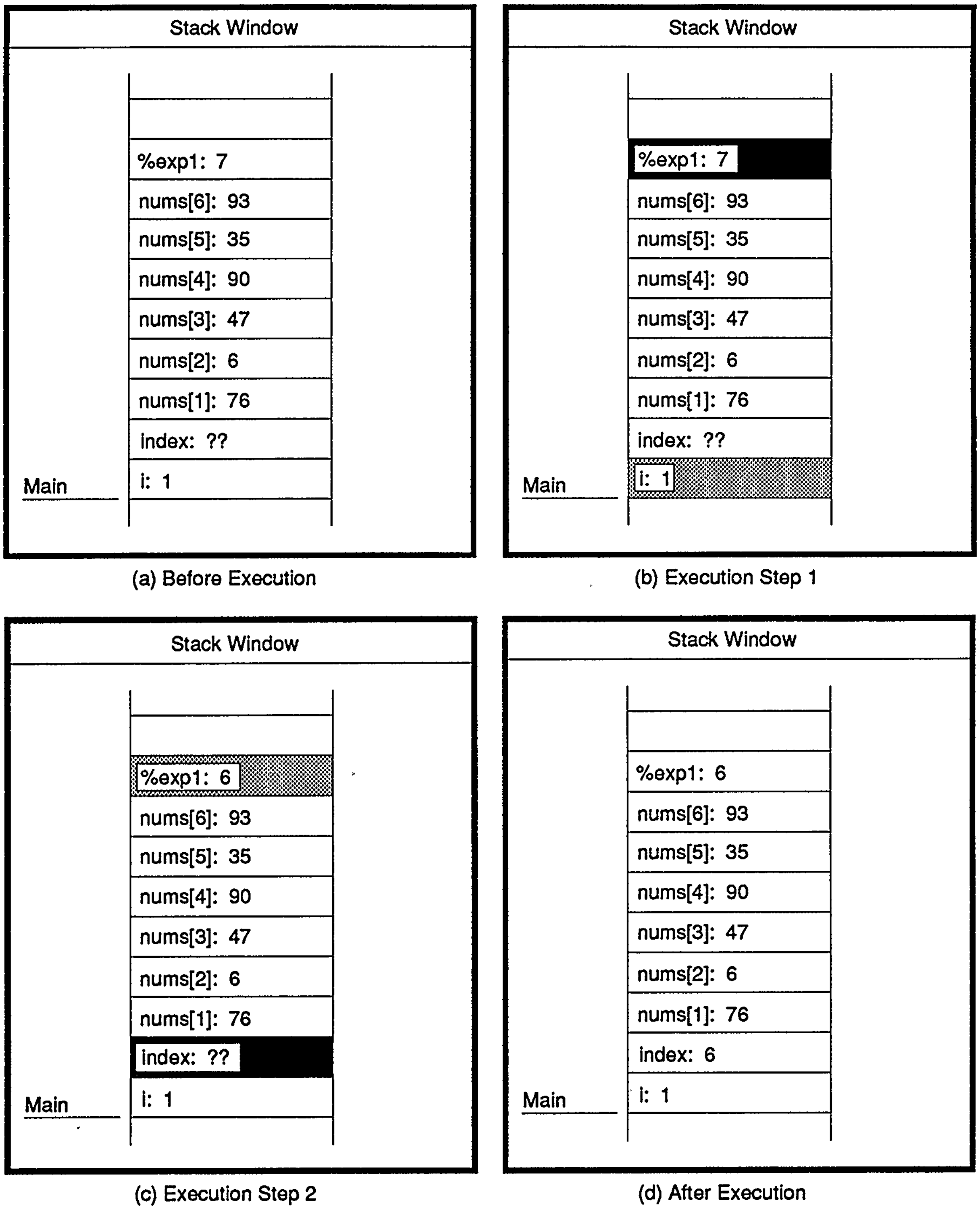


Figure 3.6: Depicting a Memory Read/Write Event

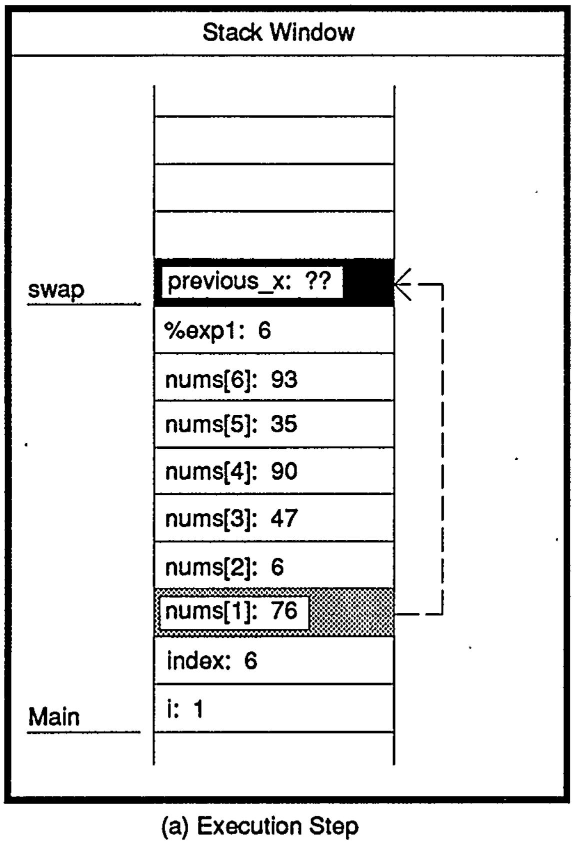
it was not written to it because 7 appears in the current expression.

In the example, $7 - i$ is 6 because i is 1. When the 6 has been written to *%exp1*, it is then read and, in accordance with the assignment statement, should be written to the variable *index*. This is shown in Execution Step 2. After execution, the value of *index* is 6.

A procedure call is the third highlighted statement. The procedure heading that corresponds to it is also highlighted. The last frame in Figure 3.6 shows the stack before the call is made. During execution, the value parameter is passed first. This is shown in the first frame of Figure 3.7. The dashed line depicts the connection from the actual parameter to the formal parameter. The dashed line disappears when the formal parameter has been assigned the actual parameter value. This result is shown in the second frame of Figure 3.7. Also, the links from the formal variable parameters to their actual parameters are displayed. The formal variable parameter links exist until the procedure is exited.

A Module End event is the fourth highlighted statement. Figure 3.8 shows the stack before and after the highlighted Module End event is executed. Local variables and the module name are actually removed gradually.

The last stack activity example is an Output event. Since Output events never write values to memory, the stack remains unchanged throughout execution. However, values are written to the Output Window. Figure 3.9 shows the constant stack state throughout the execution of the last highlighted statement. It also displays the changes to the Output Window.



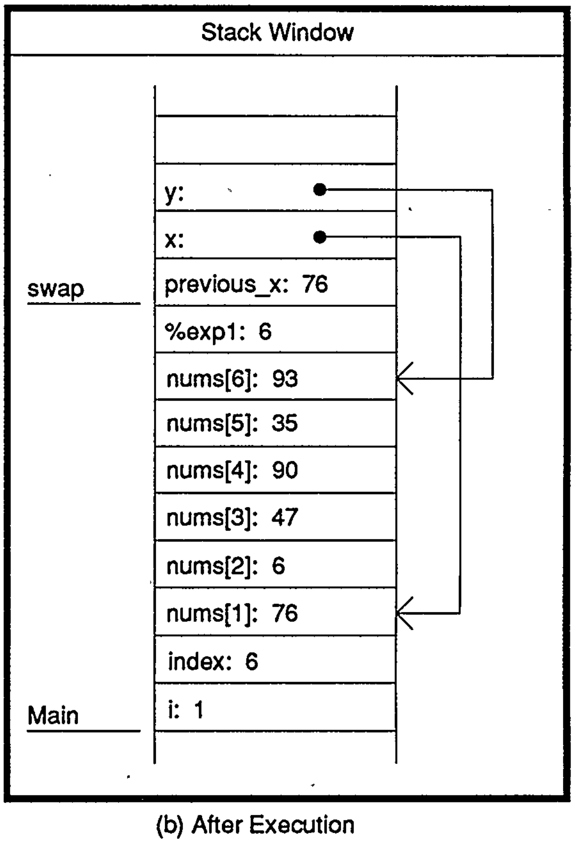


Figure 3.7: Depicting a Procedure Call

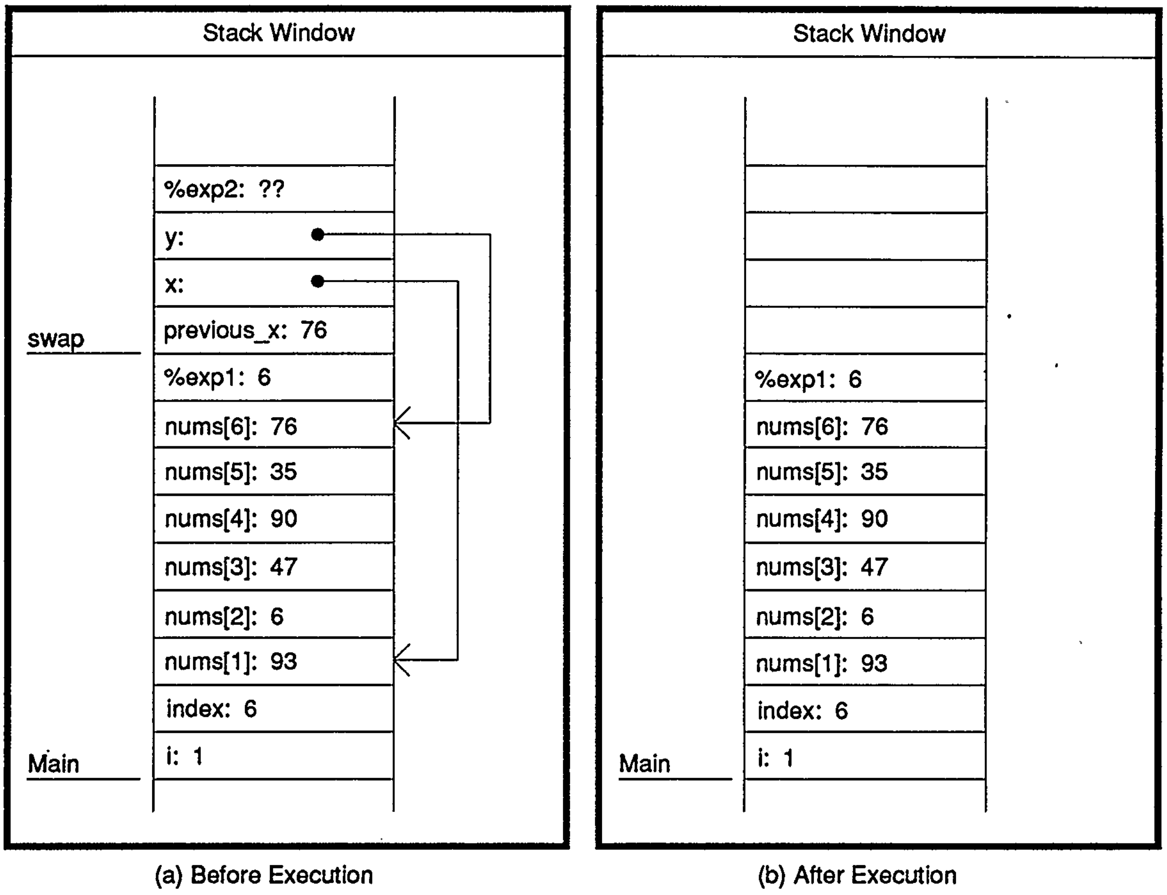
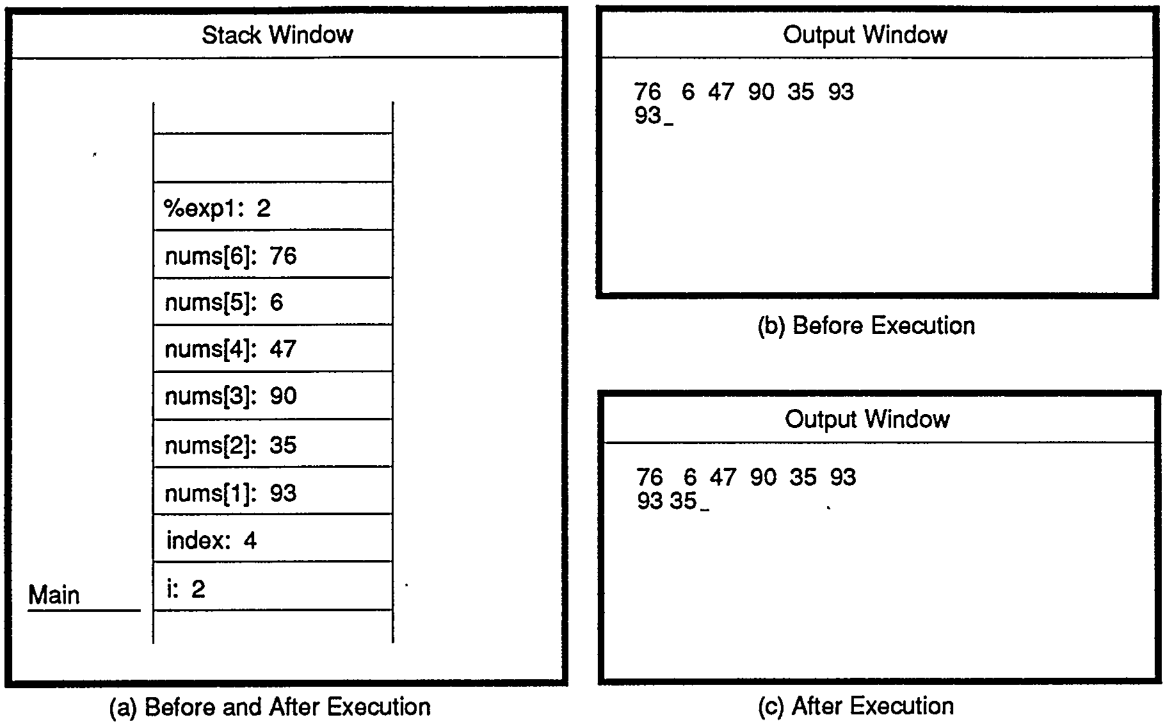


Figure 3.8: Depicting a Module End Event



```
76  6 47 90 35 93
93_
```

```
76  6 47 90 35 93
93 35_
```

Figure 3.9: Depicting an Output Event

The data file entries for the stack events described above are listed in Figure 3.10. As noted earlier, the first seven entries (i.e., Fields A to G) define program segments for highlighting purposes. Consequently those fields are not described further, except in the case of Output events. As noted below, Fields E to G serve a different purpose in Output events. Fields Y and Z are discussed in the reverse execution section.

The data file entries in the first line of Figure 3.10 are used to depict the Declaration event exemplified in Figure 3.5. Field I contains the variable named defined in the declaration. The string, "??" in Field J is displayed in the Stack Window to show that the initial value of a variable is undefined. The value in Field P defines the number of stack cell allocations. Six stack cells are allocated for this array

Highlighted Statement Number from Figure 3.4	Stack Event Field Letters																									
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	9	3	4	9	11	23	0	nums	??	"	"	"	"	"	6	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
2	1	31	9	14	–1	–1	–1	1	"	6	"	"	"	"	"	–1	2	1	8	1	–1	0	8	–1	23	2
3	2	32	9	35	11	15	46	2	swap	76	"	"	previous_x	x	y	3	–1	–1	1	2	2	2	2	7	–1	–1
4	1	19	3	3	–1	–1	–1	3	swap	"	"	"	"	"	"	4	3	–1	1	2	2	–1	2	7	–1	–1
5	1	36	6	16	1	4	–1	4	"	"	"	"	"	"	"	8	–1	1	–1	–1	–1	3	–1	–1	–1	–1

Note: Numeric fields not used contain –1
Text fields not used contain "

Field Descriptions

A	Number of program segments to highlight
B	Row number of first highlighted segment
C	Column number of first highlighted segment
D	Length of first highlighted segment
E	Row number of second highlighted segment
F	Column number of second highlighted segment
G	Length of second highlighted segment
H	Stack event identity number 0 – Declaration 1 – Memory Read/Write 2 – Procedure Call 3 – Module End 4 – Output
I–O	Utility strings
P–X	Utility values
Y	Previous pertinent stack event number for target cell 1
Z	Previous pertinent stack event number for target cell 2

Figure 3.10: Stack Event Entries

declaration event. The other fields in Declaration events are not used.

Data file entries on the second line of Figure 3.10 are used to depict the Memory Read/Write event exemplified in Figure 3.6. Fields Q and R specify the number of target and source cells respectively. The number of source cells does not include references to system expression cells. Fields S to U and V to X reference target and source stack cells. The stack cells are numbered in ascending order beginning with zero. Since there are twenty-one stack cells the top one is Cell 20.

In this Memory Read/Write event there are two target cells (as per Field Q). Consequently there are two execution steps as shown in Figure 3.6. In the first execution step, Cell 8 (Field S) is the target cell and Cell 0 (Field V) is the source cell. Whenever there are two target cells, the system expression cell targeted in the first execution step is always the source cell in the second execution step. In this case, Cell 1 (Field T) is the second target cell and Cell 8 (Field W) is the second source cell. The value written to the target cells is located in Field J.

The data file entries for the Procedure Call depicted in Figure 3.7 appear in the third line of Figure 3.10. The name of the procedure is given in Field I. The names of the formal parameters are listed in Fields M to O. The number of parameters is stated in Field P.

Values denoting parameter type are given in Fields S to U. This example uses two different parameter types. Parameter Type 1 (Field S) is a value parameter that receives a value from a variable. In contrast, Type 0 is a value parameter that receives a constant value. Type 2 (Fields T and U) is a variable parameter. Fields V to X specify actual parameter cells. These values are used to display parameter arrows. Note that Field J contains the datum for the value parameter.

The entries on the fourth line of Figure 3.10 are used to depict the Module End event in Figure 3.8. Field I contains the procedure name. Field P specifies the number of local variables, including the system expression variable. Field Q contains the number of parameters. With one exception, Fields S to X in the Procedure Call above and in this Module End event are the same. Field V in the Module End event is not used because knowledge of the actual parameter cell that mapped to the formal value parameter is not required at module termination. Recall that arrows linking actual and formal value parameter cells exist only when a procedure is called, not when it is terminated.

In Output events, such as the one on the fifth line of Figure 3.10, only one code segment is highlighted. Thus, Fields E to G are available for other information. Fields E and F contain the row and column references at which the output string will be displayed in the Output Window. Field G is not used. Field P holds a value for indexing the array of output strings. Lastly, as is the case for Memory Read/Write events, Fields R and V supply source cell information.

3.2.3 Reverse Execution

When the user selects *Previous* from the animator menu, the previous program state is restored. *Previous* can be selected repeatedly until the first statement has been reversed. The techniques for reversing the five stack activities are described below.

To reverse a declaration for an unstructured variable, the topmost cell is popped off the stack. In the case of an array declaration, the topmost set of cells is removed from the stack. Popping the topmost stack cell is a two-step process in EPAS. First, the topmost stack cell window is destroyed. Also, if the cell contained a variable parameter, the arrows pointing to the corresponding actual parameter cell are erased. Second, the pointer to the topmost array element in the stack maintained by EPAS is decremented.

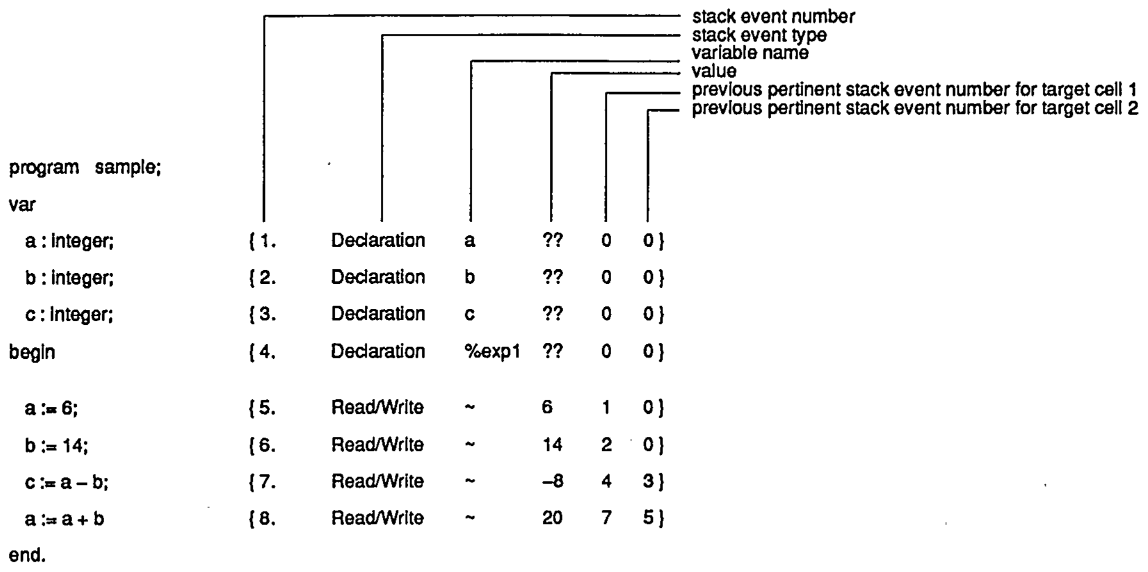
An Output event is rescinded by deleting its string from the Output Window. To accomplish this, the string is actually written to the Output Window again. However, the string disappears because the background color is used to display the text. The stack is unaffected when this event is reversed because output events do not write values to memory. Consequently, the Stack Window is not changed.

To reverse a Procedure Call, the stack is restored by popping off the cells that were allocated to parameters. Since the number of parameters is given in stack events that define Procedure Calls (see Table 3.2), the number of parameters to pop is also known. The method for popping cells was discussed in the context of declaration reversal.

In contrast to Procedure Calls, a Module End event is nullified by pushing cells back on the stack. Table 3.2 showed that the number of local variables in a module is known. The number of local variables is also the number of cells that must be restored. The two-step process for restoring cells is the converse of the method for popping them.

The following example illustrates how a Memory Read/Write event is reversed. Figure 3.11 contains a Pascal program and stack events that correspond to declarations and statements. While the stack events are incomplete, they contain the data necessary to discuss reverse execution. Note that zero entries in previous pertinent stack event numbers indicate that they would never be referenced. For instance, since stack event #5 targets one memory cell, only one previous stack event would ever be referenced. Also note that variable names are not referenced in Memory Read/Write events.

Figure 3.12(a-e) shows the state of the stack after executing Stack Events 4 through 8 in Figure 3.11. Stack Event 8 first changes the system variable, *exp1* and then it alters the programmer defined variable, *a*. After executing Stack Event 8, execution is reversed. The result of the reversal is shown in Figure 3.12(f). The previous values of *%exp1* and *a* were located in Stack Events 7 and 5, respectively. As shown by the equivalence of Frames d and f in Figure 3.12, rescinding Stack Event 8 returns the state of the stack to what it was after Stack Event 7 was executed.


stack event number
stack event type
variable name
value
previous pertinent stack event number for target cell 1
previous pertinent stack event number for target cell 2

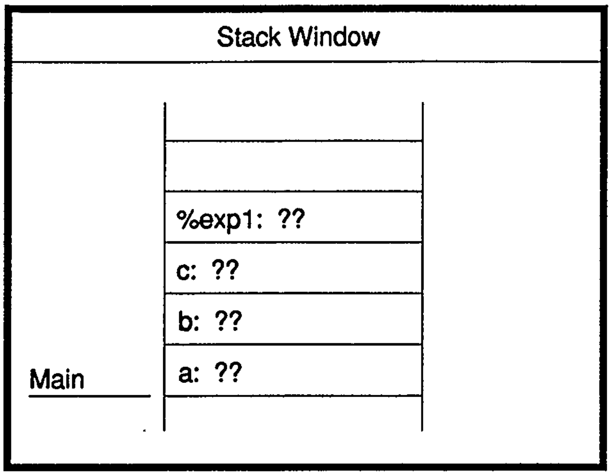
```
program  sample;
var
  a : integer;      { 1.   Declaration   a       ??   0   0 }
  b : integer;      { 2.   Declaration   b       ??   0   0 }
  c : integer;      { 3.   Declaration   c       ??   0   0 }
begin               { 4.   Declaration   %exp1   ??   0   0 }

  a := 6;           { 5.   Read/Write    ~       6    1   0 }
  b := 14;          { 6.   Read/Write    ~       14   2   0 }
  c := a – b;       { 7.   Read/Write    ~       –8   4   3 }
  a := a + b        { 8.   Read/Write    ~       20   7   5 }
end.
```

Figure 3.11: Sample Program and Stack Events

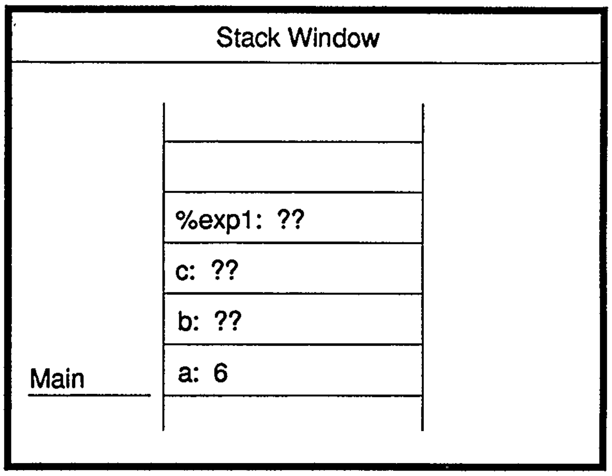
3.3 The Tutorial Module

In keeping with the instructional design model, the tutorial module begins by introducing the simulation system. This is done by presenting two sample program execution simulations. By participating in them, learners become familiar with the layout of the screen, how they should use the mouse to enter data and how the simulation will proceed.

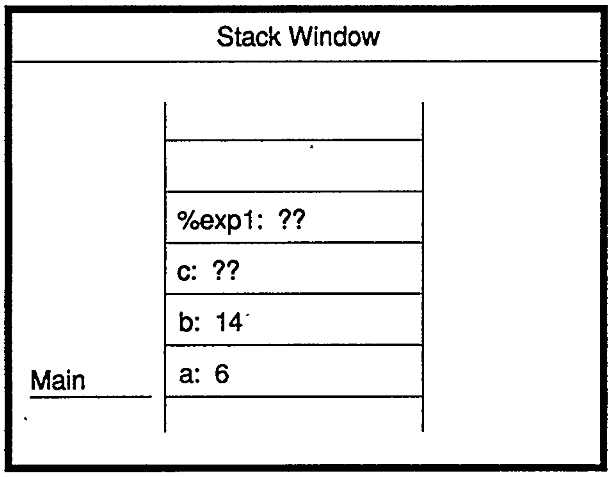
In EPAS, learners advance through three sections. The first section pertains to one-dimensional arrays, the second to value parameters and the third to variable parameters. Each section presents a brief introduction and three program execution simulations. The introduction to the section on value parameters appears in Figure 3.13. The three simulations fulfill the acquisition, application and assessment phase requirements of the instructional design model. (The instructional design model was presented in Section 2.3.)



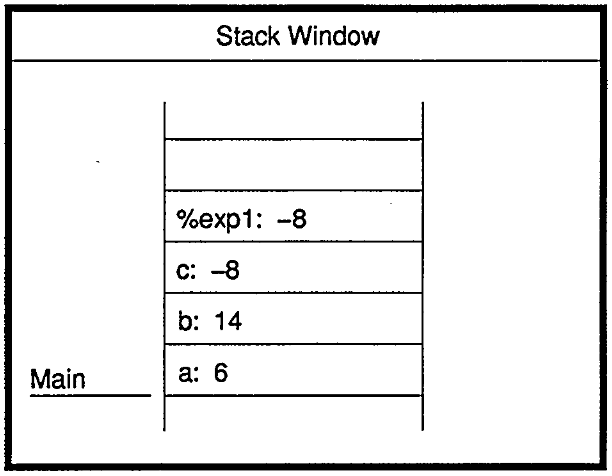
(a) After Stack Event #4



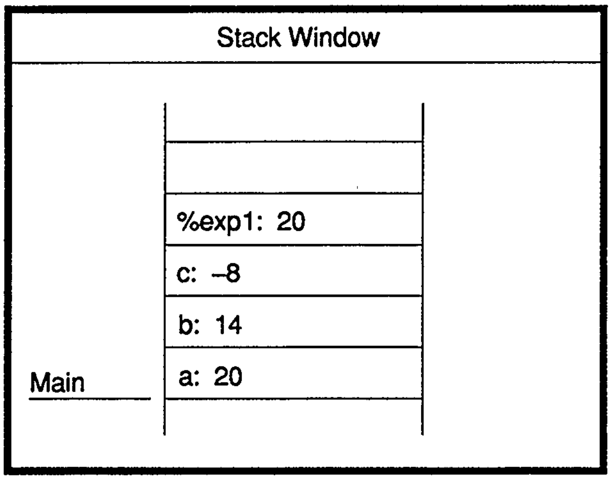
(b) After Stack Event #5



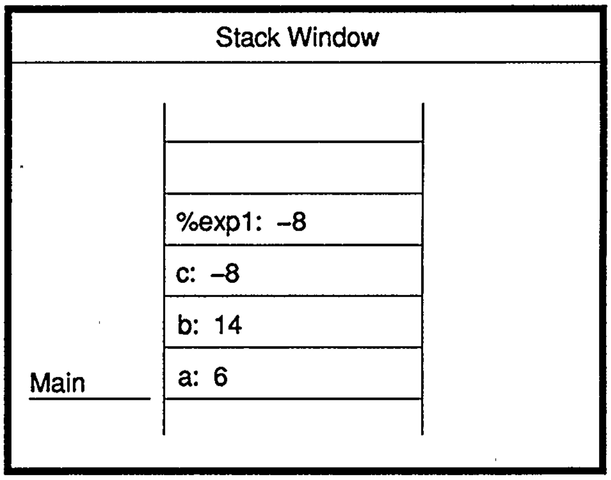
(c) After Stack Event #6



(d) After Stack Event #7

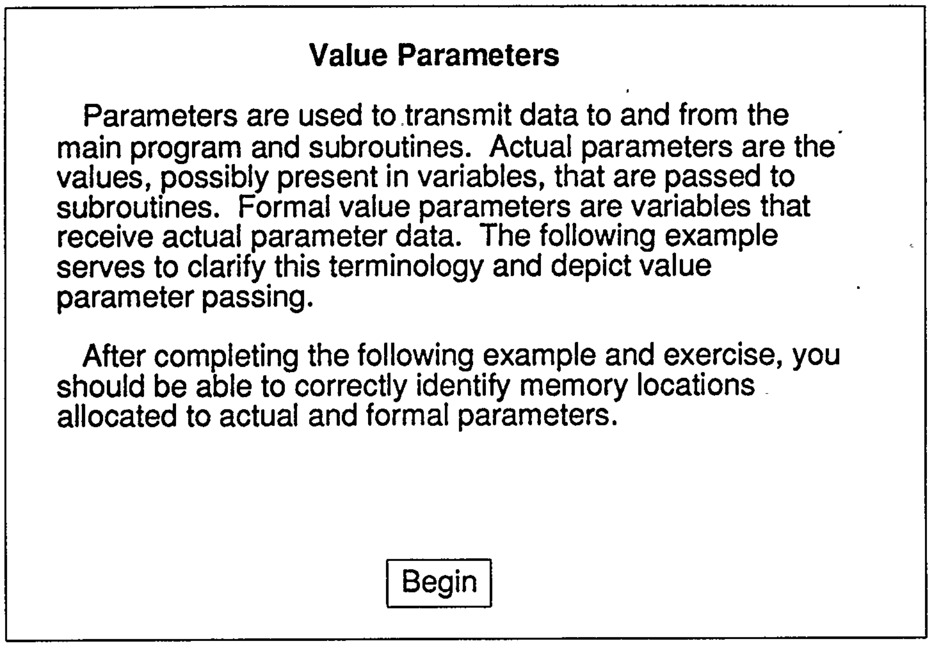


(e) After Stack Event #8



(f) After Reversing Stack Event #8

Figure 3.12: Forward and Reverse Execution



Value Parameters

Parameters are used to transmit data to and from the main program and subroutines. Actual parameters are the values, possibly present in variables, that are passed to subroutines. Formal value parameters are variables that receive actual parameter data. The following example serves to clarify this terminology and depict value parameter passing.

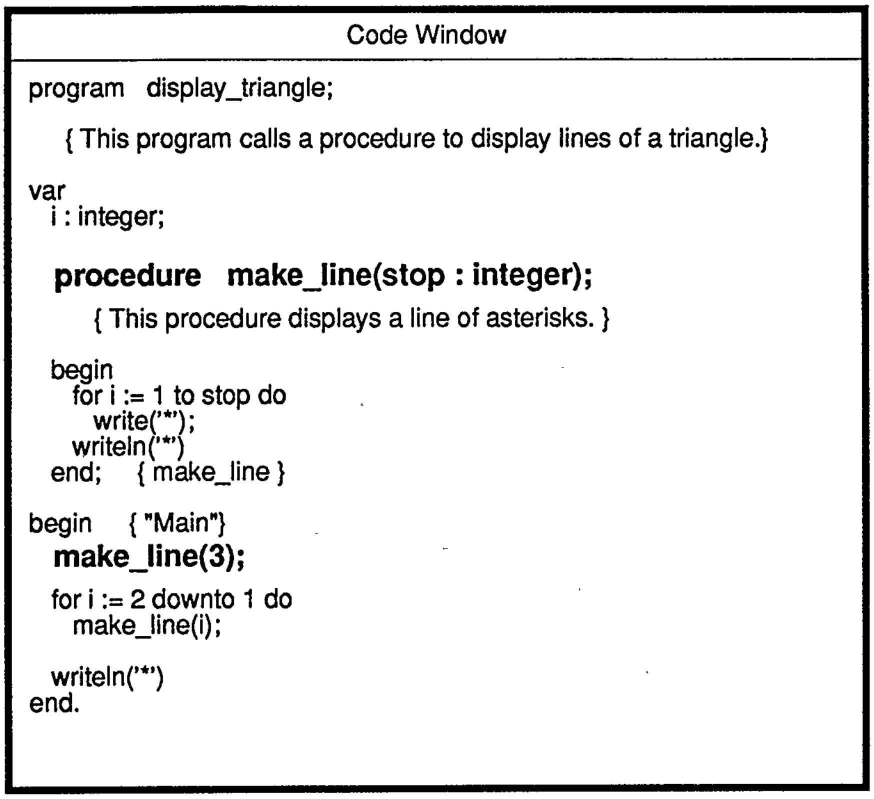
After completing the following example and exercise, you should be able to correctly identify memory locations allocated to actual and formal parameters.

Figure 3.13: The Introduction to the Value Parameter Section

In the first simulation of each section, when stack activity not previously encountered by the learner is completed, an instructional notation appears. An instructional notation explains why the most recently executed statement affected the stack in the way it did. For example, after executing the Procedure Call highlighted in Figure 3.14(a), an instructional notation pertaining to parameter passing is displayed. The actual notation appears beside the run-time stack in Figure 3.14(b).

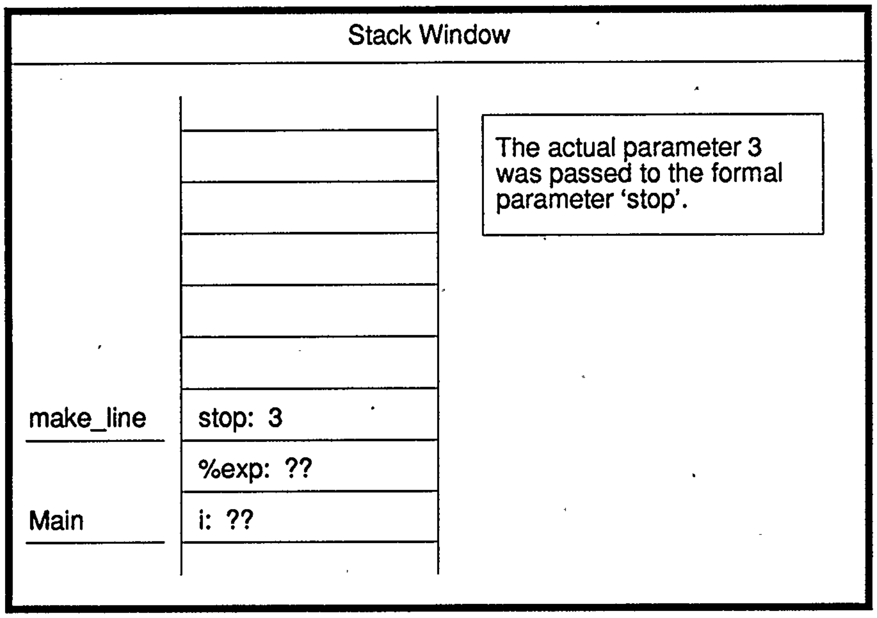
The second simulation in each section generates opportunities for practice and provides feedback based on the learner's responses. When practicing, the learner is directed to identify the source and/or target cells that will be accessed when the next statement is executed. For example, the highlighted statement in Figure 3.15(a) generates the directive in Figure 3.15(b).

The learner has the option to change stack cell selections. Eventually though, the



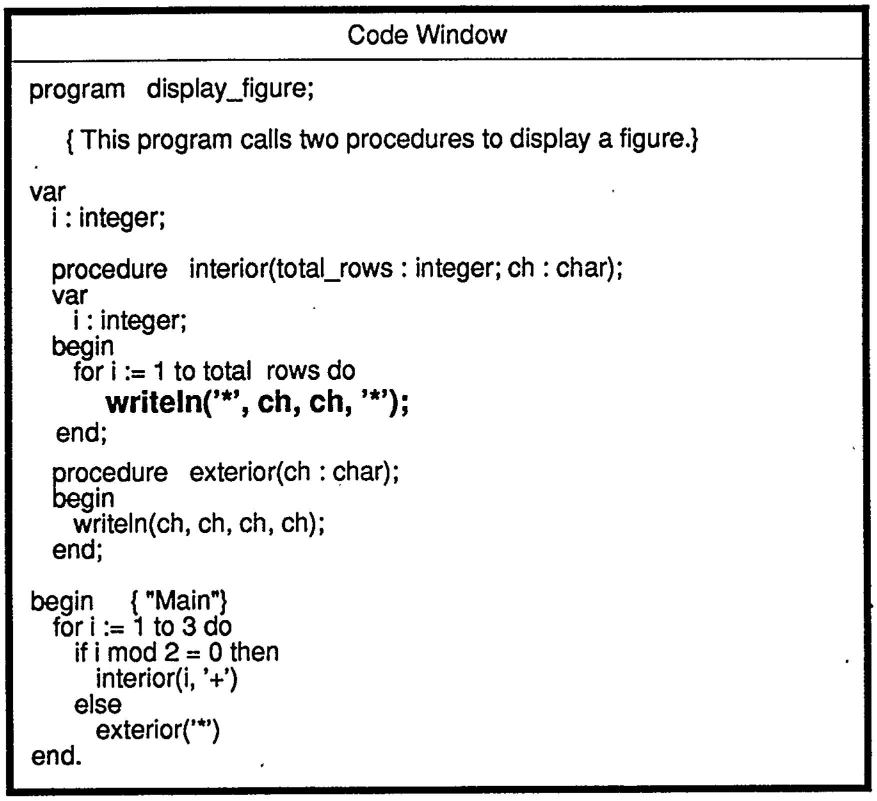
```
program   display_triangle;

   { This program calls a procedure to display lines of a triangle.}

var
  i : integer;

  procedure   make_line(stop : integer);
     { This procedure displays a line of asterisks. }

  begin
    for i := 1 to stop do
      write('*');
    writeln('*')
  end;    { make_line }

begin    { "Main"}
  make_line(3);
  for i := 2 downto 1 do
    make_line(i);

  writeln('*')
end.
```

(a)


Stack Window
The actual parameter 3 was passed to the formal parameter 'stop'.
make_line
stop: 3
%exp: ??
Main
i: ??

(b)

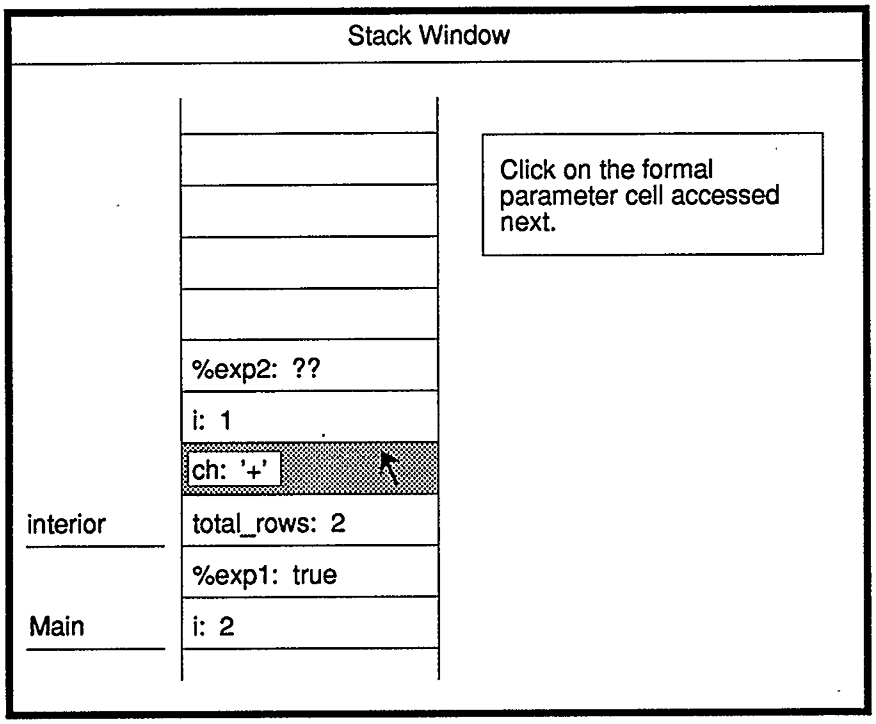
Figure 3.14: An Instructional Notation



Code Window

```
program   display_figure;

   { This program calls two procedures to display a figure.}

var
  i : integer;

  procedure   interior(total_rows : integer; ch : char);
  var
    i : integer;
  begin
    for i := 1 to total  rows do
       writeln('*', ch, ch, '*');
  end;

  procedure   exterior(ch : char);
  begin
    writeln(ch, ch, ch, ch);
  end;

begin    { "Main"}
  for i := 1 to 3 do
    if i mod 2 = 0 then
      interior(i, '+')
    else
      exterior('*')
end.
```

(a)



Stack Window

Click on the formal parameter cell accessed next.

```
              | %exp2: ??       |
              | i: 1            |
              | ch: '+'         |
interior      | total_rows: 2   |
              | %exp1: true     |
Main          | i: 2            |
```

(b)

Figure 3.15: A Directive to Identify a Stack Cell

learner must select *Next* to proceed. When *Next* is selected, source and target cells are highlighted as described in the previous section. Feedback is provided in this manner. In addition, when the learner makes an incorrect selection, a tone sounds and the word, *Incorrect* appears in the Stack Window.

In each section, the third simulation tests the learner on the material covered most recently. Again during this program execution simulation, the learner is required to identify source and target cells. The number of correct responses to the directives is calculated. This is done on a per program basis. A per program tally is used to monitor use of the *Previous* option. All data are written to a file when the last simulation has been completed.

All eleven programs incorporated into EPAS are shown in Appendix B. They are listed in order of presentation to the learner. Recall that the first two programs are presented in EPAS for introductory purposes.

Throughout the development of the animation and tutorial modules, changes were made on the advice of faculty members, graduate students and learners representative of the target group. For example, the approach to highlighting source and target cells was modified by changing the background cell color in addition to the foreground color. Also, the pace adjustment feature was enhanced to increase the range of timing delays.

3.4 The Study

Two aspects of the study conducted to test EPAS are described in this section. First, characteristics of the learners who participated in the study are noted. Second, the procedures followed during the study are discussed.

All of the participants were undergraduate students who had completed one computer literacy course prior to running EPAS. In fact, all of the students took the same course in which they completed one word processing, one spreadsheet and one database assignment. They also completed two rudimentary Pascal programming assignments.

Each student wrote a four-line program containing a single output statement to complete the first programming task. However, in writing programs to meet all of the requirements of both assignments, the students had to use input/output, assignment, looping and conditional statements. Also they had to include procedures without parameters. The student participants were never assigned tasks that involved arrays or parameter passing.

It should also be noted that three of the eight participants had no experience with a mouse input device. Before these students started working on EPAS, the researcher instructed them on how to use a mouse. They used the mouse to create, select, scale and drag objects in a drawing program to acquire mouse manipulation skills. (In EPAS, only mouse movement and single button clicking are necessary.) Lastly, with respect to the learners, all of them participated voluntarily.

Each student was scheduled to participate at some time during a three-day period. The availability of only one computer ensured that students would work individually. Upon arrival, each participant worked through the two sample program execution simulations with the researcher. At that time, the researcher advanced the following suggestions:

1. look at the statement highlighted in the code window

2. try to determine which run-time stack cells will be accessed when the highlighted statement is executed
3. select *Next* to see if you were correct

After running EPAS, the participants completed a short questionnaire and returned it to the researcher.

3.5 Summary

All of the modules in EPAS were described in this chapter. The compilation modules were documented briefly. In contrast, close attention was paid to the program visualization features implemented in the animation module. In addition, all components of the tutorial module were discussed.

The small study conducted to test EPAS was also described in this chapter. Briefly, eight undergraduate students viewed two program execution simulations to learn about EPAS. Then they worked through nine more programs to learn about arrays, value parameters and variable parameters. Collectively the nine programs presented instruction, provided learners with an opportunity to practice and assessed performance. For subsequent evaluation, EPAS stored the results for each student in a file.

Chapter 4

Results

Four types of results are discussed in this chapter. First, the extent to which the learners correctly identified stack cells is examined. Second, the tallies pertaining to the use of the *Previous* feature are noted. Third, the questionnaire data are summarized and brief statements about each item are provided. Fourth, comments written by the students who participated in the study are included.

4.1 Stack Cell Identification

The learners were required to identify stack cells during practice and quiz programs. Table 4.1 contains the percentages of correctly identified stack cells for each learner, as obtained on the three practice programs.

Table 4.1: Percentages of Correctly Identified Stack Cells (Practice Programs)

	Learner Number								
Practice Program	1	2	3	4	5	6	7	8	Mean
Arrays	75	63	75	75	88	75	25	100	72
Value Parameters	100	75	100	75	100	100	75	100	91
Variable Parameters	100	69	63	63	88	100	50	88	78
Mean	92	69	79	71	92	92	50	96	80

For practice, the learners had to identify eight stack cells on the array program. Also, they had to identify four cells on the value parameter program. The variable parameter program directed the learners to identify sixteen cells.

Considering the practice programs collectively, the students correctly identified stack cells eighty percent of the time. The students had the most difficulty identifying stack cells on the array program. On the other hand, stack cells on the value parameter program proved easiest to identify. However, recall that the learners were required to identify only four stack cells in the value parameter program.

Stack cell identification results for the three quiz programs are cited in Table 4.2. On the array, value parameter and variable parameter quiz programs, the learners were required to identify nine, ten and eighteen stack cells respectively. On these programs, the learners correctly identified stack cells eighty-three percent of the time. This represents a modest three percent improvement over the practice programs. Also with respect to the quiz programs, note that the learners performed best on the array program and worst on the variable parameter program.

Table 4.2: Percentages of Correctly Identified Stack Cells (Quiz Programs)

	Learner Number								
Quiz Program	1	2	3	4	5	6	7	8	Mean
Arrays	100	67	78	89	89	100	89	100	89
Value Parameters	100	80	80	100	60	60	80	80	80
Variable Parameters	94	78	56	78	89	96	44	100	79
Mean	98	75	71	89	79	85	71	93	83

The decrease in performance in the value parameter section is notable. However, the validity of the results obtained on the value parameter practice programs is questionable because only four stack cells had to be identified by each learner. Thus it is suspected that those results overestimate the actual abilities of the learners to identify stack cells involved in the passing of value parameters.

A reasonable procedure for analyzing the results in the two tables, given the small number of subjects, is to compare the practice and quiz program scores to determine if the students improved. Table 4.3 shows the differences in quiz and practice program scores, without regard for magnitude. The strictly positive differences in the array section indicate that all learners improved. This Sign Test result is statistically significant since the probability of eight successes in eight trials by chance is approximately 0.004. With respect to the value and variable parameter sections, no statistically significant trends are evident. Also, due to the equal number of positive and negative differences among the mean scores, no trend for overall performance is discernible.

Table 4.3: Comparing Performance on the Quiz and Practice Programs

	Learner Number							
Programming Topic	1	2	3	4	5	6	7	8
Arrays	+	+	+	+	+	+	+	+
Value Parameters	=	+	-	+	-	-	+	-
Variable Parameters	-	+	-	+	+	-	-	+
Means	+	+	-	+	-	-	+	-

However, if the questionable value parameter data are excluded, the mean scores in Tables 4.1 and 4.2 would change as shown in Tables 4.4 and 4.5. Table 4.6 shows the signs obtained when the adjusted means are compared. By chance, the probability of attaining seven or eight successes in eight trials is 0.035 (0.031 + 0.004). Thus this Sign Test result indicates that the practice exercises were probably beneficial.

Lastly in this section, the average results attained on the quiz programs, as listed

Table 4.4: Practice Program Scores (Excluding Value Parameter Data)

	Learner Number							
Practice Program	1	2	3	4	5	6	7	8
Arrays	75	63	75	75	88	75	25	100
Variable Parameters	100	69	63	63	88	100	50	88
Mean	88	66	69	69	88	88	38	94

Table 4.5: Quiz Program Scores (Excluding Value Parameter Data)

	Learner Number							
Quiz Program	1	2	3	4	5	6	7	8
Arrays	100	67	78	89	89	100	89	100
Variable Parameters	94	78	56	78	89	96	44	100
Mean	97	73	67	84	89	98	67	100

Table 4.6: Comparing Overall Performance

	Learner Number							
Tutorial Section	1	2	3	4	5	6	7	8
Practice Programs	88	66	69	69	88	88	38	94
Quiz Programs	97	73	67	84	89	98	67	100
Difference	+	+	-	+	+	+	+	+

in the final column of Table 4.2, are graphed in Figure 4.1. The graph shows that the learners, on average, achieved better than seventy-five percent accuracy (i.e., the mastery level) on all stack cell identification quiz programs. Given this result it is evident that after using EPAS, the learners were able to master stack cell identification tasks pertaining to one-dimensional arrays, value parameters and variable parameters.

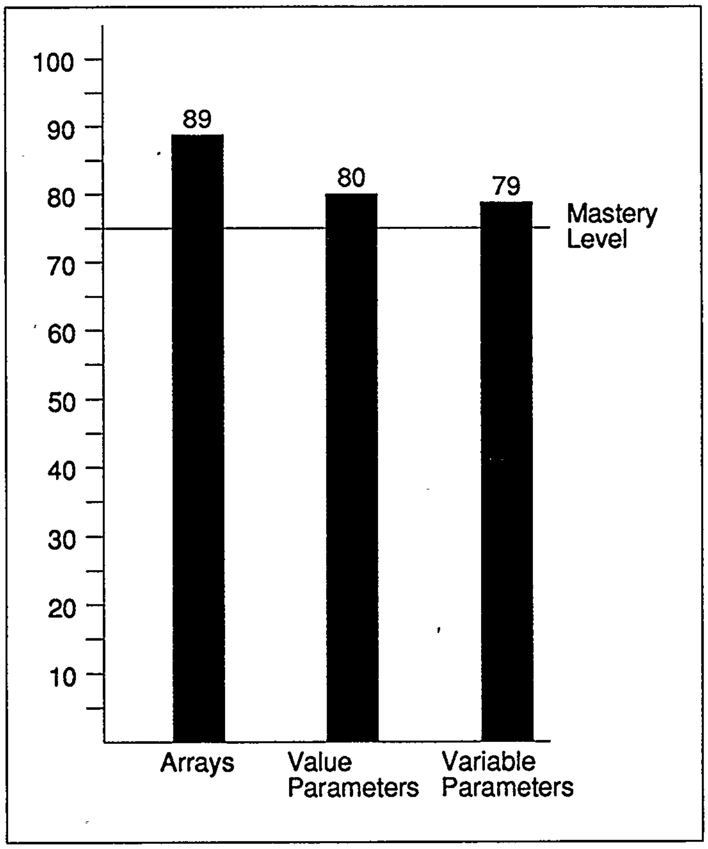


Figure 4.1: Stack Cell Identification Results for the Quizzes

4.2 Use of the *Previous* Feature

The number of times that the learners selected *Previous* is a precise indication of how often they reversed execution. Tallies accumulated while monitoring the use of the *Previous* option are given in Table 4.7. Each entry in the program identification column consists of a programming topic followed by a letter. The example, practice and quiz tutorial sections are distinguished by the letters E, P and Q respectively.

Table 4.7: *Previous* Use Tallies

Program	**Learner Number**								Program
Identification	1	2	3	4	5	6	7	8	Totals
Arrays (E)	5	3	3	10	0	0	23	4	48
Arrays (P)	1	9	3	54	13	4	19	1	104
Arrays (Q)	9	0	1	5	11	0	6	3	35
Value Parameters (E)	0	0	0	14	3	0	0	0	17
Value Parameters (P)	0	0	1	47	6	0	0	0	54
Value Parameters (Q)	0	0	0	2	4	0	0	5	11
Variable Parameters (E)	0	3	0	2	4	3	1	1	14
Variable Parameters (P)	0	5	7	49	6	0	4	0	71
Variable Parameters (Q)	0	1	1	33	1	1	0	0	37
Learner Totals	15	21	16	216	48	8	53	14	391

According to Table 4.7, *Previous* was selected 391 times. Exactly 3000 unique stack events, 375 for each of the eight learners, were executed. Thus on average, execution was reversed once for every 7.7 (3000 ÷ 391) stack events.

The totals for the learners indicate that all of them reversed execution. However, the variance of use is remarkable. One learner accounts for more use than all of the others combined. Perhaps this one learner perceived the feature as rather novel. In any case, this learner's use of *Previous* skews the results. For example, the *per learner* *use rate* drops from 48.9 ($391 \div 8$) to 25.0 ($175 \div 7$) when the tallies accumulated by Learner Number 4 are not considered.

Reverse execution data pertaining to the three tutorial sections are presented in Table 4.8. The reversal frequencies are *stack event* to *previous tally* ratios. Thus, a reversal frequency of 7.8, for example, would mean that execution was reversed once for every 7.8 stack events.

Table 4.8: Reverse Execution Data by Tutorial Section

Tutorial Section	**Number of Stack Events**	*Previous* **Tallies**	**Reversal Frequency**
Example	792	79	10.0
Practice	1024	229	4.5
Quiz	1184	83	14.3

The variability in the reversal frequencies is perplexing. In contrast to the example programs, the learners were directed to select particular cells in the practice programs. Perhaps this encouraged them to attend more closely to stack activities. This greater attention could have led them to recognize more of their misconceptions and thus increased their use of *Previous*.

However, the same contention is not plausible in accounting for the usage difference on the practice and quiz programs. In fact, the learners were directed to select stack cells in both of those settings. Motivation may account for the difference here. Perhaps the learners simply regarded the practice programs as more suitable for review than the quiz programs.

The reverse execution data are grouped by Programming Topic in Table 4.9. In the array section, the learners selected the reverse execution feature once for every five stack events, approximately. Execution was reversed about half as often in each of the parameter sections. It is believed that complexity of task and fatigue account for much of the variance in those reversal frequencies.

Table 4.9: Reverse Execution Data by Programming Topic (All Eight Learners)

Programming Topic	**Number of Stack Events**	*Previous* **Tallies**	**Reversal Frequency**
Arrays	856	187	4.6
Value Parameters	872	82	10.6
Variable Parameters	1272	122	10.4

Lastly, recall that the average scores in the value parameter section were lower on the quiz program than on the practice program. One possible explanation for that was discussed earlier. As an alternative, perhaps the students underestimated the complexity of value parameters and consequently attended to the value parameter programs less diligently than the others. Indeed, Table 4.7 reveals that, excepting Learner Number 4, the students used the reverse execution feature sparingly in the value parameter section. In fact, if the exceptionally high tallies for Learner Number 4 are not considered, the reverse execution results in Table 4.9 would be adjusted as shown in Table 4.10.

Table 4.10: Reverse Execution Data by Programming Topic (Seven Learners)

Programming Topic	**Number of Stack Events**	*Previous* **Tallies**	**Reversal Frequency**
Arrays	749	118	6.3
Value Parameters	763	19	40.2
Variable Parameters	1113	38	29.3

4.3 The Questionnaire Data

On the questionnaire, the first five items refer to specific features incorporated into EPAS. The next three items pertain to the acquisition of programming knowledge. The overall utility of the tutorial component in EPAS is the topic of the last rated item.

All items were rated on Likert scales that ranged from zero to four. Responses at the low end of the scale indicated negative feelings. In contrast, high ratings denoted positive feelings. All learner responses to the questionnaire are listed in Table 4.11. In addition, the means for each learner and item are included.

Table 4.11: Learner Responses to the Questionnaire

Item	**Learner Number**								
Number	1	2	3	4	5	6	7	8	Mean
1	4	3	4	4	4	3	3	3	3.5
2	4	4	4	4	4	4	4	3	3.9
3	4	4	4	4	4	4	2	4	3.8
4	3	4	4	4	4	3	3	2	3.4
5	4	4	4	3	3	4	4	4	3.8
6a	3	2	3	4	3	2	2	4	2.9
6b	4	3	4	4	3	3	3	4	3.5
6c	4	3	4	3	3	3	2	4	3.3
7	3	3	4	4	4	4	4	4	3.8
Mean	3.7	3.3	3.9	3.8	3.6	3.3	3.0	3.6	3.5

The responses are strikingly uniform. None of the seventy-two responses are below two. On the scale, the values 2, 3 and 4 were selected 6, 23 and 43 times, respectively. Following are brief comments about each item.

Item 1. How useful was the *Pace Adjustor* feature?

The mean response for this item is 3.5. Thus, it seems that the *Pace Adjustor* feature is worthwhile and should be retained if EPAS is developed further.

Item 2. How appropriate was the use of color for highlighting cells?

Seven of the eight learners selected the most favourable response in assessing this item. Indeed, the color highlighting technique for cells should remain the same in a complete system.

Item 3. Overall, how useful was the mouse/window interface?

Learner responses to this item were nearly identical to the previous one. Apparently, the user-interface was at least functional. Nevertheless it would need to be more flexible in a complete system. This issue is discussed further in Chapter 5.

Item 4. How useful was the *Previous* feature?

Considerable consistency in the learner responses to this item and in their use of the *Previous* feature is evident. Based on the learner responses and frequency of use, this feature is important. Techniques for incorporating it into a full system are examined in Chapter 5.

Item 5. How fair were the quizzes?

Six of the eight learners selected the maximum score for fairness. Similarly, the other two learners judged the quizzes as fair by rating them at the maximum value minus one.

Item 6a. How confident are you that you could incorporate arrays into programs?

The mean score for this item is 2.9. Notice also that there is more variability in the responses to this item than to any other. Two of the learners selected the maximum value for confidence. The others were divided equally in selecting either two or three. Considering the three programming topics, arrays, value parameters and variable parameters, the learners are least confident that they could incorporate arrays into programs. Curiously though, the average quiz score in the array section exceeds the average quiz scores in the parameter sections.

Item 6b. How confident are you that you could incorporate value parameters into programs?

Of the three programming topics covered in the EPAS tutorial, the learners are most confident that they could write code using value parameters. Again the learners were divided equally in their responses. Although this time they selected either three or four. Given that performance in the value parameter section decreased, this level of confidence may indicate a false sense of understanding.

Item 6c. How confident are you that you could incorporate variable parameters into programs?

For this item, six of the eight learners selected the same confidence rating as in the previous one. One scale point reductions are evident in the responses of the other two learners. Generally, the learners believe they could incorporate variable parameters into programs.

Item 7. Overall, how useful was the tutorial?

For this item, the mean response is 3.8. As such the learners seem to regard the tutorial as valuable.

4.4 Learners' Comments

The last item on the questionnaire called for written comments on any aspect of EPAS. All of the learners took advantage of this opportunity to express their opinions. Occasionally the learners expressed constructive criticisms. For the most part, though, they wrote complimentary remarks. A sampling of the responses appears below.

- When you get an incorrect answer, it would be nice to see the question again – also to see where you messed up.
- If you choose incorrectly on the quiz, a bit of an explanation might help when indicating what the correct choice should have been.
- The use of blue-red highlights is great.
- I think this is an interesting and useful way of learning. It is more "hands-on" than watching a black-board.
- The variable parameter section was well explained – although I tended to forget instructions given earlier – the *Previous* function was great on this aspect.
- Overall, I found it very instructive and helpful.

4.5 Summary

First in this chapter, stack cell identification results from the practice and quiz programs were discussed. Those results indicate that the learners were able to identify, to the point of mastery, target and source stack cells referenced in statements that manipulate one-dimensional arrays, value parameters and variable parameters.

Also in this chapter, data accumulated to monitor use of the *Previous* feature were cited. Those data indicate that the *Previous* feature is important. This claim is supported by responses to one of the items on the questionnaire. Other results based on the questionnaire were also discussed. Lastly, remarks written by the learners were quoted.

Chapter 5

Enhancing EPAS

Given the encouraging results presented in the previous chapter and the fact that EPAS is a prototype, one suggestion for further research seems obvious. That is, a complete compilation/animation system should be developed by enhancing EPAS. Techniques for creating such a system are developed in this chapter. Additional suggestions for further research are advanced in Chapter 6.

Before attempting to enhance the existing system, one ought to recognize how a complete program animation system based on EPAS would be limited. First note that all complete program animation systems must, at times, display only a subset of the data in a program. This realization is based on the fact that a program can generate vast quantities of data. Indeed, so much data that they cannot be displayed on one screen simultaneously.

In addition, a complete program animation system based on EPAS would represent the run-time stack in a linear manner. Although a linear stack representation is consistent with actual run-time stacks, it is not suitable for depicting multi-dimensional arrays and record structures because they are conceptually different. Also dynamically allocated data structures, which are assigned space in a set of memory locations called the *heap*, would not be depicted on the stack.

5.1 A Framework for Enhancing EPAS

Two approaches to developing a complete system based on EPAS are evident to the researcher. Both of them are discussed below. Beyond this section, though, only one approach is pursued.

First, one could enhance EPAS so that it would create data files of stack events, similar to the ones used in the existing system. To accomplish this, information from the scanner could be accessed to generate the data that would be used to highlight program declarations and statements in the Code Window. Also, variable and parameter names could be written to stack events. In addition, stack cells could be assigned numbers in a manner similar to the way variables are actually allocated memory locations. That is, stack cells could be assigned offset values relative to a frame pointer.

For this approach to be successful, two outstanding issues remain to be resolved. First, the values that would be assigned to variables must be written to stack events. Second, for each stack cell, the number of the last stack event to reference it as a target must be determined. This is necessary to enable reverse execution.

To calculate the values of variables and acquire reverse execution information, the input program must be executed. During execution, the values of variables and target stack cell reference data could be written to a file. Then, this information could be combined with the data generated by the parser to define stack events.

This approach would enable most of the existing code to be retained in some form. However, there are serious disadvantages. First, an overhead penalty would be incurred by creating files and then combining the data within them. Second, this two-step (*execute, then animate*) approach could not be used to fully implement the reverse execution feature. For example, when reversing a *readln* statement, the previous values of the input variables could be restored; however, new input values could not be assigned because stack activities would be replayed as opposed to re-executed. Another consequence of this two-step approach is that the animator could not be enhanced to include debugging functions such as the interactive assignment of values to variables and subsequent expression evaluation. For these reasons, this method for improvement will not be pursued subsequently.

As an alternative, one could enhance EPAS by writing a code generation module for execution, animation and reverse execution purposes. In contrast to the previous approach, a considerable amount of new code would have to be written. However, using the code generation approach, all stack activities could be reversed and debugging features could be added. Techniques for using this approach to enhance EPAS are examined in the next section.

5.2 Compilation and Animation Issues

A pseudocode model for code generation is shown in Figure 5.1. The model depicts the translation of an arbitrary Pascal declaration or statement. The context of the translation is nonstandard. Unlike typical translations, this one incorporates the program animation features present in EPAS (i.e., reverse execution and run-time stack display). The pseudocode representation is similar to assembly language, but it is not machine specific. Each unique line of the pseudocode model is discussed next.

```
                                          o
                                          o
                                          o
  line        REVERSE_P:     { code to reverse the
 numbers                       previous statement }

    1             START:     UPDATE_CODE_WINDOW a, b, c
    2                        GET_EXECUTION_DIRECTION
    3                        IF previous, BRANCH REVERSE_P
    4                        "embellished translated code"
    5                        UPDATE_CODE_WINDOW a, b, c
    6                        BRANCH NEXT
    7           REVERSE:     UPDATE_CODE_WINDOW d, e, f
    8                        "restoration code"
    9                        BRANCH START

   10              NEXT:     UPDATE_CODE_WINDOW d, e, f
   11                        GET_EXECUTION_DIRECTION
   12                        IF previous, BRANCH REVERSE
                                          o
                                          o
                                          o
              code_refs:     row
                             column
                             length
                                          o
                                          o
                                          o
```

Figure 5.1: Executable Code Model

REVERSE_P is the label at which code for reversing the previous stack activity would appear. It was included because the code for the translated stack activity, which begins at the label START, references it. Subsequently the model is discussed in two parts. First, statements pertaining to forward execution are examined. Then, reverse execution features are detailed.

5.2.1 Forward Execution

The first and fifth lines of a translated stack activity call a system routine to update the Code Window. Three arguments would be passed to it. One argument would be a value denoting an offset from the label, *code_refs*. Code_refs marks the beginning of the data storage area for source code references. A second value passed to this routine would be the number of code segments to display. Recall that for a procedure statement, its corresponding heading is also displayed. Third, a value that specifies the display color must also be forwarded to this routine. This value would be one of two numbers since text in the Code Window is either red or black. Note that this routine should also be able to scroll text. For instance, when the line or lines to be highlighted next do not appear in the Code Window, they should be scrolled into view.

The second translated line is a call to GET_EXECUTION_DIRECTION. This routine would ultimately return *Next* or *Previous* in accordance with the the user's desire to proceed forward or backward through a program. Also, this routine would allow the user to adjust the pace of the animation. For implementation purposes, the existing code for selecting execution direction and pace adjustment could be linked to executable files.

The third translated line is the conditional branch to code that would reverse the previous stack activity. Reverse execution is discussed subsequently. Thus, this discussion proceeds to the next line of the model.

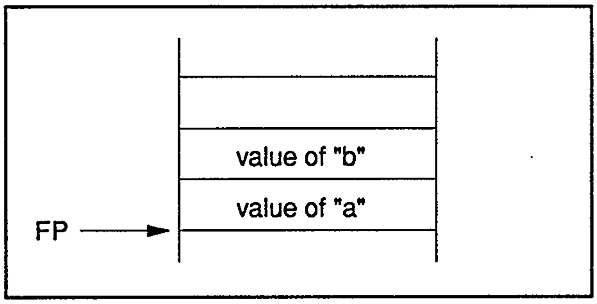
Since the model is based on an arbitrary stack activity, specific translation details are not given. However, *embellished translated code* would consist of standard code and instructions for animating stack activity and reversing execution. The following example shows how standard executable code would be embellished. The code in Figure 5.2, could be used to translate the Pascal assignment statement, $b := a + 4$. The comments in the braces serve to document the code.

The second instruction is based on the assumption that a is declared first in the current module. The fourth instruction assumes that b is declared second and that

```
LOAD    R1, 4       { Register #1 (R1) becomes 4 }
LOAD    R2, [FP]    { Register #2 (R2) becomes the value of "a" }
ADD     R1, R2      { R1 becomes R1 + R2 }
STORE   FP + 2, R1  { the value of R1 is stored in "b" }
```

Figure 5.2: Sample Translated Code

two bytes are allocated for each of the variables, a and b. Figure 5.3 depicts these run-time stack assumptions. The Frame Pointer (FP) denotes the beginning of the current module. The rather standard code in Figure 5.2 would be embellished as shown in Figure 5.4. Again, details concerning each unique line are discussed.



Note: each cell represents two bytes of memory

Figure 5.3: A Typical Run-Time Stack

Figure 5.5(a) shows how run-time stack data for one variable would be organized as a record to accommodate the embellished translated code. The second frame of Figure 5.5 shows the run-time stack after executing the statement, $b := a + 4$. The additional data in the run-time stack would be used for display purposes, as described subsequently.

The first and seventh lines of the embellished translated code are identical to the first and third lines of the standard code. Further, the second and fifteenth lines only differ from the second and fourth lines of the standard code by the offset values. For example, in the standard code, the variable a is at the frame pointer. In the

```
line
numbers

  1     LOAD      R1, 4
  2     LOAD      R2, [FP + 4]
  3     DISPLAY   FP, 1                           { display "a" in source mode }
  4     DISPLAY   FP + 12, 2                      { display "%exp1" in target mode }
  5     DELAY
  6     DISPLAY   FP, 0                           { display "a" in normal mode }
  7     ADD       R1, R2
  8     STORE     p_values_ptr += 2, [FP + 16]    { save "%exp1" }
  9     STORE     FP + 16, R1                     { "%exp1" becomes R1 (i.e. "a" + 4) }
 10     DISPLAY   FP + 12, 1                      { display "%exp1" in source mode }
 11     DISPLAY   FP + 6, 2                       { display "b" in target mode }
 12     DELAY
 13     DISPLAY   FP + 12, 0                      { display "%exp1" in normal mode }
 14     STORE     p_values_ptr += 2, [FP + 10]    { save "b" }
 15     STORE     FP + 10, R1                     { "b" becomes R1 }
 16     DISPLAY   FP + 6, 0                       { display "b" in normal mode }
```

Note: "%exp1" is a variable created by the system

Figure 5.4: Embellished Translated Code

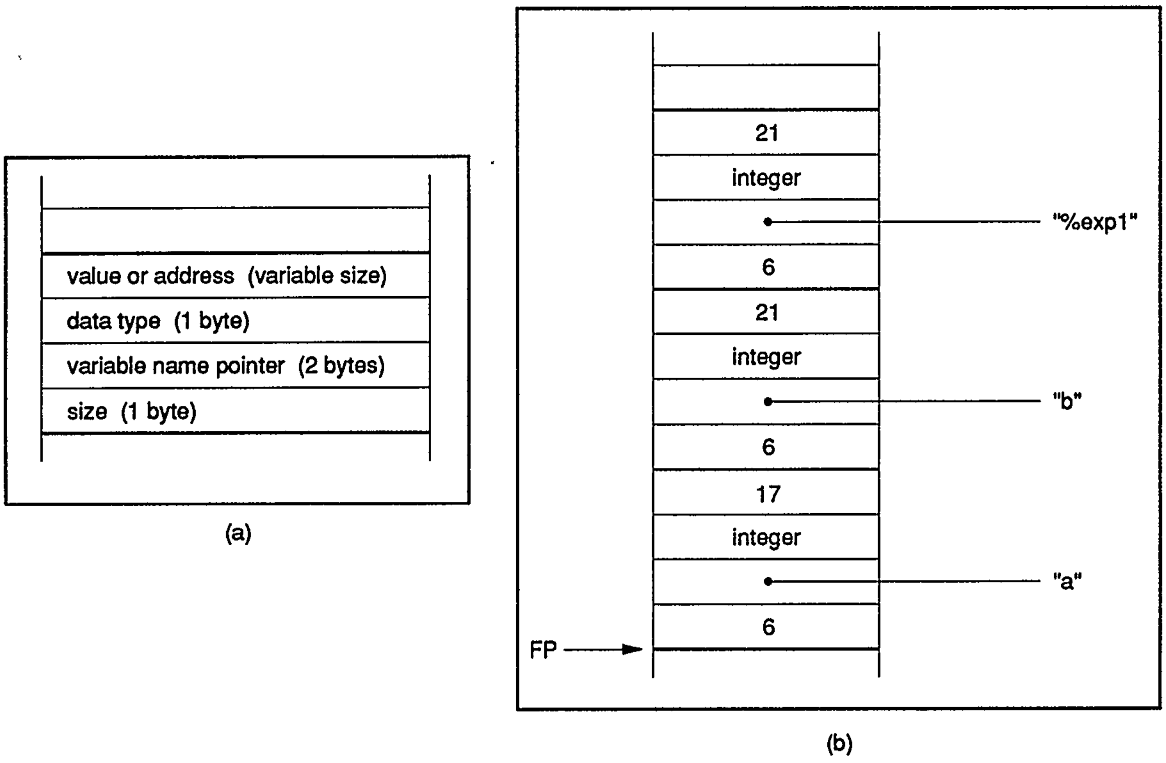


Figure 5.5: The Enhanced Run-Time Stack

embellished version, it is four bytes from the frame pointer. The twelve additional lines in the embellished code are DELAY, STORE and DISPLAY instructions.

The DELAY instruction would suspend execution for a time period set by the user via the *Adjust Pace* feature or for the default duration. STORE instructions write values to memory. In the STORE instructions, the first argument is the address at which a value is written. Optionally, a pointer to an address can be incremented after the STORE operation is completed or decremented before the operation is performed. For example, at Line 8 and Line 14, *p_values_ptr* (i.e., *previous values pointer*) is incremented by two. Actually, the STORE instructions at these lines write values to memory in case the user reverses execution. The second argument is a reference, either a register or address, to the value that is written to memory.

The calls to the DISPLAY routine are used to display data in Stack Window cells. They are also used to highlight cells. Two arguments are passed to the DISPLAY routine. The first argument is the run-time stack address of the record containing data to be displayed. The second argument defines how the data will be displayed in a cell. They could be displayed in *normal mode* (black text against a white background), *source mode* (red text against a light gray background) or *target mode* (blue text against a light gray background).

A mapping between run-time stack addresses and absolute cell locations in the Stack Window would be maintained during program execution. The mapping for the sample statement is shown in Figure 5.6. When *a*, *b* and *%exp1* were allocated space in the run-time stack, the addresses of those variables would have been written to memory. In particular, to the area in memory beginning at the pointer, *displayed_data*. Given the maximum height of a window on a SPARC station and a

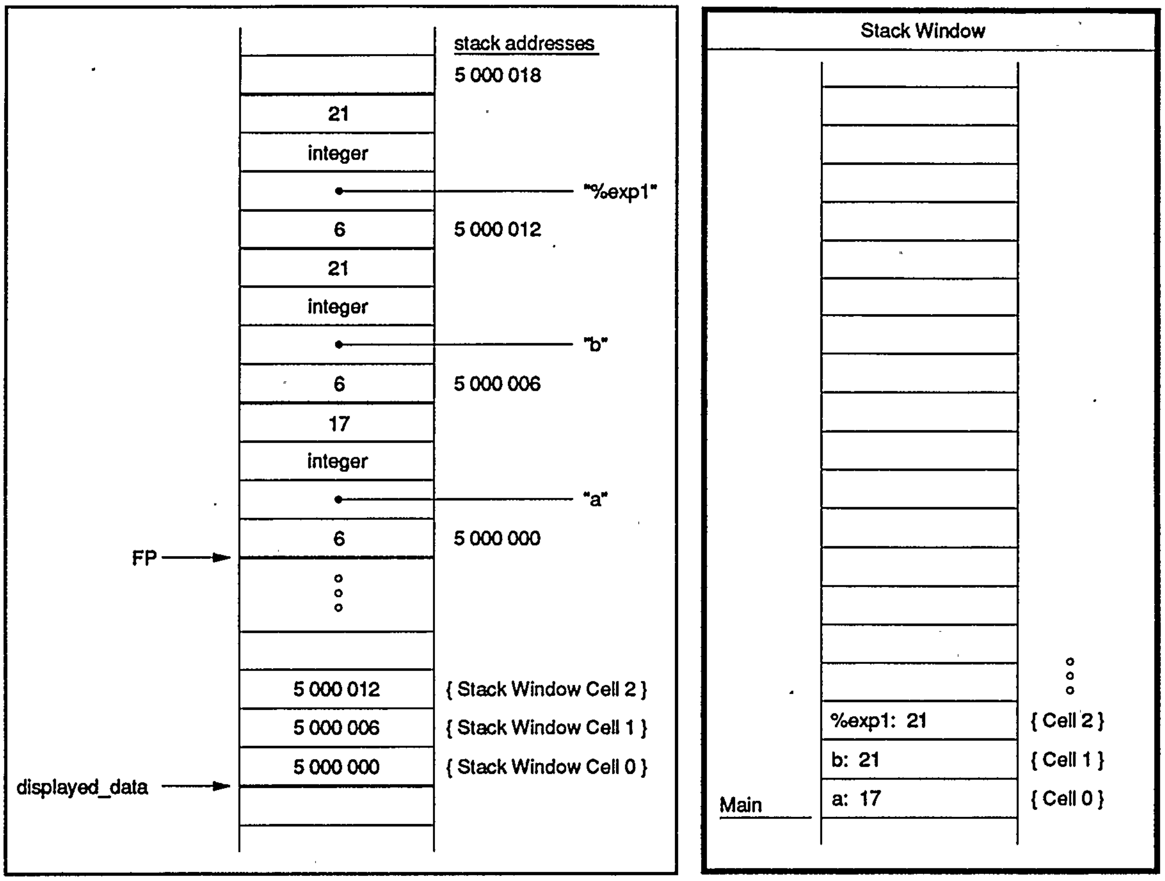


Figure 5.6: Mapping Addresses To Stack Window Cells

suitable cell height, approximately forty cells could be displayed. One address must be maintained for each cell.

All of the changes to the Stack Window, as defined by the DISPLAY instructions in the embellished translated code, could have been made by accessing data in a standard run-time stack. However, since many programs executed in an enhanced EPAS environment would contain more data than could be shown at once, the user ought to have some control over what part of the stack is displayed. Accordingly, a window scrolling mechanism could be implemented in an enhanced run-time stack environment.

For example, assume that a currently executing module contains fifty integer variables. Also, forty of them are presently displayed in the Stack Window and the user has just clicked on a scroll bar arrow to indicate that the next variable should be displayed. By adding the address of the last displayed variable to the value in its size field, the address of the next variable could be computed. Then, each stack cell mapping address could be moved down one position and the new address written at the top. Lastly, using the other fields in the enhanced run-time stack, the Stack Window could be redrawn. Given that a standard run-time stack does not contain size information, performing even the first step of that stack scrolling procedure would not be possible.

Even though the code in Figure 5.4 translates a simple Memory Read/Write event, it serves well to demonstrate how other stack activities would be implemented. Indeed, most stack activities access source and/or target stack cells. For example, code generated for Output events would often display cells in source mode. Further, Declaration events would generate code to display cells in normal mode. Procedure Calls would generate the same type of code as Declarations. For display purposes, even Module End events could be implemented by generating DISPLAY instructions that would write data to a cell using the background window color.

With respect to Stack Window displays, some consideration must be given to the various data types. Displaying values of simple data types such as *boolean, char, programmer-defined scalar* and *integer* would be straight forward. Displaying values of type *real* would be more complex. Nevertheless, they could be represented consistently to a finite number of decimal places or in exponential notation. Lastly, depicting the pointer type would be the most complicated.

As was done in EPAS, an arrow should be drawn to represent a pointer. Searching for the value of a pointer variable (i.e., an address) in the *displayed_data* area in memory would reveal if an arrow could be drawn precisely. If the address were found, an arrow would be displayed. Note that a tally for the number of currently displayed pointers must be kept to ensure that arrows would be drawn in unique vertical positions. If the address were not among the displayed ones, a line extending to the bottom of the Stack Window could be drawn to show that the appropriate cell is not presently displayed. Discussion of the issues pertaining to the DISPLAY routine and the notion of *embellished translated code* is now complete. Consequently, the discussion returns to the code generation model in Figure 5.1.

The fifth line of the model calls UPDATE_CODE_WINDOW. Highlighted text in the Code Window would be rewritten in black to indicate that execution of a stack activity had been completed. The sixth translated line, an unconditional branch to the label NEXT, completes the execution of a stack activity.

After the system highlighted the next code segment, the user would select the execution direction again. Assuming the user selects *previous*, the conditional branch after the call to GET_EXECUTION_DIRECTION would pass control to the code at the label REVERSE.

5.2.2 Reverse Execution

Removing the highlighted segment in the Code Window would be the first action taken at REVERSE. Then to rescind the previous stack activity, actions specific to its type would be taken. *Restoration code* is discussed below. The last reversal instruction in the model (Line 9), BRANCH START, would pass control to the beginning of the now current stack activity, given that restoration had been completed.

Reversing each of the five types of stack activities is discussed separately. To rescind a Memory Read/Write event, the values of cells would be restored and the Stack Window updated. Pseudocode for reversing the example statement, $b := a + 4$ is shown in Figure 5.7. The first two lines restore cells and the last two update the Stack Window. To understand stack cell restoration, visualize the state of memory after the example statement has been executed. Figure 5.6 may help in this regard, but Figure 5.8 depicts memory more fully.

```
STORE     FP + 10, [p_values_ptr -= 2]      { restore "b" }
STORE     FP + 16, [p_values_ptr -= 2]      { restore "%exp1" }
DISPLAY   FP + 6, 0                         { display "b" in normal mode }
DISPLAY   FP + 12, 0                        { display "%exp1" in normal mode }
```

Figure 5.7: Reversal Code

For this explanation, the focus will be the lowest block of memory in Figure 5.8. The block of previous values would be treated as a typical *last-in first-out* stack. Thus, since the embellished translated code targeted *%exp1* and then *b*, the STORE instructions in the reversal code restore *b* and then *%exp1*. Note that before data would be retrieved, *p_values_ptr* would be decremented.

Procedure Calls allocate space on the run-time stack for parameters. Parameters are initialized, but they do not have previous values. Thus, values would not be restored when a Procedure Call is rescinded. Rather, the cells in the Stack Window allocated for the parameters would be deleted. An exception to this technique would be made for the system input routines, *Read* and *Readln*. Calls to those procedures would be treated as Memory Read/Write events. Consequently, values would be

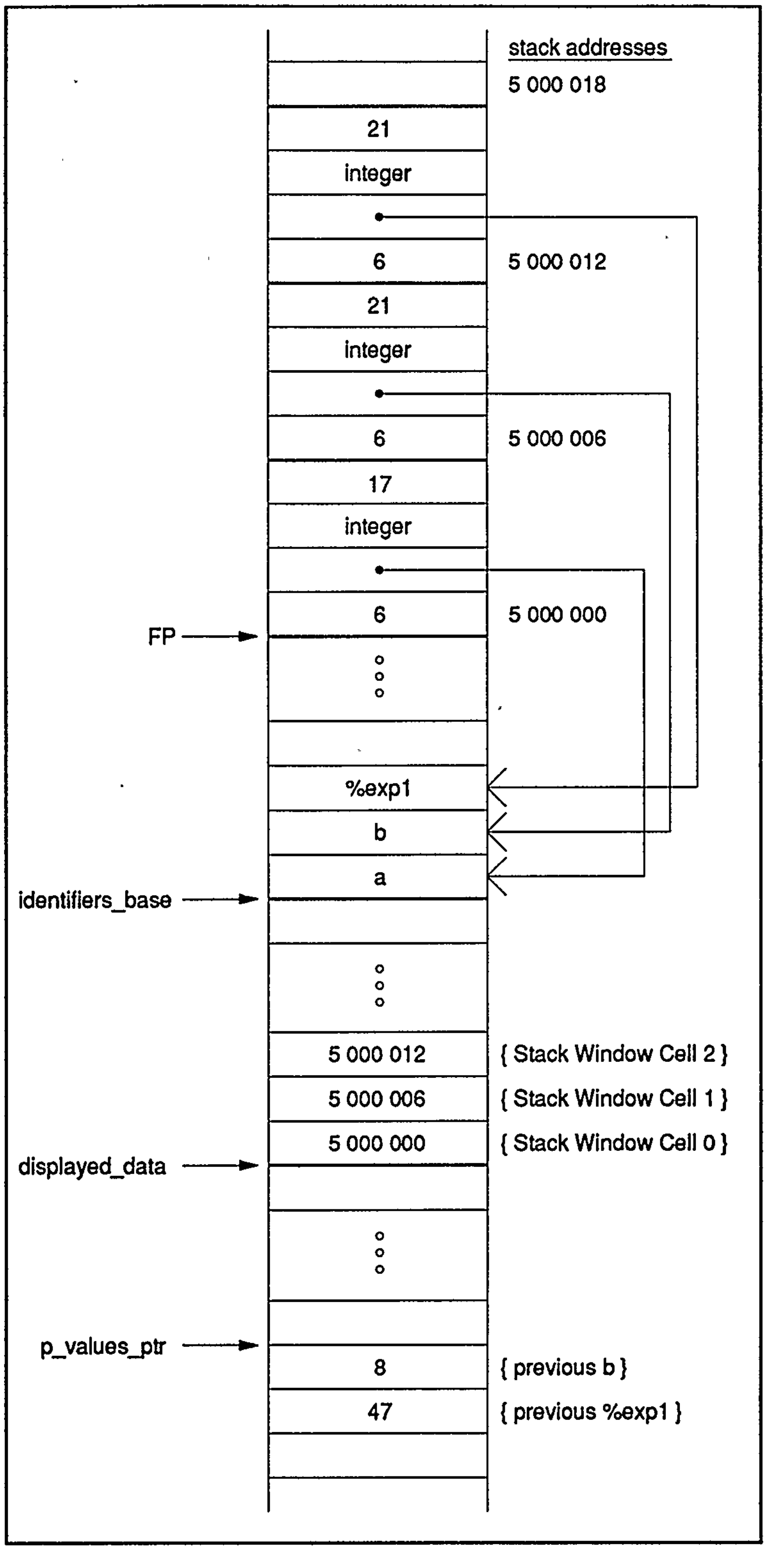


Figure 5.8: Visualizing Memory

restored when those calls were reversed. In addition, the Output Window would be updated. Updating the Output Window is described below in the context of reversing Output events.

Rescinding a Declaration event would be similar to reversing a Procedure Call that contained one parameter. Consequently, Declaration event reversal would be straight forward and is not discussed further.

To reverse a Module End event, local variables must be put back on the stack. Seemingly then, the final values of local variables should be stored in case a Module End event is reversed. However, a run-time stack does not obliterate values when a module is exited. Rather, the dynamic link of a module is followed and the frame pointer is simply changed. Thus the final values of local variables are still on the run-time stack. As such, to rescind a Module End event, a series of DISPLAY instructions would be used to update the Stack Window.

To rescind an Output event, a string from the Output Window must be deleted. Each time an Output event was executed, the string it generated would be stored in memory. Also, its row and column positions in the Code Window would be retained. Using this information, an Output event would be rescinded by writing its string in the background window color to the Output Window again. Unlike Memory Read/Write events, since Output events do not target cells, there would be no need to restore stack cells. Consequently, the Stack Window would not be changed.

Lastly, note that during programming efforts to enhance EPAS, some of the techniques discussed above would almost certainly be modified. Nevertheless, they do provide a starting point for enhancement.

5.3 Incorporating a Tutorial Module

The tutorial module in EPAS displayed instructional notations and provided practice and testing facilities. Similarly, an enhanced EPAS should provide those features. Moreover, based on student responses, an explanation feature should be added. It would serve to explain why particular stack cells were accessed as a specific statement was executed. The explanation feature could be activated when a stack cell is identified incorrectly or at the request of the user. Two methods for incorporating a tutorial module into an enhanced system are described below.

For both methods, two additional routines would be inserted into the object code file. One routine would display instructional notations, directives and explanations. The other routine would allow the user to identify stack cells in response to directives. It would also score the responses. For the first routine, an instructor would have to supply the text to be displayed. For the second routine, the number of stack cells to select and the identities of specific stack cells must be supplied. Once again an instructor would provide these data. To assist teachers, some general instructional notations could be generated whenever typical statements, such as *for, while* and *if*, were encountered. Similarly the system could supply some directives and explanations, but this could only be done for specific programs.

In the first method, an instructor would define when the tutorial routines should be called by inserting procedure statements into Pascal source programs. Data that must be passed to the tutorial routines would appear in the procedure statements as actual parameters. In particular, a text string would be passed to the first routine. The second routine would contain an integer parameter that defines an absolute stack cell location.

The second method would involve writing a *tutorial editor* in an instructor version of the enhanced EPAS. The instructor version would consist of the enhanced EPAS program and two options for inserting tutorial features. In addition to the menu selections in EPAS, *Previous, Next* and *Adjust Pace*, the instructor version would contain *Instruction* and *Directive.*

After viewing a stack event, an instructor could add an instructional notation by selecting *Instruction.* Before viewing a stack event, directives could be entered. After selecting *Directive*, the instructor would be prompted to enter text. Then the instructor would use the mouse to click on specific stack cells. Also, the instructor would be prompted to enter text for explanation purposes. Lastly, the tutorial editor would generate files that would be accessed when students used the enhanced EPAS system.

5.4 Summary

This chapter explored techniques for improving EPAS. The goal for improvement was the development of a complete program animation system consisting of compilation and animation components. Also, methods for incorporating a tutorial module in the enhanced system were discussed.

Chapter 6

Discussion

This chapter contains a summary of the thesis. In addition, suggestions for further research are advanced. Finally, conclusions are drawn.

6.1 Summary

The research conducted for this thesis pertained the acquisition of computer programming knowledge. The brief history of programming presented in Chapter 1 revealed that, in contrast to the past, programmers today use abstract processes to manipulate data. To help students acquire the skills necessary to cope with computer programming abstractions, an experimental program animation system (EPAS) was developed. Functional and technical descriptions of EPAS are summarized later.

A small study was conducted to test the effectiveness of EPAS. The study included only eight students because EPAS is a prototype. The primary purpose of the study, as defined by the hypothesis in Chapter 1, was to determine the extent to which students acquired programming knowledge pertaining to arrays, value parameters and variable parameters. The most significant results of the study are noted subsequently.

Program visualization research was reviewed in Chapter 2. Program visualizations systems can be classified as either program animators or algorithm animators. Both types of systems are similar in that they depict run-time activity.

However, the manner in which program and algorithm animators depict execution differs substantially. Program animators show how values in memory are altered by each statement. Alternatively, algorithm animators operate on a more abstract level to depict the operation of a specific routine. For example, an algorithm animator might depict sorting actions by comparing and swapping bars of varying heights (representing random values) until the bars are in ascending order. In contrast, a program animator depicting the same routine would highlight lines of code and show how each statement compares or swaps values in memory.

Since EPAS functions as a program animator, the algorithm animators were reviewed briefly. The review revealed the scope of specific algorithm animators and some of their advantages and disadvantages. The discussion of program animators proceeded along two lines. Some systems use graphics for depicting execution while others are strictly text-based. The text-based systems were referred to as program monitors to distinguish them from the graphics oriented program animators.

The discussion of specific program monitors and animators revealed that the systems vary considerably in scope and complexity. Again, advantages and disadvantages of the systems were noted. In addition, implementation details were discussed. Four specialized program monitors, called debuggers, were also reviewed to provide insight into how programmers can monitor executing programs when seeking to identify and correct errors in them.

In the final section of Chapter 2, the theoretical basis for the tutorial module in EPAS was discussed. At the foundation of the tutorial module is a relatively new instructional theory (Reigeluth and Schwartz, 1989) that defines how computer-based simulations should be designed. Only prescriptive aspects of the theory were reviewed.

In brief, Reigeluth and Schwartz maintain that computer-based simulations ought to contain the four phases, introduction, acquisition, application and assessment. Further, specific features should be included in each of the phases. For instance, a sample simulation with simultaneous explanations should be included as part of the introduction. In the acquisition phase, new material is presented. Opportunities for practice are provided in the application phase and test items are presented in the assessment phase.

Functional and Technical details of EPAS were described in Chapter 3. Functionally, EPAS depicts the execution of eleven specific Pascal programs. The first two programs, when animated, serve to introduce the system. Thus, they account for the first phase of the Reigeluth and Schwartz model. The remaining nine programs are divided into three sections for instruction on arrays, value parameters and variable parameters. Each section contains three programs. Within each section, the program animations take the learner through the acquisition, application and assessment phases of the model. One program is animated for each phase.

When EPAS is executing, the screen is divided into three windows. One of the Pascal programs is displayed in the Code Window. The Stack Window contains a consecutive set of rectangles arranged vertically. The set of rectangles depicts the run-time stack. Output that would be generated by the program in the Code Window is displayed in the Output Window.

To depict execution, one statement in the Code Window is highlighted. The values that would be read from and written to memory, if the highlighted statement were executed, are identified in the Stack Window. If the highlighted statement is a *write* or *writeln* statement, output data are displayed in the Output Window. Note that Stack Window activity does not always depict memory reading and writing. For example, when a declaration is highlighted, a variable name is placed in a stack cell. In Chapter 3, five stack activities were defined to account for all of the statements found in the eleven Pascal programs animated by EPAS. The manner in which each stack activity changes the stack was specified in Table 3.1.

Technically, EPAS contains three major components that were written in the C programming language. The three modules are shown in Figure 6.1. The size of each module, with respect to the entire system, is also given in the figure.

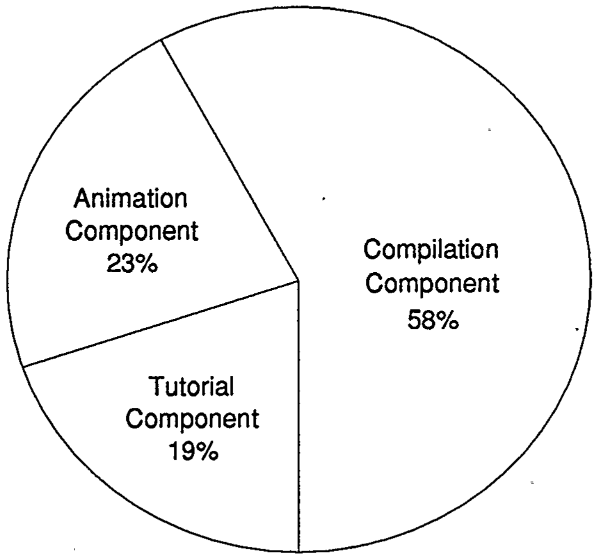


Figure 6.1: The Major Components of EPAS

The compilation modules consist of a scanner, a parser and a symbol table generator. Even though they are operational, they exist in EPAS primarily for enhancement potential. EPAS could have been written without them, but discussing enhancements to the compilation modules would have been difficult if they were not included. The animation module controls Stack Window activity. Specifically, it contains X-Window calls to display data in stack cells, highlight stack cells and draw parameter arrows. When appropriate, the animation module performs the converse of those functions.

The tutorial module ensures that all phases of the instructional model are presented appropriately. This module also contains X-Window calls. In particular, the calls display instructional material and permit the user to respond to directives. Further, the tutorial module provides feedback during the practice phase and scores the responses entered during the quiz program animations.

Also described in Chapter 3 were the procedures followed during the EPAS assessment study. In short, eight students with little computing and programming experience viewed the eleven program animations described above. The program animations presented new programming processes to the students. At various times during the animations, the students were directed to identify specific stack cells by moving and clicking the mouse input device.

Analyzing the extent to which the students correctly identified stack cells provided insight into their ability to determine how statements manipulate run-time stack data. The methods and findings of the analysis were reported in Chapter 4. On average, the students correctly identified stack cells more than seventy-five percent of the time. This level of achievement was regarded as satisfactory because it exceeded the seventy-five percent mastery level established before the study was conducted.

Since the future viability of EPAS was regarded as important throughout this research, data were collected to determine if the reverse execution feature in EPAS should be retained in an enhanced system. In particular, tallies were kept to determine precisely the number of times execution was reversed. It appears that the feature should be retained because the students, on average, reversed execution once for every eight stack events.

Methods for enhancing EPAS were discussed in Chapter 5. By creating additional data files, EPAS could animate more programs. However, this approach to improvement would never serve programmers seeking to visualize their code. Thus, the ultimate goal for improvement is the development of a system that would animate any Pascal program.

Most of the discussion addressed compilation and animation issues. More specifically, a method for generating code that could animate stack activity and reverse execution was detailed. The method involves modifying a typical run-time stack by adding data for animation purposes. Also to reverse execution, a data store for previous values must be maintained and additional executable code generated.

Lastly and briefly, approaches to updating the tutorial module were described. The main enhancement theme in this discussion was the development of an editing tool to assist instructors. By using the editing tool, teachers could insert instructional notations and directives into tutorial programs.

6.2 Suggestions for Further Research

This section proposes four paths for further research. The first two paths could be pursued immediately. In contrast, the last two options could only be considered if EPAS were enhanced.

First, additional insights into the utility of EPAS could be gained by testing the effectiveness of animations that depict recursion. This could be done by creating data files for recursive programs and following the study methodology described in Section 3.4. In this case, though, it may be beneficial to seek second-year computer science students for testing purposes. In addition, another aspect of testing could be incorporated. In particular, the students could be asked to write a recursive program after completing the stack cell identification quiz program.

Second, one could seek to incorporate EPAS into various instructional contexts. For example, by making only minor modifications to the existing data files, programs in languages such as C could be animated. In fact, only Code Window display information would have to be changed. Given the popularity of C, this ease of modification feature is important. As a second example, one could pursue the use of EPAS in a compiler writing course. Initially, just by changing the existing instructional notations, students could be shown how variables are assigned run-time stack locations. Subsequently, modifying the animation module so that it would depict static and dynamic links could prove beneficial.

Third, it is important to note that one could embellish a combination compiler/animator based on EPAS by adding a debugger. Unlike most debuggers, the one in the enhanced system would not have to display values of variables because the animator would fulfill that function. However, it should contain other standard debugging features. For instance, the debugger should allow a programmer to change the values of variables and then resume execution. Further, it should permit the use of breakpoints. Interestingly, one should be able to incorporate forward and reverse breakpoints into an enhanced system. Programmers may be able to reduce debugging time by using reverse breakpoints. For example, using such a technique could reveal an incorrect value faster than a forward execution breakpoint.

Fourth, when a stable enhanced system has been developed, summative evaluations should be performed. For example, one could compare the effectiveness of traditional classroom instruction versus the enhanced system. Additionally or alternatively, one could compare the quality of programs developed in standard debugging settings versus the combination compiler/animator/debugger environment. Further, one could compare development times in both of those environments.

6.3 Conclusion

This research showed that students gained knowledge about programming concepts and techniques by viewing program animations. Depicting run-time stack activity was the focus of the program animations. To further this research, the prototypic program animation system developed for this study, called EPAS, should be enhanced to accept a broad range of Pascal programs as input. An enhanced system may help students debug simple programs. In addition, the tutorial module in EPAS should be incorporated into the complete program animation system. It is recognized that developing a system with those components would be complex. Therefore, changes to the existing system should be made over a considerable length of time. Further, the system should be monitored throughout the development process to maximize its potential.

References

Aho, A.V., Sethi, R. & Ullman, J.D (1988). *Compilers: principles, techniques and tools.* Menlo Park, California: Addison-Wesley.

Amenda, D.K. (1990). *The graphical pascal execution model.* Unpublished manuscript, The University of Calgary, Department of Computer Science.

Baecker, R. (1975). Two systems which produce animated representations of the execution of computer programs. *SIGCSE Bulletin*, 7(1), 158-167.

Baecker, R. (1981). *Sorting out sorting.* 16 mm colour and sound film. Dynamic Graphics Project, The University of Toronto. Presented at ACM Siggraph Conference, Dallas, Texas, August, 1981.

Baecker, R. (1988). Enhancing program readability and comprehensibility with tools for program visualization. *Proceedings of the 10th International Conference on Software Engineering*, Singapore, April 11-15, 356-366.

Boehm, B.W. (1981). *Software engineering economics.* Englewood Cliffs, New Jersey: Prentice-Hall.

Borland. (1988). *Turbo Pascal reference guide (Version 5.0).* Scotts Valley, California.

Brown, G.P., Carling, R.T., Herot, C.F., Kramlich, D.A. & Souza, P. (1985). Program visualization: Graphical support for software development. *IEEE Computer*, 18(8), 27-35.

Brown, M.H. (1988). Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5), 14-36.

Brown, M.H. & Sedgewick, R. (1984). A system for algorithm animation. *Computer Graphics*, 18(3), 177-186.

Cherry, G.W. (1980). *Pascal programming structures: an introduction to systematic programming.* Reston, Virginia: Reston.

Chicelli, R.J. (1980). Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1), 17-19.

Delisle, N.M., Menicosy, D.E. & Schwartz, M.D. (1984). Viewing a programming environment as a single tool. *Proceedings of the ACM Sigsoft/Sigplan*

Software Engineering Symposium on Practical Software Development Environments, April, 1984; *Software Engineering Notes*, 9(3); and *Sigplan Notices*, 19(5), both May, 1984, 49-56.

Eckhouse, R.H. & Morris, L.R. (1979). *Minicomputer systems.* Englewood Cliffs, New Jersey: Prentice-Hall.

Fischer, C.N. & LeBlanc, R.J. (1988). *Crafting a compiler.* Menlo Park, California: Benjamin/Cummings.

Gurwitz, R.F., Fleming, R.T. & Van Dam, A. (1981). MIDAS: A microprocessor display and animation system. *IEEE Transactions on Education*, E-24(2), 126-133.

Hannafin, M.J. & Rieber, L.P. (1989). Psychological foundations of instructional design for emerging computer-based instructional technologies: Part I. *Educational Technology – Research and Development*, 37(2), 91-101.

Hille, R.F. & Higginbottom, T.F. (1983). A system for visible execution of pascal programs. *The Australian Computer Journal*, 15(2), 76-77.

Knuth, D.E. & Pardo, L.T. (1980). The early development of programming languages. In N. Metropolis, J. Howlett & G. Rota (Eds.), *A history of computing in the twentieth century* (pp. 197-273). New York, New York: Academic Press.

Krishnamoorthy, M.S. & Swaminathan, R. (1989). Program tools for algorithm animation. *Software - Practice and Experience*, 19(6), 505-513.

Lafore, R.W. (1984). *Assembly language primer for the IBM PC & XT.* Scarborough, Ontario: Plume.

London, R.L. & Duisberg, R.A. (1985). Animating programs using smalltalk. *IEEE Computer*, 18(8), 61-71.

Merrill, M.D. (1987). The new component design theory: instructional design for courseware authoring. *Instructional Science*, 16(1), 19-34.

Mills, H.D. (1975). The new math of computer programming. *Communications of the ACM*, 18(1), 43-48.

Myers, B.A. (1989). The state of the art in visual programming and program visualization. In A. Kilgour & R. Earnshaw (Eds.), *Graphics tools for software engineers* (pp. 3-26). Cambridge, Great Britain: Cambridge University Press.

Nassi, I. & Schneiderman, B. (1973). Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8), 12-26.

Plattner, B. & Nievergelt, J. (1981). Monitoring program execution: A survey. *IEEE Computer*, 14(11), 76-93.

Protsko, L.B., Sorenson, P.G., Tremblay J.P. & Schaefer, D.A. (1991). Towards the automatic generation of software diagrams. *IEEE Transactions on Software Engineering*, 17(1), 10-21.

Reigeluth, C.M. (1983). Instructional design: what is it and why is it? In C.M. Reigeluth (Ed), *Instructional-design theories and models: an overview of their current status* (pp. 3-37). Hillsdale, New Jersey: Lawrence Erlbaum.

Reigeluth, C.M. & Schwartz, E. (1989). An instructional theory for the design of computer-based simulations. *Journal of Computer-Based Instruction*, 16(1), 1-10.

Reiss, S.P. (1985). PECAN: program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3), 276-284.

Reiss, S.P. (1990a). Interacting with the FIELD environment. *Software - Practice and Experience*, 20(S1), 89-115.

Reiss, S.P. (1990b). Connecting tools using message passing in the FIELD program development environment. *IEEE software*, 7(4), 57-66.

Ross, R.J. (1991). Experience with the DYNAMOD program animator. *The Papers of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education*, San Antonio, Texas, March 7-8; SIGCSE Bulletin, 23(1), 35-42.

Savitch, W.J. (1991). *Pascal (An introduction to the art and science of programming)*. Redwood City, California: Benjamin/Cummings.

Shu, N.C. (1988). *Visual programming.* New York, NY: Van Nostrand Reinhold.

Stasko, J.T. (1990). Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9), 27-39.

Sun Microsystems (1989). *Pascal reference manual (Revision A)*. Mountain View, CA.

Sun Microsystems (1990a). *Debugging tools manual (Revision A)*. Mountain View, CA.

Sun Microsystems (1990b). *Sunview programmer's guide (Revision A).* Mountain View, California.

van Berkum, J.J. & de Jong, T. (1991). Instructional environments for simulations. *Education and Computing*, 6(3,4), 305-358.

Visible Software. (1989). *Dr. Pascal user manual (Version 2).* Princeton, New Jersey.

Williams, M.R. (1985). *A history of computing technology.* Englewood Cliffs, New Jersey: Prentice-Hall.

Wirth, N. (1976). *Algorithms + data structures = programs.* Englewood Cliffs, New Jersey: Prentice-Hall.

Appendix A

The Grammar Parsed By EPAS

⟨program⟩ ::⇒ ⟨program heading⟩ ⟨declarations⟩ ⟨block⟩ .

⟨program heading⟩ ::⇒ **program** ⟨identifier⟩ ⟨program parameters⟩ ;

⟨program parameters⟩ ::⇒ (⟨identifier list⟩) | ⟨empty⟩

⟨identifier list⟩ ::⇒ ⟨identifier⟩ ⟨identifier list tail⟩

⟨identifier list tail⟩ ::⇒ , ⟨identifier⟩ ⟨identifier list tail⟩ | ⟨empty⟩

⟨declarations⟩ ::⇒ ⟨label declaration part⟩ ⟨constant declaration part⟩
⟨type declaration part⟩ ⟨variable declaration part⟩
⟨subroutine declaration part⟩

⟨label declaration part⟩ ::⇒ **label** ⟨label list⟩ ; | ⟨empty⟩

⟨label list⟩ ::⇒ ⟨label⟩ ⟨label list tail⟩

⟨label list tail⟩ ::⇒ , ⟨label⟩ ⟨label list tail⟩ | ⟨empty⟩

⟨label⟩ ::⇒ ⟨unsigned integer⟩

⟨constant declaration part⟩ ::⇒ **const** ⟨constant declaration list⟩ | ⟨empty⟩

⟨constant declaration list⟩ ::⇒ ⟨constant declaration⟩
⟨constant declaration list tail⟩

⟨constant declaration⟩ ::⇒ ⟨identifier⟩ = ⟨constant⟩ ;

⟨constant declaration list tail⟩ ::⇒ ⟨constant declaration⟩
⟨constant declaration list tail⟩ | ⟨empty⟩

⟨constant⟩ ::⇒ ⟨sign⟩ ⟨constant tail⟩ | ⟨character string⟩

⟨constant tail⟩ ::⇒ ⟨unsigned number⟩ | ⟨constant identifier⟩

⟨unsigned number⟩ ::⇒ ⟨unsigned integer⟩ | ⟨unsigned real⟩

⟨sign⟩ ::⇒ + | - | ⟨empty⟩

⟨constant identifier⟩ ::⇒ ⟨identifier⟩

⟨type declaration part⟩ ::⇒ **type** ⟨type declaration list⟩ | ⟨empty⟩

⟨type declaration list⟩ ::⇒ ⟨type declaration⟩ ⟨type declaration list tail⟩

⟨type declaration⟩ ::⇒ ⟨identifier⟩ = ⟨type⟩ ;

⟨type declaration list tail⟩ ::⇒ ⟨type declaration⟩
  ⟨type declaration list tail⟩ | ⟨empty⟩

⟨type⟩ ::⇒ ⟨simple type⟩ | ^ ⟨type identifier⟩ |
  ⟨structured type⟩ | **packed** ⟨structured type⟩

⟨simple type⟩ ::⇒ ⟨identifier⟩ ⟨simple type identifier tail⟩ |
  + ⟨simple type sign tail⟩ | − ⟨simple type sign tail⟩ |
  ⟨character string⟩ ⟨partial range type⟩ | ⟨unsigned number⟩
  ⟨partial range type⟩ | ⟨enumerated type⟩

⟨simple type identifier tail⟩ ::⇒ ⟨partial range type⟩ | ⟨empty⟩

⟨simple type sign tail⟩ ::⇒ ⟨unsigned number⟩ ⟨partial range type⟩ |
  ⟨constant identifier⟩ ⟨partial range type⟩

⟨partial range type⟩ ::⇒ .. ⟨constant⟩

⟨enumerated type⟩ ::⇒ (⟨identifier list⟩)

⟨type identifier⟩ ::⇒ ⟨identifier⟩

⟨structured type⟩ ::⇒ **array** [⟨simple type list⟩] **of** ⟨type⟩ | **file of** ⟨type⟩ |
  set of ⟨type⟩ | **record** ⟨field list⟩ **end**

⟨simple type list⟩ ::⇒ ⟨simple type⟩ ⟨simple type list tail⟩

⟨simple type list tail⟩ ::⇒ , ⟨simple type⟩ ⟨simple type list tail⟩ | ⟨empty⟩

⟨field list⟩ ::⇒ ⟨field⟩ ⟨field list tail⟩ | ⟨variant part⟩

⟨field list tail⟩ ::⇒ ; ⟨field list⟩ | ⟨empty⟩

⟨field⟩ ::⇒ ⟨identifier list⟩ : ⟨type⟩ | ⟨empty⟩

⟨variant part⟩ ::⇒ **case** ⟨identifier⟩ ⟨variant part interior⟩ **of** ⟨variant list⟩

⟨variant part interior⟩ ::⇒ : ⟨type identifier⟩ | ⟨empty⟩

⟨variant list⟩ ::⇒ ⟨variant⟩ ⟨variant list tail⟩

⟨variant list tail⟩ ::⇒ ; ⟨variant⟩ ⟨variant list tail⟩ | ⟨empty⟩

⟨variant⟩ ::⇒ ⟨constant list⟩ : (⟨field list⟩) | ⟨empty⟩

⟨constant list⟩ ::⇒ ⟨constant⟩ ⟨constant list tail⟩

⟨constant list tail⟩ ::⇒ , ⟨constant⟩ ⟨constant list tail⟩ | ⟨empty⟩

⟨variable declaration part⟩ ::⇒ **var** ⟨variable declaration list⟩ | ⟨empty⟩

⟨variable declaration list⟩ ::⇒ ⟨variable declaration⟩
    ⟨variable declaration list tail⟩

⟨variable declaration⟩ ::⇒ ⟨identifier list⟩ : ⟨type⟩ ;

⟨variable declaration list tail⟩ ::⇒ ⟨variable declaration⟩
    ⟨variable declaration list tail⟩ | ⟨empty⟩

⟨subroutine declaration part⟩ ::⇒ ⟨procedure heading⟩ ;
    ⟨subroutine declaration tail⟩ ; ⟨subroutine declaration part⟩ |
    ⟨function heading⟩ ;
    ⟨subroutine declaration tail⟩ ; ⟨subroutine declaration part⟩ | ⟨empty⟩

⟨subroutine declaration tail⟩ ::⇒ ⟨identifier⟩ | ⟨declarations⟩ ⟨block⟩

⟨procedure heading⟩ ::⇒ **procedure** ⟨identifier⟩ ⟨formal parameters⟩

⟨function heading⟩ ::⇒ **function** ⟨identifier⟩ ⟨formal parameters⟩ :
    ⟨type identifier⟩

⟨formal parameters⟩ ::⇒ (⟨parameter list⟩) | ⟨empty⟩

⟨parameter list⟩ ::⇒ ⟨parameter⟩ ⟨parameter list tail⟩

⟨parameter list tail⟩ ::⇒ ; ⟨parameter⟩ ⟨parameter list tail⟩ | ⟨empty⟩

⟨parameter⟩ ::⇒ ⟨identifier list⟩ : ⟨type identifier⟩ |
    var ⟨identifier list⟩ : ⟨type identifier⟩ |
    ⟨procedure heading⟩ | ⟨function heading⟩

⟨block⟩ ::⇒ **begin** ⟨statement list⟩ **end**

⟨statement list⟩ ::⇒ ⟨statement⟩ ⟨statement list tail⟩

⟨statement list tail⟩ ::⇒ ; ⟨statement⟩ ⟨statement list tail⟩ | ⟨empty⟩

⟨statement⟩ ::⇒ ⟨identifier⟩ ⟨statement identifier tail⟩ |
    for ⟨identifier⟩ := ⟨expression⟩ ⟨for loop tail⟩ |
    while ⟨expression⟩ **do** ⟨statement⟩ |
    repeat ⟨statement list⟩ **until** ⟨expression⟩ |
    if ⟨expression⟩ **then** ⟨statement⟩ ⟨if tail⟩ |
    case ⟨expression⟩ **of** ⟨case statement list⟩ **end** |
    with ⟨variable list⟩ **do** ⟨statement⟩ |
    ⟨block⟩ |
    ⟨label⟩ : ⟨statement⟩ |
    goto ⟨label⟩ |
    ⟨empty⟩

⟨statement identifier tail⟩ ::⇒ (⟨actual parameter list⟩) |
    ⟨variable tail⟩ := ⟨expression⟩ |
    ⟨empty⟩

⟨actual parameter list⟩ ::⇒ ⟨actual parameter⟩ ⟨actual parameter list tail⟩

⟨actual parameter list tail⟩ ::⇒ , ⟨actual parameter⟩ ⟨actual parameter list tail⟩
    | ⟨empty⟩

⟨actual parameter⟩ ::⇒ ⟨expression⟩ ⟨actual parameter tail1⟩

⟨actual parameter tail1⟩ ::⇒ : ⟨expression⟩ ⟨actual parameter tail2⟩ | ⟨empty⟩

⟨actual parameter tail2⟩ ::⇒ : ⟨expression⟩ | ⟨empty⟩

⟨variable⟩ ::⇒ ⟨identifier⟩ ⟨variable tail⟩

⟨variable tail⟩ ::⇒ [⟨expression list⟩] ⟨variable tail⟩ |
    . ⟨field identifier⟩ ⟨variable tail⟩ |
    ^ ⟨variable tail⟩ | ⟨empty⟩

⟨variable list⟩ ::⇒ ⟨variable⟩ ⟨variable list tail⟩

⟨variable list tail⟩ ::⇒ , ⟨variable⟩ ⟨variable list tail⟩ | ⟨empty⟩

⟨for loop tail⟩ ::⇒ **to** ⟨expression⟩ **do** ⟨statement⟩ |
    downto ⟨expression⟩ **do** ⟨statement⟩

⟨if tail⟩ ::⇒ **else** ⟨statement⟩ | ⟨empty⟩

⟨case statement list⟩ ::⇒ ⟨case list element⟩ ⟨case statement list tail⟩

⟨case statement list tail⟩ ::⇒ ; ⟨case list element⟩ ⟨case statement list tail⟩ |
⟨empty⟩

⟨case list element⟩ ::⇒ ⟨constant list⟩ : ⟨statement⟩ |
otherwise ⟨statement⟩ | ⟨empty⟩

⟨field identifier⟩ ::⇒ ⟨identifier⟩

⟨expression⟩ ::⇒ ⟨simple expression⟩ ⟨expression tail⟩

⟨expression tail⟩ ::⇒ ⟨relational operator⟩ ⟨simple expression⟩ | ⟨empty⟩

⟨simple expression⟩ ::⇒ ⟨term⟩ ⟨simple expression tail⟩

⟨simple expression tail⟩ ::⇒ ⟨adding operator⟩ ⟨term⟩ ⟨simple expression tail⟩ |
⟨empty⟩

⟨term⟩ ::⇒ ⟨factor⟩ ⟨term tail⟩

⟨term tail⟩ ::⇒ ⟨multiplying operator⟩ ⟨factor⟩ ⟨term tail⟩ | ⟨empty⟩

⟨factor⟩ ::⇒ **nil** | **not** ⟨factor⟩ | ⟨character string⟩ | ⟨sign⟩ ⟨factor sign tail⟩ |
[⟨set element list⟩]

⟨factor sign tail⟩ ::⇒ (⟨expression⟩) | ⟨unsigned number⟩ |
⟨identifier⟩ ⟨factor identifier tail⟩

⟨factor identifier tail⟩ ::⇒ (⟨actual parameter list⟩) | ⟨variable tail⟩

⟨set element list⟩ ::⇒ ⟨set element⟩ ⟨set element list tail⟩ | ⟨empty⟩

⟨set element list tail⟩ ::⇒ , ⟨set element⟩ ⟨set element list tail⟩ | ⟨empty⟩

⟨set element⟩ ::⇒ ⟨expression⟩ ⟨set element tail⟩

⟨set element tail⟩ ::⇒ .. ⟨expression⟩ | ⟨empty⟩

⟨relational operator⟩ ::⇒ = | ⟨⟩ | ⟨ | ⟨= | ⟩ = | ⟩ | **in**

⟨adding operator⟩ ::⇒ + | − | **or**

⟨multiplying operator⟩ ::⇒ ***** | **/** | **div** | **mod** | **and**

⟨expression list⟩ ::⇒ ⟨expression⟩ ⟨expression list tail⟩

⟨expression list tail⟩ ::⇒ **,** ⟨expression⟩ ⟨expression list tail⟩ | ⟨empty⟩

⟨empty⟩ ::⇒

⟨identifier⟩ ::⇒ ⟨letter⟩ ⟨identifier tail⟩

⟨identifier tail⟩ ::⇒ ⟨letter⟩ ⟨identifier tail⟩ | ⟨digit⟩ ⟨identifier tail⟩ |
  _ ⟨identifier tail⟩ | ⟨empty⟩

⟨unsigned integer⟩ ::⇒ ⟨digit⟩ ⟨unsigned integer tail⟩

⟨unsigned integer tail⟩ ::⇒ ⟨digit⟩ ⟨unsigned integer tail⟩ | ⟨empty⟩

⟨unsigned real⟩ ::⇒ ⟨unsigned integer⟩ ⟨unsigned real interior⟩

⟨unsigned real interior⟩ ::⇒ **.** ⟨unsigned integer⟩ ⟨unsigned real interior tail⟩ |
  ⟨unsigned real tail⟩

⟨unsigned real interior tail⟩ ::⇒ ⟨unsigned real tail⟩ | ⟨empty⟩

⟨unsigned real tail⟩ ::⇒ **e** ⟨scale factor⟩ | **E** ⟨scale factor⟩

⟨scale factor⟩ ::⇒ ⟨sign⟩ ⟨unsigned integer⟩

⟨character string⟩ ::⇒ **'** ⟨string element list⟩ **'**

⟨string element list⟩ ::⇒ ⟨string element⟩ ⟨string element list tail⟩

⟨string element list tail⟩ ::⇒ ⟨string element⟩ ⟨string element list tail⟩ | ⟨empty⟩

⟨string element⟩ ::⇒ ⟨any character except apostrophe or newline⟩ |
  ⟨apostrophe image⟩

⟨apostrophe image⟩ ::⇒ **''**

⟨letter⟩ ::⇒ **A** | **B** | **C** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** |
  R | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z** | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** |
  k | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**

⟨digit⟩ ::⇒ **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Appendix B

The Pascal Programs Animated By EPAS

```
program division_by_repeated_subtraction;

   { This program computes 27 divided by 7.
     The quotient and remainder are calculated
     by repeated subtraction. }

var
   dividend, quotient : integer;

begin { "Main" }
   dividend := 27;
   quotient := 0;

   while dividend ≥ 7 do
      begin
         dividend := dividend - 7;
         quotient := quotient + 1
      end;

   write('Seven goes into twenty-seven, ', quotient, ' times. ');
   writeln('The remainder is ', dividend, '.')
end.
```

```
program swap2numbers;

   { This program uses a procedure to swap two numbers. }

var
   a, b, s1, s2 : real;

   procedure swap;

      { This procedure uses the local variable "temp" to
        swap the two numbers in the global variables, "s1" & "s2." }

   var
      temp : real;
```

```
    begin
        temp := s1;
        s1 := s2;
        s2 := temp
    end; { Swap }

begin { "Main" }
    a := -7.61;
    b := 3.2;
    writeln('a = ', a:2:2, ' b = ', b:2:2);
    writeln;

    s1 := a;
    s2 := b;
    swap;
    a := s1;
    b := s2;
    writeln('a = ', a:2:2, ' b = ', b:2:2)
end.
```

```
program array_demo;

      { This program initializes an array to 6 random numbers between
         1 and 4. It then searches the entire array for the number 3.
         The positions in the array containing the number 3 are displayed. }

var
    i : integer;
    nums : array[1..6] of integer;

begin { "Main" }
    for i := 1 to 6 do
        nums[i] := random(4) + 1;

    for i := 1 to 6 do
        if nums[i] = 3 then
            writeln('A 3 is in position ', i)
end.
```

```
program tally_random_numbers;

    { This program generates 6 random numbers between 1 and 3.
      It also computes the number of times each random number is drawn. }

var
   sums : array[1..3] of integer;
   i : integer;
   value : integer;

begin { "Main" }
   for i := 1 to 3 do
      sums[i] := 0;

   for i := 1 to 6 do
      begin
         value := random(3) + 1;
         sums[value] := sums[value] + 1;
      end;

   for i := 1 to 3 do
      writeln(i, ' was drawn ', sums[i], ' time(s).');
end.
```

```
program quiz1;
var
   i : integer;
   nums : array[1..3] of integer;
   sum : real;
   difference : real;
   mean : real;
   result : real;

begin { "Main" }
   sum := 0.0;
```

```
    for i := 1 to 3 do
        begin
            nums[i] := random(10);
            sum := sum + nums[i]
        end;
    mean := sum / 3;

    sum := 0.0;
    for i := 1 to 3 do
        begin
            difference := abs(nums[i] - mean);
            sum := sum + difference
        end;
    result := sum / 3;

    writeln('The mean is ', mean:3:3);
    writeln('The result is ', result:3:3);
end.



program display_triangle;

    { This program calls a procedure to display lines of a triangle. }

var
    i : integer;

    procedure make_line(stop : integer);

       { This procedure displays a line of asterisks. }

    var
        i : integer;
    begin
        for i := 1 to stop do
            write('*');
        writeln('*')
    end; { make_line }
```

```
begin { "Main" }
    make_line(3);

    for i := 2 downto 1 do
        make_line(i);

    writeln('*')
end.
```

```
program display_figure;

      { This program calls two procedures to display a figure. }

var
    i : integer;

    procedure interior(total_rows : integer; ch : char);
    var
        i : integer;
    begin
        for i := 1 to total_rows do
            writeln('*', ch, ch, '*');
    end;

    procedure exterior(ch : char);
    begin
        writeln(ch, ch, ch, ch);
    end;

begin { "Main" }
    for i := 1 to 3 do
        if i mod 2 = 0 then
            interior(i, '+')
        else
            exterior('*')
end.
```

```
program quiz2;
var
    chars : array[1..2] of char;
    i, j : integer;

    procedure display(ch1, ch2 : char);
    var
        i : integer;
    begin
        for i := 1 to 2 do
            write(ch1, ch2);
        writeln
    end; { "display" }

begin { "Main" }
    chars[1] := '/';
    chars[2] := '\';

    for i := 1 to 2 do
        begin
            j := 3 - i;
            display(chars[i], chars[j])
        end;

    display('+', '+')
end.
```

```
program loan_payments;

    { This program computes the number of months it takes to
      pay off a $2000 loan, given monthly payments of $750.00 and
      interest charged at 12% per year. }

var
    principal : real;
    months : integer;
    procedure payment(var principal : real; var time : integer);

        { This procedure simulates a monthly payment. }

    var
        interest : real;
    begin
        interest := principal * 0.01;
        principal := principal + interest;
        principal := principal - 750.00;
        time := time + 1
    end; { payment }

begin { "Main" }
    principal := 2000.00;
    months := 1;

    repeat
        payment(principal, months);
    until principal < 750.00;

    writeln('It will take ', months, ' months to pay off the loan.');
    writeln('However, in the last month only $', principal:2:2);
    writeln('plus interest must be paid.')
end.



program reverse;

    { This program assigns 6 random numbers between 0 and 99 to
      array, "nums." Then it reverses the order of the numbers
      in "nums." }

var
    i : integer;
    index : integer;
    nums : array[1..6] of integer;

    procedure swap(previous_x : integer; var x, y : integer);
```

```
        { This procedure swaps the two numbers mapped
          by the variable parameters, x and y. }

    begin
        x := y;
        y := previous_x
    end; { swap }

begin { "Main" }
    for i := 1 to 6 do
        begin
            nums[i] := random(100);
            write(nums[i]:4)
        end;
    writeln;

    for i := 1 to 3 do
        begin
            index := 7 - i;
            swap(nums[i], nums[i], nums[index])
        end;

    for i := 1 to 6 do
        write(nums[i]:4)
end.



program quiz3;
var
    i : integer;
    nums : array[1..4] of integer;
    counter : integer;
    index : integer;

    procedure move(var num1, num2, ctr : integer);
    var
        temp : integer;
    begin
```

```
        temp := num1;
        num1 := num2;
        num2 := temp;

        ctr := ctr + 1
    end; { move }

begin { "Main" }
    for i := 1 to 4 do
        begin
            nums[i] := random(5);
            write(nums[i]:3)
        end;
    writeln;

    repeat
        counter := 0;
        for i := 1 to 3 do
            begin
                index := i + 1;
                if nums[i] > nums[index] then
                    move(nums[i], nums[index], counter)
            end
    until counter = 0;

    for i := 1 to 4 do
        write(nums[i]:3)
end.
```