An ANSI C compiler for the E-machine
by Torlief James Eneboe

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Torlief James Eneboe (1995)

Abstract:
This thesis is part of the third 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. Frances Goosey designed the first compiler for the E-machine, described in her Master's thesis, A miniPascal Compiler for the E-machine. David Poole designed the second compiler for the E-machine, described in his Master's thesis, An Ada/CS Compiler for the E-machine. This thesis presents the design and implementation of the third compiler for the E-machine. The compiler's source language is ANSI C. The fourth, ongoing phase of the DYNALAB project is the development of animators, used to animate the E-machine code files produced by the compilers. Craig Pratt designed the first animator for the DYNALAB project, described in his Master's thesis, An OSF/Motif Program Animator for the DYNALAB System. Chris Boroni is currently designing an animator for the Microsoft Windows platform.

The ANSI C compiler was developed using C++ and the Purdue Compiler Construction Tool Set (PCCTS) parser development tool. It has successfully generated many object files ready for execution on the E-machine. This thesis focuses on the compilation aspects that are unique to ANSI C, the E-machine architecture, and the planned animation environment.

An ANSI C Compiler for the E-machine

by

Torlief James Eneboe

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

of

Master of Science

in

Computer Science

Montana State University
Bozeman, Montana

June 1995

N378
En272

APPROVAL

of a thesis submitted by

Torlief James Eneboe

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.

6/12/95 — Date

Rockford J. Ross — Chairperson, Graduate Committee

Approved for the Major Department

June 12th 95 — Date

J. Denbigh Starkey — Head, Major Department

Approved for the College of Graduate Studies

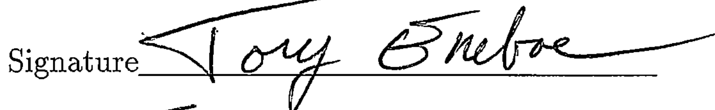
6/23/95 — Date

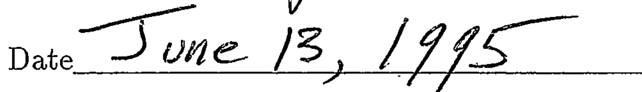
R L Brown — Graduate Dean

STATEMENT OF PERMISSION TO USE

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

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

Signature 

Date 

ACKNOWLEDGMENTS

This thesis is part of a larger software development project, called DYNALAB. The DYNALAB project evolved from an earlier pilot project called DYNAMOD [Ross 91], a program animation system that has been used 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]. Craig Pratt developed the first DYNALAB animator (OSF/Motif) [Pratt 95]. As this thesis is being completed, Chris Boroni is developing a DYNALAB animator for the Microsoft Windows platform.

I would like to take this opportunity to thank my graduate committee members, Dr. Rockford Ross, Dr. Gary Harkin, and Mr. Ray Babcock, and the rest of the faculty members from the Department of Computer Science for their help and guidance during my graduate program. I would also like to thank my thesis advisor, Dr. Ross, and DYNALAB team members, Frances Goosey, Craig Pratt, David Poole, and Chris Boroni, for their help and suggestions for my thesis.

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

Contents

Table of Contents v

List of Tables viii

List of Figures ix

Abstract xi

1 Introduction **1**
- 1.1 The DYNALAB System 1
- 1.2 Preview 3

2 The E-machine **5**
- 2.1 E-machine Design Considerations 6
- 2.2 E-machine Architecture 8
- 2.3 E-machine Emulator 14
- 2.4 E-machine Object File Sections 15
  - 2.4.1 The HEADERSECTION 16
  - 2.4.2 The CODESECTION 16
  - 2.4.3 The PACKETSECTION 16
  - 2.4.4 The VARIABLESECTION 17
  - 2.4.5 The LABELSECTION 17
  - 2.4.6 The SOURCESECTION 18
  - 2.4.7 The STATSCOPESECTION 18
  - 2.4.8 The STRINGSECTION 19
- 2.5 E-machine Compilation Considerations 20
  - 2.5.1 Program Animation Units and E-code Packets 20
  - 2.5.2 Identifying Program Animation Units 21
  - 2.5.3 Translating Program Animation Units into E-code Packets 21
  - 2.5.4 Generation of the Static Scope Table 28
  - 2.5.5 The ProcNum Field 33
  - 2.5.6 The ScopeIndex 38
  - 2.5.7 Translating Enumerated Type Variables 41
  - 2.5.8 Identifying Critical and Non-critical E-code Instructions 43

3 Introduction to the ANSI C Compiler **45**
- 3.1 Unimplemented Features of ANSI C 45
- 3.2 Overview of the ANSI C Compiler 47

4 Parsing ANSI C Using PCCTS **49**
4.1 Overview of PCCTS 49
4.1.1 The PCCTS Scanner 50
4.1.2 The PCCTS Parser 51
4.2 Changes Made to PCCTS 56

5 The ANSI C Symbol Table **58**
5.1 The Symbol Class 58
5.2 The SymbolTable Class 63
5.3 The CrossLinks Class 65
5.4 The HashTable Class 66
5.5 The Specifier Class 67
5.6 The Declarator Class 69
5.7 The BaseSpecOrDecl Class 70

6 Declarations **72**
6.1 The GlobalVarBranch Class 72
6.1.1 Methods Used to Jump Between Global Variable Declarations 74
6.1.2 Methods Used to Jump to Where `main` is Called 75
6.1.3 Methods Used to Jump to the Last Packet 75
6.2 The StaticVariable Class 76

7 Statements **81**
7.1 The SwitchStatement Class 82
7.1.1 Matching a `case` or `default` Label with the Proper `switch` Statement 82
7.1.2 Jumping to the Proper `case` or `default` Label 83
7.1.3 Readjusting the Dynamic Scope of a `case` or `default` Label 84
7.2 The JumpStatements Class 85
7.2.1 Matching a `break` or `continue` Statement with the Proper Loop or `switch` Statement 87
7.2.2 Readjusting the Dynamic Scope of a `break`, `continue`, or `return` Statement 89

8 Expressions **92**
8.1 The PostIncDec Class 93
8.2 The ImplicitFuncCall Class 94
8.3 The EvaluationStack Class 96
8.3.1 Knowing When and When Not to Generate E-code . . 96
8.3.2 Tracking the Types of Expressions 98

9 E-machine Code Generation **103**
9.1 The HeaderSection Class 103
9.2 The StringSection Class 104
9.3 The SourceSection Class 104
9.4 The LabelSection Class 104
9.5 The VariableSection Class 105
9.6 The CodeSection Class 105
9.7 The PacketSection Class 105
9.8 The StatScopeSection Class 106

10 Conclusions and Future Enhancements **108**
10.1 Conclusions 108
10.2 Future Enhancements 108

Bibliography **110**

Appendix A The E-machine Instruction Set **112**

Appendix B The E-machine Addressing Modes **124**

List of Tables

2.1 Packet Table Resulting from Compilation of Program Samp1 . . 26
2.2 Static Scope Table Resulting from Compilation of Program Samp1 29
2.3 Static Scope Block for One Dimensional Array 31
2.4 Static Scope Block for Two Dimensional Array 32
2.5 Scope Block of Record Complex 33
2.6 Static Scope Table Resulting from Compilation of Program Ftrl 36

List of Figures

2.1 The E-machine 9
2.2 Source Code for Program `Samp1` 22
2.3 Animation Units Identified in Program `Samp1` 23
2.4 E-code Instructions Resulting from Compilation of Program `Samp1` 25
2.5 Source Code for Program `Ftrl` 34
2.6 Animation Display After Final Recursive Call of Function `Fact` 35
2.7 Procedure Count Array and Dynamic Scope Stack 37
2.8 Animation Display After Execution of `temp = num1;` 40
2.9 Source Code for Program `Payroll'` 42
2.10 Animation Display After Execution of Program `Payroll` . . . 42
2.11 E-code Instructions Translating `a -= b--;` 44

4.1 Example PCCTS Token Specifications 50
4.2 PCCTS Token Specification for ANSI C Integers 50
4.3 A General PCCTS Grammar Rule 51
4.4 Example PCCTS Grammar Rules 52
4.5 PCCTS Grammar Rule with Actions 52
4.6 PCCTS Grammar Rule with Parameters 53
4.7 Example PCCTS Syntactic Predicate 55
4.8 Example PCCTS Parser Class 57

5.1 Source Code for Partial C Program `Cartoon` 62
5.2 Symbol Structure of Function Declaration `Anderson` 62
5.3 Symbol Table Structure of Partial C Program `Cartoon` 64

6.1 Source Code for Program `Globals` 73
6.2 E-code Instructions Resulting from Compilation of Program `Globals` 77
6.3 Source Code for Program `Statics` 78
6.4 E-code Instructions Resulting from Compilation of Program `Statics` 80

7.1 Source Code for Program `Switch` 85
7.2 E-code Instructions Resulting from Compilation of Program `Switch` 86
7.3 Source Code for Program `Jumps` 90
7.4 E-code Instructions Resulting from Compilation of Program `Jumps` 91

8.1 Source Code for Program `Posts` 94

8.2 E-code Instructions Resulting from Compilation of Program Posts 95
8.3 Source Code for Program Implicit 96
8.4 Source Code for Program Evaluation 100

Abstract

This thesis is part of the third 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. Frances Goosey designed the first compiler for the E-machine, described in her Master's thesis, *A miniPascal Compiler for the E-machine.* David Poole designed the second compiler for the E-machine, described in his Master's thesis, *An Ada/CS Compiler for the E-machine.* This thesis presents the design and implementation of the third compiler for the E-machine. The compiler's source language is ANSI C. The fourth, ongoing phase of the DYNALAB project is the development of animators, used to animate the E-machine code files produced by the compilers. Craig Pratt designed the first animator for the DYNALAB project, described in his Master's thesis, *An OSF/Motif Program Animator for the DYNALAB System.* Chris Boroni is currently designing an animator for the Microsoft Windows platform.

The ANSI C compiler was developed using C++ and the Purdue Compiler Construction Tool Set (PCCTS) parser development tool. It has successfully generated many object files ready for execution on the E-machine. This thesis focuses on the compilation aspects that are unique to ANSI C, the E-machine architecture, and the planned animation environment.

Chapter 1

Introduction

1.1 The DYNALAB System

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

- providing students with facilities for studying the dynamics of programming language constructs—such as iteration, selection, recursion, parameter passing mechanisms, pointers, 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 animated programs, the E-machine had to be designed to be driven by an animator system that controls the execution of programs and displays pertinent information dynamically in animated fashion. 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]. Frances has since extended her work to a nearly complete ISO standard Pascal compiler. The second compiler—Ada/CS, a subset of Ada—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 these compilers, the E-machine and its emulator were again modified as practical considerations uncovered new design issues.

Continuing DYNALAB's third phase, an ANSI C subset compiler and this thesis were written. The ANSI C compiler was developed using C++, and takes advantage of its object oriented features. As happened during the development of the previous two compilers, deficiencies in the E-machine were uncovered and corrected.

The fourth phase of the DYNALAB project, also currently in progress, 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. The first animator—OSF/Motif—was created by Craig Pratt and described in his Master's thesis *An OSF/Motif Program Animator for the DYNALAB System* [Pratt 95]. An animator for the Microsoft Windows platform is currently being developed by Chris Boroni.

The DYNALAB project will not end at this point. A compiler for C++ is in the initial stages of planning. Also, work will continue on the compilers and animators already developed to make them more functional. Algorithm animation (as opposed to program animation—see for example, [Brown 88-1, Brown 88-2]) is also a planned extension to DYNALAB. In fact, the DYNALAB project will likely never be finished, as new ideas and pedagogical conveniences are incorporated as they become apparent.

1.2 Preview

This thesis consists of ten chapters and two appendices. Chapter 1 presents an overview of the thesis and the DYNALAB project in general. Since a thorough understanding of the target virtual computer's architecture and instruction set is required for compiler development, a summary of the E-machine and its emulator is given in chapter 2. Much of the information in chapter 2 is taken from the Patton, Birch, Goosey, and Poole theses. During the ANSI C

compiler development process, it became apparent that some new E-machine features and modifications were necessary or desirable. These changes have been made and are so noted in chapter 2. For a more detailed explanation of the E-machine and its emulator, the reader is referred to the above-mentioned theses.

Chapter 3 provides a brief introduction to the ANSI C compiler. Chapter 4 covers the Purdue Compiler Construction Tool Set (PCCTS), the scanner/parser tool used in the development of the ANSI C compiler. Chapter 5 describes the ANSI C compiler's symbol table. Chapter 6, chapter 7, and chapter 8 describe some of the unique problems, and their solutions, faced by the ANSI C compiler when implementing declarations, expressions, and statements, respectively. Chapter 9 covers E-machine code generation. Finally, chapter 10 draws some conclusions and gives some ideas for future enhancements.

Since there are many E-code examples used throughout this thesis, appendices A and B are included for completeness. Appendix A describes the E-machine instruction set and appendix B describes the E-machine addressing modes. Both of these appendices are adapted from the Patton, Birch, Goosey, and Poole theses.

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], chapters 2 and 3 of Goosey's thesis [Goosey 93], and chapter 2 of Poole's thesis [Poole 94]. This chapter is a summary and update of information from those four theses (much of the material is taken verbatim).

The E-machine is a virtual computer with its own machine language, called E-code. The E-code instructions are described in appendix A; these instructions may reference various E-machine addressing modes, which are described in appendix B. The E-machine's task is to execute E-code translations of high level language programs. The miniPascal language was the first language to be translated into E-code, Ada/CS was the second, and now ANSI C is the third. The real purpose of the E-machine is to support the DYNALAB program animation system, as described more fully in [Birch, *et al* 95], [Ross 91], [Ross 93], [Ross 95], and in Patton's thesis [Patton 89], where it was called a "dynamic display system."

2.1 E-machine Design Considerations

The fact that the E-machine's sole purpose is to support program animation was central to its design. The E-machine operates as follows. After the E-machine is loaded with a compiled E-code translation of a high level language program, it awaits a call from a driver program (the *animator*). A call from the animator causes a group of E-code instructions, called a *packet*, to be executed by the E-machine. A packet contains the E-code translation of a single high level language construct, or *animation unit*, that is to be highlighted by the animator. An animation unit could be a complete high level language assignment statement, for example

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

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

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 and parameters;
- the ability to execute either forward or in reverse.

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

One other aspect of program animation substantially influenced the design of the reversing mechanism of the E-machine. Since the animator is meant to animate high level language programs, the E-machine actually has to be able to effect reversal only through high level language animation units in one reversal step, not each low level E-machine instruction in the packet that is the translation of an animation unit. This observation led to further efficiencies in the design of the E-machine and the incorporation of two classes of E-machine code instructions, critical and 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. However, it should also be noted that the E-machine has the flexibility to accurately execute E-code in reverse, instruction by instruction (rather than a packet at a time), by simply designating each E-code instruction as critical.

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 an E-machine object code file, which is described later in this chapter. The *program counter* contains the address in program memory of the next E-code instruction to be executed. The *previous program counter*, needed for reverse execution, contains the address in program memory of the most recently executed E-code instruction.

Packet memory contains information about the translated E-code packets and their corresponding source language animation units. Packet memory, which is loaded with the information found in the PACKETSECTION of an E-machine object code file, essentially effects the "packetization" of the E-code program found in source memory. Packet information includes the starting and

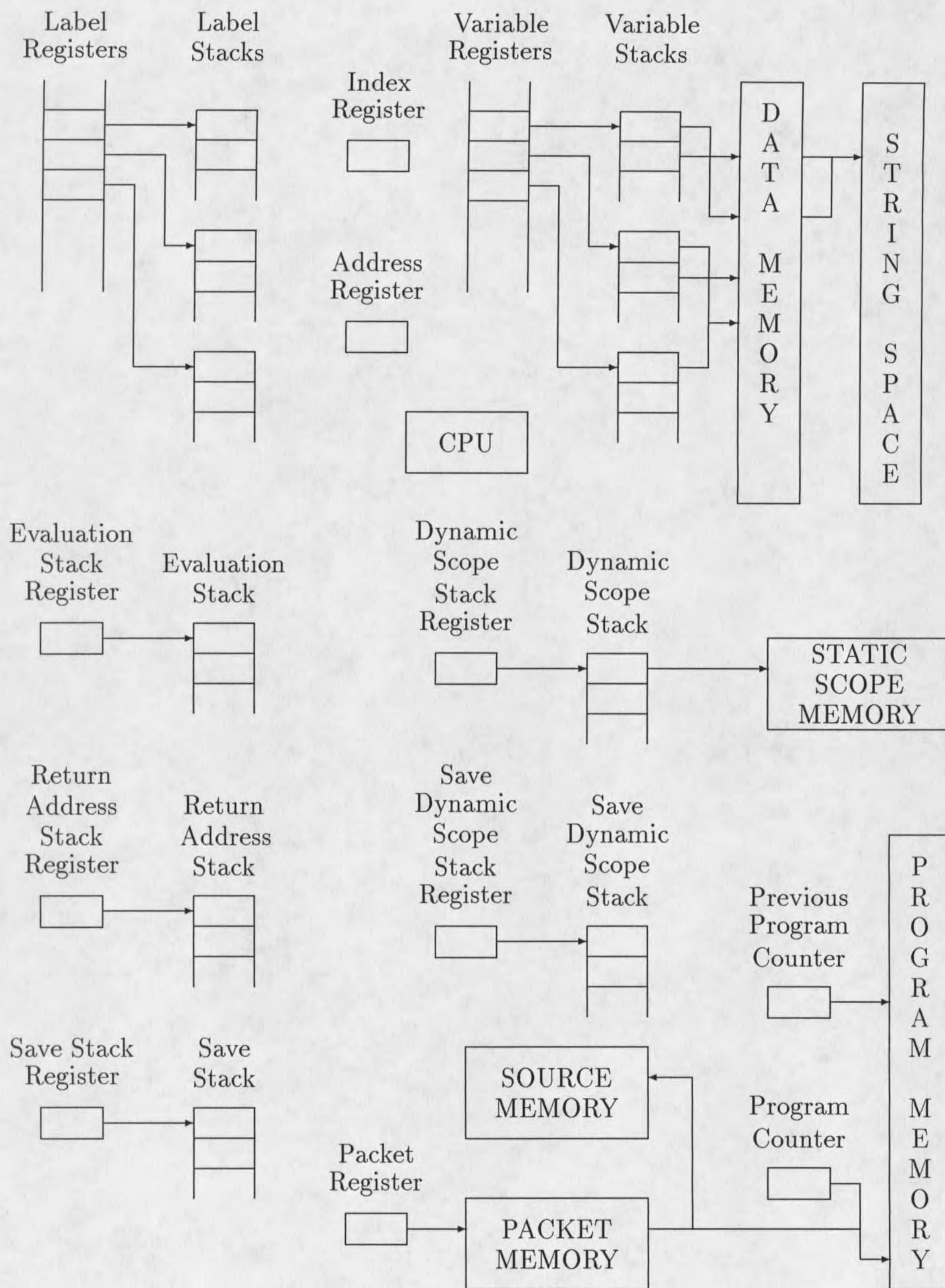


Figure 2.1: The E-machine

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 holds 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.

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

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 being declared in recursive 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. Again, register stacks are needed for reverse execution. 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 an E-machine object file. Currently, this string space is used only by the animator when displaying string constant and enumerated constant values. A more detailed discussion of the interaction of the string space and variable registers is found later in this chapter.

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 ques-

tion. 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 C is a variable that contains a data address. Data at that address can be accessed using the address register via the appropriate E-machine addressing mode. The address register can be used in place of variable registers for any of the addressing modes.

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

The *return address stack* (or *call stack*) is the E-machine's mechanism for

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

The *save stack* contains information necessary for reverse execution. Whenever some critical information (as determined by the execution of a critical instruction) is about to be destroyed, the required information is pushed onto the save stack. This ensures that when backing up, the instruction that most recently destroyed some critical information can be reversed by retrieving that critical information from the save stack. The *save stack 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 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 an 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 function). However, the specific calling sequence resulting in a particular invocation of a 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. (Even though the E-machine's return address stack could have been used

to hold this information, a separate dynamic scope stack was included in the E-machine architecture for clarity.) 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 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 routines 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 in the previous sections of this chapter. The emulator was written in ANSI Standard C for portability and has been compiled on a variety of hardware platforms ranging from an MS-DOS based IBM PC with a variety of C/C++ compilers, to Silicon Graphics and DEC Alpha workstations using GNU C and the system C compilers. 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 from the animator.

2.4 E-machine Object File Sections

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, which may occur in any order. Each section is preceded by an object file record containing the section's name followed by a record that contains a count of the number of records in that particular section. Each of these 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 things will find their way into this section as time goes on.

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:

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

- the packet's starting and ending E-code instruction addresses in program memory;
- the starting and ending line and column numbers in the original source file of the program animation unit corresponding to the packet;
- an index into the current scope block of the Static Scope Table (discussed later in this chapter);
- a variable describing how the animator should display information when the packet is executed in the forward direction (discussed later in this chapter);
- a variable describing how the animator should display information when the packet is executed in the reverse direction (discussed later in this chapter);
- a variable register number that will hold the result of the execution of a conditional expression;
- two variables that are used in conjunction with each other to allow the user to step over language constructs such as functions and loops.

2.4.4 The VARIABLESECTION

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

2.4.5 The LABELSECTION

The LABELSECTION consists of label structures describing the label numbers generated by the compiled program. A label structure consists of a single field that contains the program address at which the corresponding label is defined.

This information is used to initialize the label program address information held in the E-machine's label registers.

2.4.6 The SOURCESECTION

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

2.4.7 The STATSCOPESECTION

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

The Static Scope Table is logically divided into scope blocks, each of which describes identifiers declared within a single static scope of the source program. A more complete discussion of this section is found later in this chapter. Each Static Scope Table entry contains the following information:

- the name of the identifier being described (e.g., a variable name or a function name);

- upper and lower bounds (for array variables);
- the index of the static scope table entry containing the next array index bounds (for multidimensional arrays);
- the offset value (for record fields);
- an enumerated value indicating the data type (e.g., INTEGER, RECORD, or STRING);
- the record size (for arrays of records);
- a pointer to this entry's parent Static Scope Entry;
- a pointer to the child of this entry (e.g., if this static scope entry describes a function, this field would hold the index of the first entry in the static scope block describing the variables declared local to the function);
- a variable register number (for variable names);
- a number statically assigned to program and functions entries; this number is used in determining the dynamic scoping level at execution time;
- a value denoting whether a variable name is an array, and if so, whether it is static or dynamic;
- a value that is an index into the string space (used by variables that are enumerated types or strings);
- a value describing the index type of an array variable (e.g. integer, enumerated, or character). In C, enumerated types and characters are treated as integers. However, languages such as Pascal and Ada have stronger type checking, and thus this field is needed.

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 string space allows the animator to have dynamic access to the names of an enumerated type as well as the internal numeric values corresponding to the names. The animator can also retrieve the values of string constants from the string space.

2.5 E-machine Compilation Considerations

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

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

2.5.1 Program Animation Units and E-code Packets

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

When a high level language program is animated, the animator begins execution by displaying the first several lines of the source code and highlighting the first animation unit in the program. The animator then awaits a response from the user. When the user responds, the animator calls the E-machine to execute the currently highlighted animation unit of the program. Actually, what the E-machine executes is the packet of instructions corresponding to the animation unit. When the E-machine has completed execution of the instructions contained in the packet, control is returned to the animator. The

animator then performs various animation tasks (e.g., displaying pertinent data memory values) and then again awaits a user response before repeating this process by highlighting the next animation unit and so forth. Thus, two of the challenging tasks facing the compiler designer are identifying animation units and properly translating them into E-code packets for successful animation. The following two sections present an example program to illustrate how the C compiler accomplishes these two tasks.

2.5.2 Identifying Program Animation Units

The compiler identifies individual animation units as it is parsing the high level language source code. Consider the C program in figure 2.2 (the numbers on the left correspond to line numbers in the source program file). For this program, the ANSI C compiler identifies the twenty-four animation units shown in figure 2.3 (the numbers on the left correspond to each animation unit's associated packet structure, as discussed in the next section). These animation units will be successively highlighted (in the original source program of figure 2.2) by the animator as it performs the animation of the program. It should be noted that the determination of animation units is arbitrary and can vary from one compiler to another based on subjective esthetics of program animation. As can be seen from this example, an animation unit can correspond to a "chunk" of source code representing a single keyword, an entire program statement, the conditional part of an `if` statement, and so forth.

2.5.3 Translating Program Animation Units into E-code Packets

Once the compiler has identified an animation unit, it must then translate this unit into a corresponding packet of E-code instructions along with an

```
0   int a;
1
2   int func1 (int num1, int num2)
3   {
4      int temp;
5
6      temp = num1;
7      num1 = num2;
8      num2 = temp;
9
10     return num1 + num2 * 8;
11  }
12
13  void main ()
14  {
15     int b = 9, c = 6;
16
17     a = func1 (b, c);
18
19     if (a > c)
20       {
21          int i;
22
23          for (i = 0; i <= 4; ++i)
24             a -= b--;
25       }
26  }
```

Figure 2.2: Source Code for Program Samp1

```
0   int a;
1   int func1 (int num1, int num2)
2   {
3   int temp;
4   temp = num1;
5   num1 = num2;
6   num2 = temp;
7   return num1 + num2 * 8;
8   }
9   void main ()
10  {
11  int b = 9, c = 6;
12  a = func1 (b, c);
13  if
14  (a > c)
15  {
16  int i;
17  for
18  (i = 0;
19  i <= 4;
20  ++i)
21  a -= b--;
22  }
23  }
```

Figure 2.3: Animation Units Identified in Program Samp1

associated descriptive packet structure. Thus, compilation of the example given in figure 2.2 would result in the generation of thirty-eight E-code packets and thirty-eight corresponding packet structures. Fourteen of these packets have no corresponding source code—a situation explained later—so there are actually only twenty-four packets with associated source code. Figure 2.4 shows the pseudo assembly language representation of the E-code instructions generated for the C program shown in figure 2.2. The numbers shown on the left in figure 2.4 correspond to program memory addresses (instruction numbers).

Table 2.1 shows the array of packet structures—called the Packet Table—describing the individual packets resulting from the translation of the program of figure 2.2. The PacketNumber field (column) is included for clarity—it is not actually part of the Packet Table. The first two fields in the Packet Table (StartAddr and EndAddr) give the starting and ending addresses in program memory of the E-code packet. The next four fields (StartLine, StartCol, EndLine, and EndCol) demark the physical location of the packet's corresponding program animation unit in the source program array. The ScopeIndex field in the Packet Table is discussed in the next section of this chapter. The final two fields (DisplayForward and DisplayReverse) provide additional information necessary for animating an animation unit. Three additional fields are not shown in table 2.1—TestResultVar, PktType, and PktScope. These fields were omitted because they are not used by the ANSI C compiler yet.

As might be guessed by the fact that there are twenty-four source animation units and thirty-eight packets, not every packet must correspond to a part of the source code. There are several different ways of displaying packets, which the animator determines by examining the DisplayPkt field of the current