A OSF/MOTIF PROGRAM ANIMATOR FOR THE DYNALAB SYSTEM

by

Craig Matthew Pratt

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

of

Master of Science

in

Computer Science

Montana State University
Bozeman, Montana

May 1995

APPROVAL

of a thesis submitted by

Craig Matthew Pratt

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.

Date	Chairperson, Graduate Committee

Approved for the Major Department

Date	Head, Major Department

Approved for the College of Graduate Studies

Date	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_____

Date_____

Acknowledgements

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 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. DYNAMOD 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 [Patton 89] and Michael Birch [Birch 90] laid the groundwork for this thesis by designing and implementing the underlying virtual machine for DYNALAB. Francis Goosey developed the first compiler (Pascal) for the E-Machine [Goosey 93]. David Poole developed the second compiler (Ada/CS) for the E-Machine [Poole 94]. As this thesis is being completed, Chris Boroni is developing the second DYNALAB animator, and Tory Eneboe is implementing a C compiler for the project.

I would like to take this opportunity to thank my graduate committee members, Dr. Rockford Ross, Dr. Gary Harkin, and Ray Babcock, and the rest of the Computer Science faculty for their knowledge, guidance, and care during my seven years at Montana State University. I would also like to thank especially my thesis advisor, Dr. Ross, and DYNALAB team members and friends, Frances Goosey, David Poole, Chris Boroni, and Tory Eneboe for making our group a team.

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

List of Tables

List of Figures

Abstract

This thesis is part of the fourth phase in the development of an interactive computer science laboratory environment called DYNALAB (an acronym for 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*. The third, ongoing phase of the DYNALAB project is the development of compilers generating E-Machine code. The first of these compilers is the Pascal compiler by Frances Goosey, described in her Master's thesis, *A miniPascal Compiler for the E-Machine* The second compiler is the Ada/CS compiler by David Poole, described in her Master's thesis, *An Ada/CS Compiler for the E-Machine*. The fourth phase of the project is the development of program animators. This thesis describes the concepts of DYNALAB program animation as well as the design and implementation of an OSF/Motif program animator.

DYNALAB program animation involves a variety of issues fundamental to computer science, some issues that apply to program animation in general, and a number of issues that apply to program animation in the DYNALAB and OSF/Motif environments. This thesis describes the current implementation of the E-Machine, the current object file format, the issues related to reversable program animation, and how they were addressed in the production of the current working version of the OSF/Motif DYNALAB program animator.

Chapter 1

Introduction

1.1 The DYNALAB System

This thesis is part of the fourth phase of the ongoing DYNALAB software development project. *DYNALAB* is an acronym for *DYNAmic LABoratory,* and its primary 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 [Ross 93] [BBGPPPR 95]. 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 repetitively animate 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 programs, the E-Machine had to be designed to be driven by an animator which controls the execution of programs, the corresponding animation, and the display of pertinent information dynamically. 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 is the design and implementation of compilers for the E-Machine. The first compiler—miniPascal, a subset of ISO Pascal—was created by Frances Goosey and described in her Master's thesis *A miniPascal Compiler for the E-Machine,* [Goosey 93]. During the development of this first compiler, changes to the E-Machine design and its emulator were required to facilitate proper program animation.

The second compiler, the Ada/CS compiler, was created by David Poole and described in his Master's thesis *An Ada/CS Compiler for the E-Machine,*

[Poole 94]. During the development of the Ada/CS compiler, the E-Machine and its emulator were further modified.

The fourth phase of the DYNALAB project is the design and implementation of program animators that drive the E-Machine and display programs in dynamic, animated fashion under control of the user. Preliminary text-based animators were implemented for the testing of the E-Machine emulator and the Pascal compiler. As with the first compiler the first official animator, which is described in this paper, uncovered issues not originally considered in the design and implementation of the E-Machine. The compilers, E-Machine, and the E-Machine emulator suffered numerous changes as these issues were resolved.

The second animator, running under Microsoft Windows, is far along in its development cycle. This, and subsequent animators, are anticipated to have little impact upon the E-Machine, its emulator, or the compilers.

This thesis does not conclude the DYNALAB project. A C compiler is in progress and a C++ compiler is in the planning stages. Work on the Ada/CS and Pascal compiler, and both animators will continue. 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 needed.

It's helpful to realize that each major component of the DYNALAB project is directly analogous to elements of a traditional computer system: The E-Machine corresponds to the CPU and memory of a traditional computer system. The compilers turn high-level languages into object files. And the animator is analogous to the operating system; it reads object files, uses the information therein to control the program's environment, runs the program,

control access to resources such as input and output streams, and handles abnormal execution conditions. The major difference between this environment and a traditional computing system is due to the fact that the E-Machine is a virtual machine and therefore the DYNALAB animation system is not dependent upon the actual system architecture. Secondly, the primary goal of each component is to provide an educational animation environment.

1.2 Preview

Chapter 1 presents an overview of the thesis and the DYNALAB project in general. A summary of the E-Machine and its emulator is given in Chapter 2. The changes and additions made to the E-Machine during animator development are noted in Chapter 2 as well. For a more detailed explanation of the E-Machine and its emulator, the reader is referred to the previous theses detailing the E-Machine [Patton 89] [Birch 90] [Goosey 93] [Poole 94]. Chapter 3 outlines the general operation of the Motif animator and its interaction with object code files, the E-Machine, and input/output streams. Chapter 4 details the logistics of program execution. To achieve the desired animation characteristics, actions must be performed in particular ways and in a particular order. This chapter describes the rules and a tested algorithm which performs this task.

Chapter 5 describes the design, operation, and use of the VarList routines and data structure. This complicated platform-independent package was written as a convenience for the animator writer. It greatly simplifies the task of displaying variables and their values. Chapter 6 goes into the details of the Motif animator itself. While the animator is written in C for compliance with the current version of Motif, its design and implementation follow an object-oriented philosophy.

Since this thesis is also intended to serve as a document describing the most recent E-Machine revision, Appendices A and B are included for completeness and are adapted from Birch's thesis. Appendix A describes the E-Machine instruction set and Appendix B lists the E-Machine addressing modes.

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], chapters 1, 2, and 3 of Birch's thesis [Birch 90], and chapters 2 and 3 of Goosey's thesis [Goosey 93]. This chapter is a summary and update of information from those three 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 function is to execute E-code programs, which generally consist of programs translated from high-level imperative languages. The first translator was written for the miniPascal language; Ada/CS is the second. 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 the "dynamic display system."

2.1 E-Machine Design Considerations

The fact that the E-Machine's sole purpose is to support program animation is 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 an associated 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 or a portion of a high level language construct, or *animation unit*. For example, an animation unit could be a complete high level language assignment statement such as

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

This animation unit would be translated into a packet of E-code instructions that perform the actual arithmetic operations and the assignment operation. In order for the animator to properly handle the proper animation activities, other information is associated with the packet. This information determines what effect execution of the packet will have upon the displayed variables of the translated source program, whether or not the packet corresponds with any visible source program region, the location of the region if any, whether the animator should pause after execution of the packet, and whether the variable display should be updated.

As another example, the conditional portion of an "if" statement in a high-level language could be translated into a packet. The 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 and translate them into corresponding E-code packets and also to properly generate the associated animation information. 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. This process will be described in more detail in chapter 4.

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, which is impractical for any non-trivial program.

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 non-critical. 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 *non-critical* 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 non-critical and their effects ignored during reverse execution. It is the compiler writer's responsibility to produce the correct E-code (involving critical and non-critical instructions) for reverse execution.

2.2 E-Machine Architecture

Figure 2.1 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

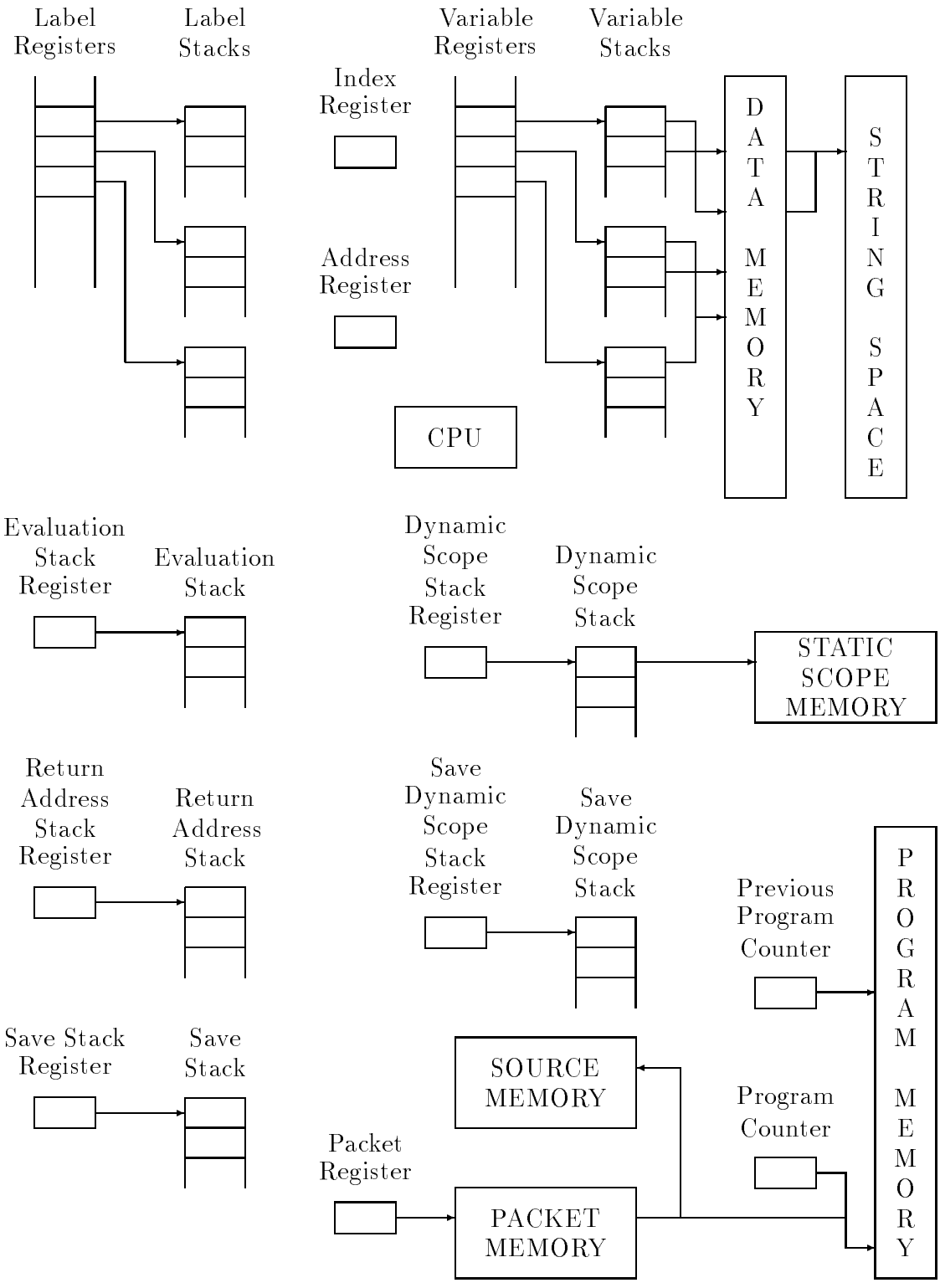


Figure 2.1: The E-Machine

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 source 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 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.

Source memory is an array of strings, 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.

Data memory contains the values of variable instantiations and dynamic storage allocation areas. Data memory actually consists of three structures: an area of data words containing the actual values, an area signifying which words are defined and which are undefined, and an area signifying which words have changed and which haven't changed recently. The last two structures are managed by the E-Machine emulator itself and are used by the animator (or

the VarList routines specifically) to determine the status of the data memory corresponding with variables. See chapter 5 for more information regarding these structures.

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. 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 its use 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 contains the values of all string literals and enumerated constant names encountered during the compilation of a 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.

The *label registers* are another unique component of the E-Machine re-

quired 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.

The operands and 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, it pops the top value from the return address stack into the program counter. 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 is about to be destroyed, as determined by the execution of a critical instruction, 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 the critical information from the top of the save stack. The *save stack register* points to the top of the save stack.

The *dynamic scope stack* allows the animator to determine all currently active scopes for memory display. 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) is obviously not available in the static scope memory.

To keep track of the set of active scopes at any point during program execution, the dynamic scope stack provides the dynamic chain as found in the run time stack of activation records generated by most conventional compilers. 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. 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, the CPU is what executes E-Machine instructions. It is the E-Machine emulator originally programmed by Birch and is described in the next section.

2.3 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 C for portability and has been compiled on a wide variety of environments and compilers including MS-DOS-, OS/2- and Unix-based IBM PCs, Silicon Graphics IRIX, DEC Alpha OSF/1, DECStation Ultrix, and MicroVAX Ultrix.

Within the complete DYNALAB environment, the emulator acts as a slave to the program animator, executing a packet of E-code instructions upon each call.

2.3.1 The E-Machine Emulator API

The program animator controls the E-Machine emulator through a defined set of calls. These calls allow the animator to initialize, control, and examine the state of the emulator. The C Application Program Interface (API) to the E-Machine emulator consists of the following routines:

- **FaultType getfault()** – Return current fault status (see description of **FaultType** below)
- **char * faultmsg(FaultType SourceFault)** – Return a pointer to the message associated with **SourceFault**.
- **LoadObjfile(char *FileName)** – Load the specified object file into the E-Machine emulator and reset the state of all the E-Machine components to their initial state. If **getfault()** returns anything except **NOFAULT**, the load was unsuccessful and the state of the E-Machine is undefined until a **LoadObjFile** is performed successfully.
- **getcurrpacket(Packet *Target)** – Copy the next packet descriptor to be executed into ***Target**

- **executepacket()** – Execute the next packet. A fault is raised if an error condition occurs during execution. Depending upon the fault type, the state of the emulator may be undefined.

- **reverse()** – Change the direction of execution. Note that the packet next to execute will be different than before the **reverse()**.

- **getlastaddr(ProgAddress *Target)** – Assign ***Target** the address of the last program instruction in program memory

- **Boolean PauseDir(PktDirectives Source)** – Return **TRUE** if packet directive **Source** contains a **DIR_PAUSE** directive. The meaning of these directives can be found in the section detailing the packet section of the object file.

- **Boolean ShowSourceDir(PktDirectives Source)** – Return **TRUE** if packet directive **Source** contains a **DIR_SHOWSOURCE** directive.

- **Boolean VarUpdateDir(PktDirectives Source)** – Return **TRUE** if packet directive **Source** contains a **DIR_UPDATEVARS** directive.

(Note: The function of the following calls are generally limited to accessing the names and values of variables. The VarList routines remove much of their usage for animators.)

- **getvaraddress (VariableReg Source, DataAddress *Target)** – Assign **Target** the data memory location corresponding to variable register **Source**.

- **getprevvaraddress(VariableReg Source, IntegerType Depth, DataAddress *Target)** – Assign **Target** the data memory location corresponding to the **Depth**-th last instantiation of variable register **Source**.

- **getvarsize(VariableReg Source, IntegerType *Target)** – Assign **Target** the size of a instantiation of a variable **Source**.

- **getstatscopeentry(int Source, ST *Target)** – Copy static scope entry **Source** into **Target**.

- **getnumprocs()** – Return the number of subroutines defined in the program (functions+procedures+programs).

- **getdynamlevel(int *Target)** – Assign **Target** the current number of dynamic levels. This number generally corresponds to the number of subroutine calls.

- **datasize(DataType Source)** – Returns the size of data type **Source** in bytes.
- **readdata (DataType SourceType, DataAddress Source, DataValue *Target, DataWord *TargetDefined)** – Reads data type **SourceType** from data memory at location **Source** and copy the contents to **Target** and its corresponding define bits to **TargetDefined**.
- **writedata(DataType TargetType, DataAddress Target, DataValue Source)** – Writes **Source** of type **TargetType** to data memory location **Target**.
- **EMClearChangeRecord()** – Resets the change record, setting all data locations as "unchanged".
- **int EMCheckChangeRecord (DataType SourceType, DataAddress Source)** – Returns 0 if the variable at data memory location **Source** of type **SourceType** is unchanged since the last **EMClearChangeRecord** and non-0 if it has changed.
- **char stringchar(int Source)** – Returns the character at position **Source** within the string space

The majority of the E-Machine emulator API calls are related to accessing data memory. This is by far the most complicated interaction between the animator and the emulator. This prompted the development of the animator-independent VarList routines discussed in chapter 5. How these API calls are used to read the current values of the program variables is discussed therein. The remaining calls and their use in controlling the animation are discussed in chapter 4.

2.3.2 Relevant E-Machine Emulator Data Types

The following type definitions are relevant to the API:

- **DataType** – Set of enumerated types designating the fundamental E-Machine data type elements. Valid enumerations are: **BOOLEAN, INTEGER, REAL, ADDRESS, CHARACTER**.

- **FaultType** – Enumerated type which designates the type of the last E-Machine error. Valid enumerations are: **NOFAULT, EVALOVRFLW, EVALEMPTY, CALLOVRFLW, CALLEMPTY, ILLEGALINSTR, ILLEGALTYPE, BADVARREG, BADLABELREG, BADPROGADDR, BADDATAADDR, BADFILE, OUTMEM, VARNOTALLOC, ILLEGALFLAG, SAVEUNDRFLW, UNDEFDATA, ILLEGALMODE, DIVBYZERO, MEMALLOC, DYNAMOVRFLW, DYNAMEMPTY, SAVDYNAMOVRFLW, SAVDYNAMEMPTY, FREEMEMOVRFLW, FILENOTOPEN, NOMAGICNUM, NOHEADER, MULTSTRING, MULTCODE, MULTVAR, MULTLAB, MULTSOURCE, MULTSTATSCOP, MULTPACKET, BADSECT, NOCODE, NOVAR, NOLAB, NOSTATSCOP, NOPACKET, NOSOURCE, NOSTRING, BADLABELS, BADVARS, BADSTRINGS, BADPACKET, BADSOURCE, BADSTATSCOPE, BADCODE, EF_BADLOGNUM, EF_BADROOTNUM**.

- **Packet** – Structure describing a single E-Machine packet and consisting of the following fields:

  1. **ProgAddress startaddr** – Starting address of the E-Machine instruction the packet references in source memory
  2. **ProgAddress endaddr** – Ending address of the E-Machine instruction the packet references in source memory
  3. **int startline** – Starting line number of source code the packet references in source memory
  4. **int startcol** – Starting column number of the source code the packet references in source memory
  5. **int endline** – Ending line number of the source code the packet references in source memory
  6. **int endcol** – Ending column number of the source code the packet references in source memory
  7. **StatScopeEntry scope** – Number of entries from the top of the static scope block to display. Basically, this number tells the animator how many entries to look at in the current scope from the top. This determines which variables should be shown as "declared."
  8. **PktDirectives forward_directives** – Directives to apply when running in the forward direction. This field is a bitmap composed of a logical union (*or*) of any of the following symbols:
     (a) **DIR_UPDATEVARS** – The animator should update the variable display *after* forward execution of this packet

(b) **DIR_PAUSE** – The animator should pause for user interaction before execution of this packet if running in an incremental execution mode

(c) **DIR_SHOWSOURCE** – The animator should highlight the source code associated with this packet before execution

9. **PktDirectives reverse_directives** – Directives to apply when running in the reverse direction. This field is a bitmap composed by a logical union (*or*) of any of the following symbols:

   (a) **DIR_UPDATEVARS** – The animator should update the variable display *before* reverse execution (unexecution) of this packet

   (b) **DIR_PAUSE** – The animator should pause for user interaction before execution of this packet if running in an incremental execution mode

   (c) **DIR_SHOWSOURCE** – The animator should highlight the source code associated with this packet before execution

10. **IntegerType TestResultVar** – If positive, this field defines the variable register to examine for the intermediate result of an expression resulting in a True or False result. If defined, the animator should display the contents of this register after execution of this packet when running forward. In reverse execution, the animator is expected to ignore this field.

- **ST** – The **ST** structure defines an element of the *Static Scope Table*. As a whole, the static scope table describes every data structure in the source language program and the location of the elements in data memory. How this is done is described in chapter 5. The **ST** type contains the following elements:

  1. **char name[ ]** – The element name, if any
  2. **StatScopeEntryType type** – The type of the element. **StatScopeEntryType** is defined to be one of: **PROCEDURE, FUNCTION, INTFUNCTION, REALFUNCTION, BOOLFUNCTION, CHARFUNCTION, ENUMFUNCTION, PTRFUNCTION, HEADER, END, RECORD, STRING, INTCONST, REALCONST, BOOLCONST, CHARCONST, STRINGCONST, ENUMCONST, PTRCONST, ENUMINT**.
  3. **IntegerType upperbound** – Upper value of index if the element is a static array or the register that *contains* the index if the element describes a dynamic array.

4. **IntegerType lowerbound** – Lower value of the index if the element is a static array or the register that *contains* the index if the element describes a dynamic array.
5. **int nextindex** – If positive, this field is used as the index of the static scope table entry that describes the next dimension in multi-dimensional arrays.
6. **IntegerType offset** – For elements of aggregate data types, such as record fields, the **offset** field stores the offset of the field from the beginning of the record.
7. **IntegerType VarSize** – Size of the element in bytes
8. **int parent** – If positive, this field is used as the index of the **HEADER** element of the enclosing scope block
9. **int child** – If positive, this field is used as the index of the *child* scope block.
10. **VariableReg varreg** – If positive, the **varreg** field designates the variable register that contains the address of the element in data memory.
11. **int ProcNum** – Unique procedure identifier. This identifier is used to identify procedures on the dynamic scope stack.
12. **int IndexType** – The index type if the element describes an array dimension
13. **StatScopeArrayType ArrayType** – The type of array, either **STATIC** or **DYNAMIC**. This controls how the index fields are interpreted.

2.4 The E-Machine Object File

The E-Machine emulator defines the object file format that must be generated by a compiler. A single E-code object file ready for execution on the E-Machine consists of eight sections, seven of which may occur in any order. The file is stored using 7-bit ASCII and is formatted to be human-readable to facilitate debugging.

Each section of an object code file 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 eight 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 HEADERSECTION, which is loaded into animator memory
- 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.

2.4.1 The HEADERSECTION

The HEADERSECTION is a repository for specific information about the program, such as the E-Machine version number and the compiler version number with which the program was compiled, as well as general information about the program itself (e.g. a description of the program such as "this program illustrates a linked list"). The HEADERSECTION is not yet fully implemented and new elements will find their way into this section as time goes on. The HEADERSECTION must be the first section in the object file.

2.4.2 The CODESECTION

The CODESECTION contains the translated program—the E-code instruction stream. Even though the instruction stream can be thought of as a 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:

- The instruction number followed by a colon
- An operation code (e.g., push or pop)
- The instruction mode (critical or non-critical)
- The data type of the operand (e.g., I indicates INTEGER)
- Either a numeric data value or an addressing mode

2.4.3 The PACKETSECTION

The PACKETSECTION consists of lines which are used to define the 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 line of the packet section contains the following information:

- The packet number followed by a colon
- 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
- The forward and reverse directives (in hexadecimal notation)
- a variable register number that will hold the result of the execution of a conditional expression.

2.4.4 The VARIABLESECTION

The VARIABLESECTION consists of lines which are used to define the variable registers used by the compiled program. A variable register definition consists of a variable number field followed by a colon and the size of the data represented by the register. This information is used to initialize size information held in the E-Machine's variable registers.

2.4.5 The LABELSECTION

The LABELSECTION consists of lines used to define label structures. A label definition consists of a label number followed by a colon and 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.

2.4.6 The SOURCESECTION

The SOURCESECTION consists of lines defining the source code of the compiled program. Each line consists of a line number followed by a colon and the source code for that line. This section is referenced exclusively by the animator for display purposes.

2.4.7 The STATSCOPESECTION

The STATSCOPESECTION was originally named the SYMBOLSECTION in Birch's thesis. It contains lines which are used to define the static scope table. The Static Scope Table (called the "symbol table" in Birch's thesis) 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 a compiler's symbol table. The STATSCOPESECTION records are loaded into the E-Machine's static scope memory at run time.

Each line of the STATSCOPESECTION contains the following information:

- The entry number followed by a colon
- Upper and lower bounds (for array variables)
- The entry number of the Static Scope Table entry containing the next array index bounds (for multidimensional arrays)
- The offset value (for aggregate types)
- An integer representation of the enumerated value indicating the data type
- The size of the variable in bytes
- The entry number of the entry's parent Static Scope Entry
- The entry number of 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)
- The variable register number that contains the address of the variable's value
- The procedure identifier number (used for dynamic scoping)
- An integer value corresponding to an enumerated type denoting whether a variable name is an array, and if so, whether it is static or dynamic
- A value describing the index type of an array variable (e.g. integer, enumerated, or character).
- The name of the identifier being described (e.g., a variable name or a procedure name)

2.4.8 The STRINGSECTION

The STRINGSECTION contains the values of string literals and enumerated constant names. The contents of the STRINGSECTION are loaded into the E-Machine's string space at run time. The animator retrieves the values of string constants from the string space.

2.4.9 The E-Machine Object File API

The DYNALAB compilers create object files and the E-Machine emulator reads object files using a common library of routines. This allows the format of the object file to change without requiring recompilation of the compilers and the E-Machine emulator.

Each object file section has a corresponding write and read routine to generate and parse the section, respectively. There is no required order for the sections except the header section, which must come first. The C object file API calls are as follows:

- **BooleanType open_code_file(char *FileName)** – Opens the object file given by FileName. Any subsequent reads reference this file. If the open is successful, **open_code_file** returns **TRUE**, otherwise it returns **FALSE**.
- **void close_code_file(void)** – Closes file opened with **open_code_file**.
- **int write_code(FILE * TargetFile, int NumInstructions, Instruction * Source)** – Writes a code section out to a file. **Source** is the base of the instruction array to write, **TargetFile** is the file to write the section into, and **NumInstructions** is the number of instructions to write. **write_code** returns 1 if successful.
- **int read_code(int NumInstructions, Instruction ** Target)** – Reads the code section into E-Machine memory. **NumInstructions** is the number of instructions to read and **Target** is a pointer to an array of instruction structures (with enough room for **NumInstructions** elements) where the read instructions are stored. **read_code** returns 1 if successful.

- **void write_labels(FILE * TargetFile, int NumLabels, LabelReg * Source)** – Writes a label section out to a file. **Source** is the base of the label array to write, **TargetFile** is the file to write the label section into, and **NumLabels** is the number of labels to write.

- **BooleanType read_labels(int NumLabels, LR ** Target)** – Reads the label section into E-Machine memory. **NumLabels** is the number of labels to read and **Target** is a pointer to an array of label register structures (with enough room for **NumLabels** elements) where the read label parameters are stored. **read_labels** returns **TRUE** if successful.

- **void write_packets(FILE * TargetFile, int NumPackets, Packet * Source)** – Writes a packet section out to a file. **Source** is the base of the packet array to write, **TargetFile** is the file to write the packet section into, and **NumLabels** is the number of labels to write.

- **BooleanType read_packets(int NumPackets, Packet ** Target, ProgAddress * LastAddress)** – Reads the packet section into E-Machine memory. **NumPackets** is the number of packets to read and **Target** is a pointer to an array of packet structures (with enough room for **NumPackets** elements) where the packets are stored. **LastAddress** is assigned the highest numbered address referenced by a packet. **read_packets** returns **TRUE** if successful.

- **void write_source(FILE * TargetFile, int NumSourceLines, char ** Source)** – Writes a source section out to a file. **Source** is a pointer to an array of string pointers to write, **TargetFile** is the file to write the source section into, and **NumSourceLines** is the number of source lines to write.

- **BooleanType read_source(int NumsourceLines, char ** Target)** – Reads the source section into E-Machine memory. **NumSourceLines** is the number of source lines to read and **Target** is a pointer to an array of character pointers (with enough pointers for **NumSourceLines** elements) where pointers to the strings (which are **malloc**ed by **read_source**) are stored. **read_source** returns **TRUE** if successful.

- **void write_statscope(FILE * TargetFile, int NumEntries, ST * Source)** – Writes a static scope section out to a file. **Source** is the base of the static scope structure array to write, **TargetFile** is the file to write the static scope section into, and **NumEntries** is the number of entries to write.

- **BooleanType read_statscope(int NumEntries, ST ** Target, * int NumProcs)** – Reads the static scope section into E-Machine * memory. **NumEntries** is the number of static scope * structures to read and **Target** is a pointer to an array of * packet structures (with enough room for **NumElements** * elements) where the packets are stored. **NumProcs** is * assigned the number of **PROCEDURE** and **FUNCTION** entry * types seen. **read_statscope** returns **TRUE** if * successful.

- **void write_strings(FILE * TargetFile, int Length, char * Source)** – Writes a string section out to a file. **Source** is a pointer to an array of characters to write, **TargetFile** is the file to write the string section into, and **Length** is the number of characters in the string section.

- **BooleanType read_strings(int Length, char * Target)** – Reads the string section into E-Machine memory. **Length** is the number of characters to read and **Target** is a pointer to an array of characters (with enough room for **Length** elements) where the characters are stored. **read_strings** returns **TRUE** if successful.

- **void write_variables(FILE * TargetFile, int NumElements, VariableReg * Source)** – Writes a variable section out to a file. **Source** is the base of the variable array to write, **TargetFile** is the file to write the variable section into, and **NumElements** is the number of variable elements to write.

- **BooleanType read_variables(int NumElements, VR ** Target)** – Reads the variable section into E-Machine memory. **NumElements** is the number of variable elements to read and **Target** is a pointer to an array of variable register structures (with enough room for **NumElements** elements) where the read variable parameters are stored. **read_variables** returns **TRUE** if successful.

The **read** routines are used by the E-Machine emulator to load the sections into its internal structures. But the animator and other utility programs may use these routines to access the contents of code files. Specifically, the **HEADERSECTION**, when fully implemented, should be read by E-Machine object file browsers and the like to obtain detailed information about the object file. This information includes the author, a short description, a long description, date of creation, and other information as deemed useful.

2.4.10 Relevant Object File Data Types

Since the E-Machine emulator already has a representation for the structures used in the object file API, the object file API simply uses those definitions instead of re-inventing them. The following type definitions are relevant to the object file API:

- **DataType** – Set of enumerated types designating the fundamental E-Machine data type elements. Valid enumerations are: **BOOLEAN, INTEGER, REAL, ADDRESS, CHARACTER**.
- **Instruction** – Structure containing the information for a single E-Machine instruction. It contains the following fields:
  1. **Opcode opcode** – Operator code for instruction. This is one of **PUSH, PUSHA, POP, POPIR, POPAR, LOADIR, LOADAR, ADD, SUB, MULT, DIV, NEG, AND, OR, XOR, NOT, SHL, SHR, MOD, CAST, LABEL, BR, BRT, BRF, EQL, NEQL, LESS, LEQL, GTR, GEQL, CALL, RETURN, ALLOC, UNALLOC, INST, UNINST, LINK, UNLINK, NOP, PUSHD, POPD, OPEN, READ, WRITE, CLOSE, PUSHAR, OP_ATAN, OP_COS, OP_EXP, OP_LN, OP_SIN, OP_SQRT**. See Appendix A for a description of the E-Machine instructions.
  2. **ModeType mode** – The *mode* of an instruction is either **CRITICAL** or **NONCRITICAL**. See the description earlier in this chapter.
  3. **unsigned char type** – The type of the operand (see description of **DataType** above)
  4. **DataValue data** – The operand for the instruction. A **DataValue** can be any of **AddressType**, **BooleanType**, **CharacterType**, **IntegerType**, or **RealType**.
  5. **int addrmode** – The addressing mode for the instruction. This may be a logical union between **IMMEDIATE**, **VARIABLE**, **ADDRREG**, **INDEXED**, **OFFSET**, **INDIRECT**, and/or **INDEXFIRST**. Only certain combinations are valid. See [Goosey 93] and [Poole 94] for details on their use.
- **LabelReg** – Integer referring to a particular E-Machine label.
- **LR** – A structure containing information about a label register. It consists of the following fields:

1. **ProgAddress address** – The address in code memory corresponding to the label.
2. **LabelStack * stack** – A pointer to the label stack for this label. It is necessary to store each branch to a label for reverse execution.

- **Packet** – See description above.
- **ST** – See description above.
- **VariableReg** – Integer designating a variable register in the E-Machine.
- **VR** – A structure containing information pertaining to a particular variable register. It consists of the following fields:

  1. **IntegerType size** – The size of the element stored in the variable register
  2. **VarStack * stack** – A pointer to the variable stack for this register. This is used for multiple instantiations of the same variable.

Chapter 3

The OSF/Motif DYNALAB Animator

This chapter is included to provide a description of the functionality of Version 1 Release 1 of the Motif-based DYNALAB Program Animator. The details of the implementation can be found in chapters 4 and 6.

The program animator's purpose is to illustrate the run-time behavior of programs compiled with one of the DYNALAB compilers. The DYNALAB compilers generate DYNALAB animation object files which describe both the static and dynamic nature of the source program. As outlined in chapter 2, an object file contains E-Machine assembly instructions that implement the program, information about the variables declared in the program, the original source code, a textual description of the program, and other information the E-Machine and animator require to properly animate the program.

The purpose of the DYNALAB project is to help students understand the complexities of programming. The philosophy behind the design of the animator interface gives highest priority to this fact. The Motif DYNALAB animator is designed to add as little complexity as possible to the task of analyzing a program. To this end, each element of the animator and how it functions is intended to clarify and simplify the understanding of the animated

program more than it complicates the use of the animator itself. How this is realized for each element is described below.

A typical use of the animator by a student in a lab environment would involve him or her performing the following:

- Retrieving a pre-constructed or compiled program into the animator from an on-line library of programs or constructing a source program and translating the source program using one of the DYNALAB compilers into a DYNALAB object code file
- Forward and reverse executing through the program to study new programming concepts in action
- Performing a time complexity analysis by entering various values and running the program or a portion of the program to acquire execution cost values.

Figure 3.1 depicts a typical animation session. The goal of the animation window is to be simple and intuitive. The primary elements of the animation window are:

A The **File** menu allows the user to load programs from a library into the animator, reset the current animation, and exit from the animator

B The *source area* contains the source code for the animated program. The highlighted region designates the source that is about to be executed or unexecuted.

C The execution control buttons allow for forward execution and advancement (depending upon the mode) or the reverse execution (unexecution) of the currently highlighted source.

D The *variable display* area contains the current call stack and the values of all variables in each active routine. Variables that have changed since the last execution step are highlighted.

E The *execution mode selector* controls how execution is performed. The modes and their operation are detailed later in the chapter.

F The *input/output (IO) area* lists the output by the program at its current state as well any input read by the program.

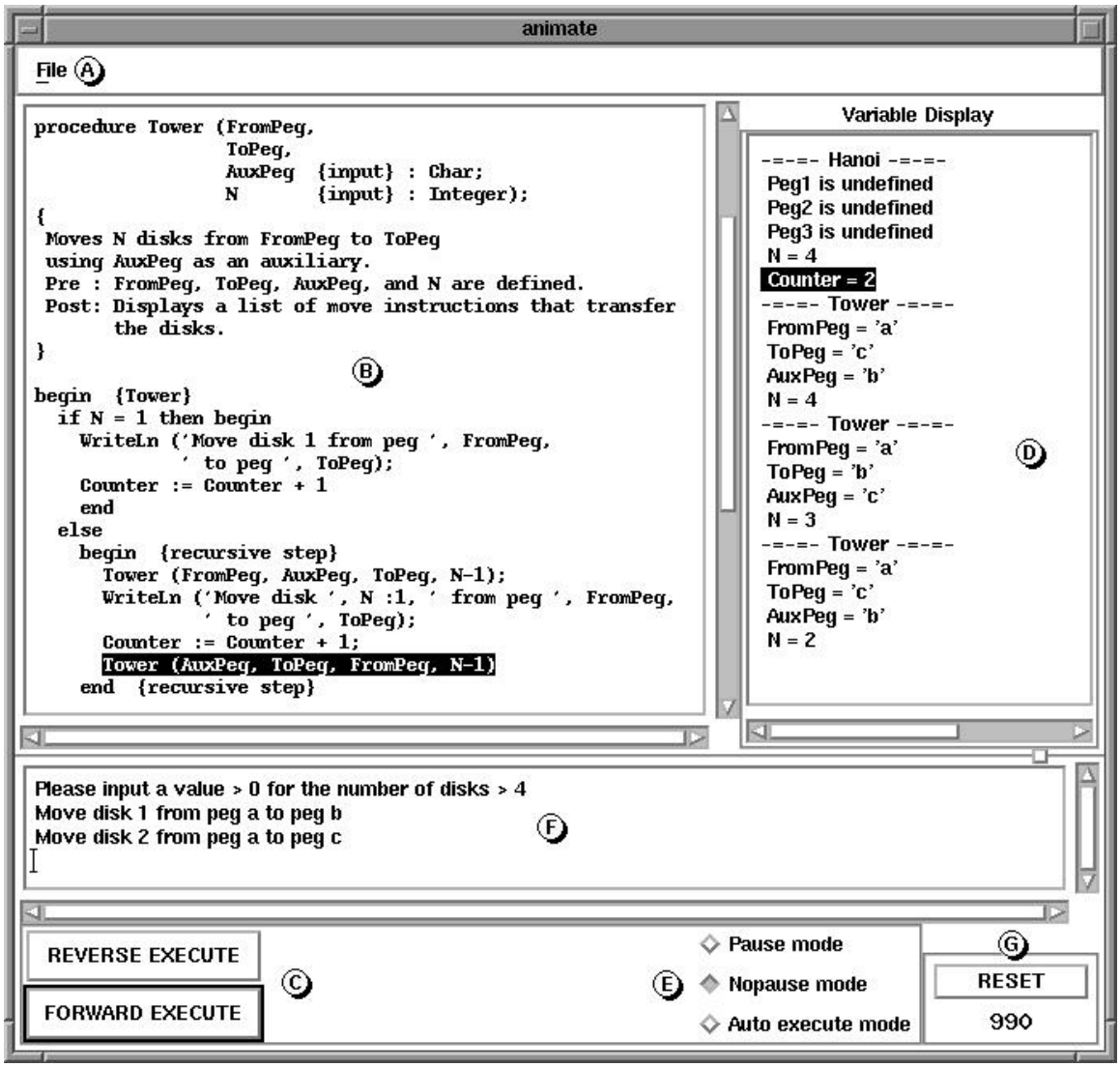


Figure 3.1: Typical Motif DYNALAB Animator Display

G The *execution cost* area displays the number of E-Machine instructions executed to the current point of execution. The **Reset** button allows the cost counter to be reset to zero so that the cost of a particular program section can be determined easily.

3.1 The Animation Controls

Control of the animator is intended to be straight-forward and predictable. That is, all actions are intended to have minimal side-effects. Since the source program is the object of study, the animator should add minimal complexity to the program animation. Only two buttons and operations on the "File" menu affect the state of execution.

3.1.1 The Execution Controls

The two principle controls are the *FORWARD EXECUTE/ADVANCE* and *REVERSE EXECUTE* buttons in the control area at the bottom of the animation window. The forward button causes execution of the current packet. It changes its label to reflect the current state of the execution when in pause mode or auto execute mode, described below. The reverse execute button unexecutes the highlighted region and advances to the previously-executed packet regardless of animation mode.

3.1.2 The Mode Selection Controls

Execution of a single packet, the highlighted portion of the source program, typically follows a sequence similar to that illustrated in Figures 3.2 - 3.5. The default execution mode, which this sequence illustrates, is the *Pause* mode. This mode is useful for allowing a student the opportunity to predict the result of the execution of the current line before executing it. And, after executing the line, the student may then predict the next line to be executed before

control advances.

When the animator is in the *nopause* mode, it automatically advances to the next packet to be executed after executing the current packet. This allows more rapid traversal of the program when desired.

In *auto execute* mode the animator runs to the end of the program, pausing only for user input. This mode is useful for examining the output of a program and determining the execution cost of the program.

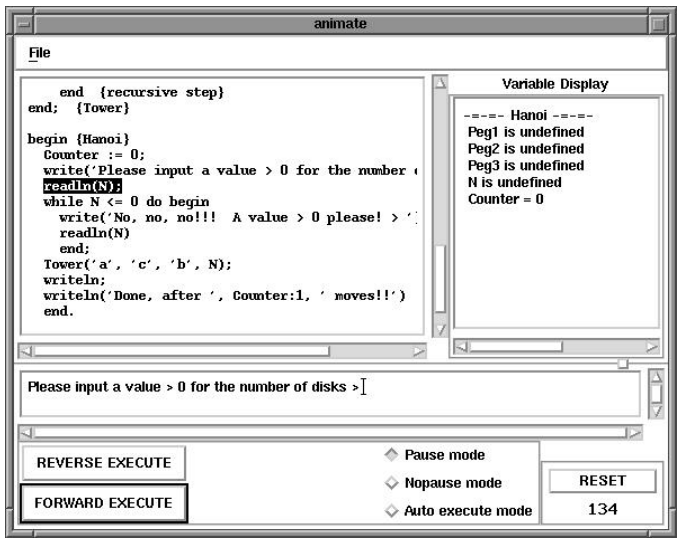


Figure 3.2: A Source Packet Before Execution

The solid reverse highlight signifies that this packet is about to be executed. Notice that the forward button is labeled "FORWARD EXECUTE", the value of "N" is "undefined", and the current execution cost is "134."

3.2 The File Menu

The **open** item in the file menu (Figure 3.6) allows the user to load a DYNALAB animation code file into the animator. The **restart** option causes the animator to restart the animation of the current code file. And the **quit**

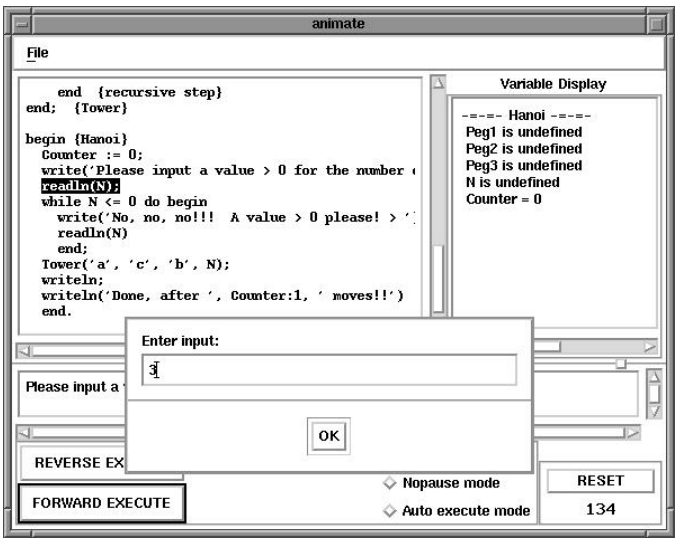


Figure 3.3: A Source Packet During Execution

After the user presses the "FORWARD EXECUTE" button, the packet is executed. Since this packet required user input, the animator brings up a requester for the user to enter a value.

option terminates the animator.

The open option brings up a file selection dialog box listing the code files in the current directory. When the user selects a file, the animator verifies the code file structure, loads it if it checks out, and initializes the displays. Figure 3.7 illustrates a typical file selection.

3.3 The Animator Displays

There are four primary displays in the animator: The source pane, the variable pane, the output pane, and the execution cost counter. With the exception of the execution cost counter, the contents of these displays are entirely non-interactive.

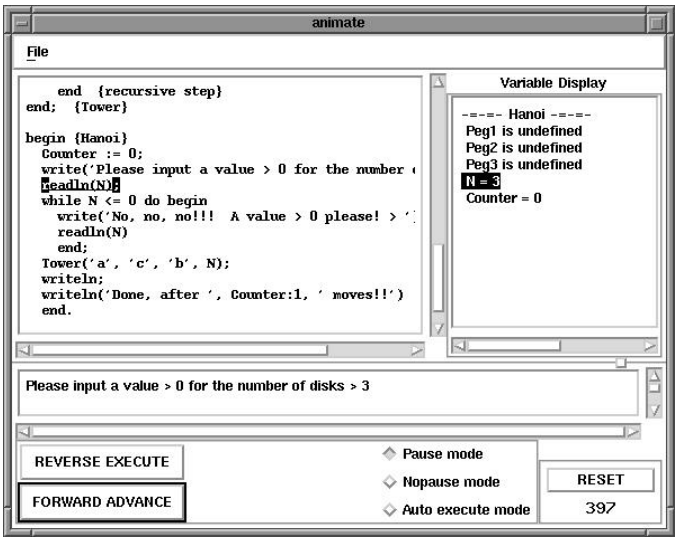


Figure 3.4: A Source Packet After Execution

The underline highlight signifies that this packet has been executed. Notice that the value of "N" is now "3" and is highlighted to signify that the value of "N" has changed. The execution cost now displays "397." Also the forward button is now labeled "FORWARD ADVANCE". Pressing the button will cause the animator to advance to and highlight the next statement to be executed.

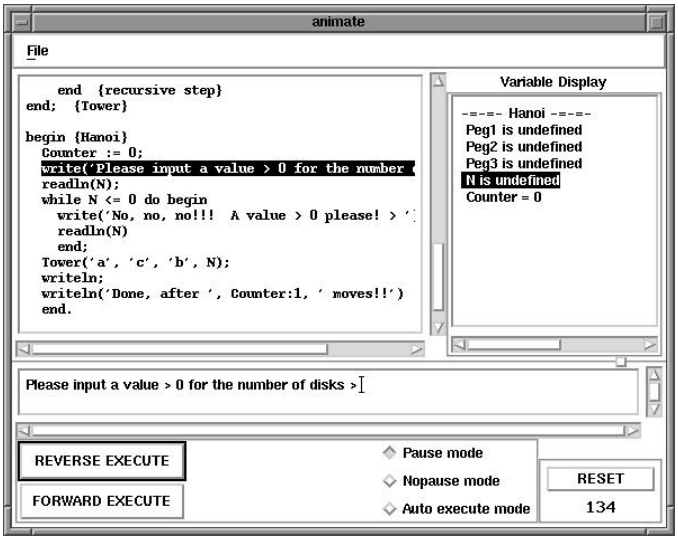


Figure 3.5: A Source Packet After Reverse Execution

This is the animator display after reverse execution of the "readln" packet. Notice that the contents of the execution cost, variable display, and IO areas are the same as in Figure 3.2. Since the previous line containing the "writeln" is the next packet to be *un*executed, it is highlighted. Also, the variable display for "N" is highlighted since it changed from "3" to "undefined."

Figure 3.6: The File Menu

3.3.1 The Source Area

The source area contains the source code for the animated program. The user may scroll through the source, but its primary function is to show the next portion of the program to be executed. Upon execution of a highlighted area, the highlight is changed according to the state of the execution and the mode of execution. The pane is scrolled automatically so that the portion being executed is always visible to the user.

3.3.2 The Variable Display Area

The variable display area contains the names and values of the active program variables at the current point of execution. Each active call frame partitions the variable display and is displayed in a top-down fashion. The initial routine is at the top of the variable display and each active routine, if any, and its variables appear below it. See Figure 3.1 item D for a typical variable display.

The user is free to scroll the variable display pane if its contents exceed the height of the pane to examine various variables. The currently-visible portion of the variable display remains in-place as program animation progresses *unless*

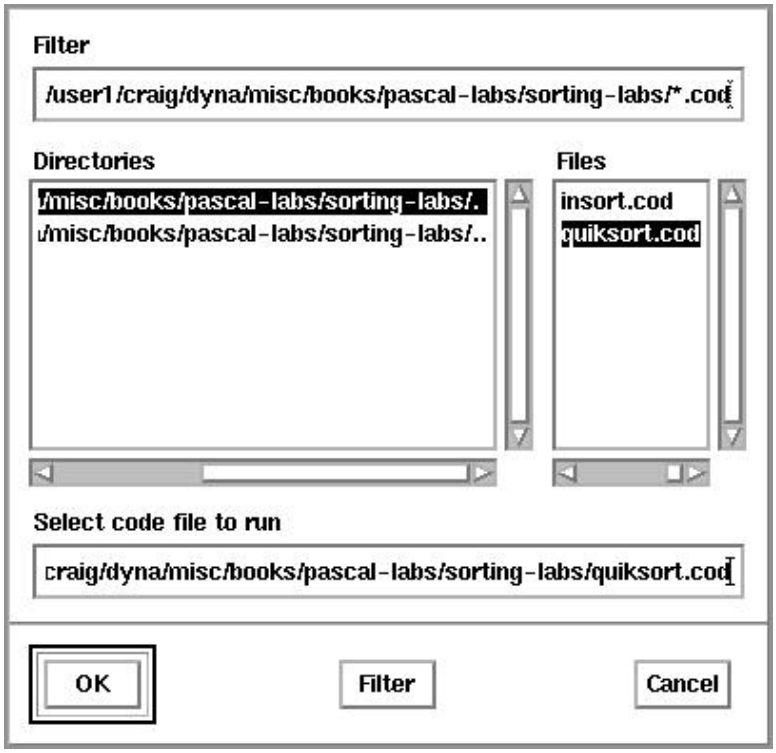


Figure 3.7: The Object File Selection Dialog

variables are instantiated in the process of animation. The variable display is automatically scrolled to the bottom when new variables appear.

It is important to note that the variable display is fundamentally different than the variable examination mechanism offered by source-level debuggers. Program animation generally displays all active variables by default while a debugger commonly shows only those variables selected by the user. To reduce the number of displayed variables in programs with a large number of variables, future enhancements to the animator include the ability to collapse sections of the variable display that aren't important to the program being studied.

3.3.3 The Input/Output Area

The input/output (IO) area displays the output of the program at the current point of execution and also all values read by the program from the user. It is important to note that the display reflects the current position in the *program* and not an output *history*.

For example, if the user were to reverse execute through the "writeln" in Figure 3.5 the IO pane would be set blank since nothing had been input or output previous to that statement in the program. It should also be noted that reverse executing through the "readln" removed the "3" from the IO pane (see Figure 3.4).

The user may scroll through the IO area if the output size exceeds the pane size. The pane is automatically scrolled to the bottom whenever the program produces output.

3.3.4 The Execution Cost Display

The execution cost display is initialized to zero at the beginning of the animation and incremented after every forward execution by the number of *E-Code* instructions associated with the executed packet. On reverse execution, it is decremented in a similar fashion.

The execution cost does not necessarily reflect the exact number of E-Machine instructions executed. But it gives a very good estimate for the cost of executing the highlighted portion. In the example execution, it is apparent that the "readln" is much more costly than *all* the statements preceding it. On all modern computer systems, it is likewise true that IO is a relatively costly operation. In the DYNALAB system, this fact comes about naturally.

Above the execution cost display is a "RESET" button. This allows the execution cost to be reset to "0" at any point in the execution. The reset

button allows the execution cost of a portion of the program to be calculated more easily.

3.4 Animator Input Dialogs

There are instances where the animator prompts the user for specific information. These are handled through Motif dialogs which appear, request user input, and then disappear. They are presented here in the order that they are typically encountered.

3.4.1 The Object File Selection Dialog

When the user selects the "open" option in the file menu, a file selection dialog appears (Figure 3.7) which lists all the object files in the currently selected directory. Currently, these files are designated with a ".cod" extension. When the user selects a file, the box disappears and the file is loaded into the animator if no errors are encountered. If an error is encountered, the animator reports the error and its controls become inactive until a valid object file is loaded. If the user selects the "Cancel" button, animation resumes where the user left off in the currently loaded animation.

3.4.2 The Input Selection Dialog

When the DYNALAB Motif animator encounters a open file command for the keyboard (standard input), the animator queries the user for an input source (see Figure 3.8). This allows the user to redirect input from a file. If the user chooses to read from a file, a file selection dialog similar to Figure 3.7 appears to allow the user to select a data file. If the user chooses to input from the keyboard, an input request dialog (Figure 3.9) appears when input is required.

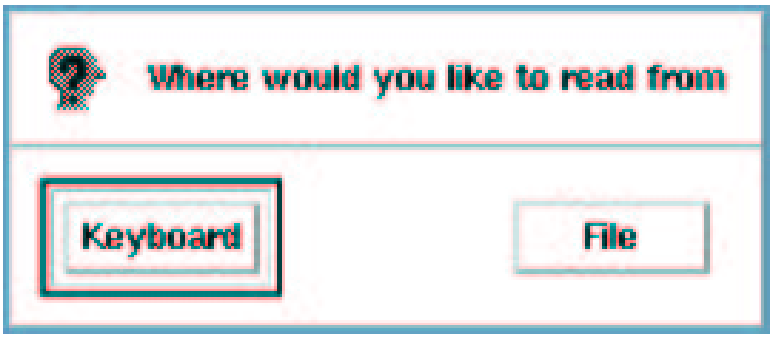


Figure 3.8: The Input Selection Dialog

3.4.3 The Input Request Dialog

When input is requested by the program from the keyboard, the animator opens the dialog shown in Figure 3.9 to receive input from the user. The characters entered by the user (followed by a newline character) are entered into an input buffer.

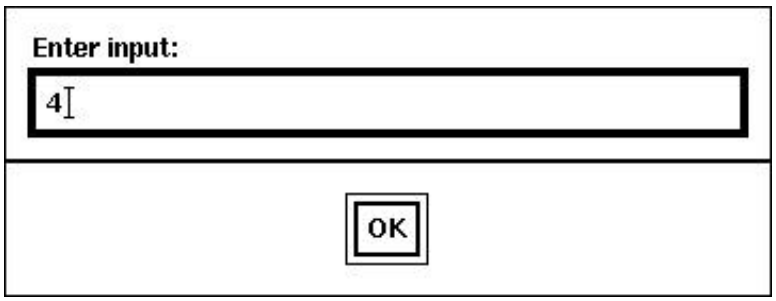


Figure 3.9: The Input Request Dialog

Subsequent requests for input by the animated program are given characters in the buffer, if any. As with any buffered input system, input is requested only if the input buffer is exhausted. When the user reverse executes through a portion of the source program that requests input, the buffer is cleared so that the user has the opportunity to enter different values upon proceeding

with execution in the forward direction. Future animator implementations may include the input buffer as an additional display to clarify how the input buffer is used. This would be beneficial since input buffering is a common source of confusion for beginning programming students.

Chapter 4

Program Animation Logistics

This chapter describes the logistics of performing proper animation of programs compiled for the DYNALAB animators. The synchronization of the variable display, source display, input/output display, and the various dialogs is a non-trivial task involving such fundamental computer science theoretical concepts as the Halting Problem.

When the E-Machine was first designed, it was presumed that program animation would simply involve executing a designated section of E-Machine code (a *packet*), highlighting the source code associated with it, updating the variable display, and moving on to the next packet [Birch 90]. As progress was made on the first compiler, it became clear that this model was too simplistic to properly animate certain programming constructs. Many changes have subsequently been made to the E-Machine to allow these constructs to be animated.

As originally conceived, the animator deals with program sections no smaller than packets. When the E-Machine is told to execute the current packet by the animator, the E-Machine executes an entire packet. The animator can then query the E-Machine to determine the number of the next packet to be executed. In order to accommodate unforeseen animation characteristics of

unimplemented DYNALAB compilers, the packet structure was redesigned to give the compiler author more control over how packets are handled by the animator and what effects they have on the animator's displays. This increased flexibility has complicated the task of program animation. But it is hoped that this flexibility will prevent additional redesigns of the E-Machine execution logic. This is imperative since any changes to the execution mechanism may require fundamental modifications to all DYNALAB compilers and animators.

4.1 Simple Animation

Figure 4.1 lists a very simple Pascal program. When compiled by the DYNALAB Pascal compiler, a code file is produced containing the E-code instructions that implement the program (Figure 4.2), and a variety of data structures used by the E-Machine and animator. Table 4.1 lists the packets, the associated source, E-Machine assembly instructions, and animator directives for a simple Pascal program.

As Table 4.1 shows, this particular program contains about one packet for each line in the source program. Notice that most of the packets have their packet directives set to "Y". (At present, only binary (two-value) directives have been necessary to encode the directives for envisioned program constructs.) These packets are basically equivalent to the original packet concept. That is, in the original packet concept, the animator would highlight each packet before execution, pause before executing the packet, and then update the variable and program counter displays after executing it.

But even this simple program illustrates the need for a more complex mechanism than originally envisioned. Some of these cases were handled by the introduction of "Packet Fragments" [Goosey 93]. But as more complicated

Packet Number	Source	First Instr	Last Instr	Forward Directives			Reverse Directives		
				H	P	U	H	P	U
0	Program Test(input)	0	11	Y	Y	Y	Y	Y	Y
1	VAR	12	12	Y	Y	Y	Y	Y	Y
2	i: Integer;	13	14	Y	Y	Y	Y	Y	Y
3	PROCEDURE dud;	15	16	Y	Y	Y	Y	Y	Y
4	BEGIN	17	17	Y	Y	Y	Y	Y	Y
5	i:=3;	18	19	Y	Y	Y	Y	Y	Y
6	END;	20	20	Y	Y	Y	Y	Y	Y
7		21	22	N	N	N	N	N	N
8	BEGIN	23	24	Y	Y	Y	Y	Y	Y
9	i := 1;	25	26	Y	Y	Y	Y	Y	Y
10	dud;	27	27	Y	Y	Y	N	N	N
11	dud;	28	28	N	N	Y	Y	Y	Y
12	i := 2112;	29	30	Y	Y	Y	Y	Y	Y
13	END.	31	36	Y	Y	Y	Y	Y	Y

- Forward directives: H - Highlight before execution, P - Pause before execution, U - Update variable display after execution
- Reverse directives: H - Highlight before un-execution, P - Pause before un-execution, U - Update variable display before unexecution

Table 4.1: Packet Information for Example 1

```
Program Test(input);
    VAR
      i: Integer;

    PROCEDURE dud;
    BEGIN
      i:=3;
    END;

    BEGIN
      i := 1;
      dud;
      i := 2112;
    END.
```

Figure 4.1: Pascal Source Code for Example 1

special cases were discovered, packet fragments and packet types [Poole 94] were removed in favor of the packet directives.

In the forward direction, proper program animation dictates that the procedure invocation, "dud" in this case, should be highlighted before being entered. Packet 10 ensures that this occurs. Proper program animation also requires that the procedure invocation should be highlighted when executing back through it in the reverse direction. Since the same packet cannot be used for both purposes, as they each contain different instructions, packet 11 is required for animation in the reverse direction. When execution leaves the procedure in either the forward or reverse direction, execution is to proceed to the statement immediately following or preceding the procedure invocation, depending upon the direction. The compiler implements this functionality by setting the reverse pause directive of packet 10 to "N". This instructs the animator to execute the packet without confirmation by the user. Packet 11

```
 0:  pushd    c, DS7
 1:  nop      c
 2:  open     c, CI0, CI0
 3:  inst     c, V1
 4:  pop      c, I, V1
 5:  inst     c, V2
 6:  push     c, I, CI0
 7:  cast     c, I, C
 8:  pop      c, C, V2
 9:  open     c, CI1, CI6
10:  inst     c, V3
11:  pop      c, I, V3
12:  nop      c
13:  inst     c, V4
14:  br       c, L0
15:  label    c, L1
16:  pushd    c, DS4
17:  nop      c
18:  push     c, I, CI3
19:  pop      c, I, V4
20:  nop      c
21:  popd     c
22:  return   c
23:  label    c, L0
24:  nop      c
25:  push     c, I, CI1
26:  pop      c, I, V4
27:  call     c, L1
28:  label    c, L2
29:  push     c, I, CI2112
30:  pop      c, I, V4
31:  nop      c
32:  uninst   c, V4
33:  uninst   c, V3
34:  uninst   c, V2
35:  uninst   c, V1
36:  popd     c
```

Figure 4.2: E-Machine Assembly Code for Example 1

has its forward pause directive set to "N" so it is skipped after leaving "dud" when executing forward. See Figures 4.3 and 4.4 for an illustration of this animation.

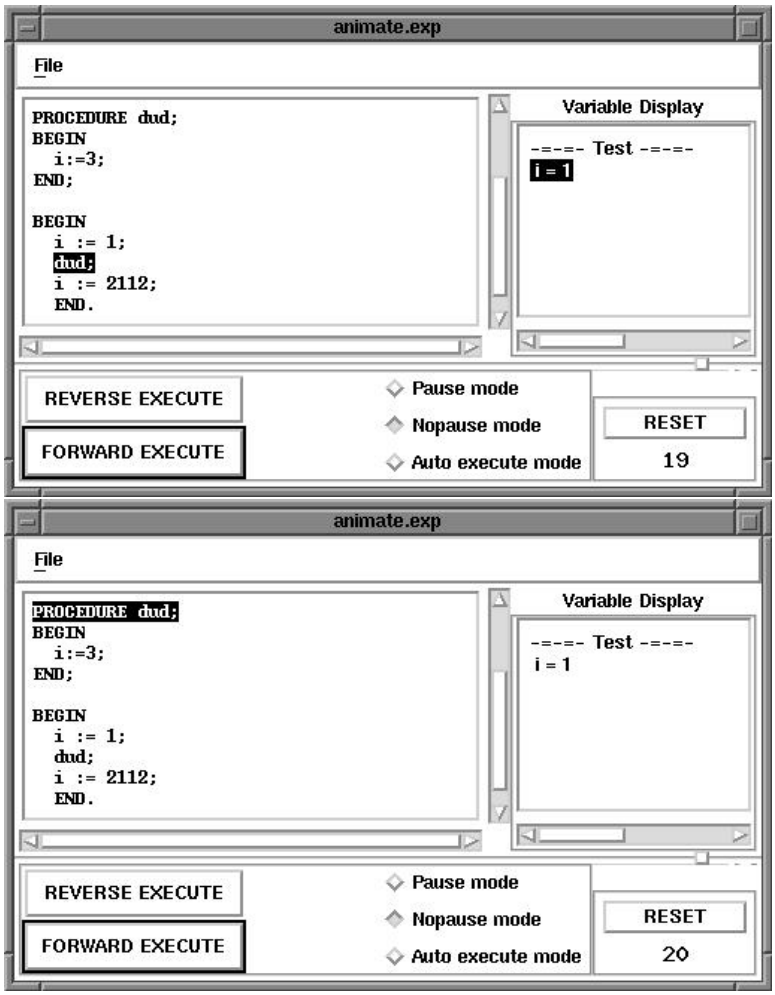


Figure 4.3: Forward Execution Through Procedure Call

Packet 7 is also of interest in this example. Once again, this packet is strategically placed here to produce proper program animation. Notice that it has no source code associated with it and has all its directives, in both directions, set to "N". This causes the animator to execute it with no user

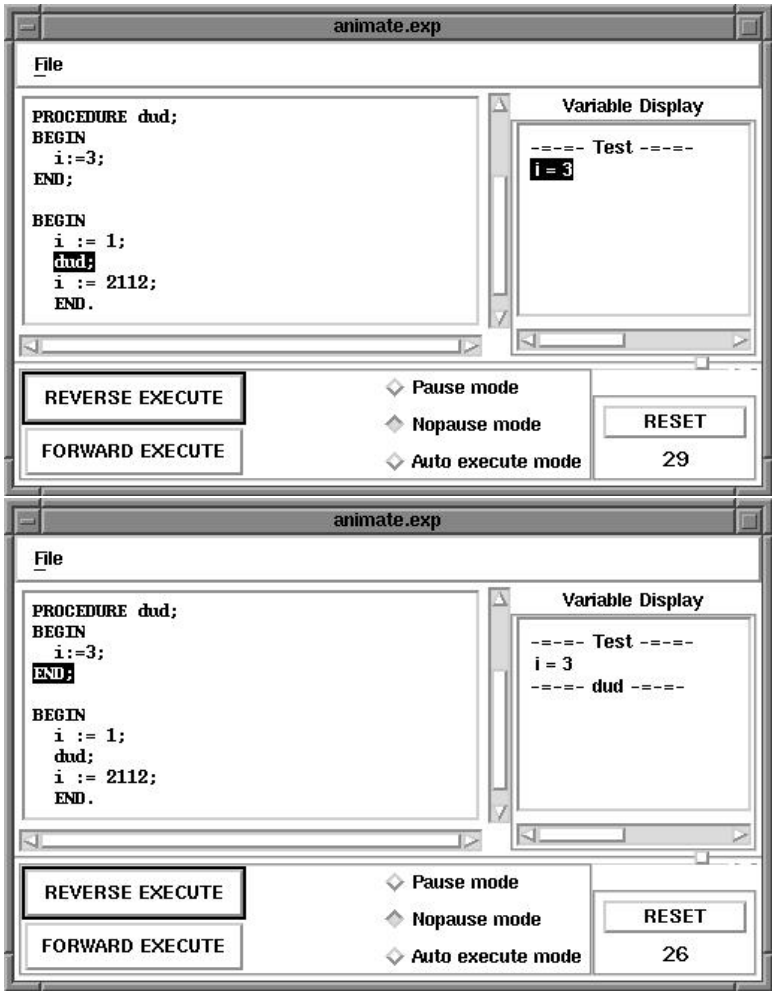


Figure 4.4: Reverse Execution Through Procedure Exit

confirmation regardless of direction. The E-code associated with packet 7 is a "popd" and a "return". These instructions remove "dud" from the call stack. When executing forward through packet 6, which contains the "END;" for the "dud" procedure, the animator will first execute packet 6, discover that packet 7 is the next packet and that its pause directive in the forward direction is "N", and execute it. The animator then queries the E-Machine and finds the next packet to be packet 11. This isn't a pause packet either, but its update directive is "Y". So, the animator updates the variable display. The previous update of the variable display had "dud" on the call stack. Now, after the "popd" has been executed, "dud" is no longer on the stack.

Actually, the above functionality could have been achieved using a single packet, but only in the forward direction. In reverse, packet 11 is a pause packet. So, when reverse executing though packet 11, the animator discovers that packet 7 is the next packet. If packet 7 were part of packet 6, the animator would update the display, execute the packet, and wait for a user event. The display would not show "dud" in the variable display since the update was done before the "popd" was unexecuted. Since packet 7 is a separate non-pause packet, the variable display correctly lists "dud" as an active subroutine when it updates the display before executing packet 6.

The point of this example is to illustrate the complications involved in properly animating even a simple program. Since the animator has no real knowledge of the source program it is animating, and should not, it must rely upon the packet information generated by the compiler.

4.2 Animation in Nopause Mode

As described in Chapter 3, *pause* mode involves two different actions: execution and advancement. In pause mode, control is first advanced to a packet

which is then executed in a second, separate step. In *nopause* mode, the steps are combined. When a packet is executed, control is automatically advanced to the next (pause) packet. The logistics of animating in nopause mode are more straight-forward than the pause mode logistics.

The animation is guided by the packet directives described in chapter 2. There are no restrictions upon the combinations of packet directives that may occur in a packet. There are a number of *common* directive combinations, but none that are invalid. It is vitally important that the animator properly deal with the directives, and all their combinations, so programs animate in the way the DYNALAB compiler writer intended.

The directive of primary interest to the animator is the *pause* directive. This directive tells the animator that the associated packet should not execute (or at least not *appear* to be executed) until the user requests it to be executed when the animator is in a stepping mode. A stepping mode is one where where the user is executing the program one step at a time. How the animator implements this depends upon the animator implementation. With an event-driven input system, simple forward execution could follow these steps.

1. Wait for user to select "Execute"
2. Execute the current packet
3. If the executed packet has the "Forward update" directive set, update the variable display to reflect the current state of the E-Machine
4. Get the next packet to be executed. While it is not a pause packet, loop:
   (a) If the packet has its "forward highlight" directive set, highlight the source corresponding to the packet
   (b) Execute the packet
   (c) If the packet has its "Forward update" directive set, update the variable display
5. If the packet has its "forward highlight" directive set, highlight the source corresponding to the packet

6. If not at end of program, go to step 1.

Of course, this algorithm is far too simple to be of much use since it doesn't handle reverse execution – one of the main features of the E-Machine and an extremely useful animation option. If the E-Machine was at the end of a program, the following algorithm could be used to step through the program in reverse.

1. Wait for user to select "Execute"
2. Unexecute the current packet
3. Get the next packet to be unexecuted. While it is not a pause packet, loop:
   (a) If the packet has its "Reverse highlight" directive set, highlight the source corresponding to the packet
   (b) If the packet has its "Reverse update" directive set, update the variable display
   (c) Execute the packet
4. If the packet has its "Reverse highlight" directive set, highlight the source corresponding to the packet
5. If executed packet has "Reverse update" directive set, update the variable display to reflect the current state of the E-Machine
6. If not at beginning of program, go to step 1

The notable difference between the forward and reverse execution nopause algorithms is the treatment of the update directive. The forward-update directive dictates that the variable display should be updated *after* execution of the associated packet while the reverse-update directive dictates that the variable display should be updated *before* unexecution of the associated packet. This difference of interpretation is necessary so the compiler can easily generate packets which have variables updated at a consistent point regardless

of direction. This could be implemented with a consistent interpretation but it would require a minimum of two packets to implement what is currently implemented in one.

Neither of these algorithms allow the changing of direction at an arbitrary point in program animation. And incorporation of directional change isn't just a simple matter of performing the appropriate algorithm depending upon the direction of execution. Some steps must be taken to ensure that the transition is done naturally. If the user was forward executing through a program using the forward algorithm above, then suddenly executed in reverse using the reverse algorithm, animation would appear to jump an instruction.

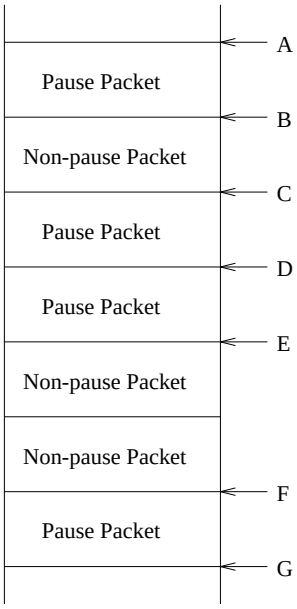


Figure 4.5: Execution States

Figure 4.5 illustrates the situation. When executing forward, the animator would pause at states A, C, D, and F. When executing in reverse, the animator would pause at states G, E, D, and B. If the direction of execution were changed from forward to reverse while the animator is in state C, the animator would

need to unexecute to state B; in D to D; and in F to E. If the direction of execution were changed from reverse to forward and animation is in state B, the animator would need to execute to state C; from D to D; and from E to F.

The algorithm to perform this is fairly simple.

1. Reverse the E-Machine
2. Get the next packet
3. Execute/unexecute until a pause packet is found, highlighting source and updating variable display as directed and according to direction of execution

With the three algorithms outlined thus far, an algorithm to implement nopause mode can be stated as:

1. Wait for user to select "Execute" or "Reverse"
2. If the user selects "Reverse"
   (a) Reverse the E-Machine
   (b) Get the next packet
   (c) Execute until a pause packet is found, highlighting the source and updating variable display as directed and according to the direction of execution
3. If the user selects "Execute"
   (a) If going forward, execute the current packet and update variable display if directive set
   (b) If going backward, execute the current packet
   (c) Execute until a pause packet is found, highlighting and updating variable display as directed according to direction of execution

This algorithm will properly animate a program on a step-by-step basis going either forward or backward. But many important and difficult factors

have been neglected. Checks are missing to ensure that execution doesn't continue past the top or bottom of the program. And I/O, instruction counting, and run-time errors have not been dealt with. These will be discussed later in the chapter.

4.3 Animation in Pause Mode

Implementing pause mode requires significantly more logic than nopause mode. The only state in nopause mode of any consequence is the state of the E-Machine. In pause mode, there are two states of execution which might be labeled "Ready to advance" and "Ready to execute." These states parallels the current highlighting style of the source display. See chapter 3 for more information about the operation of pause mode.

The complexity of pause mode arises due to the number of state combinations between the animator and E-Machine. There are a number of states the E-Machine may be in during execution, five states of execution in pause mode, the state of the I/O window, the state of the variable display, the state of the program counter, and four different types of control events. The minimal input events are execute/advance, change direction, switch to pause mode, and switch to nopause mode.

A finite state automaton (FSA) to control the animation is shown in Figure 4.6. Notice that this FSA doesn't include "reversal" as an event. Instead, forward execute and reverse execute are given as events. It is presumed that reversal of the E-Machine and the skipping of non-pause packets outlined above is done when a direction change is requested by the user.

The appropriate highlighting style and all valid options at any point can be determined from the current state of the animation. Specifically, when in state P1, which is the start state for the animation, the animator should

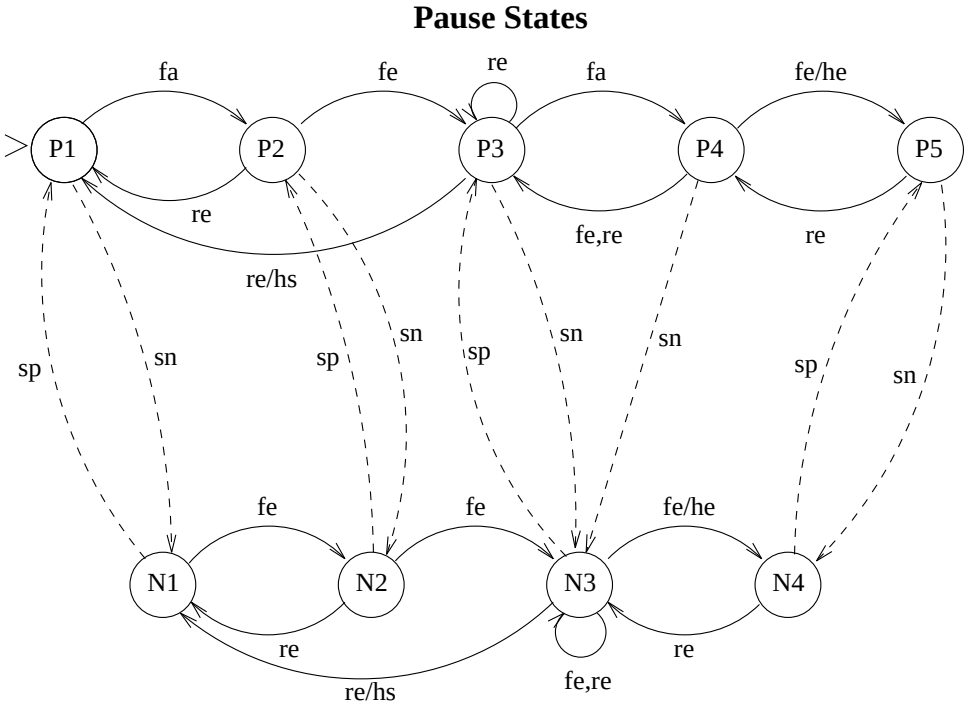


Figure 4.6: Animation States

P1-P5	The pause states
N1-N4	The nopause states
fa	Forward advance event
fe	Forward execute event
re	Reverse execute event
fe/he	Forward execute and hit end of program
re/hs	Reverse execute and hit start of program
sn	Switch to nopause mode event
sp	Switch to pause mode event

only allow the user to "advance" and nothing should be highlighted. This allows the user to guess the entry point of the program before the animator actually advances there and highlights the entry point. When in state P3, the animator should highlight the packet just executed differently to distinguish it from a highlight signifying a packet which is *about* to be executed. States P5 and N4 are similar to P1; nothing should be highlighted since execution has proceeded beyond the end of the program. All other states should highlight the next packet to be executed or unexecuted.

There are, of course, some complications.

4.3.1 Advancing Without Executing

As with any machine, it is impossible to predict the path of execution of an arbitrary program or program section without actually executing the program. This problem is, in fact, a corollary of the Halting Problem. But that is precisely the task the animator must seemingly perform in pause mode. The solution is to make the packet only *appear* to not execute until the user selects to execute it.

While it is required that all the animator displays be synchronized with each other, there isn't any requirement that the state of the E-Machine and the state the animator be synchronous. In fact, it is required that they *not* be synchronized when advancing to the next packet to be executed in pause mode.

To implement advancement, the animator may follow these steps when in state P2 or P4 and the user selects to execute the current packet (event "fe"):

1. Execute the current packet, update the variable display if directed, and distinctly highlight the executed packet to signify that it has been executed, not about to be executed. This reflects transition to state P3 or P5, if at end of program.

2. Execute all non-pause packets up to the next pause packet and remember the last seen packet with its highlight directive set

3. When the user selects advance, just highlight the last seen highlight packet (typically the current pause packet). This reflects the transition to state P2 or P4.

In order for this illusion to work, one assumption must be made. Any non-pause packets that contain instructions which query the animator for user input – such as reads and file opens – **must** follow the pause packet they are associated with. If any of these instructions were contained in non-pause packets *preceding* the packet with the associated source code, it would appear to the user that they were being asked for input before they requested execution of the source code containing the input statement. This also applies to variable update directives. Any update directives must reflect changes associated with the source for the previous pause packet. Figure 4.7 illustrates this fact.

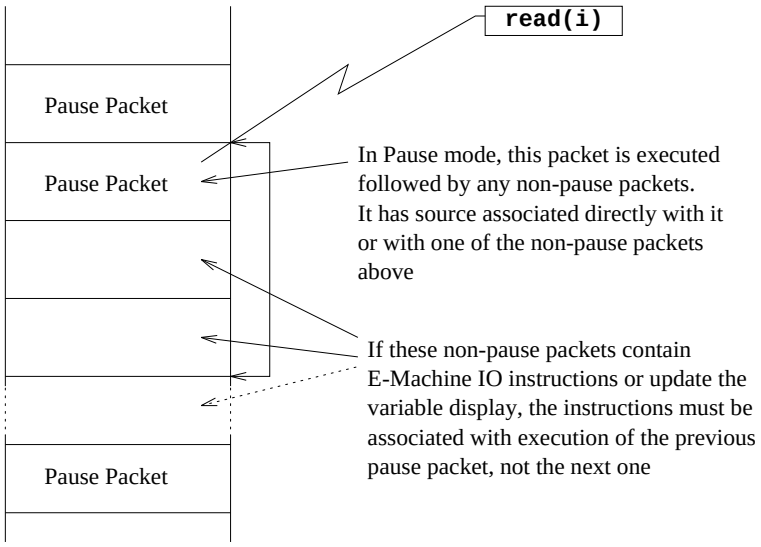


Figure 4.7: Correct Usage of Non-pause Packets

Also, it is assumed that there is a highlight packet between every adja-

cent pause packet. This assumption is also made for nopause mode. It is the compiler writer's responsibility to produce code which animates properly according to the specification of the E-Machine, the packet directives, and the above rule. The animator may make no other assumptions about the produced code.

As an example, below is a very simple Pascal program which reads a number from the user.

```
program test(input);

var
  i:integer;

begin
  read(i);
end.
```

Figure 4.8: Pascal Source Code for Example 2

As table 4.2 shows, the majority of the packets for this small program are non-pause non-highlight packets. In this case, the Pascal compiler chose to implement I/O using individual packets. The E-Machine code to implement this program isn't included due to length. The E-code from addresses 4 to 350 include instructions for user input, and packet 29 includes a variable update directive. The location of the packets containing the I/O instructions is valid in this case since the pause packet preceding the non-pause packets is associated with the Pascal "read". The location of the Update directive in packet 29 implies that packets 4-29 only affect variables associated with the implementation of the "read(i)".

In the reverse direction, execution will be paused on packet 29 since its

Packet Number	Source	First Instr	Last Instr	Forward Directives			Reverse Directives		
				H	P	U	H	P	U
0	program test(input);	0	19	Y	Y	Y	Y	Y	Y
1	var	20	20	Y	Y	Y	Y	Y	Y
2	i:integer;	21	21	Y	Y	Y	Y	Y	Y
3	begin	22	22	Y	Y	Y	Y	Y	Y
4	read(i);	23	25	Y	Y	Y	N	N	N
5		26	31	N	N	N	N	N	N
6		32	37	N	N	N	N	N	N
7		38	56	N	N	N	N	N	N
8		57	59	N	N	N	N	N	N
9		60	70	N	N	N	N	N	N
10		71	94	N	N	N	N	N	N
11		95	97	N	N	N	N	N	N
12		98	120	N	N	N	N	N	N
13		121	123	N	N	N	N	N	N
14		124	129	N	N	N	N	N	N
15		130	155	N	N	N	N	N	N
16		156	183	N	N	N	N	N	N
17		184	185	N	N	N	N	N	N
18		186	192	N	N	N	N	N	N
19		193	242	N	N	N	N	N	N
20		243	244	N	N	N	N	N	N
21		245	248	N	N	N	N	N	N
22		249	278	N	N	N	N	N	N
23		279	280	N	N	N	N	N	N
24		281	287	N	N	N	N	N	N
25		288	324	N	N	N	N	N	N
26		325	328	N	N	N	N	N	N
27		329	337	N	N	N	N	N	N
28		338	339	N	N	N	N	N	N
29	read(i);	340	342	N	N	Y	Y	Y	Y
30	end.	343	350	Y	Y	Y	Y	Y	Y

- Forward directives: H - Highlight before execution, P - Pause before execution, U - Update variable display after execution
- Reverse directives: H - Highlight before un-execution, P - Pause before un-execution, U - Update variable display before un-execution

Table 4.2: Packet Information for Example 2

reverse pause directive is set. When the user un-executes the packet, the animator will reverse execute all packets from 29 through 4 and stop on packet 3. Since packet 3 has its reverse update directive set, the variable display will be updated to reflect the new state of the variables. In this case, `i` will take on its previous value of "undefined." Since reverse pause mode is considered to be of little instructional value, no design considerations have been made for its inclusion. And, in fact, it would not be possible under the the current scheme in many situations.

Figures 4.9 and 4.10 illustrate animation of the "read" above. Notice that the entered value, "42", is displayed as the value of "i" after the read. Since the compiler set the forward Update directive in packet 29, the new value of "i" is shown immediately after the input is performed.

4.3.2 Reversing from the Forward State

Since reverse execution does not pause, it is much simpler to implement. But the transition from states P4 to P3 requires some explanation. When the animator is in state P4, the state of the execution has the pause packet associated with the next section of code as the next packet to execute. But the animator hasn't yet highlighted it since it is waiting for the user to press "Advance" before continuing.

The animator could unexecute the currently-highlighted packet and reverse advance to the previous packet. But it was decided that it would be more natural if it simply changed the highlighting of the currently-highlighted packet before unexecuting it.

So, when the user requests reverse execution during the advance state of pause mode, the animator unexecutes all the non-pause packets it executed during the execution state and highlights the next pause packet it finds. Note

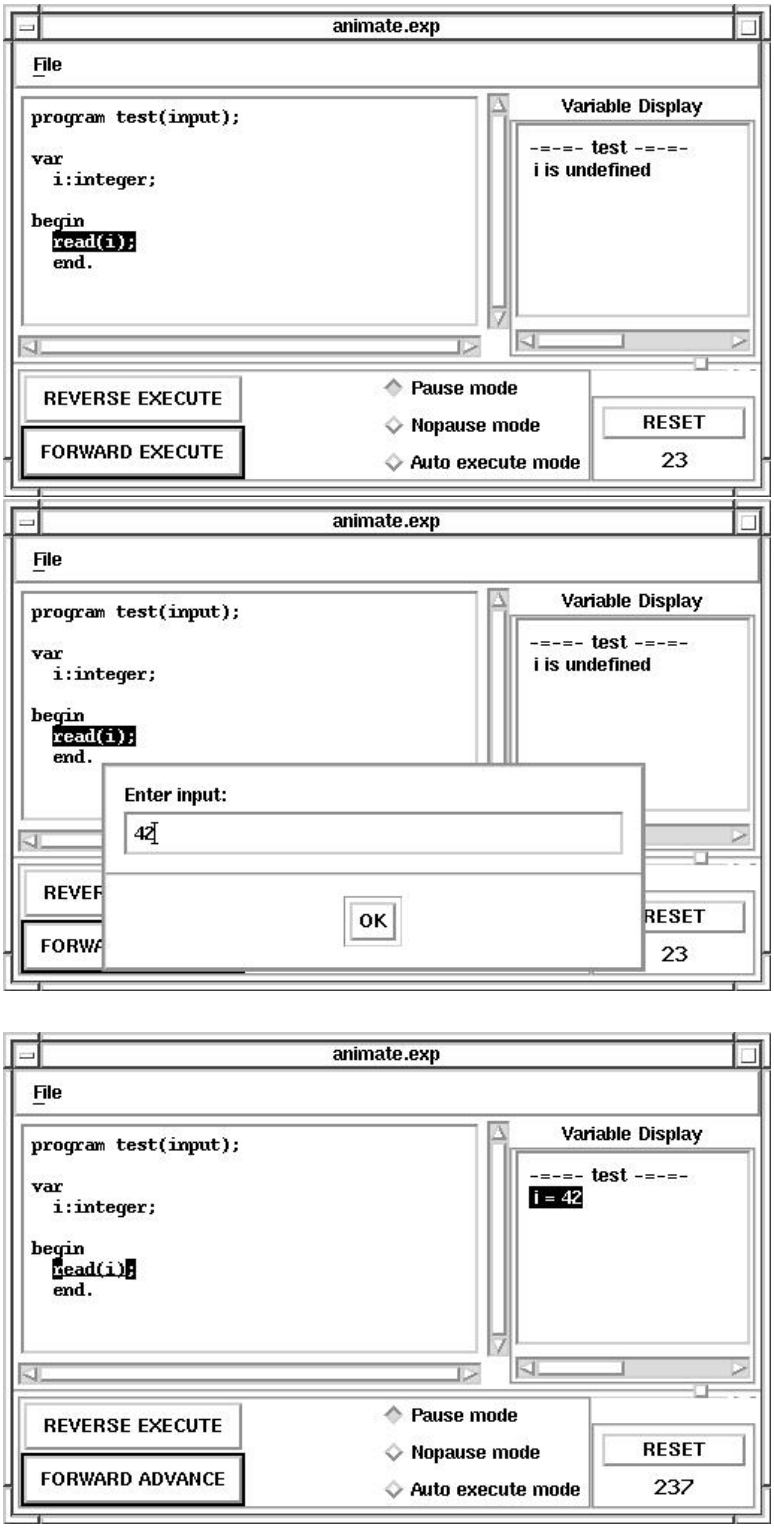


Figure 4.9: Forward Execution Through `read`

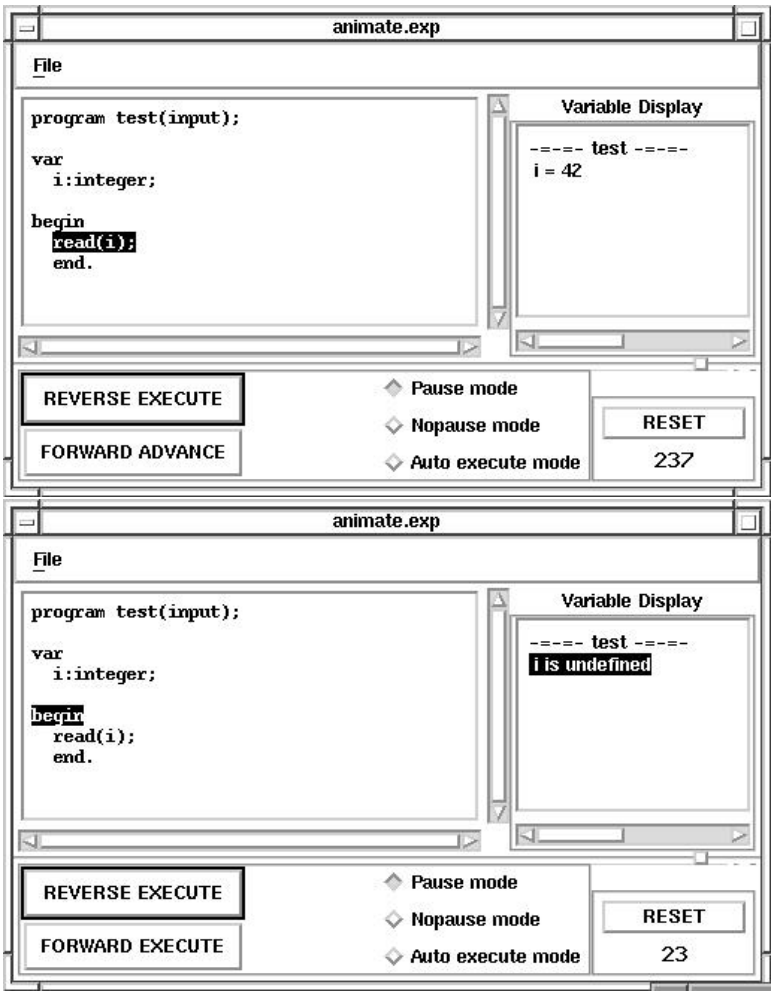


Figure 4.10: Reverse Execution Through `read`

that this packet may not be the same as the packet encountered in the forward direction.

4.4 An Algorithm for Animation Control

The following algorithms parallel the operation of the DYNALAB OSF/Motif program animator. The C/Motif implementation of this algorithm can be found with the source code in the file `control.c`. This algorithm is presumed to be invoked each time the user chooses to execute in either the forward or reverse direction. Mode changes can be made at any time between the exit of this algorithm and its invocation.

A base algorithm used by the animator, described below, is RunToNextPause. It takes the current state as a parameter and returns the number of instructions associated with the packets executed.

```
Function RunToNextPause(AnimationState) returns a scalar
  NextPacket <- Get the current packet from the E-Machine
  Cost <- 0
  If E-Machine is in forward execution mode
    Loop while NextPacket's forward pause directive isn't set
      If EndOfProgram flag is set, return Cost
      If the next packet is the last packet then
        Raise EndOfProgram flag
      If NextPacket's forward Show Source directive is set,
         highlight the source code associated with NextPacket
         according to AnimationState (no highlight if null
         state)
      Tell the E-Machine to execute the current packet
      If E-Machine fault occured, call fault handler and
         return Cost
      Increment Cost by number of instructions in NextPacket
      If NextPacket's forward variable update directive is set,
         update the variable display
      NextPacket <- Get the current packet from the E-Machine
      End loop
    If NextPacket's forward Show Source directive is set,
```

```
      highlight the source code associated with NextPacket
      according to AnimationState
  Otherwise, the E-Machine is in reverse execution mode
    Loop while NextPacket's reverse pause directive isn't set
      If TopOfProgram flag is set, return Cost
      If the next packet is the top packet then
         Raise TopOfProgram flag
      If NextPacket's reverse Show Source directive is set,
          highlight the source code associated with NextPacket
          according to AnimationState
      If NextPacket's reverse variable update directive is set,
         update the variable display
      Tell the E-Machine to execute the current packet
      If E-Machine fault occured, call fault handler and
         return Cost
      Increment Cost by number of instructions in NextPacket
      NextPacket <- Get the current packet from the E-Machine
      End loop
    If NextPacket's reverse variable update directive is set,
       update the variable display
    If NextPacket's reverse Show Source directive is set,
        highlight the source code associated with NextPacket
        according to AnimationState
  Return Cost
```

The algorithm to perform the actual animation control is greatly simplified by using RunToNextPause. The initial mode is assumed to be pause mode, the initial animation state is Start, the E-Machine direction to be forward, and the next packet is the first packet of the program.

```
Procedure Execution
  Cost <- 0
  NextPacket <- Get the current packet from the E-Machine
  If the direction of execution the user selected conflicts with
     the current execution direction of the E-Machine
    Reverse the E-Machine (changing the next packet to be executed)
    If the E-Machine is now going in reverse and the animator is in
       pause mode, change the forward button to read
       "forward advance"
```

```
    Otherwise set it to "forward execute"
  Call RunToNextPause(Execute state) and increment Cost by its
     return value
  Return from Execution procedure
If the E-Machine is in forward execution mode
  If EndOfProgram flag is set (state P5), return
  If TopOfProgram flag is set, reset it
  If current animation mode is pause mode
    If the animation is in the Advance state (state P1 or P4)
      Call RunToNextPause(Execute state) and increment Cost by
         its return value
      Set current state to Execute state
    Otherwise animation is in the Execute state (state P2 or P4)
    Tell the E-Machine to execute the current packet
    If E-Machine fault occured, call fault handler and return
    If NextPacket's forward variable update directive is set,
       update the variable display
    Increment Cost by number of instructions in NextPacket
    Call RunToNextPause(null state) and increment Cost by its
       return value
    Increment the instruction counter by Cost and set Cost to 0
    If E-Machine fault occured, call fault handler and return
    If the next packet is the last packet then
       Raise EndOfProgram flag
    Set current state to Advance state
    Set the highlight style of the currently-highlighted area for
       advancement
    Change the forward button to read "forward advance"
  Otherwise the animator is in nopause mode
    If current animation state is Advance state (P1 or P4)
      Set current state to Execution state (N1 or N3)
      Set the highlight style of the currently-highlighted area for
         execution
      Change the forward button to read "forward execute"
    Otherwise the current animation state is Execute
      Tell the E-Machine to execute the current packet
      Increment Cost by number of instructions in NextPacket
      If E-Machine fault occured, call fault handler and return
    If NextPacket's forward variable update directive is set,
       update the variable display
    Call RunToNextPause(null state) and increment Cost by its
```

```
        return value
      Increment the instruction counter by Cost and set Cost to 0
      If the next packet is the last packet then
        Raise EndOfProgram flag
Otherwise, the E-Machine is in reverse execution mode
  If TopOfProgram flag is set (state P1 and N1), return
  Tell the E-Machine to execute the current packet
  If E-Machine fault occured, call fault handler and return
  Increment Cost by number of instructions in NextPacket
  Call RunToNextPause (null state) and increment Cost by its
     return value
  Decrement the instruction counter by Cost as set Cost to 0
  If the next packet is the first packet
     Raise TopOfProgram flag
     Remove the current highlight
     Set the next state to the Start state (state P1/N1)
End Execution procedure
```

Of course, this is only one of many ways to handle the execution logistics. The philosophy behind the method outlined in this chapter is that the state of the animation and execution is only effected by the execution procedure. Mode changes can be made with no visual affect; only the forward and reverse execution events have any visual effect on the animator and no effect whatsoever on the state of the execution. The intention of this design philosophy is to make program animation as natural as possible with as little unexpected functionality as possible.

Chapter 5

The VarList Package

Control of program animation is a non-trivial task. However, the parsing of the E-Machine static scope table and the accessing of the E-Machine data and string space to extract variable names and values is a fundamentally more difficult problem. The animator must properly display the variables at each point in the execution, each of which has an arbitrary data type.

For the first two animators – the text-based DOS and Unix/curses animators – variable parsing and extraction was implemented according to the characteristics of the display environments. Due to the complexity of the process, implementation was both time-consuming and error-prone. As the development of the Motif animator progressed and the MS-Windows animator began, it was deemed necessary to create an animator-independent package that would remove the burden of writing a static scope parser specific to the animator's display environment. The result was the VarList package.

The VarList package maintains a data structure containing the names, values, and all associated E-Machine information regarding each variable in the program. The animator can use this structure to display the active scopes, variable names, and the variable values.

5.1 The VarList Structure

The VarList structure is designed to accommodate all foreseeable variable display features. Hence, its structure contains a number of elements and optional extensions. To use the basic features of the structure only three simple elements need to be accessed. The structure elements are described by their order of importance:

- **itemtype** – Type descriptor of item. This designates the type of the VarList entry, which can be any one of: **VLBOOLEAN**, **VLINTEGER**, **VLREAL**, **VLPOINTER**, **VLCHARACTER**, **VLSTRING**, **VLUNDEFINED**, **VLPROGRAM**, **VLPROCHEAD**, **VLPROCTAIL**, **VLFUNCHEAD**, **VLFUNCTAIL**, **VLENUMHEAD**, **VLENUMELEMENT**, **VLENUMTAIL**, **VLARRAYHEAD**, **VLARRAYELEMENT**, **VLARRAYTAIL**, **VLRECORDHEAD**, **VLRECORDELEMENT**, **VLRECORDTAIL**, **VLTYPEHEAD**, **VLTYPEELEMENT**, **VLTYPETAIL**, or **VLCOMMENT**.
- **level** – Integer representing the nesting or indention level for the item (negative numbers mean "not visible")
- **itemstring** – String constructed for this item containing the completely-specified variable name, an "=" sign, and the value of the variable.
- **ssentry** – Offset of entry in the static scope table for direct access to the variable information. **ssentry** is only valid if positive.
- **varreg** – The number variable register associated with this item (if positive).
- **vardepth** – How deep the item is located on the **varreg** variable register stack
- **changed** – Tells whether or not the item has changed since the last EMClearChangeRecord. This field is zero if the item is unchanged and nonzero otherwise.
- **extra** – Generic pointer for any extra information that may be associated with the variable
- **next** – Pointer to next element in the VarList (nil if at end of list).

- **prev** – Pointer to previous element in the VarList (nil if at beginning of list).

As an example, Figure 5.1 illustrates a snapshot of the animation of a Pascal program containing a simple record. The record is passed in-out to a routine. Since the value of `P.Age` changed during the execution of the previous packet, it is highlighted in the variable display. Since `Personl` is the actual parameter, it is highlighted as well.

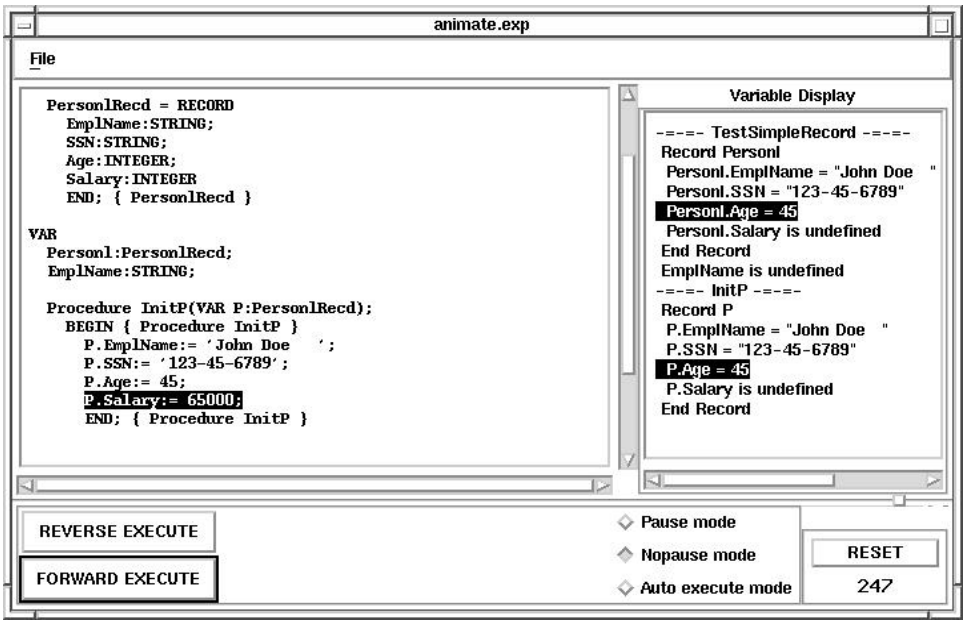
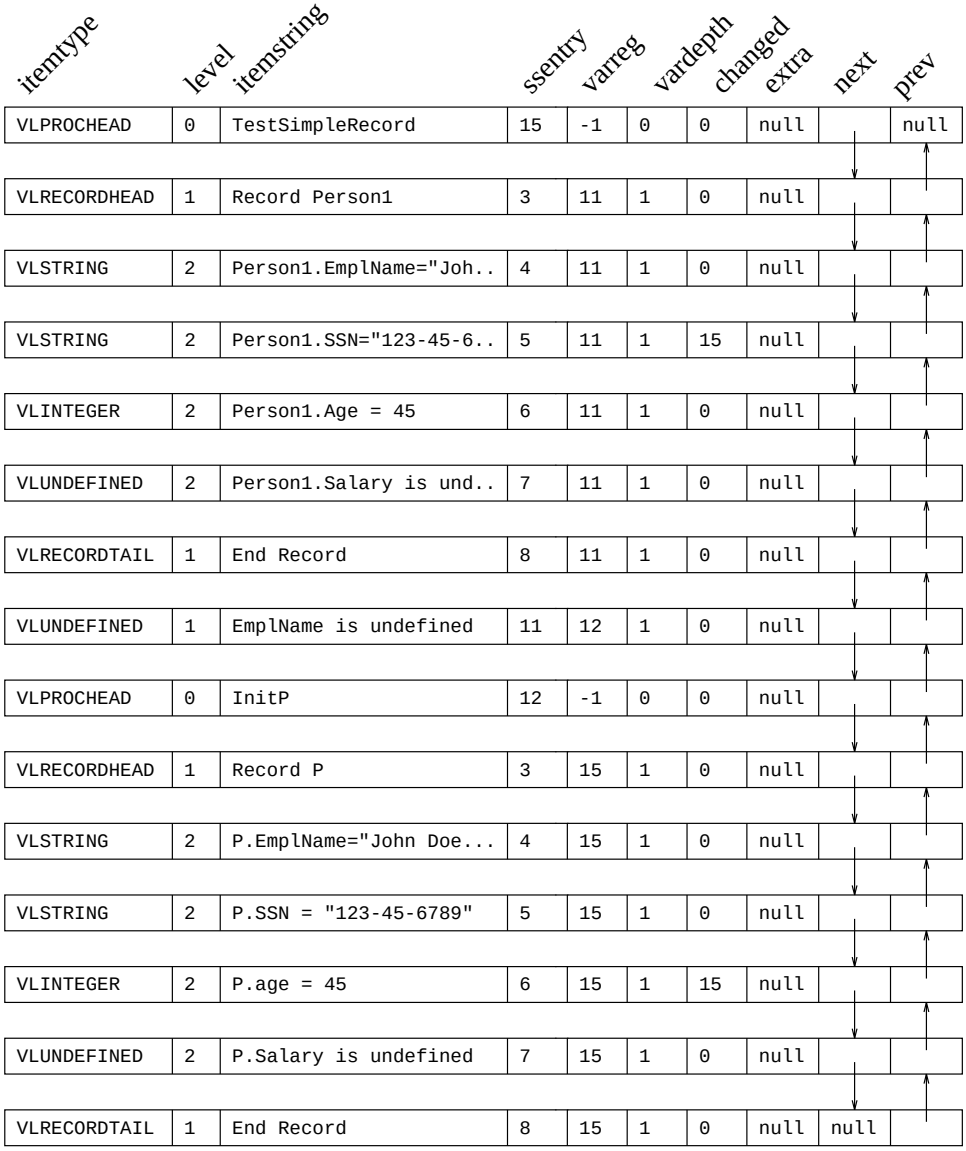


Figure 5.1: Variable Display of Simple Record

The corresponding VarList used to create the variable display in figure 5.1 is shown in figure 5.2. Each line of the variable display corresponds to one element of the VarList structure. The only fields used to create the display were **itemtype**, **level**, **itemstring**, **changed**, and **next**. Notice that the highlighted entries of the variable display correspond to those lines which have a non-zero **changed** field.



itemtype	level	itemstring	ssentry	varreg	vardepth	changed	extra	next	prev
VLPROCHEAD	0	TestSimpleRecord	15	-1	0	0	null		null
VLRECORDHEAD	1	Record Person1	3	11	1	0	null		
VLSTRING	2	Person1.EmplName="Joh..	4	11	1	0	null		
VLSTRING	2	Person1.SSN="123-45-6..	5	11	1	15	null		
VLINTEGER	2	Person1.Age = 45	6	11	1	0	null		
VLUNDEFINED	2	Person1.Salary is und..	7	11	1	0	null		
VLRECORDTAIL	1	End Record	8	11	1	0	null		
VLUNDEFINED	1	EmplName is undefined	11	12	1	0	null		
VLPROCHEAD	0	InitP	12	-1	0	0	null		
VLRECORDHEAD	1	Record P	3	15	1	0	null		
VLSTRING	2	P.EmplName="John Doe...	4	15	1	0	null		
VLSTRING	2	P.SSN = "123-45-6789"	5	15	1	0	null		
VLINTEGER	2	P.age = 45	6	15	1	15	null		
VLUNDEFINED	2	P.Salary is undefined	7	15	1	0	null		
VLRECORDTAIL	1	End Record	8	15	1	0	null	null	

Figure 5.2: Variable List Used to Generate Variable Display

Even using this subset of the available fields provides a very functional variable display. The other fields are intended for advanced animator features which have yet to be explored. The animator could, at its choosing, directly access the names and value of variables using the **ssentry**, **varreg**, and **vardepth** fields and format the names and values in whatever fashion it chooses. The **itemstring** field is included as a pure convenience.

5.2 The VarList API

There are only a few defined operations for the VarList package since the structure is directly accessed by the animator. There are just three routines that operate on VarLists.

- **int VLBuild(StatScopeEntry ssScope, VLElementType * *vlHead, int iDebugLevel)** – VLBuild will build a Variable List for E-Machine scope **ssScope** and return a pointer to the first element of it in ***vlepHead**. If **iDebugLevel** is greater then zero, debugging information will be input as **VLCOMMENT** entries in the Variable List. If **VLBuild** is unsuccessful, **VLBuild** will return a negative value and a positive value otherwise. (See chapter 2 for information on the **StatScopeEntry** type.)

- **int VLDestroy(VLElementType *vlHead)** – VLDestroy will deallocate the Variable List pointed to by **vlHead**. If unsuccessful, **VLDestroy** will return a negative value and a positive value otherwise.

- **VLElementType * VLAddElement(VLElementType * vlTarget, VLItemType vliType, short int iLevel, char * cpString, StatScopeEntry ssSource, VariableReg vrSourceReg, int iVarDepth, int iChanged, void * vpExtra)** – VLAddElement will create a new element and put it at the end of the Variable List pointed to by vlTarget and assign its data values from the remaining parameters.

Chapter 6

Animator Design

The high-level design of the DYNALAB OSF/Motif program animator follows a simple and intuitive pattern common to graphical user interface (GUI) designs. Each major element of the animator is implemented as a separate package or object. The primary element is the animation window itself. The constituent elements of the animation window are the source code area, the variable display area, the input/output (I/O) area, the instruction counter, and the animation control area (see Figure 3.1 for their visual appearance). A number of OSF/Motif and custom objects are used in the construction of these areas, including the VarList, E-Machine, and object file packages discussed in earlier chapters.

The animator is implemented in ANSI C. The current instability of the OSF/Motif C++ bindings made the use of C++ infeasible. But every effort has been made to follow the primary maxims of object-oriented design. Encapsulation, data-hiding, and polymorphism are employed throughout the design.

6.1 The Animation Window

The animation window is the containing window for all the visible animation elements. Visually, the animation window contains the drop-down menus and the top-level widget managers, which in turn contain the source, variable display, I/O, control, and instruction counter areas. The main window is also responsible for parsing command-line parameters, initializing the X-Windows connection, initializing the OSF/Motif toolkit, and starting the event loop.

The animation window supports only one public (externally-accessible) routine:

- **AnimatorError** – Reports a major animation error to the user and disables controls based on the error (some errors can be backed out of by reverse executing).

The primary private routines for the animation window are:

- **CreateMainWindow** – Constructs the main window. The drop-down menu, widget managers and all the containing areas are constructed. The callback routines for the drop down menu items are defined and associated here as well.
- **InitAnimator** – Initializes the animation. The state of the animator is initialized with the code file path given as a parameter. The initialization routines for the E-Machine and all the areas are called.
- **main** – The main routine parses the command line parameters, calls CreateMainWindow, and starts the main loop.

6.2 The Control Area

The control area contains the common animation controls. This currently includes the forward execute/advance button, reverse execute button, and the execution mode selector. The callback routines in the control area implement

the algorithms described in chapter 4. Hence, the majority of the animator complexity is contained within the control object.

The public routines for the control area consist of the following:

- **CreateControlArea** – Creates the control area as a child widget of the widget given as a parameter. This routine generates the buttons and mode selectors and adds the callbacks to the corresponding controls.
- **InitControl** – Initializes the control area to its default state. This enables the forward button, disables the reverse button, sets the execution mode to "pause" mode, and initializes the execution state machine.
- **GetControlAreaWidget** – Returns a pointer to the control area widget. This allows other objects to reference the top-level control area widget.
- **GetForwardButton, GetBackwardButton** – Returns a pointer to the control buttons. This allows other objects to reference the execution controls.
- **ForwardButtonOn, ReverseButtonOn, AutoExecButtonOn** – Enables particular control buttons.
- **ForwardButtonOff, ReverseButtonOff, AutoExecButtonOff** – Disable particular control buttons.
- **InputRequestHook** – Routine to be called whenever input is requested. This allows the control object to update displays as it sees fit.

The private routines for the control area consist of:

- **ExecutionCB** – Routine called whenever the user requests execution. This routine is responsible for adjusting the current state and calling the update routines for the various displays appropriately.
- **ModeChangeCB** – Routine called whenever the user requests a change in the execution mode. It is responsible for changing the state of the animation and the source highlight, if necessary.
- **SetForwardButtonState** - Changes the state of the forward execution button to "advance" or "execute."
- **RunTimeError** – Called whenever an E-Machine run-time error occurs to report it and set the state accordingly.

- **ChangeModeBoxes** – Changes the state of the execution mode boxes.
- **RunToEnd** – Ignores any pause directives, executes the E-Machine to the end of the program, and updates the state and displays accordingly.
- **RunToNextPause** – Executes the E-Machine in the current direction until a packet with a pause directive is encountered. Variable and source display are updated as directed.
- **HighlightForState** – Calls the source window highlight set routine according to the current state of the execution.

6.3 The Source Area

The source area is responsible for loading source code into the source area, setting the highlight region, changing the style of the highlight region, and scrolling the pane to display the region containing the currently highlighted region. The area is non-interactive. But the user is allowed to scroll the pane to display any region of the source code.

The source area object defines the following public routines:

- **CreateSourceArea** – Creates the source area. The source area consists of a text pane and scroll bars to views different regions of the source area.
- **InitSourceWin** – Initializes the source area. The contents are initialized from the contents of the E-Machine source memory and the pane is scrolled to the top of the area.
- **GetSourceAreaWidget** – Returns a pointer to the top-level source area widget. This allows other objects to reference the source area.
- **HighlightSource** – Sets the current highlight to correspond with the portion of the source code given in the packet parameter according to the given highlight type parameter. This routine will also scroll the window to display the source *if* the highlight type is non-nil.
- **ClearSourceHighlight** – Removes the current highlight.
- **ChangeVisibleSourceHighlight** – Changes the highlight style of the currently-visible highlighted region.

6.4 The Variable Display Area

The variable display area contains a textual representation of all the currently-active variables and routines of the animated program. It is responsible for updating itself to reflect the current state of the animated program's variables. It does this by calling the VarList VLCreate routine, parsing the resultant data structure, and setting its contents to reflect the values in the VarList. The variable display area is also non-interactive. But the user is allowed to scroll the pane to display any region of the source code. The pane will remain in the same location unless new variables or call frames appear. In this case, the pane will scroll to the end of the variable display area.

The variable display supports the following public routines:

- **CreateVarArea** – Creates the variable display area and its scroll bars.
- **GetVarAreaWidget** – Returns a pointer to the top-level variable display widget. This allows external objects to reference the variable display area.
- **InitVariableWin** – Initializes the variable display by clearing its contents.
- **UpdateVarWin** – Updates the variable window to reflect the state of the E-Machine data memory. A VarList, which describes the values of the active variables, is created and then used to update the variable display. See chapter 5 for a more detailed description of the VarList package.
- **DisplayVars** – Sends the contents of the variable display to standard output (for debugging purposes).

6.5 The Input/Output Area

The I/O object is responsible for maintaining the I/O area. When the E-Machine interprets an OPEN, READ, WRITE, or CLOSE E-code instruction, it calls a corresponding routine in the I/O object. These corresponding

routines, EM_Open_File, EM_Read_File, EM_Write_File, and EM_Close_File, perform the actual I/O operations. Since these instructions can also be reverse executed, the routines EM_Unopen_File, EM_Unread_File, EM_Unwrite_File, and EM_Unclose_File are called when the corresponding E-code instruction is reverse executed. All of these routines operate on an internal string space. To improve animation performance, the string space is mapped to the I/O area only when instructed by a call to UpdateIOWin.

Since I/O can also be performed on external files, the I/O object is also responsible for maintaining information about open files. Since operations on external files may also be reverse executed, the animator must save information so that it may restore the data and position in the file. It is assumed that for any I/O operation that the E-Machine requests from the animator, it will call the corresponding reverse operation during reverse execution. It is also assumed that these operations will be called in exactly the reverse sequence of their forward order. These assumptions allow the I/O object to use a simple stack model for saving information regarding file positions and contents.

As a record of data values read, the I/O object also echos all read values to the I/O window. This is intended to clarify where data values are actually read by the animated program. See figures 3.3 and 3.4 for an illustration of this mechanism.

The I/O object supports the following public routines:

- **CreateIOArea** – Creates the I/O area widget and its scroll bars.
- **InitIOArea** – Initializes the I/O area. This creates the string space and clears the contents of the I/O area.
- **GetIOAreaWidget** – Returns a pointer to the top-level I/O display widget. This allows external objects to reference the I/O display.
- **UpdateIOWin** – Updates the contents of the I/O area from the string space. Since the open, read, write, and close routines work on the string

space and not the I/O display itself, this allows the updating of the I/O area to be controlled by the animator control routines.

- **EM_Open_File, EM_Unopen_File** – These routines are called whenever the E-Machine requests a file open or reverse executes through an open instruction, respectively. These routines receive a filename and open mode as parameters. The animator picks a file descriptor to use for the file and returns it to the E-Machine which then places it on the E-Machine stack. The animated program then uses the descriptor to reference the file in any subsequent READ or WRITE instructions.

- **EM_Read_File, EM_Unread_File** – The Read and Unread routines are called by the E-Machine whenever it encounters a READ instruction while executing forward or while executing in reverse, respectively. The EM_Read_File routine receive a data type, file descriptor, and pointer as parameters. The I/O object then reads in a variable of the given type from the stream identified by the file descriptor, and places the result in the E-Machine data memory at the location designated by the pointer. During reverse execution, the Unread routine receives just a file descriptor so that the I/O object can adjust the information associated with the stream. The I/O object must restore the stream to the state preceding the READ.

- **EM_Write_File, EM_Unwrite_File** – The Write and Unwrite routines are called by the E-Machine whenever it encounters a WRITE instruction while executing forward or while executing in reverse, respectively. The EM_Write_File routine receive a data type, file descriptor, and pointer as parameters. The I/O object then writes a variable of the given type to the stream identified by the file descriptor from the location designated by the pointer in E-Machine data memory. During reverse execution, the Unwrite routine receives just a file descriptor so that the I/O object can adjust the information associated with the stream. The I/O object must restore the stream to the state preceding the WRITE.

- **EM_Close_File, EM_Unclose_File** – The Close and Unclose routines are called by the E-Machine whenever it encounters a CLOSE instruction during forward or reverse execution, respectively. Each receives a file descriptor as a parameter. When a stream is closed, the I/O object simply marks it as "closed" so that any subsequent reads or writes fail. But all information regarding the file must be maintained since the close may eventually be unexecuted. When the Unclose routine is called, the routine simply marks the designated stream as "open" once again.

The I/O object uses the following private routines:

- **OpenStdInput** – This routine is called by the EM_Open_File package whenever a request is made by the animated program to open the standard input (keyboard). It will open a dialog box which allows the user to redirect standard input from a file.
- **PushFilePos, PopFilePos** – The PushFilePos routine saves the current position of the file associated with the descriptor, which is given as a parameter, onto a stack for later retrieval with PopFilePos.
- **ReadChar** – This routine is called whenever the animated program requests the reading of a single character value. If the source descriptor designates an external file or redirected standard input, the next character is read from the source and returned. If the source is not redirected standard input, it will first attempt to get the character from the input buffer. If the input buffer is empty, it will open a dialog box to request a line of input from the user which is then used to fill the input buffer.

6.6 The Instruction Counter

The instruction counter object is responsible for storing the current value of the instruction counter, and for adjusting the counter as requested.

The instruction counter object supports the following public routines:

- **CreateInstCounter** – Creates the instruction counter and the "Reset" button.
- **GetInstCounterWidget** – Returns a pointer to the instruction counter widget so that other objects may reference it.
- **InitInstCounter** – Initializes the instruction counter widget. This sets the value of the counter to "0".
- **IncrementInstCounter, DecrementInstCounter** – Increments or decrements the instruction counter by the given parameter value.
- **SetInstCounter** – Sets the value of the instruction counter to the value of the given parameter.
- **GetInstCounter** – Returns the current value of the instruction counter.

The instruction counter object uses the following private routines:

- **UpdateInstCounter** – Sets the counter to the specified value. This centralizes all the code that modifies the value of the widget.
- **CounterResetCB** – This is the routine called when the "Reset" button is activated. It sets the value of the counter to "0" when called.

BIBLIOGRAPHY

Bibliography

[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.

[Goosey 93] F. W. Goosey. *A miniPascal Compiler for the E-Machine*. Master's thesis. Computer Science Department, Montana State University. April, 1993.

[Heller 91] Dan Heller. *Motif Programming Manual*. O'Reilly and Associates, Inc. 1991.

[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.

[Poole 94] D. K. Poole. *An Ada/CS Compiler for the E-Machine*. Master's thesis. Computer Science Department, Montana State University. July, 1994.

[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", Chapter in the AACE monograph, *Scientific Visualization in Mathematics and Science Education*. 1993.

[BBGPPPR 95] M. R. Birch, C. M. Boroni, F. W. Goosey, S. D. Patton, D. K. Poole, C. M. Pratt, R. J. Ross. "DYNALAB: A Dynamic Computer Science Laboratory Infrastructure Featuring Program Animation", *Proceedings of the Twenty-sixth Symposium on Computer Science Education, ACM SIGCSE Bulletin.*, 27(1):29–33. March 1995.

Appendix A

The E-Machine Instruction Set

This appendix, which is adapted from chapter 2 of Birch's thesis [Birch 90], appendix A of Goosey's thesis [Goosey 93], and appendix A of Poole's thesis [Poole 94], lists all of the instructions in the instruction set of the E-Machine. A pseudo assembly language format is used to describe the instructions and closely resembles the format used in the CODESECTION of the E-Machine object file. The object file is described in detail in chapter 2 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 as critical or noncritical (CFLAG);
- a field denoting the data type of the operand(s) of 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 other fields 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, ADDR

Allocates a block of memory of positive integer size.

Forward: Attempts to allocate the amount of computer words of storage referenced by ADDR. 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 ADDR 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 CFLAG, #

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: Pushes DUMMY onto the evaluation stack.

call CFLAG, #

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 true, according to the instruction, 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 CFLAG, #

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 CFLAG, #

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 as 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 # 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 CFLAG, 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 CFLAG

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 CFLAG

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 CFLAG, 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 CFLAG, 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 CFLAG, #

Pushes # 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, ADDR

Reads a value from the user. The first TYPE is the type of the data to read. The ADDR field is the integer file handle to read from.

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 CFLAG

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 shifts 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 shifts 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 CFLAG, #

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, ADDR

Displays a value for the user. The first TYPE is the type of data to write. The ADDR field is an integer file handle to write to.

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.

End of appendix A

Appendix B

The E-Machine Addressing Modes

This appendix, which is adapted from chapter 2 of Birch's thesis [Birch 90], appendix B of Goosey's thesis [Goosey 93], and chapter 2 of Poole's thesis [Poole 94] 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 the following, where V1 is the variable register associated with A:

```
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
```