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
© Copyright by Craig Matthew Pratt (1995)

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.

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

N378
P8879

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.

6/2/95	Rockford J. Ross
Date	Chairperson, Graduate Committee

Approved for the Major Department

6/2/95	[illegible]
Date	Head, Major Department

Approved for the College of Graduate Studies

6/8/95	[illegible]
Date	Graduate Dean

Statement of Permission to Use

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

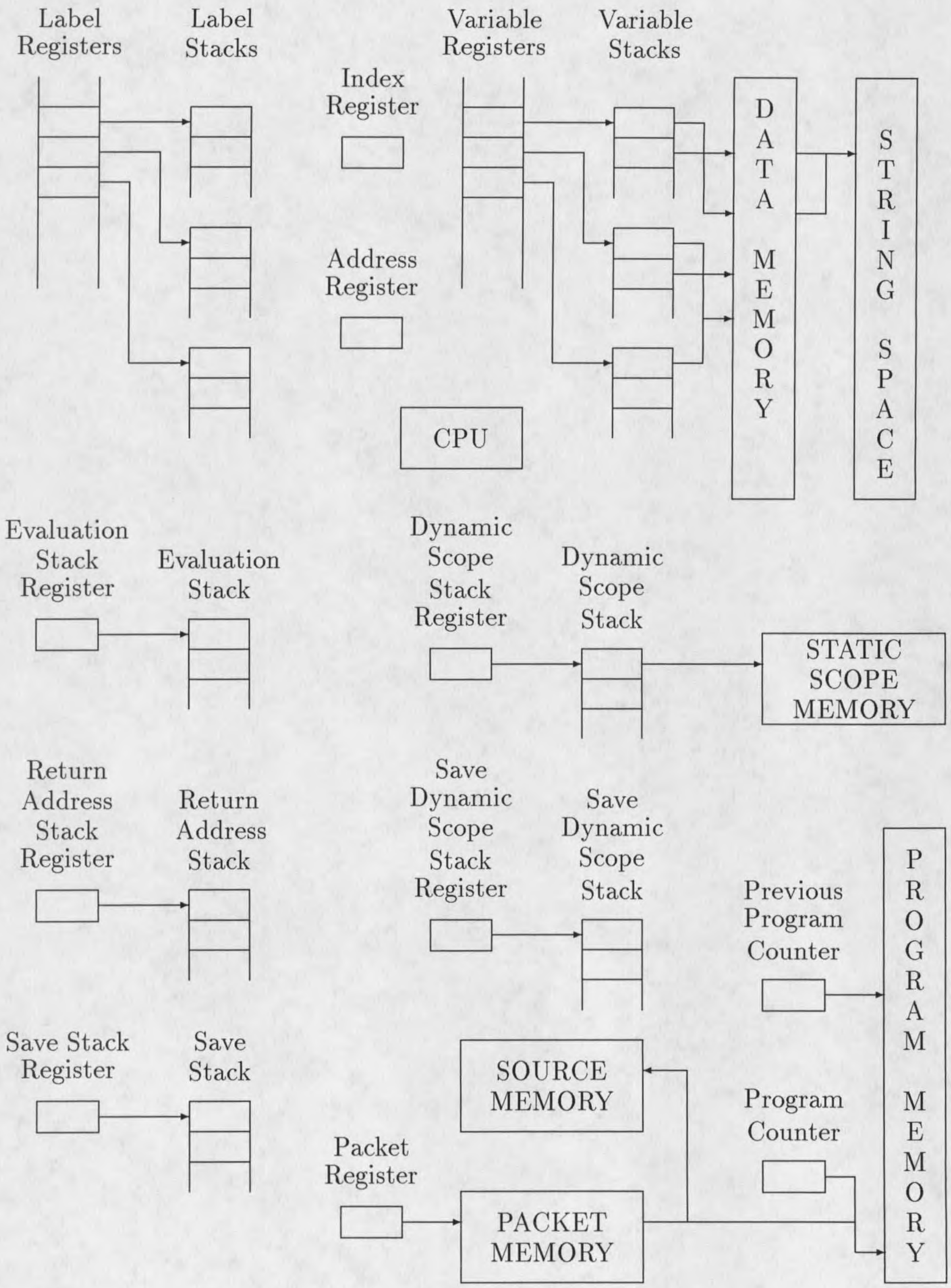

Figure 2.1: The E-Machine

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

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

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

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 multidimensional 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.