A miniPascal compiler for the E-machine
by Frances Wren Goosey

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Frances Wren Goosey (1993)

Abstract:
This thesis is the third phase in the development of a program animation system called DYNALAB (DYNAmic LABoratory). DYNALAB is an interactive software system that demonstrates programming and computer science concepts at an introductory level. The first DYNALAB development phase was the design of a virtual computer—the E-machine (Education Machine). The E-machine was designed by Samuel D. Patton and is presented in his Master's thesis, The E-machine: Supporting the Teaching of Program Execution Dynamics. In order to facilitate the support of program animation activities, the E-machine has many unique features, notably the ability to execute in reverse. The second phase in the development of DYNALAB was the design and implementation of an E-machine emulator, which is presented in Michael L. Birch's Master's thesis, An Emulator for the E-machine. This thesis presents the design and implementation of a compiler for the E-machine. The compiler's source language is miniPascal, which is a subset of ISO Standard Pascal.

The miniPascal compiler was developed using the Unix lex and yacc compiler development tools. It has successfully generated object files ready for execution on the E-machine. This thesis focuses on the compilation aspects that are unique to the E-machine architecture and the planned animation environment.

A miniPASCAL COMPILER FOR THE E-MACHINE

by

Frances Wren Goosey

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

Montana State University
Bozeman, Montana

April 1993

T378
G644

APPROVAL

of a thesis submitted by

Frances Wren Goosey

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

4/27/93
Date

Rockford J. Ross
Chairperson, Graduate Committee

Approved for the Major Department

4/27/93
Date

J Dunbigh Stanley
Head, Major Department

Approved for the College of Graduate Studies

5/7/93
Date

R L Brown
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature Frances Wm. Dooley

Date 4/27/93

ACKNOWLEDGMENTS

This thesis is part of a larger software development project, called DYNALAB. The DYNALAB project evolved from an earlier pilot project called DYNAMOD [Ross 91], a program animation system that has been used extensively at Montana State University in introductory Pascal programming classes. DYNAMOD was originally developed by Cheng Ng [Ng 82-1, Ng 82-2] and later extended and ported to various computing environments by a number of students, including Lih-nah Meng, Jim McInerny, Larry Morris, and Dean Gehnert. Experience with DYNAMOD proved the worth of program animation as a tool for teaching and learning programming and computer science concepts. It also provided extensive insight into the facilities needed in a fully functional program animation system and the inspiration for the subsequent DYNALAB project and this thesis.

Many people have contributed to the DYNALAB project. Samuel Patton and Michael Birch laid the groundwork for this thesis by designing and implementing the underlying virtual machine for DYNALAB in their Masters' theses. As this thesis is being completed, Craig Pratt is developing the animator portion of DYNALAB, and Robin Winslett and David Poole are implementing new compilers for the project.

I would like to take this opportunity to thank my graduate committee members, Dr. Rockford Ross, Dr. Gary Harkin, and Dr. Year Back Yoo, and the rest of the faculty members from the Department of Computer Science for their help and guidance during my graduate program. I would also like to thank my thesis advisor, Dr. Ross, and DYNALAB team members, David Poole, Craig Pratt, Robin Winslett, and Michael Woodring, for their help and suggestions for my thesis.

The original DYNAMOD project was supported by the National Science Foundation, grant number SPE-8320677. Work on this thesis was also supported in part by a grant from the National Science Foundation, grant number USE-9150298.

Contents

Contents—Continued

Contents—Continued

List of Tables

Table Page

List of Figures

ABSTRACT

This thesis is the third phase in the development of a program animation system called DYNALAB (DYNAmic LABoratory). DYNALAB is an interactive software system that demonstrates programming and computer science concepts at an introductory level. The first DYNALAB development phase was the design of a virtual computer—the E-machine (Education Machine). The E-machine was designed by Samuel D. Patton and is presented in his Master's thesis, *The E-machine: Supporting the Teaching of Program Execution Dynamics.* In order to facilitate the support of program animation activities, the E-machine has many unique features, notably the ability to execute in reverse. The second phase in the development of DYNALAB was the design and implementation of an E-machine emulator, which is presented in Michael L. Birch's Master's thesis, *An Emulator for the E-machine.* This thesis presents the design and implementation of a compiler for the E-machine. The compiler's source language is *miniPascal*, which is a subset of ISO Standard Pascal.

The miniPascal compiler was developed using the Unix lex and yacc compiler development tools. It has successfully generated object files ready for execution on the E-machine. This thesis focuses on the compilation aspects that are unique to the E-machine architecture and the planned animation environment.

CHAPTER 1

INTRODUCTION

The DYNALAB System

This thesis represents the third phase of the ongoing DYNALAB software development project. *DYNALAB* is an acronym for *DYNAmic LABoratory,* and its purpose is to support formal computer science laboratories at the introductory undergraduate level. Students will use DYNALAB to experiment with and explore programs and fundamental concepts of computer science. The current objectives of DYNALAB include:

- providing students with facilities for studying the dynamics of programming language constructs—such as iteration, selection, recursion, parameter passing mechanisms, and so forth—in an animated and interactive fashion;
- providing students with capabilities to validate or empirically determine the run time complexities of algorithms interactively in the experimental setting of a laboratory;
- extending to instructors the capability of incorporating animation into lectures on programming and algorithm analysis.

In order to meet these immediate objectives, the DYNALAB project was divided into four phases. The first phase was the design of a virtual computer, called the *Education Machine,* or *E-machine,* that would support the animation activities

envisioned for DYNALAB. The two primary technical problems to overcome in the design of the E-machine were the incorporation of features for reverse execution and provisions for coordination with a program animator. Reverse execution was engineered into the E-machine to allow students and instructors to animate repetitively sections of a program that were unclear without requiring that the entire program be restarted. Also, since the purpose of DYNALAB is to allow user interaction with animated programs, the E-machine had to be designed to be driven by an animator system that controls the execution of programs and displays pertinent information dynamically in animated fashion on a video screen. This first phase was completed by Samuel Patton in his Master's thesis, *The E-machine: Supporting the Teaching of Program Execution Dynamics* [Patton 89].

The second phase of the DYNALAB project was the implementation of an emulator for the E-machine. This was accomplished by Michael Birch in his Master's thesis, *An Emulator for the E-Machine,* [Birch 90]. As the emulator was implemented, Birch also included some modifications and extensions to the E-machine.

The third phase of the DYNALAB project, and the subject of this thesis, is the design and implementation of a Pascal compiler for the E-machine. The source language for the compiler is a subset of ISO Standard Pascal, called *miniPascal,* and the object language is *E-code,* the machine language of the E-machine. During compiler development, the E-machine and its emulator were again modified somewhat as practical considerations uncovered new design issues.

The fourth phase of the DYNALAB project, currently in progress, is the design and implementation of a program animator that will drive the E-machine and display miniPascal programs in dynamic, animated fashion under control of the user. Once the animator is complete, the first functional version of DYNALAB will be ready for use in introductory computer science laboratory and lecture courses by students and instructors alike.

The DYNALAB project will not end at this point. Compilers for other programming languages, such as C, Ada, and Juno—a pseudolanguage used purely for teaching [Winslett 93]—are in the initial stages of development. Algorithm animation (as opposed to program animation—see for example, [Brown 88-1, Brown 88-2]) is also a planned extension to DYNALAB. In fact, the DYNALAB project will likely never be finished, as new ideas and pedagogical conveniences are incorporated as they become apparent.

Preview

The thesis consists of five chapters and three appendices. Chapter 1 presents an overview of the thesis. Since a thorough understanding of the target computer's architecture and instruction set is required for compiler development, a summary of the E-machine and its emulator is given in chapter 2. Much of the information in chapter 2 is taken from the Patton and Birch theses. During the compiler development process, it became apparent that several additional E-machine features and modifications were necessary or desirable. These changes have been made and are so noted in chapter 2. For a more detailed explanation of the E-machine and its emulator, the reader is referred to the above-mentioned theses.

Chapter 3 describes the special considerations that E-machine compilers must address in order to function within the DYNALAB animation environment. Chapter 4 contains a description of the miniPascal compiler. The Pascal subset comprising the miniPascal language is presented, followed by an overview of the compiler design. It is the intent of chapter 4 to focus on the solutions to the compilation considerations unique to the DYNALAB animation environment. The current status of the miniPascal compiler is given in chapter 5. Chapter 5 also includes suggestions for future enhancements.

Since there are many E-code examples used throughout the thesis, appendices A and B are included for completeness. Appendix A describes the E-machine instruction set and appendix B describes the E-machine addressing modes. Both of these appendices are adapted from chapter 2 of Birch's thesis. Appendix C presents a complete miniPascal compilation example.

CHAPTER 2

THE E-MACHINE

This chapter is included to provide a description of the E-machine and is adapted from chapter 5 of Patton's thesis [Patton 89] and chapters 1, 2, and 3 of Birch's thesis [Birch 90]. This chapter is a summary and update of information from those two theses (much of the material is taken verbatim). New E-machine features that have been added as a result of this thesis are noted by a leading asterisk (*).

The E-machine is a virtual computer with its own machine language, called E-code. The E-code instructions are described in appendix A; these instructions may reference various E-machine addressing modes, which are described in appendix B. The E-machine's task is to execute E-code translations of high level language programs. The miniPascal language is the first language to be translated into E-code. The real purpose of the E-machine is to support the DYNALAB program animation system, as described more fully in [Ross 91], [Birch 90], [Ross 93] and in Patton's thesis [Patton 89], where it was called a "dynamic display system".

E-machine Design Considerations

The fact that the E-machine's sole purpose is to support program animation was central to its design. The E-machine operates as follows. After the E-machine

is loaded with a compiled E-code translation of a high level language program, it awaits a call from a driver program (the *animator*). A call from the animator causes a group of E-code instructions, called a *packet*, to be executed by the E-machine. A packet contains the E-code translation of a single high level language construct, or *animation unit*, that is to be highlighted by the animator. An animation unit could be a complete high level language assignment statement, for example

```
A := X + 2*Y;
```

which is to be highlighted as a result of a single call from the animator; the corresponding packet would be the E-code instructions that translate this assignment statement. Another animation unit could be just the conditional part of an if statement; in this case the corresponding packet would be just the E-code instructions translating the conditional expression. It is the compiler writer's responsibility to identify the animation units in the source program so that corresponding E-code packets can be generated. After the E-machine executes a packet, control is returned to the animator, which then performs the necessary animation activities before repeating the process by again calling the E-machine to execute the packet corresponding to the next animation unit. Chapter 3 describes this process in more detail.

Since the E-machine's purpose is to enable program execution dynamics of high level programming languages to be displayed easily by a program animator, it had to incorporate the following:

- structures for easy implementation of high level programming language constructs;
- a simple method for implementing functions, procedures, and parameters;
- the ability to execute either forward or in reverse.

The driving force in the design of the E-machine was the requirement for reverse execution. The approach taken by the E-machine to accomplish reverse execution

is to save the minimal amount of information necessary to recover just the previous E-machine state from the current state in a given reversal step. The E-machine can then be restored to an arbitrary prior state by doing the reversal one state at a time until the desired prior state is obtained. This one-step-at-a-time reversal means that it is necessary only to store successive differences between the previous state and the current state, instead of storing the entire state of the E-machine for each step of execution.

One other aspect of program animation substantially influenced the design of the reversing mechanism of the E-machine. Since the animator is meant to animate high level language programs, the E-machine actually has to be able to effect reversal only through high level language animation units in one reversal step, not each low level E-machine instruction in the packet that is the translation of an animation unit. This observation led to further efficiencies in the design of the E-machine and the incorporation of two classes of E-machine code instructions, critical and noncritical. An E-machine instruction within a packet is classified as *critical* if it destroys information essential to reversing through the corresponding high level language animation unit; it is classified as *noncritical* otherwise. For example, in translating the animation unit corresponding to an arithmetic assignment statement, a number of intermediate values are likely to be generated in the corresponding E-code packet. These intermediate values are needed in computing the value on the right-hand side of the assignment statement before this value can be assigned to the variable on the left-hand side. However, the only value that needs to be restored during reverse execution as far as the animation unit is concerned is the original value of the variable on the left-hand side. The intermediate values computed by various E-code instructions are of no consequence. Hence, E-code instructions generating intermediate values can be classified as noncritical and their effects ignored during reverse execution. It is the compiler writer's responsibility to produce the correct

E-code (involving critical and noncritical instructions) for reverse execution. However, it should also be noted that the E-machine has the flexibility to accurately execute E-code in reverse, instruction by instruction (rather than a packet at a time), by simply designating each E-code instruction as critical.

E-machine Architecture

Figure 1 shows the logical structure of the E-machine. A stack-based architecture was chosen for the E-machine; however, a number of components that are not found in real stack-based computers were included.

Program memory contains the E-code program currently being executed by the E-machine. Program memory is loaded with the instruction stream found in the CODESECTION of the E-machine object code file, which is described later in this chapter. The *program counter* contains the address in program memory of the next E-code instruction to be executed. The *previous program counter*, needed for reverse execution, contains the address in program memory of the most recently executed E-code instruction.

Packet memory contains information about the translated E-code packets and their corresponding source language animation units. Packet memory, which is loaded with the information found in the PACKETSECTION of the E-machine object code file, essentially effects the "packetization" of the E-code program found in program memory. Packet information includes the starting and ending line and column numbers of the original source program animation unit (e.g, an entire assignment statement, or just the conditional expression in an if statement) whose translation is the packet of E-code instructions about to be executed. Other packet information includes the starting and ending program memory addresses for the E-code packet, which are used internally to determine when execution of the packet

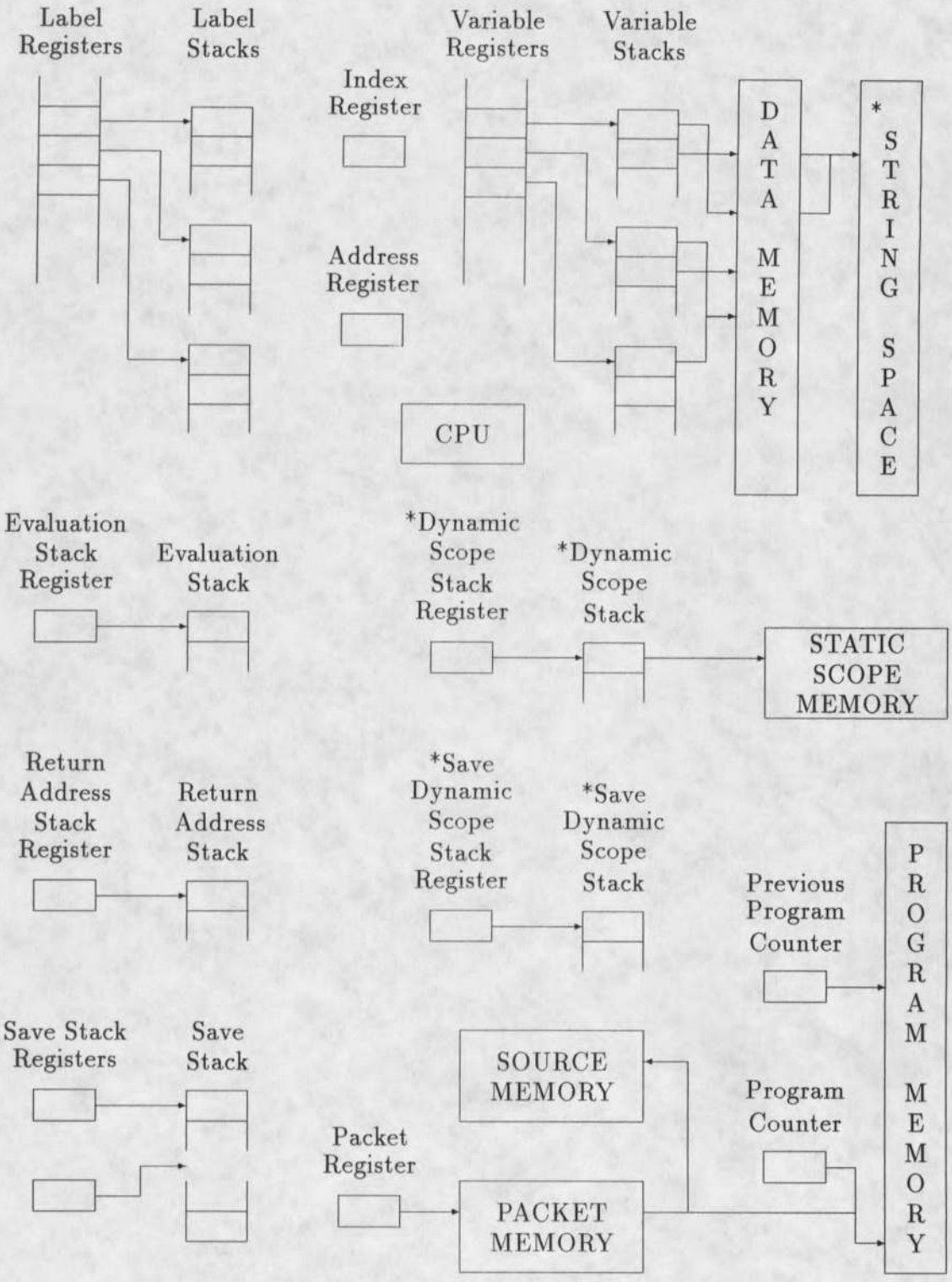

Figure 1: The E-machine

is complete. The *packet register* contains the packet memory address of the packet information corresponding to either the next packet to be executed, or the packet that is currently being executed.

The *variable registers* are an unbounded number of registers that are assigned to source program variables, constants, and parameters during compilation of a source program into E-code. Each identifier name representing memory in the source program will be assigned its own unique variable register in the E-machine. For example, in a miniPascal program, a variable named `Result` might be declared in the current program scope and another variable—also named `Result`—might be declared in another enclosing procedure scope. The compiler will assign a unique variable register to each of these two variables. Once a variable is assigned a variable register, the register remains associated with the variable for the duration of the program's compilation and subsequent execution, regardless of whether the variable is currently active or not.

The information held in a variable register consists of the corresponding variable's size (e.g., number of bytes) as well as a pointer to a corresponding *variable stack*. Each variable stack entry, in turn, holds a pointer into *data memory*, where the actual variable values are stored. The variable stacks are necessary because a particular variable may have multiple associated instances due to being declared in recursive procedures or functions. In such instances, the top of a particular variable's register stack points to the value of the current instance of the associated variable in data memory; the second stack element points to the value of the previous instantiation of the variable, and so on. The E-machine's data memory represents the usual random access memory found on real computers. The E-machine, however, uses data memory only to hold data values (it does not hold any of the program instructions).

*The *string space* component of the E-machine's architecture was added as a result of the miniPascal compiler development. The string space contains the values of all string literals and enumerated constant names encountered during the compilation of a miniPascal program. The string space is loaded with the information contained in the STRINGSECTION of the E-machine object file. Currently, this string space is used only by the animator when displaying string constant and enumerated constant values. A more detailed discussion of the interaction of the string space and variable registers is found in chapter 4

The *label registers* are another unique component of the E-machine required for reverse execution. There are an unbounded number of these registers, and they are used to keep track of labeled E-code instructions. Each E-code `label` instruction is assigned a unique label register at compile time. The information held in a label register consists of the program memory address of the corresponding E-code `label` instruction as well as a pointer to a *label stack*. A label stack essentially maintains a history of previous instructions that caused a branch to the label represented by the label register in question. During reverse execution, the top of the label stack allows for correct determination of the instruction that previously caused the branch to the label instruction.

The *index register* is found in real computers and serves the same purpose in the E-machine. In many circumstances, the data in a variable is accessed directly through the appropriate variable register. However, in the translation of a high level language data structure, such as an array or record, the address of the beginning of the structure is in a variable register; to access an individual data value in the structure, an offset—stored in the index register—is used. When necessary, the compiler can therefore utilize the index register so that the E-machine can access the proper memory location via one of the indexed addressing modes.

The *address register* is provided to allow access to memory areas that are not accessible through variable registers. For example, a pointer in Pascal is a variable that contains a data address. Data at that address can be accessed using the address register via the appropriate E-machine addressing mode. The address register can be used in place of variable registers for any of the addressing modes.

As in many real computers, the results of all arithmetic and logical operations are maintained on the *evaluation stack*; the *evaluation stack register* keeps track of the top of this stack. For example, in an arithmetic operation, the operands are pushed onto the evaluation stack and the appropriate operation is performed on them. The operands are consumed by the operation and the result is pushed onto the top of the stack. An assignment is performed by popping the top value of the evaluation stack and placing it into the proper location in data memory.

The *return address stack* (or *call stack*) is the E-machine's mechanism for implementing procedure and function calls. When a subroutine call is made, the program counter plus one is pushed onto the return address stack. Then, when the E-machine executes a return from subroutine instruction, all it has to do is load the program counter with the top of the return address stack. A pointer to the top of the return address stack is kept in the *return address stack register*.

The *save stack* contains information necessary for reverse execution. Whenever some critical information (as determined by the execution of a critical instruction) is about to be destroyed, the required information is pushed onto the save stack. This ensures that when backing up, the instruction that most recently destroyed some critical information can be reversed by retrieving that critical information from the save stack. The *save stack registers* point to the top and bottom of the save stack.

*The *dynamic scope stack* was added to the original E-machine architecture as a result of the miniPascal compiler development. The original E-machine did not provide a way for the animator to determine (for display) the currently

active program scopes. The animator must be able to display variable values associated with the execution of a packet both from within the current invocation of a procedure (or function) and from within the calling scope(s). That is, the animator must have the ability to illustrate a program's run time stack during execution. The Static Scope Table, which is loaded into *static scope memory* from the E-machine object file's STATSCOPESECTION, provides the animator with the information relevant to the static nature of a program (e.g., information pertaining to variable names local to a given procedure). However, the specific calling sequence resulting in a particular invocation of a procedure (or function) was not available.

The dynamic scope stack provides the dynamic chain as found in the run time stack activation records generated by most conventional compilers. Even though the E-machine's return address stack could be used to hold this information, a separate dynamic scope stack was added to the E-machine architecture in order to minimize the impact on the existing E-machine and its emulator. At any given point during program execution, the dynamic scope stack entries reflect the currently active scopes. Each dynamic scope stack entry—corresponding to a program name, a procedure name, or a function name—contains the index of the Static Scope Table entry describing that name (i.e., a static scope name). Once these indices are available, the animator can then use the Static Scope Table information to determine the variables whose values must be displayed following the execution of a packet. The animator needs access to the entire dynamic scope stack in order to display all pertinent data memory information following the execution of any given packet. A more detailed discussion of this process is found in chapter 4. The *dynamic scope stack register* points to the top of the dynamic scope stack.

*In order to handle reverse execution, a *save dynamic scope stack* was added to the E-machine architecture. This stack records the history of procedures and/or

functions that have been called and subsequently returned from. The *save dynamic stack register* points to the top of this stack.

Finally, *source memory* holds an array of records, each of which is a copy of a line of source code for the compiled program. Source memory is loaded from the E-machine object file's SOURCESECTION at run time and is referenced only by the animator for display purposes.

E-machine Emulator

The E-machine emulator was designed and written by Michael Birch and is described in his thesis [Birch 90]. The emulator's design essentially follows the design of the E-machine presented the previous sections of this chapter. The emulator was written in ANSI Standard C for portability and has been compiled in both Turbo C 2.0 and Borland C++ 3.1 by the current author. Within the complete DYNALAB environment, the emulator will act as a slave to the program animator, executing a packet of E-code instructions upon each call. The current author has written a simple DOS animator to drive the emulator in order to test compiled miniPascal programs. This animator/emulator has successfully run compiled miniPascal programs on several IBM PC compatible computers including 286, 386, and 486 architectures.

E-machine Object File Sections

The E-machine emulator defines the object file format that must be generated by a compiler. As a result of the miniPascal compiler development, several changes were made to the original E-machine object file definition and are denoted with a

leading asterisk (*) in the following discussion. A single E-code object file ready for execution on the E-machine consists of seven sections, which may occur in any order. Each section is preceded by an object file record containing the section's name followed by a record that contains a count of the number of records in that particular section. Each of these seven sections (whose names are shown in capital letters) holds information which is loaded into a corresponding E-machine component at run time as follows:

- the CODESECTION, which is loaded into program memory;
- the PACKETSECTION, which is loaded into packet memory;
- the VARIABLESECTION, which is loaded into the size information associated with the variable registers;
- the LABELSECTION, which is loaded into the label program address information associated with the label registers;
- the SOURCESECTION, which is loaded into source memory;
- the STATSCOPESECTION, which is loaded into static scope memory;
- the STRINGSECTION, which is loaded into the string space.

The file sections are described below.

The CODESECTION

The CODESECTION contains the translated program—the E-code instruction stream. Even though the instruction stream can be thought of as stream of pseudo assembly language instructions, the instructions are actually contained in an array of C structures, and are loaded from the CODESECTION into the E-machine's program memory at run time. Each E-code instruction structure contains the following information:

- an operation code (e.g., push or pop);
- the instruction mode (critical or noncritical);

- The data type of the operand (e.g., I indicates INTEGER);
- Either a numeric data value or an addressing mode.

*The PACKETSECTION

The PACKETSECTION consists of packet structures describing source program animation units and their translated E-code packets. These structures are loaded into the E-machine's packet memory at run time. Each packet structure contains the following information:

- the packet's starting and ending E-code instruction addresses in program memory;
- the starting and ending line and column numbers in the original source file of the program animation unit corresponding to the packet;
- *an index into the current scope block of the Static Scope Table (discussed in chapter 3);
- *the program memory address at which the packet may be "fragmented" (discussed in chapter 4);
- *a flag indicating whether or not the animator should display information when the packet is executed (discussed in chapter 4).

The VARIABLESECTION

The VARIABLESECTION consists of structures describing the variable registers used by the compiled program. A variable register structure consists of a single field that contains the size of the data represented by the register. For example, on a DOS machine where the addressable unit is a byte, a variable representing a 32-bit integer would have a size of 4. This information is used to initialize size information held in the E-machine's variable registers.

The LABELSECTION

The LABELSECTION consists of label structures describing the label numbers generated by the compiled program. A label structure consists of a single field that contains the program address at which the corresponding label is defined. This information is used to initialize the label program address information held in the E-machine's label registers.

The SOURCESECTION

The SOURCESECTION contains a copy of the source program being executed. Each record in this section corresponds to a line of original source code, and is loaded into the E-machine's source memory at run time. Source memory is referenced only by the animator for display purposes. The animator references source memory via packet memory information that describes correlations between the currently executing E-code packet and the corresponding source program animation unit. The animator references the packet structure fields that hold starting and ending line and column numbers in source memory to determine the animation unit to highlight.

*The STATSCOPESECTION

The STATSCOPESECTION was originally named the SYMBOLSECTION in Birch's thesis. It contains a complex structure—the Static Scope Table (called the symbol table in Birch's thesis)—which is used by the animator to determine the variable values that should be displayed upon execution of a packet. The name was changed to Static Scope Table in order to avoid confusion with the compiler's symbol table. The STATSCOPESECTION records are loaded into the E-machine's static scope memory at run time.

A number of additions and changes were made to the Static Scope Table's structure during miniPascal compiler development. These changes deal primarily with making information available so that the animator can display both the dynamic and static information that are appropriate at various stages of program execution. The Static Scope Table is logically divided into "scope blocks," each of which describes identifiers declared within a single static scope of the source program. A more complete discussion of this section is found in chapters 3 and 4. Each Static Scope Table entry contains the following information:

- the name of the identifier being described (e.g., a variable name or a procedure name);
- upper and lower bounds (for array variables);
- *the index of the Static Scope Table entry containing the next array index bounds (for multidimensional arrays);
- the offset value (for record fields);
- an enumerated value indicating the data type (e.g., INTEGER, RECORD, or STRING);
- *the record size (for arrays of records);
- a pointer to this entry's parent Static Scope Entry;
- a pointer to the child of this entry (e.g., if this static scope entry describes a procedure, this field would hold the index of the first entry in the static scope block describing the variables declared local to the procedure);
- a variable register number (for variable names);
- *a number statically assigned to procedure and functions entries; this number is used in determining the dynamic scoping level at execution time.

*The STRINGSECTION

The STRINGSECTION, which contains the values of string literals and enumerated constant names, was added as a result of miniPascal compiler development. The

contents of the STRINGSECTION are loaded into the E-machine's string space at run time. The string space allows the animator to have dynamic access to the names of an enumerated type as well as the internal numeric values corresponding to the names. The animator can also retrieve the values of string constants from the string space.

CHAPTER 3

E-MACHINE COMPILATION CONSIDERATIONS

Many of the compilation concerns confronting E-machine compiler writers are the same as those faced by writers of compilers for conventional machines. There are, however, several unique factors that must be addressed when compiling for the E-machine's animation environment, including:

- identification and translation of program animation units into E-code packets;
- generation of the Static Scope Table;
- providing access to names associated with enumerated type variables;
- identifying critical and noncritical E-code instructions.

Program Animation Units and E-code Packets

As briefly described in chapter 2, the animation of a high level language program is accomplished by dividing its source code into program "chunks" called *animation units*. The compiler is responsible for isolating a source program's animation units. Each animation unit, in turn, must be translated into a group—or *packet*—of E-code instructions along with corresponding descriptions of the animation unit and its translated E-code packet via a *packet structure*.

When a high level language program is animated, the animator begins execution by displaying the first several lines of the source code and highlighting the first animation unit in the program. The animator then awaits a response from the user. When the user responds, the animator calls the E-machine to execute the currently highlighted animation unit of the program. Actually, what the E-machine executes is the packet of instructions corresponding to the animation unit. When the E-machine has completed execution of the instructions contained in the packet, control is returned to the animator. The animator then performs various animation tasks (e.g., displaying pertinent data memory values) and then again awaits a user response before repeating this process by highlighting the next animation unit and so forth. Thus, two of the challenging tasks facing the compiler designer are identifying animation units and properly translating them into E-code packets for successful animation. The following two sections present an example program to illustrate how the miniPascal compiler accomplishes these two tasks. Although this example program posed no particular problems for the compiler, a number of subtle problems relative to identifying and translating program animation units were encountered during the compiler's development. These problems and their solutions are discussed in detail in chapter 4.

Identifying Program Animation Units

The compiler identifies individual animation units as it is parsing the high level language source code. Consider the miniPascal program in figure 2 (the numbers on the left correspond to line numbers in the source program file). For this program, the miniPascal compiler identifies the nineteen animation units shown in figure 3 (the numbers on the left correspond to each animation unit's associated packet structure, as discussed in the next section). These animation units will be successively highlighted (in the original source program of figure 2) by the

```
 0  Program Samp1;
 1
 2    VAR
 3      I,J,K:INTEGER;
 4      N:INTEGER;
 5
 6    Procedure Init(VAR X,Y:INTEGER);
 7      BEGIN
 8        X := 1;
 9        Y := 2;
10        END;
11
12    BEGIN
13      Init(I,J);
14      IF I < 10
15        THEN K := 100
16        ELSE K := 0;
17      N := K + I*J
18      END.
```

Figure 2: Source Code for Program Samp1

```
 0  Program Samp1;
 1  VAR
 2  I,J,K:INTEGER;
 3  N:INTEGER;
 4  Procedure Init
 5  (VAR X,Y:INTEGER);
 6  BEGIN
 7  X := 1;
 8  Y := 2;
 9  END;
10  BEGIN
11  Init(I,J);
12  IF I < 10
13  THEN
14  K := 100
15  ELSE
16  K := 0
17  N := K + I*J
18  END.
```

Figure 3: Animation Units Identified in Program Samp1

animator as it performs the animation of the program. It should be noted that the determination of animation units is arbitrary and can vary from one compiler to another based on subjective aesthetics of program animation. As can be seen from this example, an animation unit can correspond to "chunks" of source code representing a single keyword, an entire program statement, the conditional part of an if statement, and so forth.

Translating Program Animation Units into E-code Packets

Once the compiler has identified an animation unit, it must then translate this unit into a corresponding packet of E-code instructions along with an associated descriptive packet structure. Thus, compilation of the example given in figure 2, would result in the generation of nineteen E-code packets and nineteen corresponding packet structures. Figure 4 shows the pseudo assembly language representation of the E-code instructions generated for the miniPascal program shown in figure 2. The numbers shown on the left in figure 4 correspond to program memory addresses (instruction numbers). The individual packets, corresponding to the animation units of figure 3, are shown separated by blank lines in figure 4.

Table 1 shows the array of packet structures—called the Packet Table—describing the individual packets resulting from the translation of the program of figure 2. The PacketNumber field (column) is included for clarity—it is not part of the Packet Table. The first two fields in the Packet Table (StartAddr and EndAddr) give the starting and ending addresses in program memory of the E-code packet. The next four fields (StartLine, StartCol, EndLine, and EndCol) demark the physical location of the packet's corresponding program animation unit in the source program array. The ScopeIndex field in the Packet Table is discussed in the next section of this chapter. The final two fields (FragAddr and DisplayPacket) provide

```
 0   pushd C12
 1   nop

 2   nop

 3   inst c,V0
 4   inst c,V1
 5   inst c,V2

 6   inst c,V3
 7   br 0

 8   label L1
 9   pushd C9

10   link V5
11   link V4

12   nop

13   push I,C1
14   pop c,I,V5

15   push I,C2
16   pop c,I,V4

17   nop
18   unlink V4
19   unlink V5
20   popd
21   return

22   label 0
23   nop

24   pusha V1
25   pusha V2
26   call 1

27   label 2
28   label 3
29   inst c,V6
30   push I,V2
31   push I,C10
32   less c,I
33   pop c,B,V6
34   push B,V6
35   brf c,4

36   nop

37   push I,C100
38   pop c,I,V0
39   br 5

40   label 4
41   nop

42   push I,C0
43   pop c,I,V0

44   label L5
45   inst c,V7
46   push I,V2
47   push I,V1
48   mult c,I
49   pop c,I,V7
50   inst c,V8
51   push I,V0
52   push I,V7
53   add c,I
54   pop c,I,V8
55   push I,V8
56   pop c,I,V3

57   nop
58   uninst c,V8
59   uninst c,V7
60   uninst c,V6
61   uninst c,V3
62   uninst c,V0
63   uninst c,V1
64   uninst c,V2
65   popd
```

Figure 4: E-code Instructions Resulting from Compilation of Program Samp1

Packet Number	Start Addr	End Addr	Start Line	Start Col	End Line	End Col	Scope Index	Frag Addr	Display Packet
0	0	1	0	0	0	14	0	-1	TRUE
1	2	2	2	2	2	4	0	-1	TRUE
2	3	5	3	4	3	17	3	-1	TRUE
3	6	7	4	4	4	13	4	-1	TRUE
4	8	9	6	2	6	15	0	-1	TRUE
5	10	11	6	17	6	33	2	-1	TRUE
6	12	12	7	4	7	8	2	-1	TRUE
7	13	14	8	6	8	12	2	-1	TRUE
8	15	16	9	6	9	12	2	-1	TRUE
9	17	21	10	6	10	9	5	-1	TRUE
10	22	23	12	2	12	6	5	-1	TRUE
11	24	26	13	4	13	13	5	-1	TRUE
12	27	35	14	4	14	12	5	-1	TRUE
13	36	36	15	6	15	9	5	-1	TRUE
14	37	39	15	11	15	18	5	-1	TRUE
15	40	41	16	6	16	9	5	-1	TRUE
16	42	43	16	11	16	17	5	-1	TRUE
17	44	56	17	4	17	15	5	-1	TRUE
18	57	65	18	4	18	7	5	-1	TRUE

Table 1: Packet Table Resulting from Compilation of Program Samp1

additional information necessary for animating an animation unit and are discussed in chapter 4.

Generation of the Static Scope Table

The compiler writer must also provide information describing all of the data memory variables that the animator must display. This information is provided in the Static Scope Table, a linear array which is, in turn, logically divided into numerous scope blocks. Each scope block describes the identifiers (e.g., variable names and procedure names) declared in a single static scope in a program. Even though this information is obtained from the compiler's symbol table, the generation of the

Static Scope Table is not a straightforward task due to scope nesting characteristics of many high level languages, such as miniPascal.

Table 2 shows the Static Scope Table that is generated as a result of compiling the miniPascal program given in figure 2. The Entry (entry number) column, or field, is included for clarity—it is not part of the Static Scope Table. This Static Scope Table consists of three scope blocks—a block describing the identifiers declared within the scope of procedure Init (entries 0–3), a block describing the identifiers declared within the scope of program Samp1 (entries 4–10), and a "bootstrap" block describing the main program entry (entries 11-13).

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Off set	Type	Rec Siz	Par ent	Ch ild	Var Reg	Proc Num
Scope block describing procedure Init											
0		-	-	-	-	HEADER	-	4	-	-	-
1	X	-	-	-	-	INTEGER	-	-	-	5	-
2	Y	-	-	-	-	INTEGER	-	-	-	4	-
3		-	-	-	-	END	-	-	-	-	-
Scope block describing program Samp1											
4		-	-	-	-	HEADER	-	11	-	-	-
5	I	-	-	-	-	INTEGER	-	-	-	2	-
6	J	-	-	-	-	INTEGER	-	-	-	1	-
7	K	-	-	-	-	INTEGER	-	-	-	0	-
8	N	-	-	-	-	INTEGER	-	-	-	3	-
9	Init	-	-	-	-	PROCEDURE	-	-	0	-	1
10		-	-	-	-	END	-	-	-	-	-
Bootstrap scope block											
11		-	-	-	-	HEADER	-	-	-	-	-
12	Samp1	-	-	-	-	PROGRAM	-	-	4	-	0
13		-	-	-	-	END	-	-	-	-	-

Table 2: Static Scope Table Resulting from Compilation of Program Samp1

The bootstrap block contains three entries: the HEADER and END entries that delimit the scope block and a PROGRAM entry containing information about the program itself. There are two fields of interest in the PROGRAM entry; these are the

child pointer field (Child) and the procedure number field (ProcNum). The Child field contains the index of the first entry of the scope block describing the identifiers declared in the program. The ProcNum field contains a compiler-generated number that is used in conjunction with dynamic scoping; this field is discussed in chapter 4.

The entries in the scope block describing the identifiers declared in the program scope consist of the HEADER and END delimiter entries as well as entries describing each of the scope's identifiers. The Parent field of the HEADER entry in this scope block contains the index of the first entry of the bootstrap scope block. This scope block's PROCEDURE entry—describing procedure Init—uses the Child field, which contains the index of the first entry of the scope block describing the identifiers declared in procedure Init. The ProcNum field is also used in the PROCEDURE entry; it contains a compiler-generated number to be used in conjunction with dynamic scoping.

The entries in the scope block describing the identifiers declared in procedure Init consist of the HEADER and END delimiter entries as well as entries describing each identifier declared in the scope, in this case the procedure's parameters. The Parent field of the HEADER entry of this scope block contains the index of the first entry of the scope block containing the procedure's declaration.

There must also be some way to relate a high level language program's dynamic nature to the static information found in the Static Scope Table. That is, the animator must be able to determine all of the active scopes at any given point during execution of the program. The animator can then display the data memory values pertinent to the most current scope as well as the data memory values associated with the scopes in the calling sequence leading to the most current scope.

The animator retrieves dynamic scoping information from the E-machine's dynamic scope stack. For instance, suppose that the animator has just highlighted

the animation unit

```
X := 1;
```

in procedure Init. After receiving a response from the user, the animator then calls the E-machine to execute the E-code packet corresponding to this animation unit. When the E-machine returns control to the animator, the animator must then determine the relevant data memory values to be displayed following any changes that resulted from execution of the packet. This task is accomplished by querying the E-machine's dynamic scope stack, which contains a history of the active scopes. In this example, the dynamic scope stack currently consists of two entries, each containing an index into the Static Scope Table. The top entry contains the value 9 and the bottom entry contains the value 12. These values indicate to the animator that procedure Init (Static Scope Table entry number 9) is the most current active scope and that program Samp1 (entry number 12) is the calling scope. By using the child pointers associated with these two Static Scope Table entries, the animator can now determine the appropriate data memory values to be displayed. Figure 5 shows a possible animation resulting from the execution of this animation unit. The arrow (==>) pointing to the instruction `Y := 2;` indicates where animation proceeds.

The ScopeIndex field of the packet structure can now be explained. Suppose that the E-machine has completed execution of the packet corresponding to the animation unit

```
I,J,K:INTEGER;
```

and has returned control to the animator. The animator, via a query of the dynamic scope stack, now determines that only the values of the variables contained in the outer program scope should be displayed. The variables listed in the block describing this scope's variables are `I`, `J`, `K`, and `N`. However, at this point in the program's execution, variable `N` has not yet been declared, and thus should not be displayed. The ScopeIndex field of the packet structure associated with the above animation

```
Program Samp1;

  VAR
    I,J,K:INTEGER;
    N:INTEGER;

  Procedure Init(VAR X,Y:INTEGER);
    BEGIN
      X := 1;
==> Y := 2;
      END;

  BEGIN
    Init(I,J);
    IF I < 10
      THEN K := 100
      ELSE K := 0;
    N := K + I*J
    END.
```

```
   Program Samp1
I = 1
J is undefined
K is undefined
N is undefined
-----------------------
   Procedure Init
X = 1
Y is undefined
```

Figure 5: Animation Display After Execution of `X := 1;`

unit contains the value 3. This value indicates to the animator that it should only display data memory values for entries numbered 0, 1, 2, and 3 in the window associated with the *most current* active scope block. Hence, the animator will display the values of the variables `I`, `J`, and `K` (0 stands for the HEADER entry). In this case, all of these variables would have the value "undefined," as they have only just been declared and have not yet had values assigned to them.

Translating Enumerated Type Variables

Ordinarily, only the internal numeric value of an enumerated type variable is required in translated object code. It is desirable, however, for program animation purposes to have the animator display the enumerated constant name rather than just the internal numeric value of a variable of an enumerated type. Thus, when translating an enumerated type variable, the compiler must provide a way for the

animator to relate the variable's internal numeric value to its corresponding constant name. This task was accomplished by the addition of the string space to the E-machine's architecture. The string space holds the enumerated constant names (as well as string literals) defined in a miniPascal program. The method that the miniPascal compiler uses to relate an enumerated type variable's internal numeric value to the appropriate name in the string space is discussed in chapter 4.

Identifying Critical and Noncritical E-code Instructions

The final major E-machine compilation concern is that of identifying the E-code instructions that would destroy information that is needed (i.e., critical) for successful reverse execution. Since the immediate concern for the miniPascal compiler was to produce a usable compiler, the current version of the compiler treats all E-code instructions as critical. For example, the animation unit

```
N := K + I*J;
```

in figure 2 corresponds to the packet of E-code instructions numbered 44 through 56 in figure 4. All of these instructions are marked critical via the "c" operand. Only instruction number 56 is actually critical, however, as only it results in critical information being destroyed. That is, the old value of N is being destroyed by popping a new value into it in instruction 56; for reverse execution, this old value of N must be saved. Thus, the packet of E-code instructions corresponding to this animation unit could be generated as shown in figure 6, where the operand "n" indicates a noncritical instruction.

```
44    label L5
45    inst n,V7
46    push I,V2
47    push I,V1
48    mult n,I
49    pop n,I,V7
50    inst n,V8
51    push I,V0
52    push I,V7
53    add n,I
54    pop n,I,V8
55    push I,V8
56    pop c,I,V3
```

Figure 6: E-code Instructions Translating `N := K + I*J`

CHAPTER 4

THE DESIGN OF THE miniPASCAL COMPILER

The miniPascal compiler is a one-pass compiler written in ANSI Standard C and developed with Borland C++ 3.1 on an IBM PC compatible computer. E-machine object files (E-code files) generated by the miniPascal compiler have been tested using a simple DOS animator driving the E-machine emulator. Even though the capabilities of this animator are quite limited, a significant number of miniPascal programs have been compiled, executed, and animated successfully.

The miniPascal Language

The miniPascal language is a subset (with a few noted extensions) of ISO Standard Pascal as defined in the book *Pascal User Manual and Report* by Jensen and Wirth [Jensen 91]. The following Pascal features are supported by miniPascal:

- constant, type, and variable declarations;
- procedure and function declarations;
- simple types including integer, real, character, boolean, enumerated types, and subrange types;

- structured types:
  - single and multidimensional arrays,
  - strings, including arrays of strings,
  - fixed-part records including records whose fields are arrays, records, strings, or enumerated types (arrays of records are also supported);
- boolean expressions, unary expressions, and infix expressions;
- assignment statements;
- procedure and function calls;
- control statements:
  - the if-then and if-then-else statements,
  - the while loop,
  - the repeat loop,
  - the for loop,
  - the case statement (with the extension of an others clause).

The following Pascal features are not currently supported in miniPascal:

- records with variant parts;
- the with statement;
- pointers;
- sets;
- labels;
- the goto statement;
- external files;
- the forward directive;
- predeclared functions and procedures;
- procedure or function names as parameters;
- conformant-array parameters.

Overview of the miniPascal Compiler

The miniPascal compiler was developed using the lex and yacc compiler development tools [Mason 90]. Lex is a scanner generator written by M.E. Lesk and E. Schmidt of Bell Laboratories [Lesk 75] and yacc is a parser generator written by S.C. Johnson, also of Bell Laboratories [Johnson 75]. Lex reads a specification file of regular expressions identifying the tokens in a language and generates a C module containing a scanner for those tokens. Yacc reads a specification file containing a context-free grammar (and associated semantic actions) for a language and produces a C module containing an LALR(1) parser for the language. The basic lex and yacc specifications for ISO Standard Pascal were obtained from the ftp network site primost.cs.wisc.edu. The semantic stack definition and semantic actions were then added to these specifications.

Both lex and yacc are standard utilities available on Unix machines. Even though there are versions of these utilities available for DOS machines, the lex and yacc specifications for miniPascal have been run exclusively on a Unix machine, with the resulting C modules being downloaded to a DOS machine. These C modules were then compiled and linked with numerous other C modules containing the semantic analysis and code generation routines.

The compiler consists of a total of sixteen modules. Figure 7 is a schematic diagram showing the interactions among the various modules—the directions of the arrows indicate calls to a module. Three of the sixteen modules are omitted from the figure for the sake of clarity. These are the Error Message module, the Memory Allocation module, and the module that produces a text file containing the pseudo assembly language instructions translating the source program (used for compiler debugging purposes). A brief description of the compiler operation is given below.

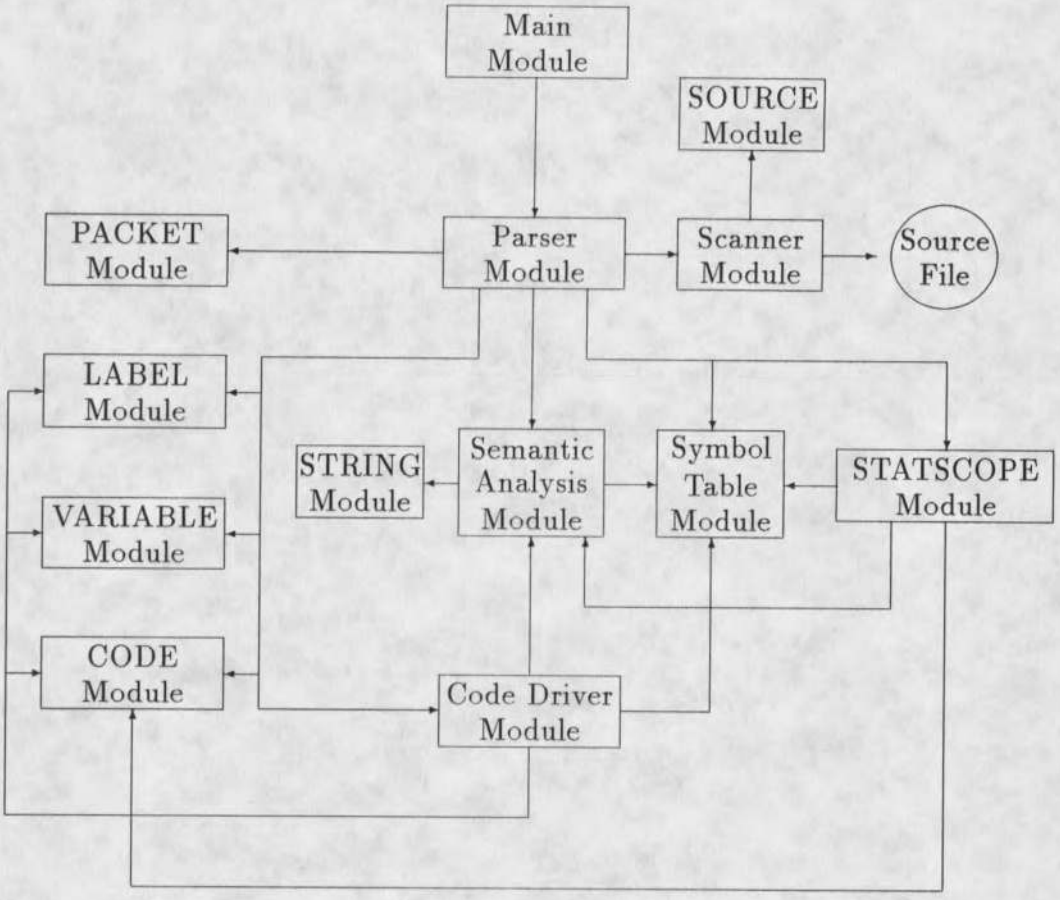

Figure 7: Schematic Diagram of the miniPascal Compiler

After the Main module opens appropriate files, it calls the Parser module, which drives the compilation process by requesting tokens from the Scanner module and by calling various semantic analysis and code generation routines, notably the Semantic Analysis module and the Code Driver module. As can be seen in figure 7, the Symbol Table module plays a central role during semantic analysis and code generation.

Seven of the modules are dedicated to producing the E-code object file. These modules are:

- the PACKET module, which produces the PACKETSECTION;
- the LABEL module, which produces the LABELSECTION;
- the VARIABLE module, which produces the VARIABLESECTION;

- the CODE module, which produces the CODESECTION;
- the STRING module, which produces the STRINGSECTION;
- the SOURCE module, which produces the SOURCESECTION;
- the STATSCOPE module, which produces the STATSCOPESECTION.

When compilation is complete, control is returned to the Main module, which then calls routines in each of these seven E-code production modules in order to generate the final E-code file (these calls are not indicated in figure 7). If the compiler encounters an error during compilation, a call is made to the Error module (omitted from figure 7), which prints an error message and then calls a routine in the Main module for immediate termination of compilation.

Error Detection and Recovery

When the compiler detects an error in a miniPascal source file, an appropriate message is printed and the compilation is halted. The initial users of this compiler will be instructors preparing laboratory exercises—not students developing programs. Thus, minimal error reporting with no recovery was considered to be sufficient.

Optimization

There are no provisions for optimization in the compiler. There is no real need for optimization in the animation environment, and many optimizations would alter the E-code/source language relationship too severely for animation to be successful.

The Compiler Modules

The remainder of this chapter describes the individual compiler modules in more detail. The discussion is focused on the role each module plays in the generation of the seven sections of the E-code file, giving particular attention to those sections that presented problems unique to this compiler. The E-code's CODESECTION is essentially the equivalent of the intermediate code files generated by many compilers; the problems encountered in generating this section were the same as would be found in the development of any compiler. The four sections, VARIABLESECTION, LABELSECTION, SOURCESECTION, and STRINGSECTION, are unique to the E-machine; they, however, posed no particular problems and are generated in a straightforward manner. The PACKETSECTION, also unique to the E-machine, did present some problems, which are discussed below in the Parser Module description. The problems presented by the STATSCOPESECTION are discussed in the STATSCOPE Module description. Another E-code generation problem occurred due to the desire to have the animator display an enumerated type variable's constant name as well as its internal numeric value. The solution to this problem was to add the STRINGSECTION to the E-machine object file as discussed in the STRING Module description.

The Main Module

When the miniPascal compiler is invoked, control passes to the main routine in the Main module. The Main module consists of the main routine, routines that handle the opening and closing of files, and a routine to handle abnormal end of compilation. The Main module opens the miniPascal source file, whose name is

obtained from a command line argument when the compiler is invoked. The Main module then creates three files to hold the output from the compilation: the E-code (object) file, a file to hold the pseudo assembly language instruction stream (for compiler debugging), and a temporary file to hold output from the module producing the CODESECTION of the E-code file. Next, the Main module calls the yacc-generated yyparse routine in the Parser module to begin the compilation. When yyparse returns successfully to the Main module, the compilation is complete, and the Main module then calls routines in the code producing modules to write the various E-code sections to the E-code file. Finally, the files are closed and the compiler exits normally. If a return marking an unsuccessful compilation is made to the main Module, the miniPascal source file is closed, the output files are deleted, and an abnormal exit is indicated.

The Parser Module

As indicated above, the yacc-generated Parser module is responsible for driving the compilation process. Yacc produces an LALR(1) parser by processing a specification file containing a context free grammar that generates the source language. Calls to semantic routines, written in C, are interspersed among the grammar production rules given in the miniPascal yacc specification. The yacc-generated parser maintains a parser-controlled semantic stack, whose records hold information corresponding to each token and non-terminal found in the grammar productions. The parser has access to the information in the semantic stack records via pseudo variables used in the semantic actions. The yacc specification provides a union structure to define the different types of semantic stack records necessary to describe the various semantic information required for each symbol in the grammar. In the case of the miniPascal specification, this union structure consists primarily of pointers to dynamically allocated structures containing information needed to produce the

E-code for the animation of a miniPascal program. The yacc-generated Parser module consists of one very large routine, yyparse. Two small user-supplied supporting routines are also included in this module.

Calls to the Scanner. As in conventional compilers, the Parser module requests the next token from the Scanner module by calling the lex-generated yylex routine. The Parser module has access to the value of a token through the external variable yytext, whose value is produced in the Scanner module. Since the miniPascal language is not case-sensitive, the Scanner converts all letters in an identifier name token to lower case before returning the token (and its value) to the Parser module. The numeric values of integer and real literal tokens are available to the Parser module via the external variable yylval, also produced in the Scanner module.

Interface to the Symbol Table. The Parser module interfaces directly with the Symbol Table module to enter and retrieve identifier names. The Parser module enters and retrieves an identifier's symbol table attributes by calling routines in the Semantic Analysis module.

Initiating Semantic Actions. Many of the semantic actions initiated by the Parser module are accomplished by calls to routines in the Code Driver module. These routines perform further semantic analysis (via calls to routines in the Semantic Analysis module) and then generate code (via calls to the code production routines). For example, when the Parser module recognizes an assignment production, it calls the GenAssign routine in the Code Driver module. The parameters passed to GenAssign are pointers to the semantic stack structures corresponding to the symbols involved in the assignment production rule of the grammar. GenAssign can then determine whether the assignment is valid, determine what value (if any) to load into the index register, and generate appropriate E-code by calling routines in

the code production modules. There are also situations in which the Parser module itself can cause E-code generation directly by calling code production routines.

Providing for Dynamic Scoping. The Parser module provides dynamic scoping information to the E-machine by generating code to manipulate the E-machine's dynamic scope stack. (Static scoping information is contained in the Static Scope Table and is discussed in detail in the STATSCOPE module description.) When the Parser module encounters the beginning of a program, function, or procedure scope, it calls the GenInstr routine in the CODE module to generate the **pushd** instruction. At run time, the **pushd** instruction causes an entry to be pushed onto the E-machine's dynamic scope stack. This entry contains the index of the scope's declaration (e.g., procedure name description) in the Static Scope Table; this index must be passed as an operand in the **pushd** instruction. (Recall from the discussion of the dynamic scope stack in chapter 2 that this is necessary in order that identifiers in calling scopes be accessible at run time.) At this point in the parse, however, this index value is not known because the scope's declaration is "owned" by the containing scope, whose Static Scope Table entries will not be generated until that entire scope has been parsed. This means that the Parser must associate a dummy index value with the **pushd** instruction, and the instruction must be "patched" when the actual value becomes available. When the Parser module encounters the end of a scope, it generates the **popd** instruction and then calls the GenStatScopeBlock routine in the STATSCOPE module to generate the Static Scope Table entries for the scope. When the Parser module finally encounters the end of a containing scope, the STATSCOPE module can calculate the index of any nested procedure or function scope declarations and patch their corresponding **pushd** instructions via a call to the CODE module.

Translating Animation Units into Packets. The Parser module controls the identification and subsequent translation of a miniPascal program's animation units. This translation involves the generation of a packet of E-code instructions (via calls to the CODE and Code Driver modules) as well as the construction of an associated packet structure describing the animation unit. The Parser calls the following routines in the PACKET module to construct a packet structure:

- StartPacket;
- EndPacket;
- AdjustStartPacket;
- AdjustEndPacket;
- AddPktFragInstr;

A packet structure's delimiter values—pertaining to the source file line and column number boundaries of the animation unit that the packet translates, as well as the starting and ending program memory addresses of the E-code packet itself—are determined by the Parser module, which then passes these values to the appropriate PACKET module routine. The Parser module has access to the source file line and column number values via external variables that are initialized when the Scanner module recognizes a token; the Parser has access to the current program memory address (instruction number) via an external variable maintained by the CODE module as E-code instructions are generated. When the Parser module recognizes a token that marks the beginning of an animation unit, it calls the StartPacket routine, passing as parameters the source file line and column numbers corresponding to the first character of this token as well as (in the general case) the number of the *next* E-code instruction to be generated. The PACKET module maintains an internal variable, PktNum, containing the number of the packet structure currently under construction. This variable, which serves as the index into the PACKET module's

array of packet structures, is incremented in the StartPacket routine. Subsequent calls to any of the other PACKET module routines listed above refer to the previously "started" packet structure. Thus, for a given animation unit, StartPacket is called only once, while the remaining routines (including EndPacket) may be called any number of times while the animation unit's packet structure is being constructed.

In the simplest case, upon recognizing a token that marks the beginning of an animation unit, the Parser module first calls the StartPacket routine and then generates any corresponding E-code instructions. As other tokens within the animation unit are recognized, the Parser continues to generate E-code instructions. When the Parser recognizes the token marking the end of the animation unit, it calls the EndPacket routine, passing as parameters the source file line and column numbers corresponding to the last character of this token as well as the number of the current E-code instruction. For example, the `BEGIN` keyword is considered to be a complete packet. Thus, upon recognizing the `BEGIN` token, the Parser first calls the StartPacket routine, passing the source file line and column numbers associated with the letter B as well as the number of the next E-code instruction to be generated. Next, the Parser generates the E-code nop instruction (via a call to the CODE module). Finally, the Parser calls the EndPacket routine, passing the source file line and column numbers associated with the letter N as well as the current E-code instruction number (corresponding to the nop instruction). There are many cases in which an animation unit's delimiters can be determined in such a straightforward manner; there were, however, a number of subtle animation unit translation problems encountered during compiler development.

The Lookahead Problem in Animation Unit Translation. One of these problems occurs when the Parser module is assigning source file line and column

numbers to an animation unit. For those tokens that delimit an animation unit, the parser calls either StartPacket or EndPacket, passing the token's line and column numbers and the appropriate instruction number. The values in the external variables containing the line and column numbers, however, are incorrect when the Parser module must examine the lookahead token in order to determine which production to reduce. (Recall that yacc produces an LALR(1) parser that sometimes requires a one-symbol lookahead for proper parsing actions.) In these cases, the current line and column numbers reflect the location of the lookahead token instead of the token delineating the animation unit. This problem was solved by identifying the tokens that were involved in these situations and replacing them with non-terminals on the right-hand sides of productions. Each new non-terminal becomes the left-hand side of a unit production whose right-hand side is just the token the non-terminal replaced. During reduction of one of these new unit productions, the token's line and column numbers can be captured and placed in the semantic record belonging to the token because no lookahead is required to reduce these unit productions. Yacc places this record on the semantic stack, which allows access by routines processing the stack. Thus, when a production that has one of the new non-terminals on its right-hand side is reduced, the correct values can be retrieved from the semantic stack and passed to the PACKET module routines.

The Semicolon Problem in Animation Unit Translation. Another problem was easily solved by calling the EndPacket routine more than once for the same animation unit. This situation can be illustrated by Pascal's use of semicolons to separate, rather than terminate, statements (recall that the semicolon is not part of a Pascal statement [Jensen 91]). For example, in the Pascal code fragment shown in figure 8, the semicolon at the end of `J := 2;` is unnecessary. It really separates `J := 2` from a null statement, and the null statement precedes the `END` statement.

However, the statement (including the semicolon)

```
J := 2;
```

is considered to be an animation unit. The yacc production associated with assignment statements however, does not immediately associate the semicolon with the statement (e.g., `J := 2`). Rather, an enclosing production eventually accomplishes this association. Thus, in this case, the Parser first issues a call to EndPacket, passing the source file line and column numbers associated with the token "2", as well as the number of the final instruction within the E-code translation of the animation unit. Then, when the Parser reduces the enclosing production recognizing the semicolon, the Parser again calls the EndPacket routine, passing the source file line and column numbers associated with the token ";", as well as the current instruction number (this number will not have changed since no code is generated for the semicolon). Hence, the animation unit is "ended" correctly. As noted for this case, since the statement precedes an `END` statement, the semicolon is optional. If the semicolon is omitted, the Parser performs correctly by calling the EndPacket routine only once because the enclosing production in this case has no semicolon.

```
.
.
.
BEGIN
  I := 1;
  J := 2;
  END;
.
.
.
```

Figure 8: Code Fragment Illustrating the Semicolon Problem

Adjusting an Animation Unit's Ending Delimiter. There are instances when an animation unit's ending delimiters must be adjusted after the Parser has

already ended construction of its associated packet structure. For example, if there are procedure declarations following a scope's variable declarations, the Parser must generate an instruction to branch around the code translating the nested procedures in order to achieve the correct program flow. During program execution, the animator will need to highlight the animation units corresponding to the variable declarations in this routine, then skip the procedure declarations and continue by highlighting the body of this routine, demonstrating execution flow. In such cases, the Parser has already (correctly) ended the construction of the current packet structure—describing the animation unit consisting of the last variable declaration in the scope—prior to reaching the first procedure declaration. The branch instruction number, however, must now be included in the current packet structure as the ending program memory address of the corresponding E-code instruction packet to ensure proper animation around the procedure declarations. In this situation, the packet structure cannot simply be "ended" again, because the current source line and column numbers now reflect the location of the beginning of the animation unit corresponding to the subsequent procedure declaration. Hence, the AdjustEndPacket routine was designed to alter the ending program memory address associated with the current packet structure. For the example given above, the Parser calls the AdjustEndPacket routine, passing as a parameter the E-code instruction number associated with the branch instruction. The Parser then continues by calling the StartPacket routine to begin construction of the packet structure describing the procedure declaration animation unit.

Adjusting an Animation Unit's Beginning Delimiter. The routine, AdjustStartpacket, was provided to support the situation in which an animation unit is nested within another animation unit. This situation occurs when there is a function call within another miniPascal statement. For example, consider the

following statement:

```
Result := Min(x,y) + 2;
```

Upon recognizing the assignment production, the Parser issues a call to the StartPacket routine to begin construction of the packet structure describing the animation unit consisting of the entire statement. For animation purposes, however, it is desirable to highlight the function call, `Min(x,y)`, separately in order to illustrate the fact that the function Min must be called before the assignment statement can be completed (i.e., `Min(x,y)` should be treated as a separate animation unit). Thus, when the Parser recognizes the function call, it issues a call to the AdjustStartPacket routine, passing the source line and column numbers associated with the beginning of the function call. The AdjustStartPacket routine returns (via parameters) the previous source line and column numbers associated with the original StartPacket call for the current packet structure. The Parser then continues to control the construction of the current packet structure, which now describes only the `Min(x,y)` portion of the statement, generating an E-code instruction packet translating `Min(x,y)`. When the Parser completes the processing of the function call production, it calls the EndPacket routine to end the current packet structure and then calls the StartPacket routine to start construction of the next packet structure. The line and column numbers passed to this call to StartPacket are the previous source line and column numbers returned by the AdjustStartPacket routine; the instruction number of the *next* E-code instruction is also passed to StartPacket. Thus, the source code associated with the now current packet structure is the entire assignment statement; the E-code packet translating this assignment statement does not include the instructions translating the function call.

Adjusting the Starting Memory Address of a Packet. There is a case when it is necessary to retain the value of the current E-code instruction number so

that it can be used as the starting program memory address of the packet translating the next animation unit. Normally, a packet's starting program memory address is the next instruction number to be generated. However, when an E-code `label` instruction is the current instruction, there are situations when this instruction must become the first instruction in the next E-code packet. Unfortunately, since it is not an enclosing production that needs the memory address of the label instruction, the semantic stack cannot be used to store this value. This problem is solved by storing the current instruction number (i.e., the number of the E-code `label` instruction) in a global variable, SaveStartInstrNum, which is accessible in the appropriate production. Thus, whenever the Parser recognizes a token that marks the beginning of an animation unit, it first queries SaveStartInstrNum for a valid instruction number. If the instruction number is valid (i.e., its value is not -1), the Parser passes this instruction number value to the StartPacket routine and then sets SaveStartInstrNum's value to -1. If the value of SaveStartInstrNum is -1, the Parser passes the number of the next E-code instruction to be generated to StartPacket. It should be noted that use of global variables in the yacc parser must be done very carefully to ensure that the parser does not alter a variable's value before it is used.

Adjusting the Ending Memory Address of a Packet. Similarly, there is also a case when it is necessary to retain the value of the current E-code instruction number so that it can be used as the ending program memory address of the packet currently under construction. This situation arises when the Parser generates an E-code `br` (branch) or `call` instruction immediately preceding the generation of a `label` instruction. Due to the nature of the parse, however, the Parser has not yet determined that it is time to end construction of the animation unit corresponding to the current packet. This decision is made when the next production is processed. This next production is not an enclosing production, and thus cannot retrieve the

necessary information from the semantic stack. Here again, a global variable is used. The Parser queries this variable, SaveEndInstrNum, before calling the EndPacket routine. (This is the same situation in which the succeeding `label` instruction must become the starting program address for the next animation unit.)

Fragmented Animation Units. Another problem in translating animation units occurs when an animation unit becomes "fragmented". Fragments result when parsing either a single conditional statement or a single procedure/function call that occurs within another conditional statement alone, not within a compound (BEGIN/END) statement. This situation is best explained by an example. Consider the miniPascal program in figure 9 (the numbers on the left correspond to line numbers in the source file). This example illustrates a single procedure call statement that occurs within a while loop (line number 11). Figure 10 shows the pseudo assembly language representation of the E-code instructions translating program Increment1. The numbers shown on the left correspond to program memory addresses (instruction numbers). The individual packets are separated by a blank line in the figure.

First, assume that the animator has two options pertaining to when an animation unit should be highlighted. One of these options is to highlight an animation unit, await a response from the user before executing the corresponding E-code packet (so that the user can contemplate what will happen when the animation unit is executed), and then rehighlight the same animation unit upon completion of its execution (so that the user can ponder where execution will proceed next). This is the scenario that has been used in previous examples. However, as the user progresses in his understanding of program flow, it would also be desirable to give the animator a second option. This option would allow the animator, upon completion of the execution of an animation unit, to immediately highlight the *next* animation

```
 0  PROGRAM Increment1;
 1    VAR
 2      i:INTEGER;
 3
 4    PROCEDURE IncrI;
 5      BEGIN
 6       i:=i+1
 7       END;
 8
 9    BEGIN
10     i:=1;
11     WHILE i < 5 DO IncrI;
12     END.
```

Figure 9: Source Code for Program Increment1

```
 0   pushd C7
 1   nop

 2   nop

 3   inst c,V0
 4   br 0

 5   label 1
 6   pushd C4

 7   nop

 8   inst c,V1
 9   push I,V0
10   push I,C1
11   add c,I
12   pop c,I,V1
13   push I,V1
14   pop c,I,V0

15   nop
16   uninst c,V1
17   popd
18   return

19   label 0
20   nop

21   push I,C1
22   pop c,I,V0

23   label 2
24   inst c,V2
25   push I,V0
26   push I,C5
27   less c,I
28   pop c,B,V2
29   push B,V2
30   brf c,3

31   nop

32   call 1
33   label 5
34   br 2

35   label 3
36   nop
37   uninst c,V2
38   uninst c,V0
39   popd
```

Figure 10: E-code Translation of Program Increment1

unit to be executed. The following discussion assumes that the animator is running under this second option.

Suppose that the animator has just highlighted the animation unit `END;` of procedure IncrI. Upon receiving a response from the user, the animator calls the E-machine to execute the corresponding E-code packet (consisting of the instructions numbered 15 through 18). The E-machine returns to the animator when it completes execution of the packet. The animator now queries the E-machine's packet register in order to determine the *next* animation unit to be highlighted. Although it is not evident from figure 10, the `RETURN` instruction (number 18) causes control to pass to the `LABEL` instruction (number 33) following the `CALL` instruction that caused control to pass to procedure IncrI. (This is accomplished via the E-machine's query of its return address stack, as discussed in chapter 2.) Instruction number 33 is within the E-code packet translating the animation unit consisting of the call to procedure IncrI (instruction numbers 32 through 34). Thus, the E-machine's packet register contains the address of the packet structure describing the animation unit consisting of the call to procedure IncrI. This animation unit, however, was highlighted prior to the call to the procedure and should not be rehighlighted. The animation unit that should be highlighted is `WHILE i < 5`.

This fragmentation problem was solved by adding a new field to the packet structure definition. Table 3 shows the Packet Table containing the packet structures resulting from the compilation of program Increment1. This new field, named FragAddr in table 3, holds the first program memory address at which this fragmentation can occur. (More than one such `LABEL` instruction within the E-code packet can cause this problem to occur multiple times due to multiple nesting possibilities.) When the Parser module determines that this situation is possible, it calls the AddPktFragInstr routine to initialize the FragAddr field. The animator must now query the E-machine's program counter as well as its packet register when

determining the next animation unit to be displayed. If the program counter value is greater than or equal to the FragAddr value in the packet structure corresponding to the packet register, the animator does not change its current display (i.e., it continues to highlight the animation unit it is on, not the animation unit described by the packet structure to which the return was made). Of course, the animator must still call the E-machine to complete execution of the fragmented E-code packet even though there is no change in what the animator highlights. Once execution of the fragmented packet is completed, the next animation unit is highlighted, in this case `WHILE i < 5`.

Packet Number	Start Addr	End Addr	Start Line	Start Col	End Line	End Col	Scope Index	Frag Addr	Display Packet
0	0	1	0	0	0	18	0	-1	TRUE
1	2	2	1	2	1	4	0	-1	TRUE
2	3	4	2	4	2	13	1	-1	TRUE
3	5	6	4	2	4	22	0	-1	TRUE
4	7	7	5	4	5	8	0	-1	TRUE
5	8	14	6	5	6	10	0	-1	TRUE
6	15	18	7	5	7	8	2	-1	TRUE
7	19	20	9	2	9	6	2	-1	TRUE
8	21	22	10	3	10	7	2	-1	TRUE
9	23	30	11	3	11	13	2	-1	TRUE
10	31	31	11	15	11	16	2	-1	TRUE
11	32	34	11	18	11	28	2	33	TRUE
12	35	39	12	3	12	6	2	-1	TRUE

Table 3: Packet Table Resulting from Compilation of Program Increment1

It should be noted that the miniPascal code

```
WHILE i < 0 DO BEGIN IncrI(i) END;
```

does not produce a fragmentation problem, because the call to IncrI is contained in a compound statement (BEGIN/END) pair. Figure 11 contains the source code illustrating this situation. Figure 12 shows the pseudo assembly language

```
 0     PROGRAM Increment2
 1       VAR
 2         i:INTEGER;
 3
 4       PROCEDURE IncrI;
 5         BEGIN
 6          i:=i+1
 7          END;
 8
 9       BEGIN
10        i:=1;
11        WHILE i < 5 DO BEGIN IncrI END;
12        END.
```

Figure 11: Source Code for Program Increment2

```
 0   pushd C7
 1   nop

 2   nop

 3   inst c,V0
 4   br 0

 5   label 1
 6   pushd C4

 7   nop

 8   inst c,V1
 9   push I,V0
10   push I,C1
11   add c,I
12   pop c,I,V1
13   push I,V1
14   pop c,I,V0

15   nop
16   uninst c,V1
17   popd
18   return

19   label 0
20   nop

21   push I,C1
22   pop c,I,V0

23   label 2
24   inst c,V2
25   push I,V0
26   push I,C5
27   less c,I
28   pop c,B,V2
29   push B,V2
30   brf c,3

31   nop

32   nop

33   call 1

34   label 5
35   nop
36   br 2

37   label 3
38   nop
39   uninst c,V2
40   uninst c,V0
41   popd
```

Figure 12: E-code Translation of Program Increment2

representation of the E-code instructions translating program Increment2. As can be seen in figure 12, in this case the `LABEL` instruction following the `CALL` instruction is not within the E-code packet that contains the `CALL` instruction. This `LABEL` instruction is the first instruction in the E-code packet translating the `END` statement associated with the while loop. Thus, upon completion of the execution of the E-code packet translating the procedure's `END` statement, the animator would (correctly) highlight the animation unit containing the `END` statement of the while loop. The fact that the `LABEL` instruction is physically adjacent to an instruction involved in the translation of the next animation unit (in this case, the while loop's `END` statement) allows the Parser to "adjust" the E-code packet boundaries by querying the SaveStartInstrNum and SaveEndInstrNum variables as previously discussed.

The sample program found in appendix C illustrates another situation in which a packet becomes fragmented.

To Highlight or Not. A final field, named DisplayPacket in table 3, was added to the packet structure to indicate whether or not the animator should display (highlight) anything at all before the corresponding E-code packet is executed. There are two miniPascal situations when the animator should not change its display before execution of a packet:

- before execution of a packet associated with the return from a function call;
- before execution of a packet containing instructions that determine the case label to which a branch should be made based on the case selector value.

The following two examples illustrate these situations. First, consider the miniPascal statement,

```
Result := Min(x,y) + 2;
```

In this example, the animator will eventually highlight the animation unit `Min(x,y)` and then await a response from the user before executing the corresponding E-code packet. When execution of `Min` is complete and control returns to the calling

procedure, a "dummy" packet containing the E-code to pop the value returned by `Min` into a temporary variable is executed. Since `Min(x,y)` has already been highlighted and its corresponding E-code packet has been executed, there is no corresponding source code (i.e., animation unit) associated with this dummy packet. This situation is similar to the fragmentation problem discussed above. In this case, however, execution of the entire dummy packet should not result in any animation (highlighting) of the source program. Thus, the DisplayPacket field in the dummy packet's associated packet structure is set to FALSE. The animator would continue to highlight `Min(x,y)` until execution of the dummy packet is complete, and then highlight the animation unit containing the statement `Result := Min(x,y) + 2;` in order to illustrate the result of executing this assignment statement.

Now, consider the code in figure 13. In this example, upon completion of parse of the entire case statement (up to the `END` statement), the Parser Module calls the GenCaseSearch routine in the Code Driver module. This routine generates a packet of E-code that enables control to pass to the proper case label at run time. Here again, this is a "dummy" packet in that there is no animation unit associated directly with it. For the case statement in figure 13, the animator will first highlight the animation unit, `CASE i OF`, and then await a response from the user. Upon completion of execution of the corresponding E-code packet, the animator will subsequently call the E-machine to execute the dummy packet, without changing its display (i.e., `CASE i OF` continues to be highlighted). Then, since the value of the case selector is 2, the animator highlights the animation unit containing the case label `2:` and again awaits a response from the user.

The Scanner Module

The Scanner module performs the scanning (or lexical analysis) function for the compiler. This module consists of the lex-generated yylex routine and two

```
.
.
.
i:=2;
CASE i OF
  1:       j:=100;
  2:       j:=200;
  3:       j:=300;
  OTHERS:  j:=0
  END;
.
.
.
```

Figure 13: Source Code for a CASE Statement

user-supplied routines that handle miniPascal comments and quoted strings, respectively. The yylex routine is called by the Parser when the next token is required. The other two routines are called internally (from yylex in the Scanner module).

Lex produces a scanner by processing a specification file containing rules that consist of regular expressions. These regular expressions define the tokens in a language, in this case miniPascal. Actions, written in C, are interspersed among the rules—these actions effect the accomplishment of the two scanner tasks performed upon each call to yylex. The Scanner module's first task is to recognize and return miniPascal tokens (and their values) to the parser. Its second task is to enter the original miniPascal source code into the E-code's SOURCESECTION via calls the GenSource routine in the SOURCE module. The Scanner module is also responsible for ensuring that the miniPascal compiler is not case-sensitive. Thus, when the Scanner module recognizes a identifier token, it first enters the name of the identifier into the SOURCESECTION, and then converts the name (in the yytext variable) to all lower case characters before returning to the Parser.

The Code Driver Module

The Code Driver module drives the E-code translation of the source program. This module is a large one, consisting of thirty-two routines. Many of these routines are called by the Parser module when the parse reaches a point where code should be generated. The remaining routines in this module are called internally (from within the Code Driver module). The Code Driver module interfaces directly with the Semantic Analysis and Symbol Table modules to perform semantic analysis, and with the CODE, LABEL, and VARIABLE modules to perform code generation.

The Semantic Analysis Module

The Semantic Analysis module performs semantic analysis during compilation. This module is a large module, consisting of fifty-eight routines. These routines are called by the Parser module, the Code Driver module, and the STATSCOPE modules when semantic checking must be done. The Semantic Analysis routines may also be called internally (from within the Semantic Analysis module). These routines perform tasks relevant to both the initialization and the retrieval of semantic information. For example, the Parser module calls the SetProcAttributes routine upon encountering a procedure declaration. This routine is dedicated to associating with the procedure name its (compiler generated) starting program memory address as well as any formal parameter attributes. Later, when the Parser encounters a call to the procedure, it calls the GetProcAttributes routine to retrieve this information in order to associate the correct program memory address with the generated E-code `call` instruction and to verify the actual parameter list. The Semantic Analysis module interfaces directly with the Symbol Table module to enter and retrieve symbol table attribute information. The Semantic Analysis module also interfaces directly with the STRING module by calling the EnterString routine to enter

the values of string literals and enumerated constant names into the string space array.

The PACKET Module

The PACKET module produces the E-code packet descriptors during compilation. This module contains routines that initialize a statically allocated array of structures containing the packet descriptions. With the exception of the WritePKT routine, the PACKET module routines are called by the Parser module during the parsing process. The WritePKT routine, called by the Main module at the end of compilation, writes the packet structure array elements to records in the PACKETSECTION of the E-code file.

The SOURCE Module

The SOURCE module produces the source code array (for animation purposes). This module contains the GenSource routine, which initializes a statically allocated array containing the source code of the miniPascal program being compiled. Each element in the source code array corresponds to a single line in the miniPascal source program. The GenSource routine is called by the Scanner module during the scanning process. The WriteSOURCE routine, called by the Main module at the end of compilation, writes the source code array elements to records in the SOURCESECTION of the E-code file.

The LABEL Module

The LABEL module produces the table that maps E-code label numbers to their corresponding E-code instruction numbers (i.e., E-code `label` instructions). This module contains the GenLabRegTable routine, which initializes a statically allocated array whose elements contain the instruction number of corresponding

E-code (`label`) instructions. The GenLabRegTable routine is called by both the Parser and the Code Driver modules during the compilation process. The WriteLAB routine, called by the Main module at the end of compilation, writes the label array elements to records in the LABELSECTION of the E-code file.

The VARIABLE Module

The VARIABLE module produces the table that maps E-code variable register numbers to their corresponding data memory sizes. This module contains the GenVarRegTable routine, which initializes a statically allocated array whose elements contain the size of the data memory reserved for corresponding variable register numbers. The GenVarRegTable routine is called by both the Parser and the Code Driver modules during the compilation process. The WriteVAR routine, called by the Main module at the end of compilation, writes the variable register array elements to records in the VARIABLESECTION of the E-code file.

The STRING Module

The STRING module generates the string array found in the E-code STRINGSECTION. The miniPascal compiler's implementation of enumerated types precipitated the need for a new E-machine component (the string space), and hence the need for a corresponding section in the E-code file. This new section is called the STRINGSECTION. Ordinarily, only the internal numeric value of an enumerated type variable is required in translated object code for real computers and computing environments. It is desirable, however, for a program animation system to have the animator display the enumerated constant name rather than (or in addition to) the internal numeric value of a variable of an enumerated type. The STRINGSECTION consists of a statically allocated character array containing all of the enumerated constant names defined in a miniPascal program, as well as the values of any string

literals declared in the source program (which may also need to be displayed by the animator). When the Semantic Analysis module encounters the definition of a string literal or an enumerated constant name, it calls the EnterString routine in the STRING module. The WriteSTRINGS routine, called by the Main module at the end of compilation, writes the string character array to the STRINGSECTION of the E-code file.

When a miniPascal program is animated, the STRINGSECTION portion of the E-code file is loaded into the E-machine's string space. The string space is then accessed by the animator for displaying string constants and enumerated variable values. For example, upon completion of execution of the program in figure 14, the animator can display the enumerated type variable values as shown in figure 15.

Figure 16 illustrates the relationship of the E-machine's string space with the variable registers and data memory. This illustration assumes that a variable register associated with an enumerated type variable represents 32-bits of data memory. The 16 high-order bits of this data memory location contain the dynamically determined internal numeric value of the enumerated constant associated with this variable; the 16 low-order bits contain an index into the string space where the associated enumerated constant name can be found. As can be seen in figure 16, the index into the string space is always that of the first constant name of the enumerated type. This is due to the fact that the compiler can statically generate code to increment or decrement the numeric value of an enumerated type variable (e.g., for an enumerated type control variable in a for loop). The compiler cannot, however, statically determine in advance the absolute string space index of the enumerated constant name associated with an enumerated type variable at any given time. Instead, the animator has the capability to retrieve the variable's numeric value and the starting string space index. The animator can then step sequentially through the string space until the name corresponding to the numeric value is found; the

```
Program Payroll1;

  TYPE
    DAYS = (MON,TUES,WED,THURS,FRI);
    FREQUENCY = (WEEK,MONTH);

  VAR
    OffDay,PayDay:DAYS;
    PayFreq:FREQUENCY;

  BEGIN
    OffDay:=WED;
    PayDay:=FRI;
    PayFreq:=WEEK;
    END.
```

Figure 14: Source Code for Program Payroll1

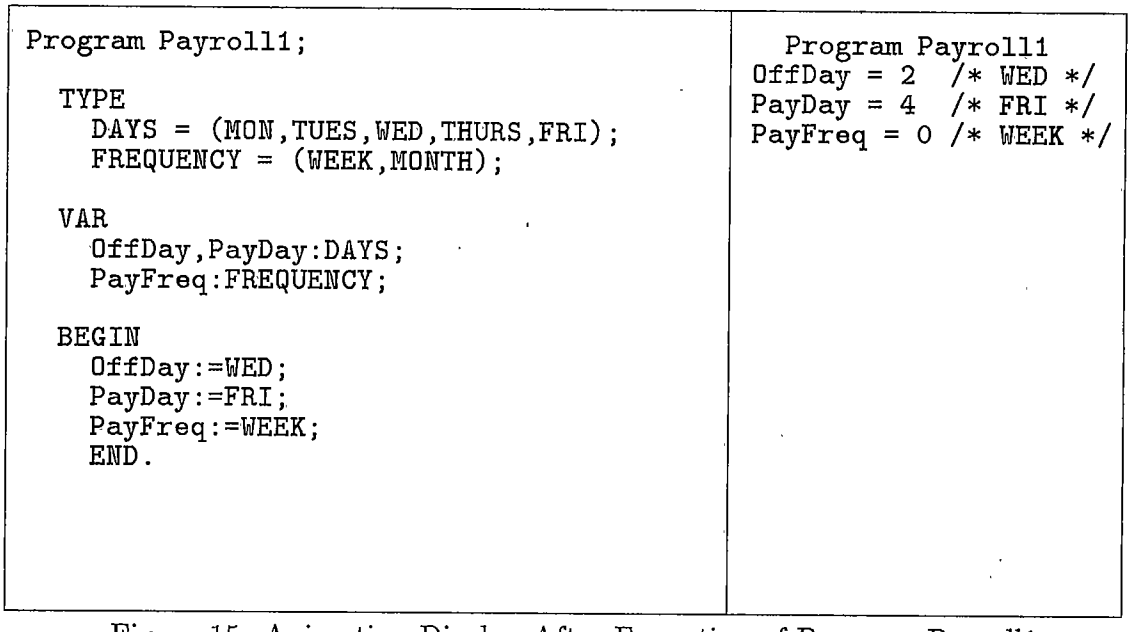

Figure 15: Animation Display After Execution of Program Payroll1

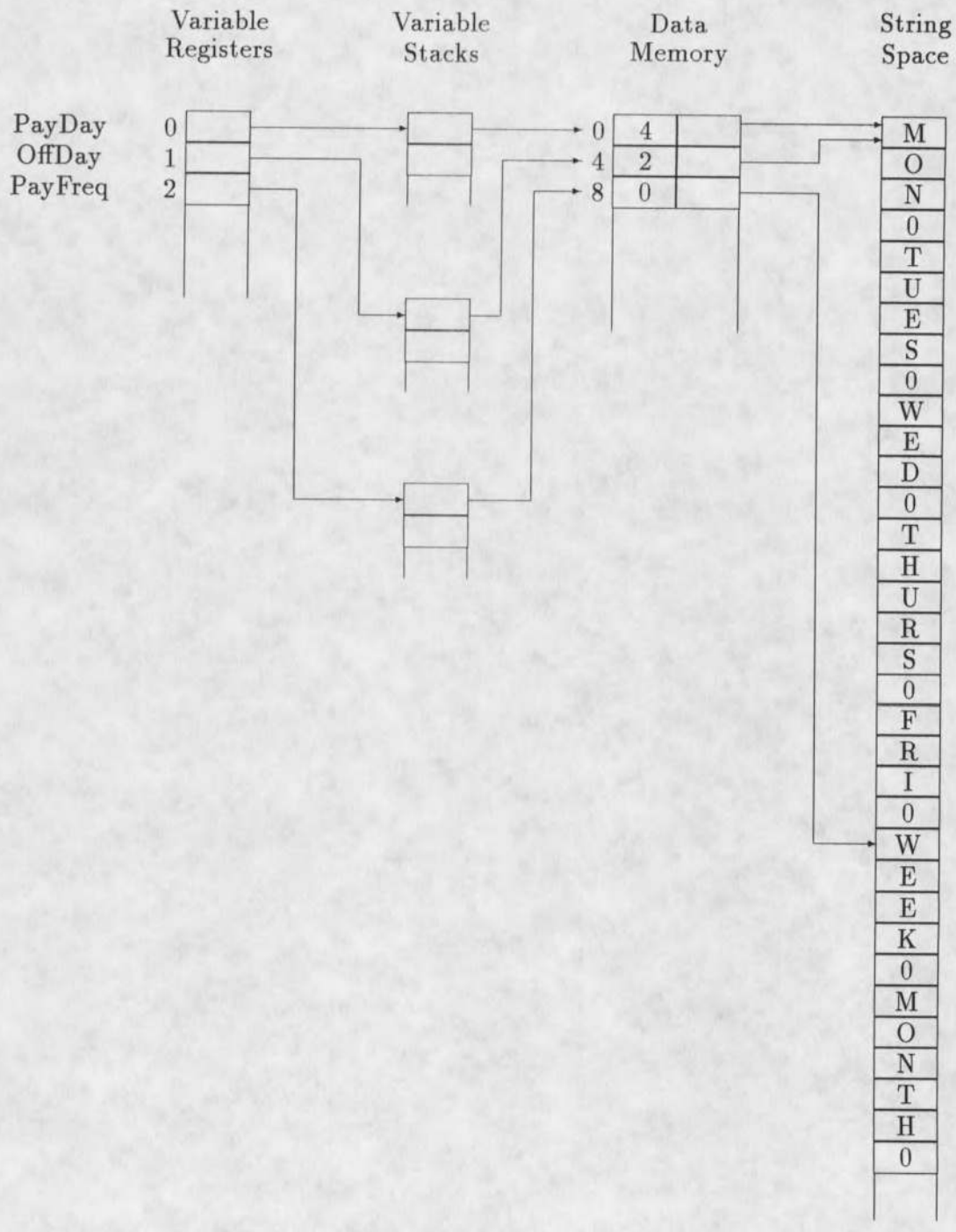

Figure 16: String Space's Relationship with Variable Registers and Data Memory

names are null-terminated, thus allowing such a search. (A similar situation will exist when the predeclared Pascal functions, pred and succ, are eventually implemented.)

The animator also accesses the string space when displaying enumerated type array indices. Thus, upon completion of the execution of the program shown in figure 17, the animator can display DayCode's value as shown in figure 18. In this case, the animator retrieves the values of the enumerated type indices through information stored in the Static Scope Table. In this example, the Static Scope Table entry for the variable DayCode contains the following information:

- Identifier Name: DayCode
- Upper array bound: 19
- Lower array bound: 0
- Entry type: INTEGERENUMI
- Variable Reg: 0

Type INTEGERENUMI means that the variable DayCode is an array with integer elements and an enumerated index type. This indicates to the animator that the array bounds are indices into the string space rather than absolute numbers.

The Error Module

The Error module produces an error message whenever an error is encountered during compilation. This module consists of routines to report the following types of errors:

- scan errors;
- parse errors;
- internal errors;
- parse warnings;
- lack of support messages.

```
Program Payroll2;

  TYPE
    DAYS = (MON,TUES,WED,THURS,FRI);
    DAYLIST = ARRAY [MON..FRI] OF INTEGER;

  VAR
    DayCode:DAYLIST:
    Day:DAYS;

  BEGIN
    FOR Day := MON TO FRI DO
      DayCode[Day] := 0;
    END.
```

Figure 17: Source Code for Program Payroll2

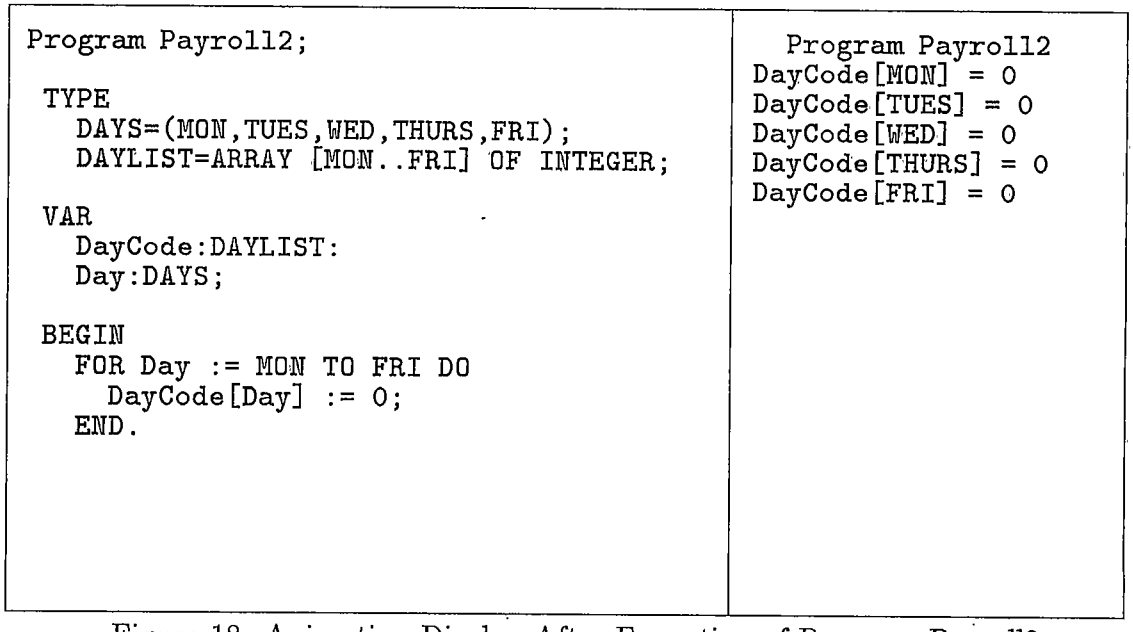

Figure 18: Animation Display After Execution of Program Payroll2

Each of these routines prints an appropriate message, and, with the exception of the parse warning routine, then calls the AbnormalEnd routine in the Main module. The Error module routines are called by various other modules during the compilation process.

The Memory Allocation Module

The Memory Allocation module is responsible for allocating and freeing memory for the various data structures required during the compilation process. This module consists of allocate and free routines associated with each data structure defined in the compiler. The Memory Allocation module routines are called by many of the other modules during the compilation process.

The Assembly Code Module

The Assembly Code module produces a text file containing the pseudo assembly language translation of a source program. The WrtAsmFile routine in this module writes a copy of the CODESECTION instructions to a text file in pseudo assembly language format. This module is not required for compilation since it does not generate any of the E-code file sections; it does, however, provide an excellent debugging tool for compiler development. The routines in this module are (optionally) called by the CODE module as the instructions are generated.

The CODE Module

The CODE module produces the array of C structures containing the E-code instructions. The CODE module contains the GenInstr routine which writes a single E-code instruction to a temporary file, and (optionally) calls the WrtAsmFile routine in the Assembly Code module (to output an equivalent pseudo assembly code instruction). The GenInstr routine is called by the Parser and Code Driver modules.

The CODE module also contains the PatchInstr routine, which maintains an array of structures that associate a "patch value" with a program memory address. There are two situations when a code patch is necessary:

1. References to label values before they are known during the generation of case statement code.
2. The association of the index of the Static Scope Table entry for a procedure/function with the appropriate pushd instruction (see the Parser module section previously in this chapter).

When compilation is complete, the Main module closes and reopens the temporary CODESECTION file and then calls the WriteCODE routine. This routine reads the temporary file and writes the records to the CODESECTION of the E-code file, incorporating any patches into the proper instructions.

The Symbol Table Module

The Symbol Table module manages the compiler's symbol table. The Symbol Table module routines are responsible for opening and closing static scopes as well as entering and retrieving identifier names and their attributes. The Symbol Table routines are called by the Parser, the Semantic Analysis, the Code Driver, and the STATSCOPE modules.

Each identifier name in the symbol table has a static scope level and, possibly, a record number, associated with it. This allows the same identifier name to be used in more than one scope, including scopes nested within each other. The current scope level is contained in a global variable in the Symbol Table Interface module. When the Parser module determines the beginning of a scope, it calls the OpenScope routine, which simply increments the scope level. When the Parser module determines an end of scope, it calls the CloseScope routine, which deletes all symbol table entries for that scope and then decrements the scope level. It should

be noted that, at the beginning of compilation, the Parser module (via the Semantic Analysis module) enters the predefined Pascal types—integer, real, boolean, and char—and the predefined Pascal constants—true, false, and maxint—into the symbol table. These predefined identifiers are associated with the outermost scope of the program.

The symbol table itself is implemented as a single hash table using chaining to resolve collisions. The chained buckets are stacked such that the identifiers declared in the most recent scope are at the top of the stack. This ensures that the proper identifier is retrieved by an outward search of the buckets associated with the same identifier name. The hashing algorithm used is "hashpjw" from P.J. Weinberger's C compiler, as presented in *Compilers: Principles, Techniques, and Tools* by Aho, Sethi, and Ullman [Aho 86].

The basic structure of the symbol table, as well as the design of the Symbol Table module, are based on the symbol table design presented in *Crafting A Compiler* by Charles N. Fischer and Richard J. LeBlanc, Jr. [Fischer 88]. There are also several symbol table features adapted from the symbol table design given in *Compiler Design in C* by Allen I. Holub [Holub 90]. Figure 19 illustrates the symbol table implementation.

A symbol table entry for an identifier consists of several structures which are chained together. The various symbol table structures are shown in figure 20. The identifier's primary structure, the Symbol Structure, is initialized as soon as the Parser module encounters the identifier's declaration. Later, when the parse has progressed to the point where the identifier's attributes become known, the remaining "descriptor" structures are initialized. Once its attributes are known, an identifier's symbol table entry will consist of at least a Symbol Structure, a Type Descriptor Structure, and a Class Descriptor Structure. The various symbol table structures are described below.

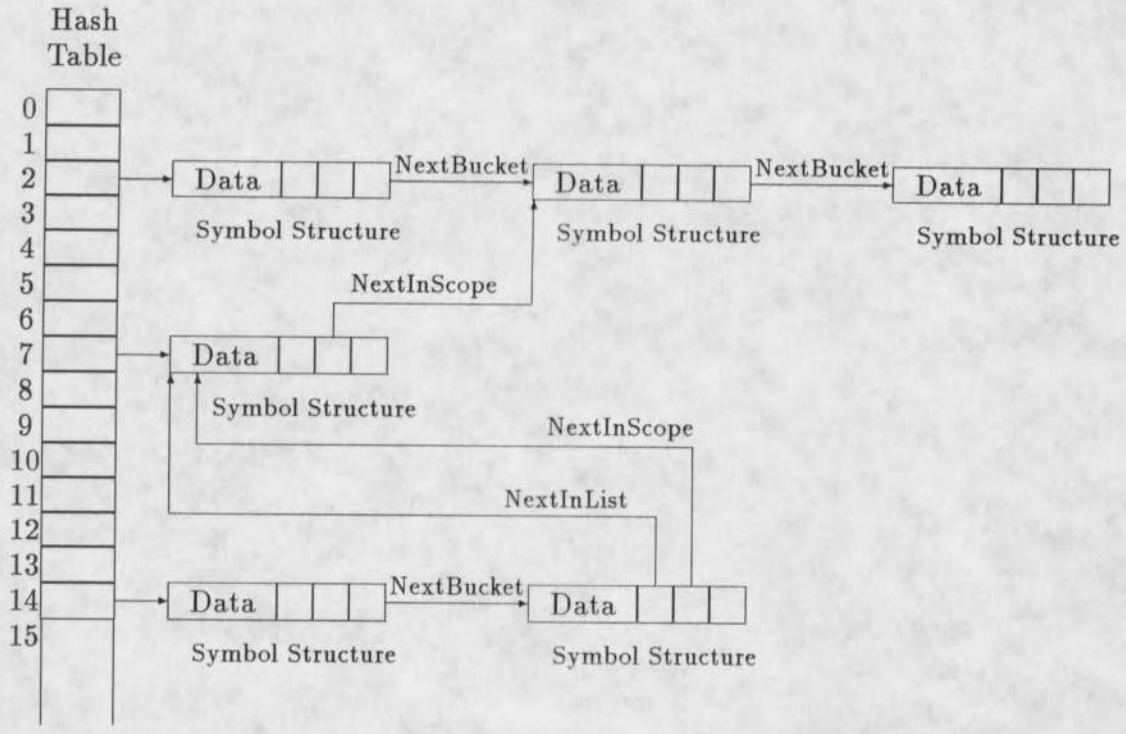

Figure 19: The Symbol Table Hash Implementation

A Symbol structure consists of the following fields:

- IdName (identifier name). This field contains a pointer to the dynamically allocated memory location containing the identifier name.
- ScopeLevel. This field contains the static scope level of the identifier.
- RecordNum. If the identifier being described is a field name, this Symbol Structure field contains the compiler-assigned number of the record containing the identifier name. If the identifier is not a field name, this number is 0.
- LineNum. This field contains the identifier's source code line number. Since all of the symbol table names are stored as lower case, this number is later used by the STATSCOPE module to retrieve the identifier's original name (for animation purposes) from the source code array, which is maintained in the SOURCE module.

- ColNum. This field contains the identifier's starting column number in the source code and is likewise used by the STATSCOPE module to retrieve the identifier's original name.

- IdType. This field contains the pointer to the identifier's Type Descriptor record.

- IdClass. This field contains a pointer to the identifier's Class Descriptor record.

- NextInList. If the identifier is declared within a list of identifiers, this field contains the pointer to the Symbol Structure of the previous identifier in the list. The Parser module can then traverse this chain to associate common attributes with the identifiers in the list. This chain is also used (in reverse order) when assigning internal numeric values to enumerated constant names.

- NextInScope. This field contains a pointer to the Symbol Structure of the previous identifier in the current static scope level. This chain is traversed by the STATSCOPE module to retrieve the identifiers declared within a given static scope (as discussed in the next section).

- NextBucket. This field contains the pointer to the previous Symbol Structure in the hash table entry's collision chain. This chain is used internally in the Symbol Table module.

A Type Descriptor structure consists of the following fields:

- UseCount. Since numerous other structures may point to the same Type Descriptor structure, the UseCount field is utilized to prevent the Memory Allocation module from freeing the structure while it is still in use.

- Size. This field contains the identifier's size (in terms of the host computer's smallest addressable memory size, normally bytes).

- Packed. This field indicates whether or not a structured type identifier is packed. The miniPascal compiler recognizes the packed attribute only if the identifier is a string variable.

- TypeName. This field contains an enumerated constant value indicating the identifier's type (e.g., INTEGERTYPE, ENUMTYPE, or ARRAYTYPE).

- SelectType. This field contains a pointer to another descriptor based on the value of the TypeName field. For example, if the TypeName is ARRAYTYPE, SelectType would point to an Array Descriptor record, which would contain further attribute information pertaining to the identifier.

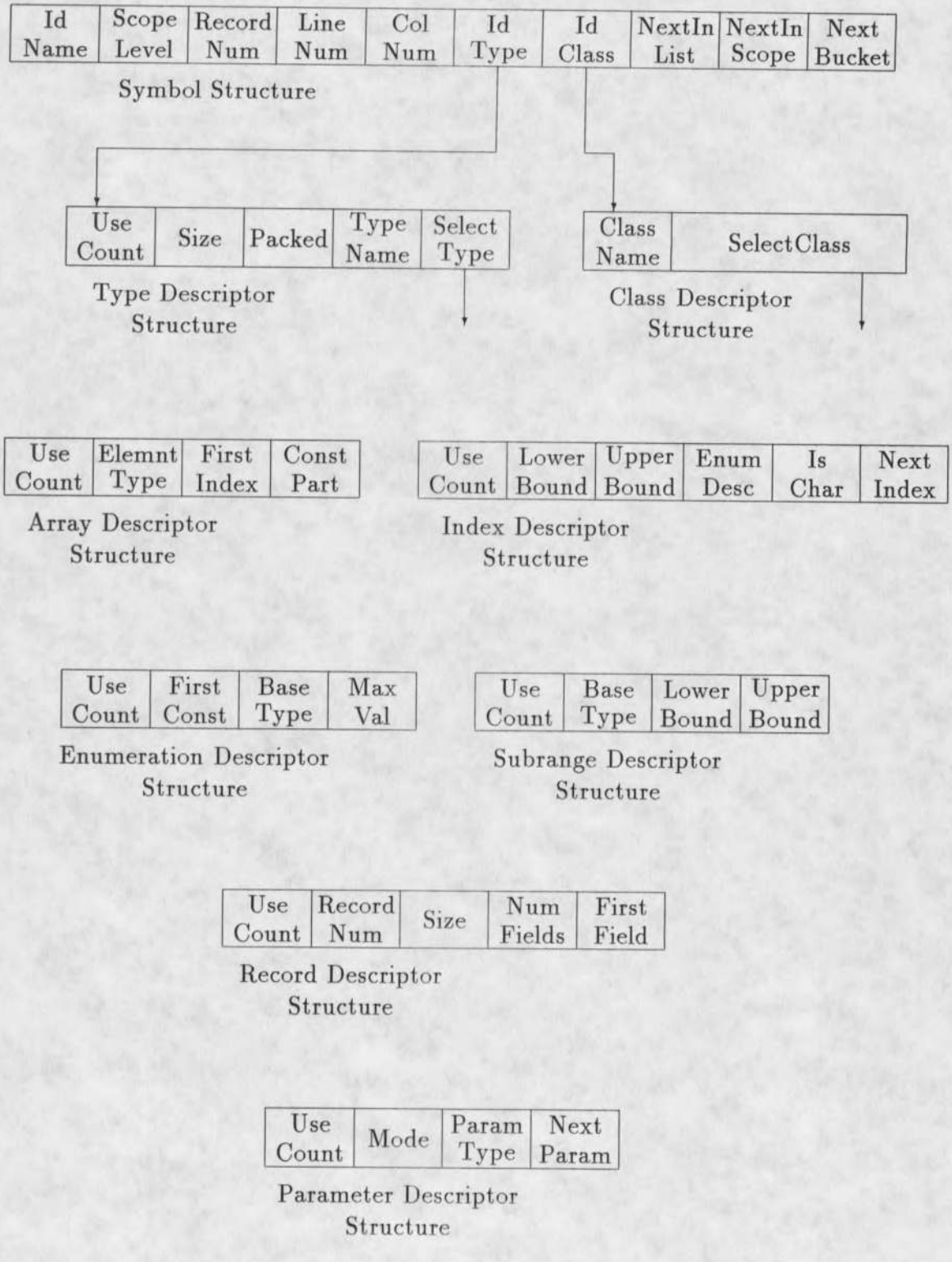

Figure 20: The Symbol Table Structures

A Class Descriptor structure consists of the following fields:

- ClassName. This field contains an enumerated constant value indicating the identifier's class (e.g., VARIABLECLASS, CONSTANTCLASS, or PROCEDURECLASS).
- SelectClass. This field is composed of a union structure containing information based on the value of the ClassName field. For example, if the ClassName is VARIABLECLASS, one of the fields in this structure would contain the variable register number associated with the variable.

The remaining symbol table structures are used in describing specific type or class attributes pertaining to a given identifier. These symbol table structures are discussed below. Figure 21 shows the enumerated names of the various miniPascal identifier types; figure 22 shows the enumerated names of the various miniPascal identifier classes. (Identifier types POINTERTYPE, SETTYPE, and FILETYPE listed in figure 21 are not currently implemented.)

```
      /* Identifier types */
typedef enum
{
  INTEGERTYPE, REALTYPE, BOOLEANTYPE, CHARTYPE, ENUMTYPE, SUBRANGETYPE,
  POINTERTYPE, SETTYPE, ARRAYTYPE, RECORDTYPE, FILETYPE, EXISTINGTYPE,
  STRINGTYPE
} IdTypes;
```

Figure 21: The miniPascal Identifier Types

```
      /* Identifier classes */
typedef enum
{
  VARIABLECLASS, FIELDCLASS, TYPENAMECLASS, CONSTANTCLASS,
  PROCEDURECLASS, FUNCTIONCLASS, INTLITCLASS, REALLITCLASS,
  CHARLITCLASS, STRINGLITCLASS, PROGRAMCLASS, TEMPCLASS, PARAMETERCLASS
} IdClasses;
```

Figure 22: The miniPascal Identifier Classes

In some cases, no additional structures are needed to describe an identifier's attributes. For example, the three basic symbol structures (defined above) suffice in describing a variable (i.e., identifier class is VARIABLECLASS) of type integer (i.e., identifier type is INTEGERTYPE).

An Array Descriptor structure is used when an identifier's TypeName field (in its Type Descriptor structure) has the value ARRAYTYPE. The SelectType field holds a pointer to the corresponding Array Descriptor structure. An Array Descriptor structure consists of the following fields:

- UseCount. Since numerous other structures may point to the same Array Descriptor structure, the UseCount field is utilized to prevent the Memory Allocation module from freeing the structure while it is still in use.
- ElemntType. This field holds a pointer to the Type Descriptor structure defining the array's element type (e.g., the predefined integer type or some previously declared user-defined type).
- FirstIndex. This field holds the pointer to the Index Descriptor structure describing the array's (first) index. The Index Descriptor structure is defined below.
- ConstPart. This field holds the "constant part" used in the algorithm that calculates the linear address for a (multidimensional) array reference. This algorithm was taken from [Fischer 88].

An Index Descriptor structure is referenced by the FirstIndex field in a corresponding Array Descriptor structure, as discussed above. It may also be referenced by the NextIndex field of another Index Descriptor structure. An Index Descriptor structure consists of the following fields:

- UseCount. Since numerous other structures may point to the same Index Descriptor structure, the UseCount field is utilized to prevent the Memory Allocation module from freeing the structure while it is still in use.
- LowerBound. This field holds the (numeric) lower bound of an array index.
- UpperBound. This field holds the (numeric) upper bound of an array index.

- EnumDesc. If the array index values are an enumerated type, this field holds a pointer to the Enumeration Descriptor structure describing the enumerated type. The Enumeration Descriptor structure is discussed below.
- IsChar. This field's value is TRUE if the array index values are CHARTYPE; otherwise this field's value is FALSE.
- NextIndex. This field holds a pointer to the Index Descriptor structure describing the next array index (if any).

An Enumeration Descriptor structure is used when an identifier's TypeName field (in its Type Descriptor structure) has the value ENUMTYPE. The SelectType field holds a pointer to the corresponding Enumeration Descriptor structure. An Enumeration Descriptor structure consists of the following fields:

- UseCount. Since numerous other structures may point to the same Enumeration Descriptor structure, the UseCount field is utilized to prevent the Memory Allocation module from freeing the structure while it is still in use.
- FirstConst. This field holds the pointer to the Symbol structure describing the first constant in the enumeration.
- BaseType. This field holds a pointer to the Type Descriptor structure that describes the first constant in the enumeration.
- MaxVal. This field holds the maximum numeric value associated with the enumeration (i.e., the number of constants in the enumeration minus 1).

A Subrange Descriptor structure is used when an identifier's TypeName field (in its Type Descriptor structure) has the value SUBRANGETYPE. The SelectType field holds the pointer to the corresponding Subrange Descriptor structure. A Subrange Descriptor structure consists of the following fields:

- UseCount. Since numerous other structures may point to the same Subrange Descriptor structure, the UseCount field is utilized to prevent the Memory Allocation module from freeing the structure while it is still in use.
- BaseType. This field holds a pointer to the Type Descriptor structure describing the base type of the subrange (e.g., a pointer to the Type Descriptor structure describing the predefined integer or char types).

- LowerBound. This field holds the (numeric) lower bound of the subrange.
- UpperBound. This field holds the (numeric) upper bound of the subrange.

A Record Descriptor structure is used when an identifier's TypeName field (in its Type Descriptor structure) has the value RECORDTYPE. The SelectType field holds a pointer to the corresponding Record Descriptor structure. A Record Descriptor structure consists of the following fields:

- UseCount. Since numerous other structures may point to the same Record Descriptor structure, the UseCount field is utilized to prevent the Memory Allocation module from freeing the structure while it is still in use.
- RecordNum. This field hold the (compiler-generated) number associated with the record.
- Size. This field holds the record's size (normally, in bytes).
- NumFields. This field holds the number of fields in the record.
- FirstField. This field holds a pointer to the Symbol structure associated with the record's first field.

A Parameter Descriptor structure is used when an identifier's ClassName field (in its Class Descriptor structure) has the value PROCEDURECLASS or FUNCTIONCLASS. A Parameter Descriptor structure describes the formal parameters associated with a procedure (or function). The union structure corresponding to the SelectClass field holds a pointer to the Parameter Descriptor structure describing the first parameter in the formal parameter list. A Parameter Descriptor structure consists of the following fields:

- UseCount. Since numerous other structures may point to the same Parameter Descriptor structure, the UseCount field is utilized to prevent the Memory Allocation module from freeing the structure while it is still in use.
- Mode. This field holds the parameter's mode (i.e., VALUE or REFERENCE).
- ParamType. This field holds a pointer to the Type Descriptor structure that describes the parameter.
- NextParam. This field holds a pointer to the Parameter Descriptor structure of the next parameter in the formal parameter list.

The STATSCOPE Module

The STATSCOPE module contains the routines that build the Static Scope Table. The animator uses the Static Scope Table in conjunction with the dynamic scope stack to determine the data memory values that should be displayed at a given point during a program's execution. The Static Scope Table was called the Symbol Table in Birch's thesis [Birch 90]; the name was changed here to avoid confusion with the compiler's symbol table. This table is a linear array of structures (or entries) which are in turn divided into numerous scope blocks. The scope blocks are chained together via parent/child pointers as discussed later. A scope block is used to describe the program identifiers associated with a single static scope. For example, a scope block would describe all of the local variable names and locally declared functions and/or procedures within a given procedure.

Generating a Static Scope Block. The Parser module calls the STATSCOPE module's GenStatScopeBlock routine whenever the end of a static scope is encountered during parsing (i.e., at the end of a procedure, function, or program). The parsing of an inner scope is always completed before the containing scope is completely parsed (a result of Pascal syntax).

The GenStatScopeBlock routine drives the generation of the static scope block in the Static Scope Table for the scope in question from information in the symbol table for the current scope. (Recall that the symbol table entries for this scope will be deleted at this point of the parse, so this information must be saved in the Static Scope Table for animation purposes.) This routine, via calls to other STATSCOPE module routines, performs the following tasks:

- dynamic allocation of a static scope block. The number of static scope entries (i.e, the size of the static scope block) is passed as a parameter to GenStatScopeBlock;

- entry of the static scope's owner name into the Scope Owner Table. The Scope Owner Table contains the information necessary to tie all of the static scope blocks together at the end of compilation. The static scope's owner name is passed as a parameter to GenStatScopeBlock;
- initialization of the descriptive information contained in the static scope block entries.

The names and descriptive attributes of the identifiers declared within a scope are retrieved by traversing the symbol table's NextInScope chain; the head of the appropriate "scope chain" is passed as a parameter to GenStatScopeBlock.

A Static Scope Table entry describing a simple variable identifier includes the variable's type attribute (e.g., INTEGER) and its variable register number attribute. For the more complicated array variable entry, additional fields are utilized to describe the array bounds. If the array's index values are simple integers or characters, the lower and upper bound values are entered directly into the corresponding fields. For arrays whose index values are enumerated type values, the appropriate indices into the STRINGSECTION array are computed and entered into the static scope entry's array bound fields (see the STRING module section previously in this chapter). For multidimensional arrays, additional scope blocks are used to describe the other index bounds. These additional index scope blocks are chained via the NxtIdx field. A corresponding entry is placed in the Scope Owner Table indicating that the array "owns" the index scope block (via the Array Descriptor field). Record variable entries also use an additional scope block to describe the fields within the record. The child pointer is used to associate a record name with its defining scope block. Here again, an entry is placed in the Scope Owner Table indicating that the record "owns" the scope block (via the Record Descriptor field).

The ProcNum Field. The Static Scope Table's ProcNum field can now be explained. As each program, procedure, and function name identifier is

encountered during compilation, it is assigned a unique "procedure number." The identifier names are referred to as *static scope names* in the following discussion. The procedure number is produced by a counter variable in the Semantic Analysis module. Thus, the procedure number assigned to a miniPascal program name is always 0. The next static scope name declaration encountered in the program would be assigned the procedure number 1, and so on. A static scope name's procedure number is stored as one of its symbol table attributes. Thus, when the GenStatScopeBlock routine encounters a static scope name while traversing a containing scope's NextInScope chain, one of the attributes it retrieves is the corresponding procedure number. This number is then placed in the ProcNum field of the Static Scope Table entry describing the static scope name.

The animator uses the ProcNum field in conjunction with the dynamic scope stack when determining the dynamics of program execution. The use of this field is best explained by an example. The program shown in figure 23 contains a recursively called function (function Fact). That Fact is recursive implies that for any given call to function Fact, the animator must be able to determine the "depth" of the pertinent data memory values associated with the variables declared in function Fact, as well as the depths of any variables in the calling (program) scope. These values are retrieved by querying the appropriate variable stacks, as discussed in chapter 2. Thus, upon the final recursive call to function Fact, the animator should be able to display data memory values as shown in figure 24.

The ProcNum field is used in the following manner when determining the depths of the variables declared in a program, procedure or function scope. After the E-machine has been loaded with the E-code translation of a source program, the animator queries the E-machine to determine the total number of static "procedure" scopes that are described in the Static Scope Table. The Static Scope Table for the example in figure 23 is shown in table 4. The animator then dynamically allocates

```
Program Ftrl;

VAR
  n,nfact:INTEGER;

  Function Fact(n:INTEGER):INTEGER;
    BEGIN
      IF n = 0
        THEN Fact:=1
        ELSE Fact:=n * Fact(n-1)
      END;

  BEGIN
    n:=3;
    nfact:=Fact(n);
    END.
```

Figure 23: Source Code for Program Ftrl

```
Program Ftrl;

VAR
  n,nfact:INTEGER;

  Function Fact(n:INTEGER):INTEGER;
    BEGIN
      IF n = 0
        THEN Fact:=1
        ELSE Fact:=n * Fact(n-1)
      END;

  BEGIN
    n:=3;
    nfact:=Fact(n);
    END.
```

```
   Program Ftrl
n = 3
nfact is undefined
-----------------------
   Function Fact
n = 3
Fact is undefined
-----------------------
   Function Fact
n = 2
Fact is undefined
-----------------------
   Function Fact
n = 1
Fact is undefined
-----------------------
   Function Fact
n = 0
Fact = 1
```

Figure 24: Animation Display After Final Recursive Call of Function Fact

a *procedure count array* that contains an entry corresponding to each of these scopes. Thus, for the program shown in figure 23, this array has two entries. Entry 0 corresponds to the program scope and entry 1 corresponds to function Fact. During program animation, the animator sets the values of the procedure count array entries to reflect the current number of active calls to the corresponding procedure or function. (This means that the animator reinitializes the values in the procedure count array *every* time control is passed to the animator.) At the same time, the E-machine's dynamic scope stack contains a history of active scopes, with the Static Scope Table entry number of the most current scope being the value at the top of this stack.

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num
Scope block describing function Fact											
0		-	-	-	-	HEADER	-	4	-	-	-
1	n	-	-	-	-	INTEGER	-	-	-	2	-
2	Fact	-	-	-	-	INTEGER	-	-	-	3	-
3		-	-	-	-	END	-	-	-	-	-
Scope block describing program Ftrl											
4		-	-	-	-	HEADER	-	9	-	-	-
5	n	-	-	-	-	INTEGER	-	-	-	1	-
6	nfact	-	-	-	-	INTEGER	-	-	-	0	-
7	Fact	-	-	-	-	FUNCTION	-	-	0	-	1
8		-	-	-	-	END	-	-	-	-	-
Bootstrap scope block											
9		-	-	-	-	HEADER	-	-	-	-	-
10	Ftrl	-	-	-	-	PROGRAM	-	-	4	-	0
11		-	-	-	-	END	-	-	-	-	-

Table 4: Static Scope Table Resulting from Compilation of Program Ftrl

Now, consider the animation of the current example. Suppose the program has executed to the point that it is in the third recursive call to function Fact. When the animator begins displaying data memory variables after the execution of the packet

translating the animation unit `Fact:=1`, the procedure count array and the dynamic scope stack are in the state shown in figure 25. The values in the procedure count array indicate that the program Ftrl has one active "call" and that function Fact has four active calls. In this example, the animator begins its retrieval of data memory values by examining the value at the *bottom* of the dynamic scope stack. The bottom stack value is 10, which means that the animator now examines the tenth entry in the Static Scope Table. This entry is a PROGRAM entry describing Ftrl. The ProcNum field in the PROGRAM entry has the value 0. Next, the animator will examine entry 0 in the procedure count array to determine the depth of the variables to be displayed for this invocation of the program scope. Since the program scope cannot be called recursively, this value will always be 1. Thus, when the animator retrieves the values of the variables described in the program's child scope block, it will instruct the E-machine to retrieve the data memory values associated with the top of the appropriate variable stacks. After these values have been displayed, the animator decrements the value in entry 0 of the procedure count array.

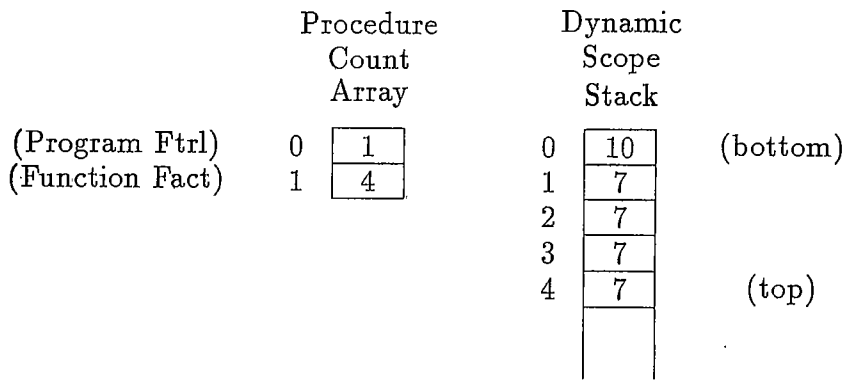

Figure 25: Procedure Count Array and Dynamic Scope Stack

Next, the animator examines the value in entry 1 in the dynamic scope stack. This value is 7, corresponding to the seventh entry in the Static Scope Table. This entry, whose ProcNum field has the value 1, describes function Fact. The animator then examines entry 1 in the procedure count array. The current value in this entry is 4, indicating that the animator should instruct the E-machine to retrieve data memory values associated with the fourth level of the appropriate variable stacks when displaying variable values described in the function's child scope block. These values reflect the function's variable values resulting from its initial call from the program scope. The animator then decrements the value in entry 1 of the procedure count array so that the next iteration will result in displaying the values associated with the first recursive call to function Fact. The animator continues this process until the dynamic scope stack is exhausted, resulting in the display shown in figure 24.

Writing the STATSCOPESECTION. When parsing is completed, the Main module calls the WriteSTATSCOPE routine. This routine first traverses the Scope Owner Table in reverse order to initialize the program, procedure, and function parent/child pointers that will appear in the final linear array containing the complete Static Scope Table. Since the nesting characteristics of miniPascal allow the same name to be given to more than one procedure or function, the reverse order traversal ensures that the proper child is found. The final entry in the Scope Owner Table describes a "bootstrap" scope block, which will become the final scope block in the completed Static Scope Table. The Scope Owner Table contains the information needed to initialize the child pointer in the bootstrap scope block; this child pointer is the computed index of the first entry in the scope block describing the local variables, procedures, and functions belonging to the program. Also, the parent pointer in the program scope block can now be initialized with the computed index of the

bootstrap scope block. Similarly, each function and procedure can have its child and parent pointers initialized.

Finally, the WriteSTATSCOPE routine traverses the Scope Owner Table in forward order to sequentially write the various scope blocks to the STATSCOPESECTION of the E-code file.

Example of STATSCOPESECTION Generation. Program Samp2, shown in figure 26, is used to illustrate the generation of the STATSCOPESECTION. This program contains two procedures, named A and B, which are at the same static scope level. Procedure A contains a nested function, also named B. Table 5 is the Scope Owner Table for this program. The Scope Owner Table holds the following information:

- Owner Name. This field contains the corresponding static scope's owner's name (for program, procedure, and function scopes);
- Pointer to Scope Block. This field contains the memory address of the corresponding scope's dynamically allocated scope block. The numbers used in this example are for illustrative purposes only;
- Scope Table Index. This field contains the computed (final) index of the first entry of the static scope block describing the corresponding static scope;
- Number of Scope Entries. This field contains the number of identifiers (e.g., variable names and function names) declared in the corresponding static scope;
- Array Descriptor (indicates the "owner" of additional scope blocks containing index descriptions for multidimensional array variables). This field contains the memory address of a dynamically allocated symbol table array descriptor structure, and thus allows array variables sharing the same user defined type to share an index scope block;
- Record Descriptor (indicates the "owner" of the additional scope containing record field descriptions). This field contains the memory address of a dynamically allocated symbol table record descriptor structure, and thus allows record variables that share the same user defined type also to share a field description scope block.

```
Program Samp2;

TYPE
  LIST = ARRAY [1..4] OF INTEGER;

VAR
  X,Y:INTEGER;
  List1:LIST;

    Procedure A(VAR X,Y:INTEGER);

            Function B(I:INTEGER):INTEGER;
              BEGIN { Function B }
                .
                .
                .
              END; { Function B }

      BEGIN { Procedure A }
        .
        .
        .
      END; { Procedure A }

    Procedure B(I,J:INTEGER);
      VAR
        List2:LIST;
      BEGIN { Procedure B }
        .
        .
        .
      END; { Procedure B }

  BEGIN { Program Samp2 }
    .
    .
    .
  END. { Program Samp2 }
```

Figure 26: Source Code for Program Samp2

Owner Name	Pointer to Scope Block	Scope Table Index	Number of Scope Entries	Array Descriptor	Record Descriptor
B	1000	0	4	-	-
A	3002	4	5	-	-
B	5001	9	5	-	-
Samp2	6240	14	7	-	-
Bootstrap	7000	21	3	-	-

Table 5: Scope Owner Table for Program Samp2

Tables 6 through 10 show the five scope blocks generated by the STATSCOPE module during compilation. Table 11 shows the completed Static Scope table as it would be written to the STATSCOPESECTION at the end of compilation.

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num
0		-	-	-	-	HEADER	-	-	-	-	-
1	I	-	-	-	-	INTEGER	-	-	-	5	-
2	B	-	-	-	-	INTEGER	-	-	-	6	-
3		-	-	-	-	END	-	-	-	-	-

Table 6: Scope Block for Function B in Procedure A in Program Samp2

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num
0		-	-	-	-	HEADER	-	-	-	-	-
1	X	-	-	-	-	INTEGER	-	-	-	4	-
2	Y	-	-	-	-	INTEGER	-	-	-	3	-
3	B	-	-	-	-	FUNCTION	-	-	-	-	2
4		-	-	-	-	END	-	-	-	-	-

Table 7: Scope Block for Procedure A in Program Samp2

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num
0		-	-	-	-	HEADER	-	-	-	-	-
1	I	-	-	-	-	INTEGER	-	-	-	8	-
2	J	-	-	-	-	INTEGER	-	-	-	7	-
3	List2	4	1	-	-	INTEGER	-	-	-	9	-
4		-	-	-	-	END	-	-	-	-	-

Table 8: Scope Block for Procedure B in Program Samp2

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num
0		-	-	-	-	HEADER	-	-	-	-	-
1	X	-	-	-	-	INTEGER	-	-	-	1	-
2	Y	-	-	-	-	INTEGER	-	-	-	0	-
3	List1	4	1	-	-	INTEGER	-	-	-	2	-
4	A	-	-	-	-	PROCEDURE	-	-	-	-	1
5	B	-	-	-	-	PROCEDURE	-	-	-	-	3
6		-	-	-	-	END	-	-	-	-	-

Table 9: Scope Block for Program Scope in Program Samp2

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num
0		-	-	-	-	HEADER	-	-	-	-	-
1	Samp2	-	-	-	-	PROGRAM	-	-	-	-	0
2		-	-	-	-	END	-	-	-	-	-

Table 10: Scope Block for "Bootstrap" Scope in Program Samp2

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num
Scope block describing function B in procedure A											
0		-	-	-	-	HEADER	-	4	-	-	-
1	I	-	-	-	-	INTEGER	-	-	-	5	-
2	B	-	-	-	-	INTEGER	-	-	-	6	-
3		-	-	-	-	END	-	-	-	-	-
Scope block describing procedure A											
4		-	-	-	-	HEADER	-	14	-	-	-
5	X	-	-	-	-	INTEGER	-	-	-	4	-
6	Y	-	-	-	-	INTEGER	-	-	-	3	-
7	B	-	-	-	-	FUNCTION	-	-	0	-	2
8		-	-	-	-	END	-	-	-	-	-
Scope block describing procedure B											
9		-	-	-	-	HEADER	-	14	-	-	-
10	I	-	-	-	-	INTEGER	-	-	-	8	-
11	J	-	-	-	-	INTEGER	-	-	-	7	-
12	List2	4	1	-	-	INTEGER	-	-	-	9	-
13		-	-	-	-	END	-	-	-	-	-
Scope block describing program Samp2											
14		-	-	-	-	HEADER	-	21	-	-	-
15	X	-	-	-	-	INTEGER	-	-	-	1	-
16	Y	-	-	-	-	INTEGER	-	-	-	0	-
17	List1	4	1	-	-	INTEGER	-	-	-	2	-
18	A	-	-	-	-	PROCEDURE	-	-	4	-	1
19	B	-	-	-	-	PROCEDURE	-	-	9	-	3
20		-	-	-	-	END	-	-	-	-	-
Bootstrap scope block											
21		-	-	-	-	HEADER	-	-	-	-	-
22	Samp2	-	-	-	-	PROGRAM	-	-	14	-	0
23		-	-	-	-	END	-	-	-	-	-

Table 11: Final Static Scope Table for Program Samp2

CHAPTER 5

CONCLUSIONS AND FUTURE ENHANCEMENTS

Conclusions

The first compiler for the E-machine has been designed and implemented. The compiler's source language, called *miniPascal*, is a subset of ISO Standard Pascal. The miniPascal compiler is a one-pass compiler written in ANSI Standard C and was developed using the Unix development tools, lex and yacc [Mason 90], [Lesk 75], [Johnson 75]. The compiler's scanner module, produced by running lex on a Unix machine, and its parser module, produced by running yacc on a Unix machine, were subsequently downloaded to a DOS machine. These two modules, compiled and linked with numerous semantic analysis and code generation modules, comprise the miniPascal compiler. A number of miniPascal programs compiled into E-machine object files have been successfully animated using a simple DOS animator to drive the E-machine.

Future Enhancements

Since miniPascal is a subset of Pascal, future versions of miniPascal will include additional Pascal features. A next logical feature to be implemented is pointers—particularly important to animate, because they are often a difficult concept for students to grasp. Other desirable features to be implemented in the future include: records with variant parts, the with statement, sets, and predeclared functions and procedures. It would be particularly useful to implement the predeclared procedure, *read*. The availability of the read procedure would greatly facilitate the initialization of data (e.g., arrays) in programs demonstrating concepts such as sorting and matrix multiplication.

One feature that is not completely implemented in the current version is the method of displaying the value returned by a function call. Currently, the code generated by the compiler allows the animator to display a function value only when displaying the variable values in a window associated with the called function itself. The function name, however, is actually declared in the calling scope, and hence its value is available in this scope. It would be desirable to have the function value also displayed in the calling scope's data memory window. A problem occurs when a function is called multiple times from the same scope, either by calls in several different statements or by multiple calls within the same statement. The question here is whether to display only the most recent value returned by the function, or to display all previous function values as well. Once this design decision is made, the compiler will require modification to produce code to support the display method.

The compiler should also be enhanced to identify the E-code instructions that are considered critical. Currently, the compiler simply designates all E-code instructions as critical, thus hampering the efficiency of the E-machine.

Another compiler enhancement is improvement of error handling. Currently, only minimal error reporting is supported by the compiler, and there is no attempt at error recovery. This minimal support is considered sufficient for the present DYNALAB system since the miniPascal programs will be prepared by expert programmers. Later, however, the DYNALAB system may be used by students preparing their own programs for animation. Thus, error handling must be enhanced to provide a more "friendly" environment for the miniPascal programmers.

Finally, since the DYNALAB system is intended to be an evolutionary system, the miniPascal compiler will continue to evolve in order to support new animation features. For example, the animator may provide visualization of expression evaluation in order to demonstrate precedence rules in a language. The animator may also display "TRUE" or "FALSE" as conditional expressions are evaluated. It may be desirable to have the programmer indicate groups of source code lines that should appear in the same source code animation window in order to clearly illustrate some concept. All of these animation features require modifications to the compiler in order to generate the supporting code.

Thus, even though the miniPascal compiler is a usable first compiler for the E-machine, its evolution is expected to continue. The compiler is also expected to serve as a pattern for developers of future E-machine compilers.

REFERENCES

References

[Aho 86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts. 1986.

[Birch 90] M. L. Birch. *An Emulator for the E-machine.* Master's thesis. Computer Science Department, Montana State University. June 1990.

[Brown 88-1] M. Brown. *Algorithm Animation.* The MIT Press, Cambridge, Massachusetts. 1988.

[Brown 88-2] M. Brown. "Exploring Algorithms Using Balsa-II", *Computer* Volume 21, Number 5. May 1988.

[Fischer 88] C. N. Fischer and R. J. LeBlanc, Jr. *Crafting a Compiler.* Benjamin/Cummings Publishing Company, Menlo Park, California. 1988.

[Holub 90] A. I. Holub. *Compiler Design in C.* Prentice Hall, Englewood Cliffs, New Jersey. 1990.

[Jensen 91] K. Jensen and N. Wirth. *Pascal: User Manual and Report.* Springer-Verlag, New York, New York. 1991.

[Johnson 75] S. C. Johnson. "Yacc: Yet Another Compiler-Compiler", *Computer Science Technical Report* Number 32. Bell Laboratories, Murray Hill, New Jersey. July 1975.

[Lesk 75] M. E. Lesk and E. Schmidt. "Lex - A Lexical Analyzer Generator", *Computer Science Technical Report* Number 39. Bell Laboratories, Murray Hill, New Jersey. October 1975.

[Mason 90] T. Mason and D. Brown. *lex & yacc.* O'Reilly and Associates, Sebastopol, California. 1990.

[Ng 82-1] C. Ng. *Ling User's Guide.* Unpublished Master's project. Computer Science Department, Washington State University. 1982.

[Ng 82-2] C. Ng. *Ling Programmer's Guide.* Unpublished Master's project. Computer Science Department, Washington State University. 1982.

[Patton 89] S. D. Patton. *The E-machine: Supporting the Teaching of Program Execution Dynamics.* Master's thesis. Computer Science Department, Montana State University. June 1989.

[Ross 91] R. J. Ross. "Experience with the DYNAMOD Program Animator", *Proceedings of the Twenty-second Symposium on Computer Science Education, SIGCSE Bulletin,* 23(1):35–42. 1991.

[Ross 93] R. J. Ross. "Visualizing Computer Science", Invited chapter to appear in the AACE monograph, *Scientific Visualization in Mathematics and Science Education*. 1993.

[Winslett 93] R. Winslett. *Juno*. Master's thesis in progress. Computer Science Department, Montana State University.

APPENDICES

APPENDIX A

THE E-MACHINE INSTRUCTION SET

This appendix, which is adapted from chapter 2 of Birch's thesis, lists all of the instructions in the instruction set of the E-machine. A pseudo assembly language format is used to describe the instructions, however the instruction stream itself is actually an array of structures loaded from the CODESECTION portion of the E-machine object file at run time. The object file is described in detail in chapters 2 and 4 of this thesis.

Each instruction is composed of four fields (or arguments):

- an opcode mnemonic (e.g., push, pop, add);
- a flag marking the instruction critical or noncritical (CFLAG);
- an field denoting the data type to be used in the instruction (TYPE);
- a field containing either a number (#) or an addressing mode (ADDR); Addressing modes and their formats are described in appendix B.

The mnemonic field is separated from the others by one or more spaces, and the remaining fields are separated by commas. The CFLAG field must be either c or n to designate whether the instruction is to be treated as critical (c) or noncritical (n). The TYPE field holds a single capital letter, I, R, B, C, or A, referring to the data types *integer, real, boolean, character*, or *address*, respectively. The # refers to a constant specifying the number of an E-code label, a constant numeric value, or an

E-machine variable register number. If the ADDR argument is used for the fourth field, it refers to any of the addressing modes described in appendix B.

In the following description of the instruction set, the effects of executing an instruction both forward and in reverse are given. The actions taken in each case will be different, depending on whether the instruction has been designated critical or noncritical. Some instructions have no critical/noncritical flag, because their execution (either forward or in reverse) would be the same in either case. Reversing through a noncritical instruction sometimes requires that something be pushed onto the evaluation stack to keep the stack of the proper size; in such cases an arbitrary value, called DUMMY is used.

add CFLAG, TYPE

Adds the top two values on the evaluation stack and places the result onto the evaluation stack.

Forward-Critical: Pops the top two values of the evaluation stack, pushes them onto the save stack, and then pushes their sum onto the evaluation stack.

Forward-Noncritical: Pops the top two values of the evaluation stack and pushes their sum onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards the value. Pops the top two elements of the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

alloc CFLAG, #

Allocates a block of memory of # size.

Forward: Attempts to allocate # computer words of storage. If successful, the address of the first word of data memory that was allocated is pushed onto the evaluation stack. Otherwise, a NULL address is pushed onto the evaluation stack.

Reverse: Pops the top value off the evaluation stack, which should be a data address, and frees # words of data memory starting at that address.

and CFLAG, TYPE

Bitwise and's the top two values of the evaluation stack and places the result onto the evaluation stack.

Forward-Critical: Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value bitwise and'ed with the top value onto the evaluation stack.

Forward-Noncritical: Pops the top two values of the evaluation stack and pushes the bottom value bitwise and'ed with the top value onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

br #

Unconditionally branches to label #.

Forward: Loads the program counter with the address of the label # instruction.

Reverse: No operation.

brt, brf CFLAG, #

Conditionally branches depending on whether the top of the evaluation stack is TRUE or FALSE.

Forward-Critical: Pops the top value off the evaluation stack and pushes it onto the save stack. If the value satisfies the conditional on the branch (TRUE for brt, FALSE for brf), the program counter is loaded with the address of the label # instruction.

Forward-Noncritical: Pops the top value off the evaluation stack. If the value agrees with the conditional branch (TRUE for brt, FALSE for brf), the program counter is loaded with the address of the label # instruction.

Reverse-Critical: Pops the top value of the save stack and pushes it onto the evaluation stack.

Reverse-Noncritical: Arbitrarily pushes DUMMY onto the evaluation stack.

call #

Branches to label # saving the program address which follows the call instruction so that execution will continue there upon execution of a return instruction.

Forward: Pushes the current program counter onto the return address stack, then loads the address of the label # instruction into the program counter.

Reverse: Pops the top value from the return address stack.

cast CFLAG, TYPE, TYPE

Changes the top value of the evaluation stack from the first TYPE to the second.

Forward-Critical: Pops the top value of the evaluation stack and pushes it onto the save stack, then transforms the value from the first TYPE to the second. The result is pushed onto the evaluation stack.

Forward-Noncritical: Pops the top value of the evaluation stack, then transforms the value from the first TYPE to the second. The result is pushed onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack. The pops the top value of the save stack and pushes it onto the evaluation stack.

Reverse-Noncritical Nothing happens.

div CFLAG, TYPE

Divides the second value from the top of the evaluation stack by the first and places the result onto the evaluation stack.

Forward-Critical: Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and pushes the bottom value divided by the top value onto the evaluation stack.

Forward-Noncritical: Pops the top two values of the evaluation stack and pushes the bottom value divided by the top value onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

eql, neql, less, leql, gtr, geql CFLAG, TYPE

If the second value from the top of the evaluation stack compares favorably with the first, then TRUE is pushed onto the evaluation stack. Otherwise FALSE is pushed onto the evaluation stack.

Forward-Critical: Pops the top two values off the evaluation stack, pushes the two values onto the save stack, compares the bottom value with the top. If the result of the comparison matches the comparison operation performed, a boolean TRUE is pushed onto the evaluation stack, otherwise, a boolean FALSE is pushed onto the evaluation stack.

Forward-Noncritical: Pops the top two values off the evaluation stack and compares the bottom value with the top value. If the result matches the comparison operation performed, a boolean TRUE is pushed onto the evaluation stack, otherwise, a boolean FALSE is pushed onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards it, then pops the top two values off the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

inst CFLAG, #

Creates an instance of the variable register #.

Forward-Critical: Allocates enough data memory for the variable represented by the variable register #. The address of the allocated memory is then pushed onto the variable register's stack.

Forward-Noncritical: Allocates enough data memory for the variable represented by the variable register #. The size of the variable is stored in the variable register. The address of the allocated memory is then pushed onto the variable register's stack.

Reverse-Critical: The data memory occupied by the variable register is freed and the top value is popped off the variable register's stack.

Reverse-Noncritical: Frees the space taken up by the variable in data memory and pops the top value off the variable register's stack.

label #

Marks the location to which a branch may be made.

Forward: Pushes the previous program counter onto the stack pointed to by label register #.

Reverse: Pops the top value of the stack pointed to by label register # and places it in the program counter.

link #

Associates one variable register with the value of another.

Forward: Pops the top value of the evaluation stack and pushes it onto the variable stack pointed to by variable register #.

Reverse: Pops the top value of the variable stack pointed to by variable register # and pushes it onto the evaluation stack.

loadar CFLAG, ADDR

Places the address ADDR in the address register.

Forward-Critical: The contents of the address register are pushed onto the save stack. Then the address computed for the addressing mode is placed in the address register. Important note: it is the address that is computed by the addressing mode that is used, not the contents of that address.

Forward-Noncritical: The address computed for the addressing mode is placed in the address register. Same note for Forward-Critical applies here.

Reverse-Critical: The address on top of the save stack is popped off and placed in the address register.

Reverse-Noncritical: Nothing happens.

loadir CFLAG, #

Places the # into the index register.

Forward-Critical: The contents of the index register are pushed onto the save stack. Then # is placed in the address register.

Forward-Noncritical: # is placed in the index register.

Reverse-Critical: The value on top of the save stack is popped off and placed in the index register.

Reverse-Noncritical: Nothing happens.

mod CFLAG, TYPE

Finds the remainder of the division of the second value from the top of the evaluation stack by the first and places the result onto the evaluation stack.

Forward-Critical: Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value modulo the top value onto the evaluation stack.

Forward-Noncritical Pops the top two values of the evaluation stack and pushes the bottom value modulo the top value onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

mult CFLAG, TYPE

Multiplies the top two values on the evaluation stack and places the result onto the evaluation stack.

Forward-Critical: Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes their product onto the evaluation stack.

Forward-Noncritical: Pops the top two values of the evaluation stack and pushes their product onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

neg TYPE

Negates the top value on the evaluation stack.

Forward: Pops the top of the evaluation stack and pushes the negation of that value onto the evaluation stack.

Reverse: Pops the top of the evaluation stack and pushes the negation of that value onto the evaluation stack.

nop This instruction is the standard no-operation instruction. It can be used to create packets for high level program text for which no E-machine instructions are generated but which nonetheless need to be highlighted for animation purposes. An example of this is the **begin** keyword in Pascal. In illustrating the flow of control during program animation, a **begin** keyword may need to be highlighted (and thus have its own underlying E-machine packet of instructions). The **nop** instruction can be used in these cases.

not CFLAG, TYPE

Bitwise complements the top value of the evaluation stack.

Forward: Pops the top of the evaluation stack and pushes the bitwise not of that value onto the evaluation stack.

Reverse: Pops the top of the evaluation stack and pushes the bitwise not of that value onto the evaluation stack.

or CFLAG, TYPE

Bitwise or's the top two values of the evaluation stack and places the result onto the evaluation stack.

Forward-Critical: Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value bitwise or'ed with the top value onto the evaluation stack.

Forward-Noncritical: Pops the top two values of the evaluation stack and pushes the bottom value bitwise or'ed with the top value onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

pop CFLAG, TYPE, ADDR

Pops the top value of the evaluation stack and places it in ADDR.

Forward-Critical: Pushes the value in ADDR onto the save stack and then pops the top value of the evaluation stack and stores it in ADDR.

Forward-Noncritical: Pops the top value of the evaluation stack and stores it in ADDR.

Reverse-Critical: Pushes the value in ADDR onto the evaluation stack and then pops the top value of the save stack and places it in ADDR.

Reverse-Noncritical: Pushes the value in ADDR onto the evaluation stack.

popar CFLAG

Pops the address on top of the evaluation stack and places it in the address register.

Forward-Critical: The contents of the address register are pushed onto the save stack. The address on top of the evaluation stack is popped and placed in the address register.

Forward-Noncritical: The address on top of the evaluation stack is popped off and placed in the address register.

Reverse-Critical: The contents of the address register are pushed onto the evaluation stack. Then the address on top of the save stack is popped off and placed in the address register.

Reverse-Noncritical: The contents of the address register are pushed onto the evaluation stack.

popd Pops the top value from the dynamic scope stack.

Forward: Pops the top value from the dynamic scope stack and pushes it onto the save dynamic scope stack.

Reverse: Pops the top value from the save dynamic scope stack and pushes it onto the dynamic scope stack.

popir CFLAG

Pops the integer on top of the evaluation stack and places it in the index register.

Forward-Critical: The contents of the index register are pushed onto the save stack. Then the integer on top of the evaluation stack is popped off and placed in the index register.

Forward-Noncritical: The integer on top of the evaluation stack is popped off and placed in the index register.

Reverse-Critical: The contents of the index register are pushed onto the evaluation stack. Then the integer on top of the save stack is popped off and placed in the index register.

Reverse-Noncritical: The contents of the index register are pushed onto the evaluation stack.

push TYPE, ADDR

Pushes the value in ADDR onto the evaluation stack.

Forward: Pushes the value in ADDR onto the evaluation stack.

Reverse: Pops the top value of the evaluation stack and stores it in ADDR.

pusha ADDR

Pushes the calculated address of ADDR onto the evaluation stack. This instruction is intended to be used for pushing the addresses of parameters passed by reference.

Forward: Pushes the calculated address of ADDR onto the evaluation stack.

Reverse: Pops and discards the address on top of the evaluation stack.

pushd #

Pushes the # onto the dynamic scope stack (where # is the index of a program, procedure, or function entry in the Static Scope Table)

Forward: Pushes # onto the dynamic scope stack.

Reverse: Pops the top value from the dynamic scope stack.

read CFLAG, TYPE

Reads a value from the user.

Forward: A user interface function is called to get input from the user. The input is converted from a string to the appropriate type and pushed onto the evaluation stack.

Reverse: The top value is popped off the evaluation stack.

return Returns to the appropriate program address following a call instruction.

Forward: Pops the top value of the return address stack and loads it into the program counter.

Reverse: Pushes the previous program counter onto the return address stack.

shl CFLAG, TYPE, #

Shifts the value on top of the evaluation stack # bits to the left filling on the right with 0's.

Forward-Critical: Pops the top value of the evaluation stack, pushes it onto the save stack, then shift it # bits to the left and pushes the result back onto the evaluation stack.

Forward-Noncritical: Pops the top value of the evaluation stack, shifts it left # bits, then pushes the result back onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack. Then pops the top value of the save stack and pushes it onto the evaluation stack.

Reverse-Noncritical: Nothing happens.

shr CFLAG, TYPE, #

Shifts the value on top of the evaluation stack # bits to the right filling on the left with 0's.

Forward-Critical: Pops the top value of the evaluation stack, pushes it onto the save stack, then shift it # bits to the right and pushes the result back onto the evaluation stack.

Forward-Noncritical: Pops the top value of the evaluation stack, shifts it right # bits, then pushes the result back onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack. Then pops the top value of the save stack and pushes it onto the evaluation stack.

Reverse-Noncritical: Nothing happens.

sub CFLAG, TYPE

Subtracts the value on the top of the evaluation stack from the second value from the top and places the result onto the evaluation stack.

Forward-Critical: Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value minus the top value onto the evaluation stack.

Forward-Noncritical: Pops the top two values of the evaluation stack, and pushes the bottom value minus the top value onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

unalloc CFLAG, #

Deallocates a block of memory of # size beginning at the data address atop the evaluation stack.

Forward-Critical: Pops the top value off the evaluation stack, which should be a data address, copies # words of data memory starting at that address to the save stack, then frees the data memory.

Forward-Noncritical: Pops the top value off the evaluation stack, which should be a data address, and frees # words of data memory starting at that address.

Reverse-Critical: Pops the top value off the save stack, which should be a data address, pushes it onto the evaluation stack and allocates # words of data memory starting at that location. # words are then moved from the save stack to this data memory.

Reverse-Noncritical: Allocates # words of data memory and pushes the address of the first word of allocated memory onto the evaluation stack.

uninst CFLAG, #

Dispose of an instance of variable register #.

Forward-Critical: Frees the memory occupied by the variable then pops the top data memory address off the variable register's stack and pushes it onto the save stack.

Forward-Noncritical: Frees the memory occupied by the variable then pops the top address off the variable register's stack.

Reverse-Critical: Pops the address off the save stack and pushes it onto the variable register's stack then reallocates enough data memory for the variable # starting at that address.

Reverse-Noncritical: Reallocates enough data memory for the variable # and pushes the address of the data memory allocated onto the variable register's stack.

unlink #

Disassociates a variable register from another.

Forward: Pops the top value of the variable stack pointed to by variable register # and pushes it onto the save stack.

Reverse: Pops the top value of the save stack and pushes it onto the variable stack pointed to by variable register #.

write CFLAG, TYPE

Displays a value for the user.

Forward-Critical: The top of the evaluation stack is popped and the value pushed onto the save stack. This value is then converted into a string and passed to a user interface function which takes appropriate action to display the value.

Forward-Noncritical: The top of the evaluation stack is popped and is converted into a string and passed to a user interface function to be displayed.

Reverse-Critical: The value on top of the save stack is popped and pushed onto the evaluation stack. Then a user interface function is called to handle undisplaying of the last value displayed.

Reverse-Noncritical: DUMMY is pushed onto the evaluation stack and then a user interface function is called to handle undisplaying of the last value displayed.

xor CFLAG, TYPE

Bitwise exclusive-or's the top two values of the evaluation stack and places the result onto the evaluation stack.

Forward-Critical: Pops the top two values of the evaluation stack, pushes the two values onto the save stack, and then pushes the bottom value bitwise exclusive or'ed with the top value onto the evaluation stack.

Forward-Noncritical: Pops the top two values of the evaluation stack and pushes the bottom value bitwise exclusive or'ed with the top value onto the evaluation stack.

Reverse-Critical: Pops the top value of the evaluation stack and discards it. Pops the top two values of the save stack and pushes them onto the evaluation stack.

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

APPENDIX B

THE E-MACHINE ADDRESSING MODES

This appendix, which is adapted from chapter 2 of Birch's thesis, describes the various addressing modes allowed in E-machine instructions. Quite a few modes are defined in order to accommodate standard high level language data structures more conveniently. Note that each addressing mode refers to either the data at the computed address or the computed address itself, depending on the instruction. That is, for those instructions that need a data value, such as **push**, the data value at the address computed from the addressing mode is used. For instructions that need an address, such as **pop**, the address that was computed from the addressing mode is used.

For each addressing mode listed below, an example of its intended use is given. Each example is given in pseudo assembly language form for clarity; it is important to remember that no assembler (and hence no assembly language) has yet been developed for the E-machine. However, the pseudo assembly language examples should be easily understood.

constant mode - C#

This mode is often called the immediate mode in other architectures; # is itself the integer, real, boolean, character, or address constant operand required in the instruction.

Example:

$A := 1.5;$

could be translated into:

```
push    R,C1.5          ; push 1.5
pop     c,R,V1          ; assign to A
```

variable mode - V#:

variable register # ⟶ top of variable stack ⟶ data memory

This mode accesses the data memory location given in the top element of the variable stack that is pointed to by variable register #. This mode is intended to address source program variables that are of one of the basic E-machine types.

Example:

$B := 1;$

could be translated into:

```
push    I,C1            ; push 1
pop     c,I,V3          ; assign to B
```

variable indirect - (V#):

variable register # ⟶ top of variable stack ⟶ data memory ⟶ data memory

This mode accesses the data in data memory whose location is stored at another data memory location, which is pointed to by the top of the variable stack pointed to by variable register #. This mode is intended for accessing the contents of a high level language pointer variables. It would be particularly useful for handling parameters in C which are passed as pointers for the intention of passing by reference.

Example:

```
int foo( C )
int *C
{
    *C = 1;
}
```

could be translated into:

```
label   c,5             ; procedure entry
inst    c,V3            ; create new instance of C
pop     c,A,V3          ; assign argument passed to *c
push    I,C1            ; push 1
pop     c,I,(V3)        ; assign to *c
uninst  c,V3            ; destroy instance of C
return                  ; return from call
```

variable offset mode - V#{offset}:

variable register # ⟶ top of variable stack + IR ⟶ data memory

This mode accesses the data pointed to by the top of the variable register # stack plus a byte offset which was previously loaded into the index register. This mode is useful for accessing fields in a structured data type such as a Pascal record or C struct.

Example:

A := D.Field2

could be translated into:

```
push    I,2             ; D is at offset of 2 in structure
popir   c               ; put offset into index register
push    R,V4{IR}        ; push D.Field2
pop     c,R,V1          ; assign to A
```

address indirect - (A):

address register ⟶ data memory

This mode provides access to data located at the data address in the address register. The address register must be loaded with a data memory address which points to data memory. This mode is useful for multiple indirection.

Example:

c = *(*g);

could be translated into:

```
loadar  c,V7            ; load addr reg with addr of g
loadar  c,(A)           ; load addr reg with addr of *g
push    I,(A)           ; push *(*g)
pop     c,I,V3          ; assign to c
```

address offset mode - A{offset}:

address register + IR ⟶ data memory

This mode provides access to structured data through the address register. The index register is added to the address register to provide an address to the data to be accessed. This mode is useful for indirection with structured data, such as pointers to records in Pascal.

Example:

I := H↑.Data

could be translated into:

```
push    A,V8          ; push H↑ (address value of H)
popar   c             ; load ar with H↑
push    I,C2          ; Data has offset of 2 in record
popir   c             ; load ir with offset
push    I,A{IR}       ; push H↑.Data
pop     c,I,V9        ; assign to I
```

variable indexed mode - V#[index]:

*variable register # ⟶ top of variable stack + IR * data size ⟶ data memory*

This address mode uses the top of the variable register # stack as a base address and adds the index register, which must be previously loaded, multiplied by the number of bytes occupied by the data type, which is a basic E-machine data type. The resulting address points to the data item. This mode is useful for accessing an array whose elements are of a basic E-machine data type.

Example:

B := L[3];

could be translated into:

```
push    n,I,3         ; put index of 3 into
popir   c             ; the index register
push    I,V12[IR]     ; push L[3]
pop     c,I,V2        ; assign to B
```

address indexed mode - A[index]:

*address register + IR * data size ⟶ data memory*

This mode provides the same function as variable indexed mode, except instead of a variable register providing the base address, the address register is loaded with the base address. This mode could be used for accessing elements of an array which is pointed to by a variable.

Example:

B := S↑[4];

could be translated into:

```
push    A,V19       ; put address of array into
popar   c           ; address register
push    I,4         ; put index of 4 into
popir   c           ; the index register
push    I,A[IR]     ; push S↑[4]
pop     c,I,V2      ; assign to B
```

APPENDIX C

A miniPASCAL COMPILATION EXAMPLE

This appendix provides an example showing the complete results of the compilation of a miniPascal program. The compilation was produced on a DOS machine. The example program, shown in figure 27, is referred to as program Samp3 throughout the remainder of this appendix. The numbers on the left refer to source program line numbers. The program, as shown in figure 27 (with the exception of the line numbers), is written to the SOURCESECTION portion of the E-machine object file (or E-code file). Program Samp3 contains several features that were not illustrated previously. These features include constant and type declarations, a record definition, a two dimensional array, and an array of records. The record definition, DRec, consists of two fields, one of which is a two-dimensional array of the previously defined Matrx type. An array of these records (DBase) is then declared, with an instance of such an array (Data) being declared in the variable declaration section of the main program. Another variable—also named Data—is declared in the formal parameter list of procedure InitD. In this case, Data is declared as only a single record of type DRec.

Program Samp3 also contains a situation in which a packet becomes fragmented (see chapter 4 for a discussion of the packet fragmentation problem). The fragmentation occurs in procedure InitD, which contains a nested for loop in which the inner for loop is a single statement within another conditionally executed statement.

The particular packet fragmentation situation found in program Samp3 is discussed later in this appendix.

Table 12 shows the array containing the program memory addresses corresponding to program Samp3's generated E-code label instructions. The column holding the label numbers (or label register numbers) is included for clarity—only the array of program memory addresses is actually written to the LABELSECTION portion of the E-code file.

Table 13 shows the array containing the data memory sizes reserved for program Samp3's variable registers. The columns holding the variable names and the variable register numbers are included for clarity—only the array of data memory sizes is actually written to the VARIABLESECTION portion of the E-code file. The variable registers whose corresponding names are blank are temporary registers needed to hold intermediate values. In this implementation, the data memory sizes are in terms of bytes; hence, the corresponding data memory size for a 32-bit integer (e.g., J) is 4. As can be seen in table 13, the full size of the array of records (variable Data represented by variable register number 3) is reserved for the array. The full size of a single record (40 bytes) is reserved for the record Data (variable register number 6) found in procedure InitD. Variable register number 18, representing a 2-byte temporary variable, holds the result of the `if` comparison found in function Fact.

Figure 28 shows the contents of program Samp3's string space array. In this example, the string literal, 'Sample Program' associated with the string constant Name, is entered into the string array. The string array is subsequently written to the STRINGSECTION portion of the E-code file.

Table 14 shows the Packet Table generated for program Samp3. The column holding the packet number is included for clarity—the remaining fields (columns) are written to the PACKETSECTION of the E-code file. As can be seen in table 14,

packet number 24 is a fragmented packet. This fragmentation situation is discussed later in this appendix. There are also two packets (numbers 36 and 43) whose execution should not result in changing the animation display. These two packets correspond to a return from a function call; this situation was discussed in the Parser module section of chapter 4.

Table 15 shows the Static Scope Table for program Samp3. The column holding the entry number is included for clarity—the remaining fields (columns) are written to the STATSCOPESECTION of the E-code file. Two previously unillustrated types of scope blocks are found in table 15. These are a record description scope block (entries 6–9) and an array index description scope block (entries 10–13).

As can be seen in table 15, two identifiers (entry 1 in procedure InitD's scope block and entry 21 in program Samp3's scope) both refer to the same child scope block, which is the record scope block describing a record of type DRec (entries 6–9). The compiler is able to determine that this record description scope block needs to be present only once (and possibly referenced multiple times) by querying the Scope Owner Table's Record Descriptor field, as discussed in the STATSCOPE module section of chapter 4.

Examine entry number 7 in table 15. This entry describes field A of a record of type DRec; field A is an array of type Matrx. The bounds of A's first index are included in entry number 7. The NxtIdx field of this entry holds the index of the first entry of the scope block describing A's second index (entries 10–13). Entry numbers 7 and 8, which describe the fields named A and B, also utilize the Offset field to denote the fields' offsets from the beginning of the record. Finally, note that the RecSiz field is utilized for a variable representing an array of records (e.g., entry number 21 describing the variable Data in program Samp3). This value is required by the animator for proper calculation of offsets when displaying values associated with arrays of records.

Figure 29 shows the pseudo assembly language representation of the E-code instructions generated for program Samp3. Figure 29 is formatted to enumerate the program's animation units, with translated E-code packets printed directly beneath corresponding animation units. Here again, the reader is reminded that the pseudo assembly language format is used for clarity—it is an array of C structures representing the E-code instruction stream that is actually written to the CODESECTION of the E-code file.

Figure 29 illustrates several situations that need to be discussed. First, examine E-code instruction numbers 3–11. Each name declared in the constant declaration section is assigned a variable register number. The constants' values are then stored in their corresponding variable registers. Thus, the compiler treats constants as though they were variable names in order to allow the animator access to their values at run time. Figure 30 shows a possible animation snapshot after the constant declarations have been executed (i.e., at this point the keyword TYPE will be highlighted, indicating that it is the next animation unit to be executed). It should be noted that as each of the subsequent type declarations are executed, the animator simply sequentially highlights the corresponding animation unit—there will be no corresponding data memory values added to the right-hand side of the display until variable names are actually declared. The nop instructions (numbers 12–15) serve as "dummy" instructions to allow the animator to highlight the appropriate animation unit.

As mentioned above, program Samp3 contains a situation in which a packet is fragmented. This packet, number 24, is the E-code translation of the animation unit

```
Data.A[I,J] := I + 101.33 * MultF;
```

This animation unit is part of a single for statement, thus illustrating the fragmentation of a packet resulting from a single for statement nested within a

conditionally executed statement, in this case another for statement. The fragmentation problem is manifested as follows. As the inner for loop is executed, the animator sequentially highlights the four animation units composing the inner for statement (i.e., the animation units translated by E-code packets numbered 21–24). The animator repeats this process upon each iteration of the inner loop. When the inner loop index eventually reaches its upper limit, the branch to label 7 (shown in instruction number 64) is taken. The E-code instruction defining label 7 (instruction number 117) is contained in packet number 24, which translates the above-mentioned animation unit. At this point, however, this animation unit should not be highlighted (since the instructions translating the assignment statement represented by the animation unit will not now be executed). The FragAddr field for packet number 24 in the Packet Table (shown in table 14) holds the value 117, indicating that packet number 24 is considered fragmented whenever control *branches into* the packet at (or beyond) instruction number 117. The animator queries the E-machine's program counter and packet register to determine which animation unit (if any) should be highlighted prior to the E-machine's execution of the corresponding packet. Thus, the animator, upon querying the E-machine's program counter (currently 117) and packet register (currently 24), determines that packet number 24 is fragmented at its current point of entry, instruction number 117. The animator must now retain its previous display while the E-machine executes instruction numbers 117–118. When the branch to label 2 (shown in instruction number 118) is accomplished, the E-machine returns control to the animator, which again queries the E-machine's program counter (currently 33) and packet register (currently 19). Since packet number 19 is not fragmented, the animator now highlights the animation unit corresponding to this packet,

```
I := 1 TO Rows
```

Finally, figures 31 and 32 show two possible animation screens occurring during the animation of program Samp3. Figure 31 shows an animation display that could occur immediately before procedure InitD is called from the main program (i.e., the animator is highlighting the animation unit `InitD(Data[Num],3);` while awaiting a response from the user). The dotted lines shown in the source program window indicate omitted source lines. Figure 32 shows an animation display that could occur immediately before a return is issued from procedure InitD (i.e., the animator is highlighting the animation unit `END;` in procedure InitD). Here again, the dotted lines indicate omitted source lines.

```
 0  Program Samp3;
 1  CONST
 2   Rows = 3;
 3   Cols = 3;
 4   Name = 'Sample Program';
 5  TYPE
 6   Matrx = ARRAY [1..Rows,1..Cols] of REAL;
 7   DRec = RECORD
 8     A:Matrx;
 9     B:INTEGER;
10     END; { DRec }
11   DBase = ARRAY [1..2] OF DRec;
12  VAR
13   Data:DBase;
14   Num,nFact:INTEGER;
15
16   Procedure InitD(VAR Data:DRec;
17                   MultF:INTEGER);
18     VAR
19      I,J:INTEGER;
20     BEGIN { Procedure InitD }
21      FOR I := 1 TO Rows DO
22       FOR J := 1 TO Cols DO
23         Data.A[I,J] := I + 101.33 * MultF;
24      Data.B := MultF;
25      END; { Procedure InitD }
26
27   Function Fact(n:INTEGER):INTEGER;
28     BEGIN { Function Fact }
29      IF n = 0
30        THEN Fact := 1
31        ELSE Fact := n * Fact(n-1)
32      END; { Function Fact }
33
34   BEGIN { Program Samp3 }
35    Num := 2;
36    InitD(Data[Num],3);
37    nFact := Fact(Data[Num].B);
38    END. { Program Samp3 }
```

Figure 27: The E-code SOURCESECTION for Program Samp3

Label Register Number	Label Program Address
L0	187
L1	21
L2	33
L3	42
L4	119
L5	51
L6	60
L7	117
L8	139
L9	145
L10	157
L11	179
L12	166
L13	208
L14	233

Table 12: The E-code LABELSECTION for Program Samp3

Variable Name	Variable Register Number	Variable Size
Rows	0	4
Cols	1	4
Name	2	4
Data	3	80
nFact	4	4
Num	5	4
Data	6	40
MultF	7	4
J	8	4
I	9	4
	10	4
	11	4
	12	4
	13	4
	14	4
	15	4
n	16	4
Fact	17	4
	18	2
	19	4
	20	4
	21	4
	22	4
	23	4
	24	4
	25	4
	26	4

Table 13: The E-code VARIABLESECTION for Program Samp3

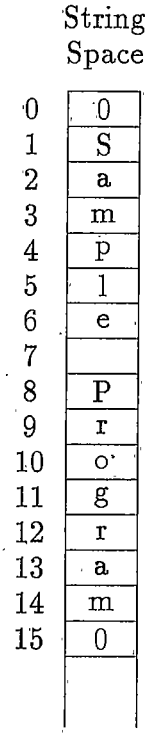

Figure 28: The E-code STRINGSECTION for Program Samp3

Packet Number	Start Addr	End Addr	Start Line	Start Col	End Line	End Col	Scope Index	Frag Addr	Display Packet
0	0	1	0	0	0	13	0	-1	TRUE
1	2	2	1	0	1	4	0	-1	TRUE
2	3	5	2	1	2	9	1	-1	TRUE
3	6	8	3	1	3	9	2	-1	TRUE
4	9	11	4	1	4	24	3	-1	TRUE
5	12	12	5	0	5	3	3	-1	TRUE
6	13	13	6	1	6	40	3	-1	TRUE
7	14	14	7	1	10	6	3	-1	TRUE
8	15	15	11	1	11	29	3	-1	TRUE
9	16	16	12	0	12	2	3	-1	TRUE
10	17	17	13	1	13	11	4	-1	TRUE
11	18	20	14	1	14	18	6	-1	TRUE
12	21	22	16	1	16	15	0	-1	TRUE
13	23	23	16	17	16	30	1	-1	TRUE
14	24	25	17	17	17	31	2	-1	TRUE
15	26	26	18	3	18	5	2	-1	TRUE
16	27	28	19	4	19	15	4	-1	TRUE
17	29	29	20	3	20	7	4	-1	TRUE
18	30	31	21	4	21	13	4	-1	TRUE
19	32	46	21	8	21	21	4	-1	TRUE
20	47	47	21	23	21	24	4	-1	TRUE
21	48	49	22	5	22	14	4	-1	TRUE
22	50	64	22	9	22	22	4	-1	TRUE
23	65	65	22	24	22	25	4	-1	TRUE
24	66	118	23	6	23	39	4	117	TRUE
25	119	130	24	4	24	19	4	-1	TRUE
26	131	138	25	4	25	7	7	-1	TRUE
27	139	140	27	1	27	13	0	-1	TRUE
28	141	142	27	15	27	24	1	-1	TRUE
29	143	143	27	25	27	33	2	-1	TRUE
30	144	144	28	3	28	7	2	-1	TRUE
31	145	152	29	4	29	11	2	-1	TRUE
32	153	153	30	6	30	9	2	-1	TRUE
33	154	156	30	11	30	19	2	-1	TRUE
34	157	158	31	6	31	9	2	-1	TRUE
35	159	165	31	23	31	31	2	-1	TRUE
36	166	168	-	-	-	-	2	-1	FALSE
37	169	178	31	11	31	31	2	-1	TRUE
38	179	186	32	4	32	7	8	-1	TRUE
39	187	188	34	1	34	5	8	-1	TRUE
40	189	190	35	2	35	10	8	-1	TRUE
41	191	207	36	2	36	20	8	-1	TRUE
42	208	232	37	11	37	27	8	-1	TRUE
43	233	235	-	-	-	-	8	-1	FALSE
44	236	237	37	2	37	28	8	-1	TRUE
45	238	250	38	2	38	5	8	-1	TRUE

Table 14: The E-code PACKETSECTION for Program Samp3

Entry	Id Name	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num
Scope block describing procedure InitD											
0		-	-	-	-	HEADER	-	17	-	-	-
1	Data	-	-	-	-	RECORD	-	-	6	6	-
2	MultF	-	-	-	-	INTEGER	-	-	-	7	-
3	I	-	-	-	-	INTEGER	-	-	-	9	-
4	J	-	-	-	-	INTEGER	-	-	-	8	-
5		-	-	-	-	END	-	-	-	-	-
Scope block describing record of type DRec											
6		-	-	-	-	HEADER	-	-	-	-	-
7	A	3	1	10	0	REAL	-	-	-	-	-
8	B	-	-	-	36	INTEGER	-	-	-	-	-
9		-	-	-	-	END	-	-	-	-	-
Scope block describing second index of array of type Matrx											
10		-	-	-	-	HEADER	-	-	-	-	-
11		3	1	-	-	-	-	-	-	-	-
12		-	-	-	-	END	-	-	-	-	-
Scope block describing function Fact											
13		-	-	-	-	HEADER	-	17	-	-	-
14	n	-	-	-	-	INTEGER	-	-	-	16	-
15	Fact	-	-	-	-	INTEGER	-	-	-	17	-
16		-	-	-	-	END	-	-	-	-	-
Scope block describing program Samp3											
17		-	-	-	-	HEADER	-	27	-	-	-
18	Rows	-	-	-	-	INTCONST	-	-	-	0	-
19	Cols	-	-	-	-	INTCONST	-	-	-	1	-
20	Name	-	-	-	-	STRINGCONST	-	-	-	2	-
21	Data	2	1	-	-	RECORD	40	-	6	3	-
22	Num	-	-	-	-	INTEGER	-	-	-	5	-
23	nFact	-	-	-	-	INTEGER	-	-	-	4	-
24	InitD	-	-	-	-	PROCEDURE	-	-	0	-	1
25	Fact	-	-	-	-	FUNCTION	-	-	13	-	2
26		-	-	-	-	END	-	-	-	-	-
Bootstrap scope block											
27		-	-	-	-	HEADER	-	-	-	-	-
28	Samp3	-	-	-	-	PROGRAM	-	-	17	-	0
29		-	-	-	-	END	-	-	-	-	-

Table 15: The E-code STATSCOPESECTION for Program Samp3

```
Pkt  Animation Unit
Num
        Instr   E-code
         Num    Instruction
 0   Program Samp3;
          0     pushd C28           ; Push program's Static Scope Table
                                    ; index onto dynamic scope stack
          1     nop

 1   CONST
          2     nop

 2    Rows = 3;
          3     inst c,V0           ; Create instance of Rows
          4     push I,C3           ; Store value of Rows in data memory
          5     pop c,I,V0

 3    Cols = 3;
          6     inst c,V1           ; Create instance of Cols
          7     push I,C3           ; Store value of Cols in data memory
          8     pop c,I,V1

 4    Name = 'Sample Program';
          9     inst c,V2           ; Create instance of Name
         10     push I,C1           ; Store Name's string space index in
         11     pop c,C,V2          ; corresponding variable register

 5   TYPE
         12     nop

 6    Matrx = ARRAY [1..Rows,1..Cols] OF REAL;
         13     nop

 7    DRec = RECORD
        A:Matrx;
        B:INTEGER;
        END;
         14     nop

 8    DBase = ARRAY [1..2] OF DRec;
         15     nop

 9   VAR
         16     nop

10    Data:DBase;
         17     inst c,V3           ; Create instance of Data
```

Figure 29: The E-code CODESECTION for Program Samp3

```
11    Num,nFact:INTEGER;
         18   inst c,V4          ; Create instance of nFact
         19   inst c,V5          ; Create instance of Num
         20   br 0               ; Branch to beginning of main program

12    Procedure InitD
         21   label 1            ; Enter Procedure InitD
         22   pushd C24          ; Push procedure's Static Scope Table
                                 ; index onto dynamic scope stack

13                (VAR Data:DRec;
         23   link V6            ; Link Data to actual param

14                   MultF:INTEGER);
         24   inst c,V7          ; Create instance of MultF
         25   pop c,I,V7         ; Put actual param into MultF

15       VAR
         26   nop

16       I,J:INTEGER;
         27   inst c,V8          ; Create instance of J
         28   inst c,V9          ; Create instance of I

17       BEGIN
         29   nop

18       FOR I := 1
         30   push I,C1          ; Initialize I with value of 1
         31   pop c,I,V9

19           I := 1 TO Rows
         32   br 3               ; Branch around MAXINT test and
                                 ; increment of I on first pass
                                 ; through the loop
         33   label 2            ; Test label of outer FOR loop
         34   push I,V9
         35   push I,C32767
         36   eql c,I            ; Test that I has not exceeded MAXINT
         37   brt c,4            ; If so, branch out of loop
         38   push I,V9
         39   push I,C1
         40   add c,I            ; Increment I
         41   pop c,I,V9
         42   label 3
         43   push I,V9
         44   push I,C3
         45   gtr c,I            ; Test for I reaching upper loop limit
         46   brt c,4            ; If so, branch out of loop
```

Figure 29 (continued)

```
20                              DO
        47   nop

21       FOR J := 1
        48   push I,C1          ; Initialize J with value of 1
        49   pop c,I,V8

22           J := 1 TO Cols
        50   br 6               ; Branch around MAXINT test and
                                ; increment of J on first pass
                                ; through the loop
        51   label 5            ; Test label of inner FOR loop
        52   push I,V8
        53   push I,C32767
        54   eql c,I            ; Test that J has not exceeded MAXINT
        55   brt c,7            ; If so, branch out of loop
        56   push I,V8
        57   push I,C1
        58   add c,I            ; Increment J
        59   pop c,I,V8
        60   label 6
        61   push I,V8
        62   push I,C3
        63   gtr c,I            ; Test for J reaching upper loop limit
        64   brt c,7            ; If so, branch out of loop

23                            DO
        65   nop

24         Data.A[I,J] := I + 101.33 * MultF;
        66   inst c,V10         ; Create instance of temporary
        67   push I,V9          ; variable (V10) and store value of
        68   pop c,I,V10        ; first index (I) in V10
        69   inst c,V11         ; Create instance of temporary
        70   push I,V8          ; variable (V11) and store value of
        71   pop c,I,V11        ; second index (J) in V11
        72   inst c,V12         ; Create instance of temporary
        73   push I,C0          ; variable (V12) and calculate the
        74   pop c,I,V12        ; final (lineal) array index value
        75   push I,V10         ; based on the values of the two
                                ; indices, I and J
        76   push I,C3
        77   mult c,I
        78   push I,V11
        79   add c,I
        80   pop c,I,V10
        81   push I,V10
        82   push I,C4
        83   sub c,I
        84   pop c,I,V10
        ( Packet number 24 continued on next page)
```

Figure 29 (continued)

```
(Continuation of packet number 24)
          85    push I,V10
          86    push I,C4
          87    mult c,I
          88    push I,V12
          89    add c,I             ; Store calculated value of final
          90    pop c,I,V12         ; index in V12
          91    push I,V12
          92    push I,C0
          93    add c,I             ; Convert index value in V12 to
          94    pop c,I,V12         ; offset value
          95    inst c,V13          ; Create instance of temporary
          96    push R,C101.33      ; variable (V13) to hold result
          97    push I,V7           ; of 101.33 * MultF
          98    cast c,I,R          ; Cast MultF to REAL
          99    mult c,R            ; 101.33 * MultF
         100    pop c,R,V13         ; Store multiplication result in V13
         101    inst c,V14          ; Create instance of temporary
         102    push I,V9           ; variable (V14) to hold result
         103    cast c,I,R          ; I + V13 and cast I to REAL
         104    push R,V13
         105    add c,R             ; I + V13
         106    pop c,R,V14         ; Store addition result in V14
         107    push R,V14
         108    push I,V12
         109    popir c             ; Put offset value in index reg
         110    pop c,R,V6{IR}      ; Put V14's value in Data.A[I,J]
         111    uninst c,V14        ; Delete instances of temporary
         112    uninst c,V13        ; variables created within the
         113    uninst c,V12        ; inner FOR loop
         114    uninst c,V11
         115    uninst c,V10
         116    br 5                ; Branch to test of inner FOR loop
         117    label 7             ; Branch out label of inner FOR loop
         118    br 2                ; Branch to test of outer FOR loop

25        Data.B := MultF;
         119    label 4             ; Branch out label of outer FOR loop
         120    inst c,V15          ; Create instance of a temporary
         121    push I,C0           ; variable (V15) to hold offset of
         122    pop c,I,V15         ; field B
         123    push I,V15          ; Calculate offset of field B
         124    push I,C36
         125    add c,I
         126    pop c,I,V15         ; Store offset of field B in V15
         127    push I,V7           ; Put MultF on evaluation stack
         128    push I,V15          ; Put offset of field B on eval stack
         129    popir c             ; Put offset of field B in index reg
         130    pop c,I,V6{IR}      ; Put MultF in Data.B
```

Figure 29 (continued)

```
26        END;
          131   nop
          132   uninst c,V15        ; Delete instance of temporary
                                    ; variable
          133   uninst c,V8         ; Delete instance of J
          134   uninst c,V9         ; Delete instance of I
          135   uninst c,V7         ; Delete instance of MultF
          136   unlink c,V6         ; Unlink Data
          137   popd                ; Pop procedure's Static Scope Table
                                    ; from the dynamic scope stack
          138   return              ; Return to calling scope

27    Function Fact
          139   label L8            ; Enter Function Fact
          140   pushd C25           ; Push function's Static Scope Table
                                    ; index onto dynamic scope stack

28                (n:INTEGER)
          141   inst c,V16          ; Create instance of n
          142   pop c,I,V16         ; Put actual param into n

29                                 :INTEGER;
          143   inst c,V17            ; Create instance of Fact (function's
                                      ; return value)

30        BEGIN
          144   nop

31         IF n = 0
          145   label L9
          146   inst c,V18            ; Create instance of temporary
          147   push I,V16            ; variable (V18) to hold comparison
          148   push I,C0             ; result
          149   eql c,I               ; Check for n = 0
          150   pop c,B,V18           ; Put comparison result in V18
          151   push B,V18
          152   brf c,10            ; If n not = 0, branch to ELSE

32          THEN
          153   nop

33                Fact := 1
          154   push I,C1         ; Put 1 in Fact
          155   pop c,I,V17
          156   br 11             ; Branch around ELSE

34          ELSE
          157   label 10          ; ELSE label
          158   nop
```

Figure 29 (continued)

```
35                                  Fact(n-1)
        159   inst c,V19            ; Create instance of temporary
        160   push I,V16            ; variable (V19) to hold n-1
        161   push I,C1
        162   sub c,I               ; Subtract 1 from n
        163   pop c,I,V19           ; Put n-1 in V19
        164   push I,V19            ; Push n-1 onto evaluation stack
        165   call 8                ; Call Fact

36
        166   label 12              ; Return from Fact
        167   inst c,V20            ; Create instance of temporary
        168   pop c,I,V20           ; variable (V20) to hold function
                                    ; value

37              Fact := n * Fact(n-1)
        169   inst c,V21            ; Create instance of temporary
        170   push I,V16            ; variable (V21) to hold n* Fact(n-1)
        171   push I,V20
        172   mult c,I
        173   pop c,I,V21           ; Put multiplication result in V21
        174   push I,V21
        175   pop c,I,V17           ; Put function value in Fact
        176   uninst c,V21          ; Delete instances of temporary
        177   uninst c,V20          ; variables created in ELSE clause
        178   uninst c,V19

38       END;
        179   label 11              ; Branch out label for ELSE
        180   nop
        181   push I,V17            ; Put function value on eval stack
                                    ; index from the dynamic scope stack
        182   uninst c,V18          ; Delete instance of temp variable
        183   uninst c,V17          ; Delete instance of Fact's result var
        184   uninst c,V16          ; Delete instance of n
        185   popd                  ; Pop function's Static Scope Table
        186   return                ; Return to calling scope

39    BEGIN
        187   label 0               ; Start label for main program
        188   nop

40     Num := 2;
        189   push I,C2             ; Put value of 2 in Num
        190   pop c,I,V5
```

Figure 29 (continued)

```
41      InitD(Data[Num],3);
         191    inst c,V22          ; Create instance of temporary
         192    push I,V5           ; variable (V22) and store value
         193    pop c,I,V22         ; of index (Num) in V22
         194    push I,V22          ; Calculate final (linear) array
         195    push I,C1           ; index
         196    sub c,I
         197    pop c,I,V22         ; Put final array index in V22
         198    push I,C3           ; Put 3 on evaluation stack
         199    inst c,V23          ; Create instance of temporary
         200    push I,V22          ; variable (V23) to hold offset of
         201    push I,C40          ; Data
         202    mult c,I            ; Calculate Data's offset and
         203    pop c,I,V23         ; put it in V23
         204    push I,V23
         205    popir c             ; Put Data's offset in index reg
         206    pusha V3{IR}        ; Put address of Data[Num] on
                                    ; eval stack
         207    call 1              ; Call InitD

42                Fact(Data[Num].B);
         208    label 13            ; Return from InitD
         209    inst c,V24          ; Create instance of temporary
         210    push I,V5           ; variable (V24) and store value
         211    pop c,I,V24         ; of index (Num) in V24
         212    inst c,V25          ; Create instance of temporary
         213    push I,C0           ; variable to hold calculated
         214    pop c,I,V25         ; offset of Data[Num].B
         215    push I,V24
         216    push I,C1
         217    sub c,I
         218    pop c,I,V24
         219    push I,V24
         220    push I,C40
         221    mult c,I
         222    push I,V25
         223    add c,I
         224    pop c,I,V25
         225    push I,V25
         226    push I,C36
         227    add c,I
         228    pop c,I,V25
         229    push I,V25          ; Put offset of Data[Num].B in
         230    popir c             ; index reg
         231    push I,V3{IR}       ; Put Data[Num].B on eval stack
         232    call 8              ; Call Fact
```

Figure 29 (continued)

```
43
        233   label 14              ; Return from Fact
        234   inst c,V26            ; Create instance of temp variable
        235   pop c,I,V26           ; (V26) to hold function value

44      nFact := Fact(Data[Num].B);
        236   push I,V26
        237   pop c,I,V4            ; Put value of Fact in nFact

45      END.
        238   nop
        239   uninst c,V26          ; Delete instances of temporary
        240   uninst c,V25          ; variables
        241   uninst c,V24
        242   uninst c,V23
        243   uninst c,V22
        244   uninst c,V4           ; Delete instance of nFact
        245   uninst c,V5           ; Delete instance of Num
        246   uninst c,V3           ; Delete instance of Data
        247   uninst c,V2           ; Delete instance of Name
        248   uninst c,V1           ; Delete instance of Cols
        249   uninst c,V0           ; Delete instance of Rows
        250   popd                  ; Pop program's Static Scope Table
                                    ; index from the dynamic scope stack
```

Figure 29 (continued)

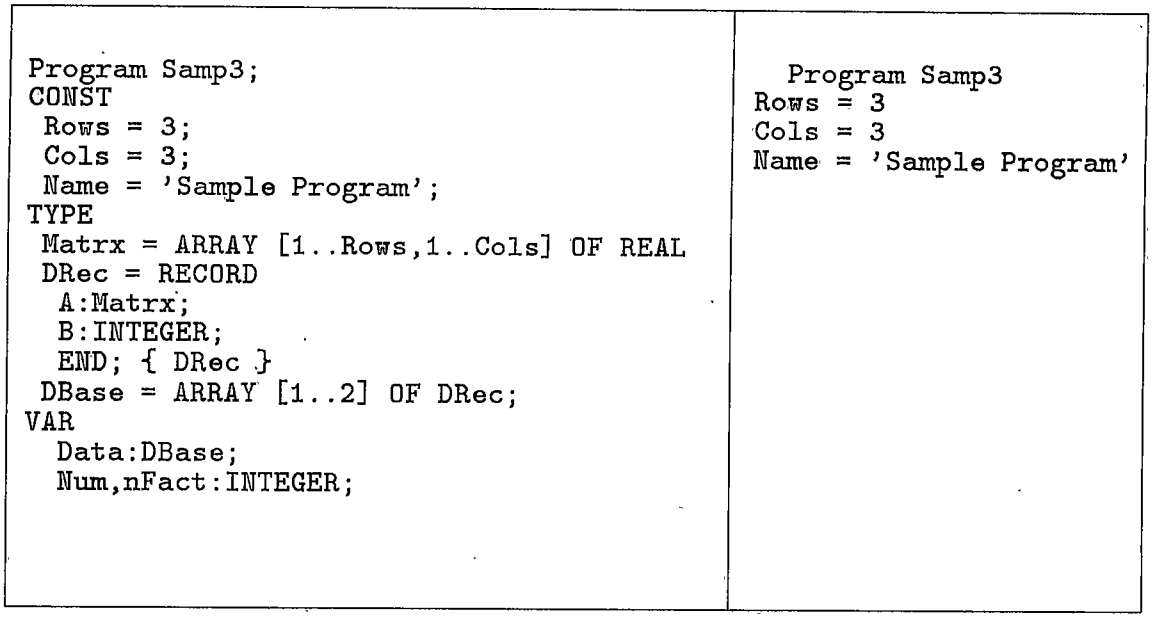

Figure 30: Animation Display After Constant Declarations in Program Samp3

```
Program Samp3;
CONST
 Rows = 3;
 Cols = 3;
 Name = 'Sample Program';
TYPE
 Matrx = ARRAY [1..Rows,1..Cols] OF REAL
 DRec = RECORD
  A:Matrx;
  B:INTEGER;
  END; { DRec }
 DBase = ARRAY [1..2] OF DRec;
VAR
  Data:DBase;
  Num,nFact:INTEGER;

   .........

BEGIN { Program Samp3 }
 Num := 2;
 InitD(Data[Num],3);

   .........
```

```
  Program Samp3
Rows = 3
Cols = 3
Name = 'Sample Program'
Data[1].A
  undef undef undef
  undef undef undef
  undef undef undef
Data[1].B is undefined
Data[2].A
  undef undef undef
  undef undef undef
  undef undef undef
Data[2].B is undefined
Num = 2
nFact is undefined
```

Figure 31: Animation Display Before Calling Procedure InitD in Program Samp3

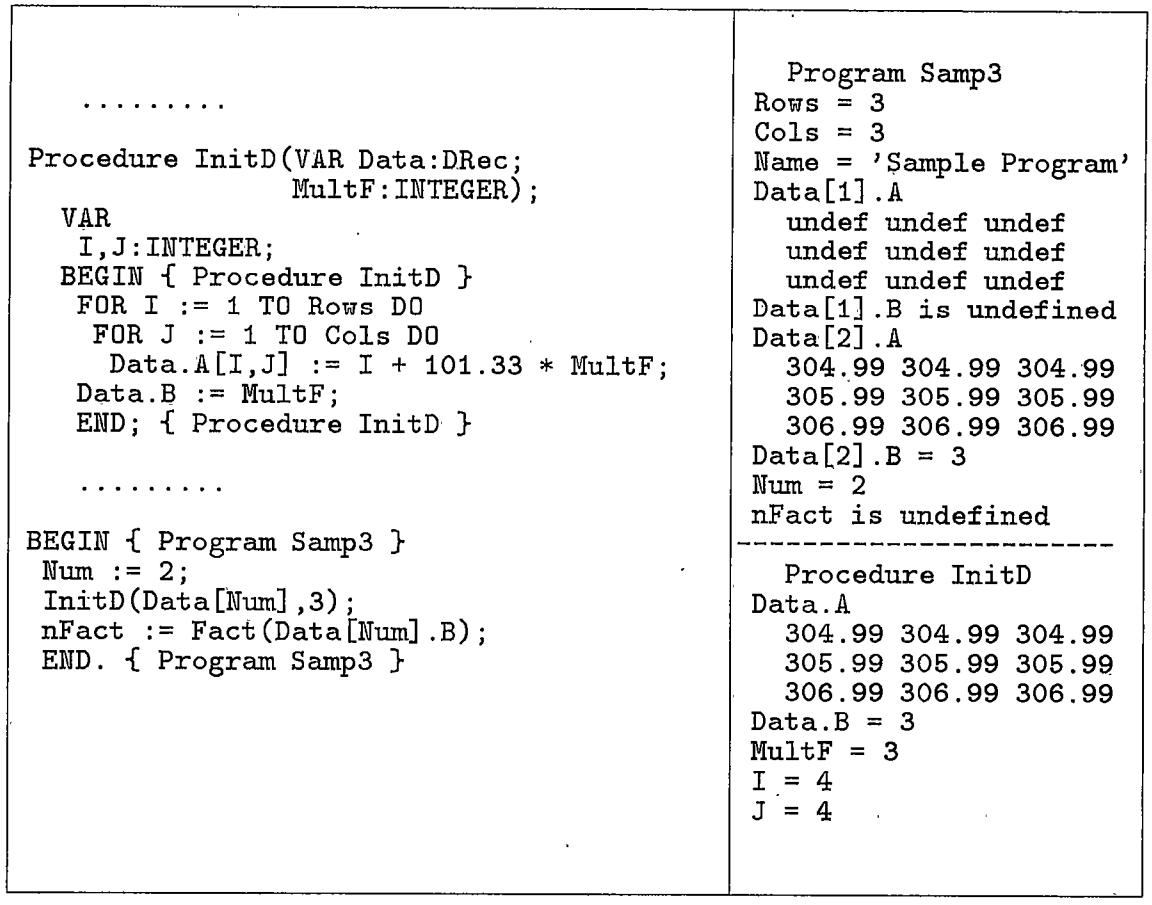

Figure 32: Animation Display at End of Procedure InitD in Program Samp3

MONTANA STATE UNIVERSITY LIBRARIES
3 1762 10200041 9

HOUCHEN
BINDERY LTD
UTICA/OMAHA
NE.