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

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

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

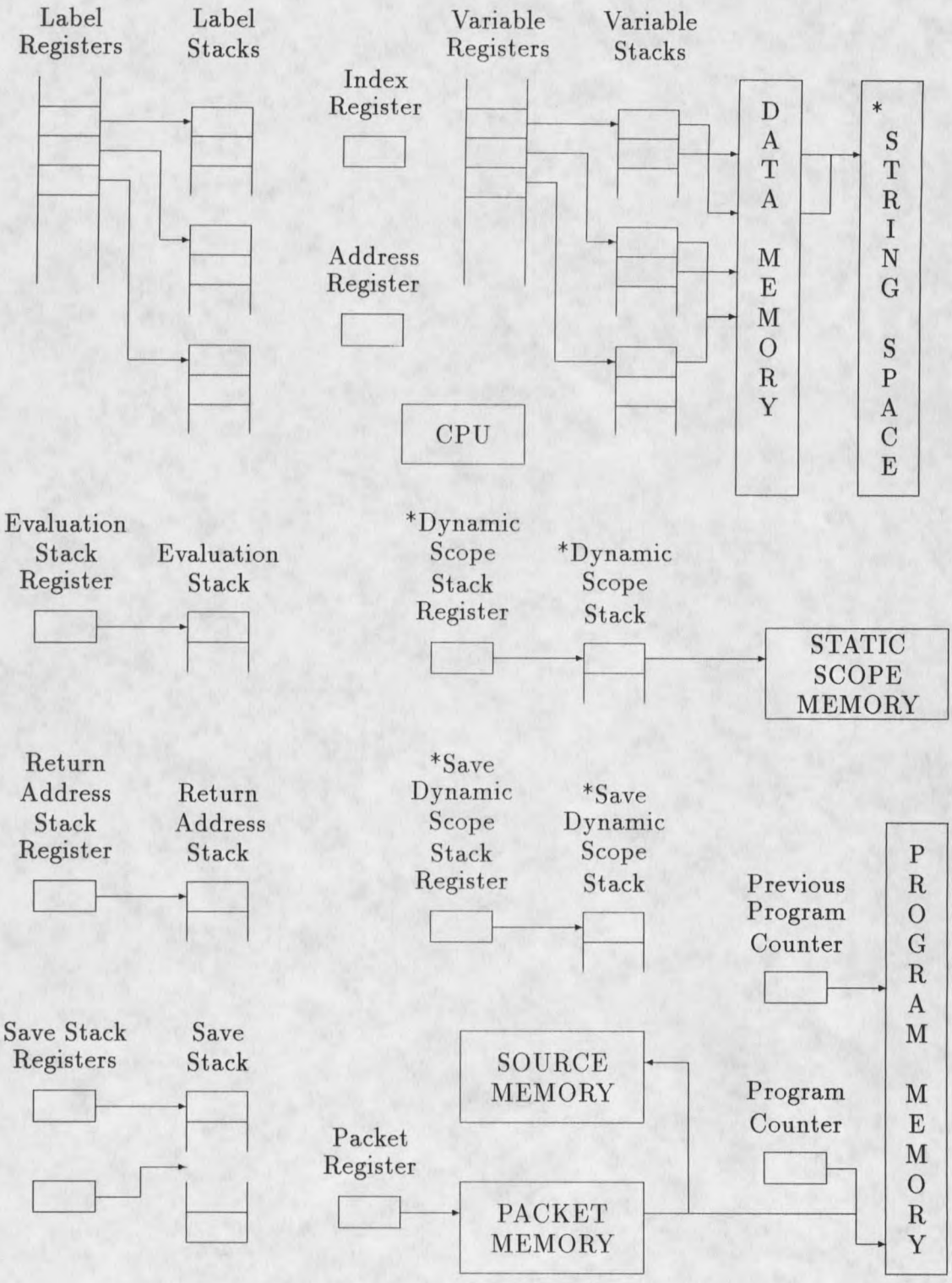

Figure 1: The E-machine

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.