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

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.

Date	Chairperson, Graduate Committee

Approved for the Major Department

Date	Head, Major Department

Approved for the College of Graduate Studies

Date	Graduate Dean

STATEMENT OF PERMISSION TO USE

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

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

Signature________________________________

Date____________________________________

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

List of Tables

List of Figures

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

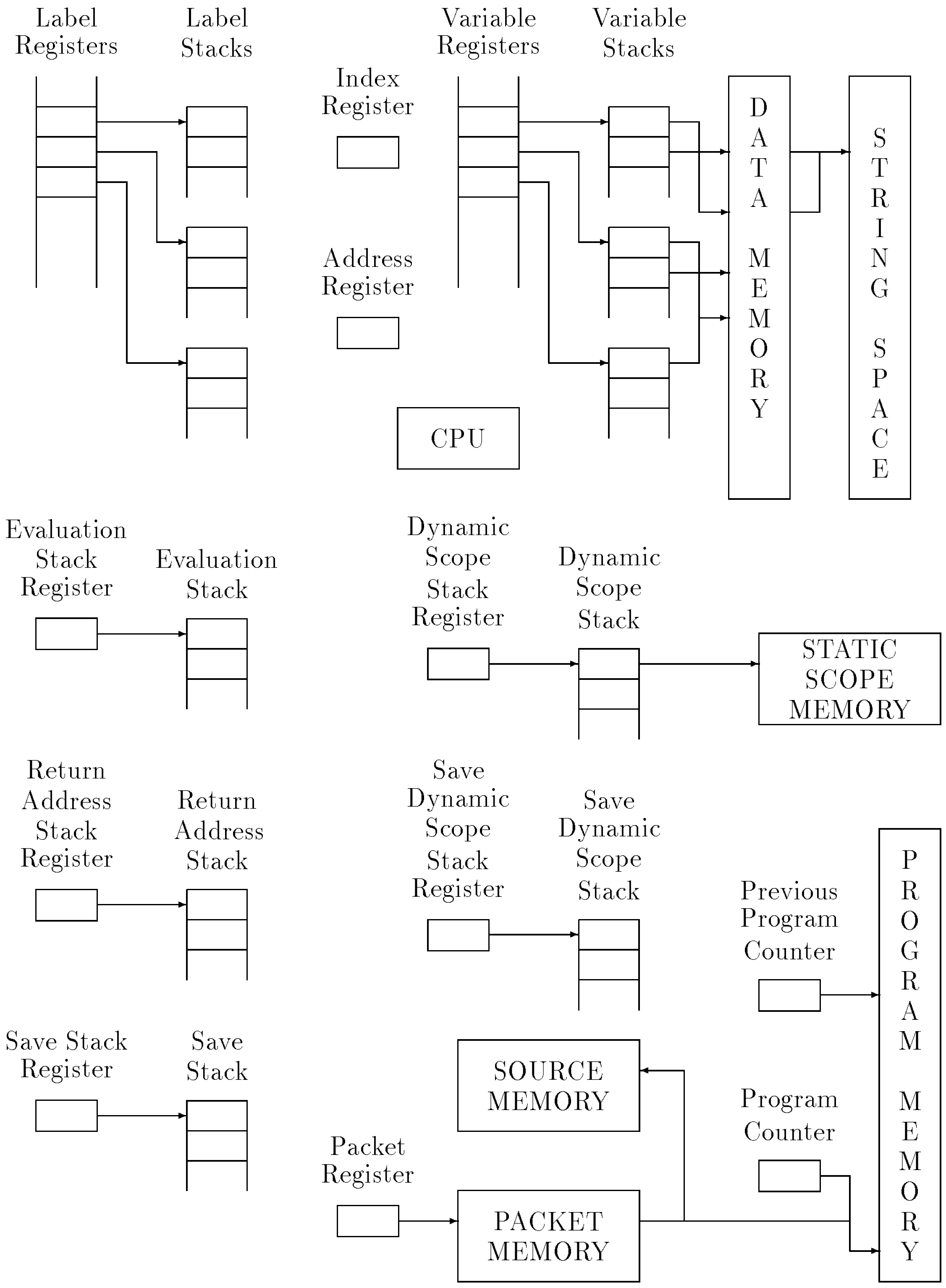


Figure 2.1: The E-machine

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 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 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

```
int a;

int func1 (int num1, int num2)
{
   int temp;

   temp = num1;
   num1 = num2;
   num2 = temp;

   return num1 + num2 * 8;
}

void main ()
{
   int b = 9, c = 6;

   a = func1 (b, c);

   if (a > c)
     {
        int i;

        for (i = 0; i <= 4; ++i)
           a -= b--;
     }
}
```

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

```
  0  pushd   c, DS19
  1  inst    c, V0
  2  inst    c, V1
  3  inst    c, V2
  4  inst    c, V3
  5  br      c, L15
  6  label   c, L0
  7  br      c, L3
  8  label   c, L1
  9  call    c, L5
 10  label   c, L2
 11  br      c, L16
 12  label   c, L3
 13  inst    c, V4
 14  push    c, I, CI0
 15  pop     c, I, V4
 16  br      c, L1
 17  label   c, L4
 18  pushd   c, DS15
 19  inst    c, V6
 20  inst    c, V7
 21  pop     c, I, V7
 22  pop     c, I, V6
 23  nop     c
 24  inst    c, V8
 25  push    c, I, V6
 26  pop     c, I, V8
 27  push    c, I, V8
 28  pop     c, I, V0
 29  push    c, I, V7
 30  pop     c, I, V6
 31  push    c, I, V6
 32  pop     c, I, V0
 33  push    c, I, V8
 34  pop     c, I, V7
 35  push    c, I, V7
 36  pop     c, I, V0
 37  push    c, I, V6
 38  push    c, I, V7
 39  push    c, I, CI8
 40  mult    c, I
 41  add     c, I
 42  pop     c, I, V5
 43  uninst  c, V6
 44  uninst  c, V7
 45  uninst  c, V8
 46  popd    c
 47  return  c
 48  push    c, I, CI0
 49  pop     c, I, V5
 50  uninst  c, V6
 51  uninst  c, V7
 52  uninst  c, V8
 53  popd    c
 54  return  c
 55  label   c, L5
 56  pushd   c, DS16
 57  nop     c
 58  inst    c, V10
 59  push    c, I, CI9
 60  pop     c, I, V10
 61  inst    c, V11
 62  push    c, I, CI6
 63  pop     c, I, V11
 64  push    c, I, V10
 65  push    c, I, V11
 66  call    c, L4
 67  label   c, L6
 68  push    c, I, V5
 69  pop     c, I, V4
 70  push    c, I, V4
 71  pop     c, I, V0
 72  nop     c
 73  push    c, I, V4
 74  push    c, I, V11
 75  gtr     c, I
 76  cast    c, B, I
 77  push    c, I, CI0
 78  eql     c, I
 79  brt     c, L13
 80  pushd   c, DS11
 81  inst    c, V12
 82  br      c, L8
 83  label   c, L7
 84  br      c, L10
 85  label   c, L8
 86  push    c, I, CI0
 87  pop     c, I, V12
 88  push    c, I, V12
 89  pop     c, I, V0
 90  label   c, L9
 91  push    c, I, V12
 92  push    c, I, CI4
 93  leql    c, I9
 94  cast    c, B, I
 95  push    c, I, CI0
 96  eql     c, I
 97  brt     c, L12
 98  br      c, L11
 99  label   c, L10
100  push    c, I, V12
101  push    c, I, CI1
102  addd    c, I
103  pop     c, I, V12
104  push    c, I, V12
105  pop     c, I, V0
106  br      c, L9
107  label   c, L11
108  push    c, I, V10
109  pop     c, I, V0
110  push    c, I, V4
111  push    c, I, V0
112  sub     c, I
113  pop     c, I, V4
114  push    c, I, V4
115  push    c, I, V10
116  push    c, I, CI1
117  sub     c, I
118  pop     c, I, V10
119  pop     c, I, V0
120  br      c, L7
121  label   c, L12
122  uninst  c, V12
123  popd    c
124  br      c, L14
125  label   c, L13
126  label   c, L14
127  push    c, I, CI0
128  cast    c, I, C
129  pop     c, C, V9
130  uninst  c, V10
131  uninst  c, V11
132  popd    c
133  return  c
134  label   c, L15
135  inst    c, V9
136  push    c, I, CI0
137  cast    c, I, C
138  pop     c, C, V9
139  inst    c, V5
140  push    c, I, CI0
141  pop     c, I, V5
142  br      c, L0
143  label   c, L16
144  uninst  c, V0
145  uninst  c, V1
146  uninst  c, V2
147  uninst  c, V3
148  uninst  c, V9
149  uninst  c, V5
150  uninst  c, V4
151  popd    c
```

Figure 2.4: E-code Instructions Resulting from Compilation of Program `Samp1`

Packet Number	Start Addr	End Addr	Start Line	Start Col	End Line	End Col	Scope Index	Display Forward	Display Reverse
0	0	5	-1	-1	-1	-1	0	00	00
1	6	7	-1	-1	-1	-1	0	00	00
2	8	9	-1	-1	-1	-1	0	00	00
3	10	11	-1	-1	-1	-1	0	00	00
4	12	16	0	0	0	5	1	07	07
5	17	22	2	0	2	29	2	07	07
6	23	23	3	0	3	0	2	07	07
7	24	24	4	3	4	11	3	07	07
8	25	28	6	3	6	14	3	07	07
9	29	32	7	3	7	14	3	07	07
10	33	36	8	3	8	14	3	07	07
11	37	46	10	3	10	25	3	07	06
12	47	47	-1	-1	-1	-1	2	00	01
13	48	54	11	0	11	0	2	07	07
14	55	56	13	0	13	11	0	07	07
15	57	57	14	0	14	0	0	07	07
16	58	63	15	3	15	19	2	07	07
17	64	71	17	3	17	19	2	07	07
18	72	72	19	3	19	4	2	07	07
19	73	79	19	6	19	12	2	07	07
20	80	80	20	6	20	6	0	07	07
21	81	81	21	9	21	14	1	07	07
22	82	82	23	9	23	11	1	07	07
23	83	83	-1	-1	-1	-1	1	01	00
24	84	84	23	9	23	11	1	07	07
25	85	89	23	13	23	19	1	07	07
26	90	98	23	21	23	27	1	07	07
27	99	106	23	29	23	32	1	07	07
28	107	107	-1	-1	-1	-1	1	00	00
29	108	119	24	12	24	20	1	07	07
30	120	120	-1	-1	-1	-1	1	00	00
31	121	121	-1	-1	-1	-1	1	01	00
32	122	123	25	6	25	6	2	07	07
33	124	124	-1	-1	-1	-1	2	00	00
34	125	126	-1	-1	-1	-1	2	00	00
35	127	133	26	0	26	0	3	07	07
36	134	142	-1	-1	-1	-1	3	00	00
37	143	151	-1	-1	-1	-1	0	01	00

Table 2.1: Packet Table Resulting from Compilation of Program `Samp1`

packet. The DisplayPkt of the packet structure is an 8-bit field made up by combining the following several flags together:

- Update variable display after execution when going forward;
- Pause before execution of this packet when going forward;
- Highlight the source code for this packet when going forward;
- Update variable display after execution when going backward;
- Pause before execution of this packet when going backward;
- Highlight the source code for this packet when going backward.

The standard display packet would have all these flags set. Packets without Highlight-Forward, Pause-Forward, Highlight-Backward, and Pause-Backward are effectively "invisible" and are executed automatically by the animator. These "invisible" packets are very useful for situations in which there is no source code animation unit for the corresponding E-code being executed. For example, in the packets in figure 2.4 for the source code in figure 2.2, packets 12 and 23, among others, are invisible. Neither of these two packets has a corresponding animation unit in the source code.

TestResultVar is a field that was added to further facilitate program animation. This field is the number of a variable register that holds the result of a conditional expression for the animator to display. For example, execution of the expression

```
if (j<5 && !k>=5 || flag) ...
```

would be difficult for a user to follow. The compiler would generate code to store the result of the conditional expression evaluation in a variable register and set the TestResultVar to that register number. The animator may use this variable register to display the result (0 or 1), making understanding the program simpler.

PktType and PktScope are two fields that were again added to facilitate program animation. These fields allow the user to step over programming language constructs such as functions and loops. For example, consider the following code:

```
for (i=0; i<10000; i++)
{
  ...
}
```

It would be very time consuming if the user had to step through all of the code associated with this `for` loop. The PktType and PktScope fields provide the animator with the information necessary to allow the user to step over constructs (such as the loop above) if desired.

2.5.4 Generation of the Static Scope Table

The compiler writer must also provide information describing all of the data memory variables that the animator must display. This information is provided in the Static Scope Table, a linear array which is, in turn, logically divided into numerous scope blocks. Each scope block describes the identifiers (e.g., variable names and function names) declared in a single static scope in a program. Even though this information is obtained from the compiler's symbol table, the generation of the Static Scope Table is not a straightforward task due to scope nesting characteristics of many high level languages.

Table 2.2 shows the Static Scope Table that is generated as a result of compiling the C program given in figure 2.2. The Entry (entry number) column, or field, is included for clarity—it is not part of the Static Scope Table. Six fields—upper bound, lower bound, next index, offset, record size, and string index—were omitted because they were all unused. This Static Scope Table consists of five scope blocks—a block describing the identifiers declared within the scope of function `func1` (entries 0–4), a block describing the identifiers declared within the scope of the unnamed block that is the `then` part of the `if` statement (entries 5–7), a block describing the identifiers declared within the scope of function `main` (entries 8–12), a block describing the identifiers declared within the scope of the program (entries 13–17), and a bootstrap block describing the program entry (entries 18–20).

Entry	Id Name	Type	Parent	Child	Var Reg	Proc Num
Scope block describing function `func1`						
0		HEADER	13	-	-	-
1	num1	INTEGER	-	-	6	-
2	num2	INTEGER	-	-	7	-
3	temp	INTEGER	-	-	8	-
4		END	-	-	-	-
Scope block describing unnamed block for `then` *part of* `if` *statement*						
5		HEADER	8	-	-	-
6	i	INTEGER	-	-	12	-
7		END	-	-	-	-
Scope block describing function `main`						
8		HEADER	13	-	-	-
9	b	INTEGER	-	-	10	-
10	c	INTEGER	-	-	11	-
11	*Unnamed*Block*	PROCEDURE	-	5	-	1
12		END	-	-	-	-
Program scope block						
13		HEADER	18	-	-	-
14	a	INTEGER	-	-	4	-
15	func1	FUNCTION	-	0	-	2
16	main	FUNCTION	-	8	-	3
17		END	-	-	-	-
Bootstrap scope block						
18		HEADER	-	-	-	-
19	*C*Program*	PROGRAM	-	13	-	0
20		END	-	-	-	-

Table 2.2: Static Scope Table Resulting from Compilation of Program `Samp1`

The bootstrap block contains three entries: the HEADER and END entries that delimit the scope block and a PROGRAM entry containing information about the program itself. There are two fields of interest in the PROGRAM entry; these are the child pointer field (Child) and the procedure number field (ProcNum). The Child field contains the index of the first entry of the scope block describing the identifiers declared in the program. The ProcNum field contains a compiler-generated number that is used in conjunction with dynamic scoping.

The entries in the scope block describing the identifiers declared in the program scope consist of the HEADER and END delimiter entries as well as entries describing each of the scope's identifiers. The Parent field of the HEADER entry in this scope block contains the index of the first entry of the bootstrap scope block. This scope block's two FUNCTION entries—describing functions `func1` and `main`—use the Child field, which contains the index of the first entry of the scope block describing the identifiers declared in those functions. The ProcNum field is also used in the FUNCTION entries; it contains a compiler-generated number to be used in conjunction with dynamic scoping.

The entries in the scope block describing the identifiers declared in function `main` consist of the HEADER and END delimiter entries as well as entries describing each identifier declared in the scope, in this case the function's local variables and an unnamed block. The Parent field of the HEADER entry of this scope block contains the index of the first entry of the scope block containing the function's declaration. The Child field of the unnamed block entry contains the index of the first entry of the scope block containing the identifiers declared within. Again, the ProcNum field is used by this entry and is a compiler-generated number to be used in conjunction with dynamic scoping.

The entries in the scope block describing the unnamed block associated with the `then` part of the `if` statement consist of the HEADER and END delimiter entries as well as entries describing each identifier declared in the scope, in this case the unnamed block's local variables. The Parent field of the Header entry of this scope block contains the index of the first entry of the scope block containing the unnamed block's declaration.

The entries in the scope block describing the identifiers declared in function `func1` consist of the HEADER and END delimiter entries as well as entries describing each identifier declared in the scope, in this case the function's parameters and a local variable. The Parent field of the HEADER entry of this scope block contains the index of the first entry of the scope block containing the function's declaration.

Simple variables such as integers, floats, and characters, may be simply described in the static scope table by a name, type, and variable register. Aggregate types, such as arrays and records, need more description. Consider, for example, the array declaration

```
int a[10];
```

In order for the animator to correctly display the elements of this array, it would have to know about the element type of the array and its ranges. The same may be said for records—the animator needs to know the names and types of the record's elements. Separate scope blocks are made to describe records and arrays greater than one dimension.

A static scope block for the above array could be as shown in table 2.3. Unused fields are omitted from the table.

En try	Id Name	Array Type	Upr Bnd	Lwr Bnd	Type	Var Reg	Proc Num	Index Type
0	ProcName	-	-	-	HEADER	-	1	-
1	a	STATIC	9	0	INTEGER	3	-	INTEGER
2	-	-	-	-	END	-	-	-

Table 2.3: Static Scope Block for One Dimensional Array

The array variable `a` at position 1 defines the type of the array (STATIC), the element type of the array (INTEGER), the constant upper and lower bounds of the array, and the index type of the array (INTEGER).

A two-dimensional array is more complex. Because there is only one set of fields for each dimension, further dimensions must be placed in their own scope blocks. Suppose we had the following array declaration, whose static scope table is shown in table 2.4 (again, unused fields are omitted).

```
int a[10][2];
```

The first dimension is described by the variable `a` in position 4 and the second dimension's static scope position is the NextIndex field of the structure, which is 0. The second dimension is described in the static scope block from 0 to 2.

En try	Id Name	Array Type	Upr Bnd	Lwr Bnd	Nxt Idx	Type	Var Reg	Proc Num	Index Type
0	-	-	-	-	-	HEADER	-	-	-
1	-	STATIC	1	0	-	-	-	-	INTEGER
2	-	-	-	-	-	END	-	-	-
3	ProcName	-	-	-	-	HEADER	-	1	-
4	a	STATIC	9	0	0	INTEGER	3	-	INTEGER
5	-	-	-	-	-	END	-	-	-

Table 2.4: Static Scope Block for Two Dimensional Array

Record static scope blocks and their variables are handled a bit differently. Static scope entries for record variables have their child field set to the static scope table position of their types. By following these child indices, the animator can find all necessary record type information. For example, suppose we have the record

```
struct Complex
{
   float real_part;
   float imag_part;
};
struct Complex c1, c2;
```

The static scope table for the record and its variables is shown in table 2.5 (again, unused fields are not shown). The record variables in entries 5 and 6 contain the record size and a child index set to the static scope block that describes their record type. Entry 0 is the start of the block describing record `Complex`. Each record member has its offset set to the byte offset from the start of the record.

En try	Id Name	Off set	Type	Rec Siz	Ch ild	Var Reg	Proc Num
0	Complex	-	HEADER	-	-	-	-
1	real_part	0	REAL	-	-	-	-
2	imag_part	4	REAL	-	-	-	-
3	-	-	END	-	-	-	-
4	ProcName	-	HEADER	-	-	-	1
5	c1	-	RECORD	8	0	7	-
6	c2	-	RECORD	8	0	8	-
7	-	-	END	-	-	-	-

Table 2.5: Scope Block of Record `Complex`

2.5.5 The ProcNum Field

As each program, function, and unnamed block is encountered during compilation, it is assigned a unique procedure number. The identifier names are referred to as *static scope names* in the following discussion. The procedure number is produced by a counter variable in the compiler's semantic analysis module. Thus, the procedure number assigned to a C program entry is always 0. The next static scope name declaration encountered in the program would be assigned the procedure number 1, and so on. A static scope name's procedure number is stored as one of its symbol table attributes. This number is then placed in the ProcNum field of the Static Scope Table entry describing the static scope name.

The animator uses the ProcNum field in conjunction with the dynamic scope stack when determining the dynamics of program execution. The use of this field is best explained by an example. The program shown in figure 2.5 contains a recursively called function (`Fact`). That `Fact` is recursive implies that for any given call to function `Fact`, the animator must be able to determine the depth of the pertinent data memory values associated with the variables declared in function `Fact`, as well as the depths of any variables in the calling (program) scope. These values are retrieved by querying the appropriate variable stacks, as discussed earlier in this chapter. Thus, upon the final recursive call to function `Fact`, the animator should be able to display data memory values as shown in figure 2.6. The arrow (`==>`) pointing to the statement `if (n == 0)` indicates where animation proceeds.

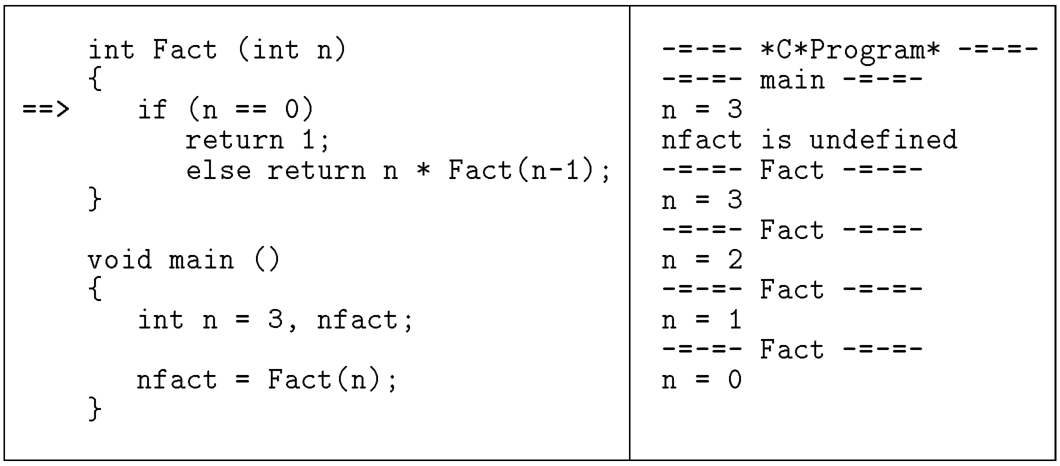
```
int Fact (int n)
{
   if (n == 0)
      return 1;
      else return n * Fact(n-1);
}

void main ()
{
   int n = 3, nfact;

   nfact = Fact(n);
}
```

Figure 2.5: Source Code for Program `Ftrl`

After the E-machine has been loaded with the E-code translation of a source program, the animator queries the E-machine to determine the total number of static "procedure" scopes that are described in the Static Scope Table. The Static Scope Table for the example in figure 2.5 is shown in table 2.6. The animator then dynamically allocates a *procedure count array* that contains an entry corresponding to each of these scopes. Thus, for the program shown in figure 2.5, this array has three entries. Entry 0 corresponds to the



```
    int Fact (int n)
    {
==>    if (n == 0)
          return 1;
          else return n * Fact(n-1);
    }

    void main ()
    {
       int n = 3, nfact;

       nfact = Fact(n);
    }
```

```
-=-=- *C*Program* -=-=-
-=-=- main -=-=-
n = 3
nfact is undefined
-=-=- Fact -=-=-
n = 3
-=-=- Fact -=-=-
n = 2
-=-=- Fact -=-=-
n = 1
-=-=- Fact -=-=-
n = 0
```

Figure 2.6: Animation Display After Final Recursive Call of Function `Fact`

program scope, entry 1 corresponds to function `Fact`, and entry 2 corresponds to function `main`. During program animation, the animator sets the values of the procedure count array entries to reflect the current number of active calls to the corresponding function. (This means that the animator reinitializes the values in the procedure count array *every* time control is passed to the animator.) At the same time, the E-machine's dynamic scope stack contains a history of active scopes, with the Static Scope Table entry number of the most current scope being the value at the top of this stack.

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

```
    if (n == 0)
```

the procedure count array and the dynamic scope stack are in the state shown in figure 2.7. The values in the procedure count array indicate that the program `Ftrl` has one active "call", the function `main` has one active "call", and that function `Fact` has four active "calls". In this example, the animator

Entry	Id Name	Type	Parent	Child	Var Reg	Proc Num
Scope block describing function `Fact`						
0		HEADER	7	-	-	-
1	n	INTEGER	-	-	5	-
2		END	-	-	-	-
Scope block describing function `main`						
3		HEADER	7	-	-	-
4	n	INTEGER	-	-	7	-
5	nfact	INTEGER	-	-	8	-
6		END	-	-	-	-
Program scope block						
7		HEADER	11	-	-	-
8	Fact	FUNCTION	-	0	-	2
9	main	FUNCTION	-	3	-	1
10		END	-	-	-	-
Bootstrap scope block						
11		HEADER	-	-	-	-
12	*C*Program*	PROGRAM	-	7	-	0
13		END	-	-	-	-

Table 2.6: Static Scope Table Resulting from Compilation of Program `Ftrl`

begins its retrieval of data memory values by examining the value at the *bottom* of the dynamic scope stack. The bottom stack value is 12, which means that the animator now examines the twelfth entry in the Static Scope Table. This entry is a PROGRAM entry describing `Ftrl`. The ProcNum field in the PROGRAM entry has the value 0. Next, the animator will examine entry 0 in the procedure count array to determine the depth of the variables to be displayed for this invocation of the program scope. Since the program scope cannot be called recursively, this value will always be 1. Thus, when the animator retrieves the values of the variables described in the program's child scope block, it will instruct the E-machine to retrieve the data memory values associated with the top of the appropriate variable stacks. After these values have been displayed, the animator decrements the value in entry 0 of the procedure count array.

Next, the animator examines the value in entry 1 in the dynamic scope stack. This value is 9, corresponding to the ninth entry in the Static Scope

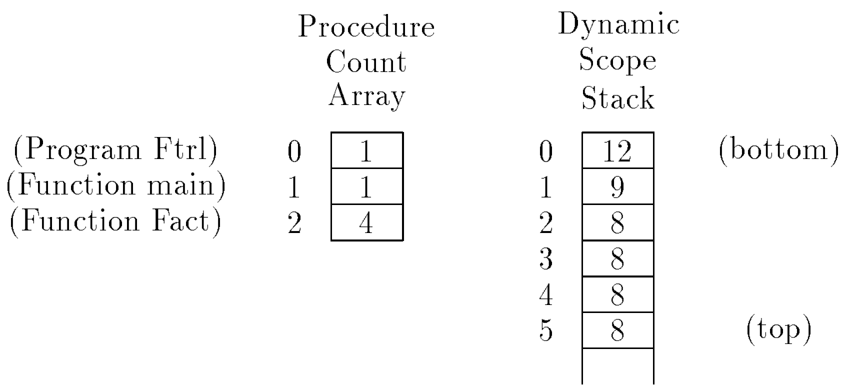


Figure 2.7: Procedure Count Array and Dynamic Scope Stack

Table. This entry, whose ProcNum field has the value 1, describes function `main`. The animator then examines entry 1 in the procedure count array. The current value in this entry is 1, indicating that the animator should instruct the E-machine to retrieve data memory values associated with the first level of the appropriate variable stacks when displaying variable values described in the function's child scope block. These values reflect the function's variable values resulting from its initial call from the program scope. The animator then decrements the value in entry 1 of the procedure count array.

Finally, the animator examines the value in entry 2 in the dynamic scope stack. This value is 8, corresponding to the eighth entry in the Static Scope Table. This entry, whose ProcNum field has the value 2, describes function `Fact`. The animator then examines entry 2 in the procedure count array. The current value in this entry is 4, indicating that the animator should instruct the E-machine to retrieve data memory values associated with the fourth level of the appropriate variable stacks when displaying variable values described in the function's child scope block. These values reflect the function's variable values resulting from its initial call from the program scope. The animator then decrements the value in entry 1 of the procedure count array. The animator continues this process three more times—until the procedure count array associated with the function `Fact` is 0, resulting in the display shown in figure 2.6.

2.5.6 The ScopeIndex

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

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

```
num1 = num2;
```

in function `func1` of figure 2.8. After receiving a response from the user, the animator then calls the E-machine to execute the E-code packet corresponding to this animation unit. When the E-machine returns control to the animator, the animator must then determine the relevant data memory values to be displayed following any changes that resulted from execution of the packet. This task is accomplished by querying the E-machine's dynamic scope stack, which contains a history of the active scopes. In this example, the dynamic scope stack currently consists of three entries, each containing an index into the Static Scope Table shown in table 2.2. The top entry contains the value 15, the second entry contains the value 16, and the bottom entry contains the value 19. These values indicate to the animator that function `func1` (Static Scope Table entry number 15) is the most current active scope, function `main` (entry number 16) is the calling scope of `func1`, and that program `Samp1` (entry number 19) is the calling scope of `main`. By using the child pointers associated with these three Static Scope Table entries, the animator can now determine the appropriate data memory values to be displayed. Figure 2.8 shows a possible animation resulting from the execution of this animation unit. The arrow (`==>`) pointing to the statement `num1 = num2;` indicates where animation proceeds.

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

```
{
```

found on line 3 of figure 2.2 and has returned control to the animator. The animator, via a query of the dynamic scope stack, now determines that the values of the variables contained in the outer program scope, the function `main` scope, and the function `func1` scope should be displayed. The variables listed in the block describing `func1`'s scope variables are `num1`, `num2`, and `temp`. However, at this point in the program's execution, variable `temp` has not yet been declared, and thus should not be displayed. The ScopeIndex field of the packet structure associated with the above animation unit contains the value 2. This value indicates to the animator that it should only display data memory values for entries numbered 0, 1, and 2, in the window associated with the *most current* active scope block which in this case starts at scope table index

```
    int a;

    int func1 (int num1, int num2)
    {
       int temp;

       temp = num1;
==>    num1 = num2;
       num2 = temp;

       return num1 + num2 * 8;
    }

    void main ()
    {
       int b = 9, c = 6;

       a = func1 (b, c);

       if (a > c)
         {
            int i;

            for (i = 0; i <= 4; ++i)
               a -= b--;
         }
    }
```

```
-=-=- *C*Program* -=-=-
a = 0
-=-=- main -=-=-
b = 9
c = 6
-=-=- func1 -=-=-
num1 = 9
num2 = 6
temp = 9
```

Figure 2.8: Animation Display After Execution of `temp = num1;`

0. Hence, the animator will display the values of the variables `num1` and `num2` (0 stands for the HEADER entry), but not the value of the variable `temp`.

2.5.7 Translating Enumerated Type Variables

Ordinarily, only the internal numeric value of an enumerated type variable is required in translated object code. It is desirable, however, for program animation purposes to have the animator display the enumerated constant name rather than just the internal numeric value of a variable of an enumerated type. Thus, when translating an enumerated type variable, the compiler must provide a way for the animator to relate the variable's internal numeric value to its corresponding constant name. This task is facilitated by the string space component of the E-machine. The string space holds the enumerated constant names (as well as string literals) defined in a C program.

The String Section consists of a statically allocated character array containing all of the enumerated constant names defined in a C program, as well as the values of any string literals declared in the source program (which may also need to be displayed by the animator). When the compiler encounters the definition of a string literal or an enumerated constant name, it stores that name in the string section.

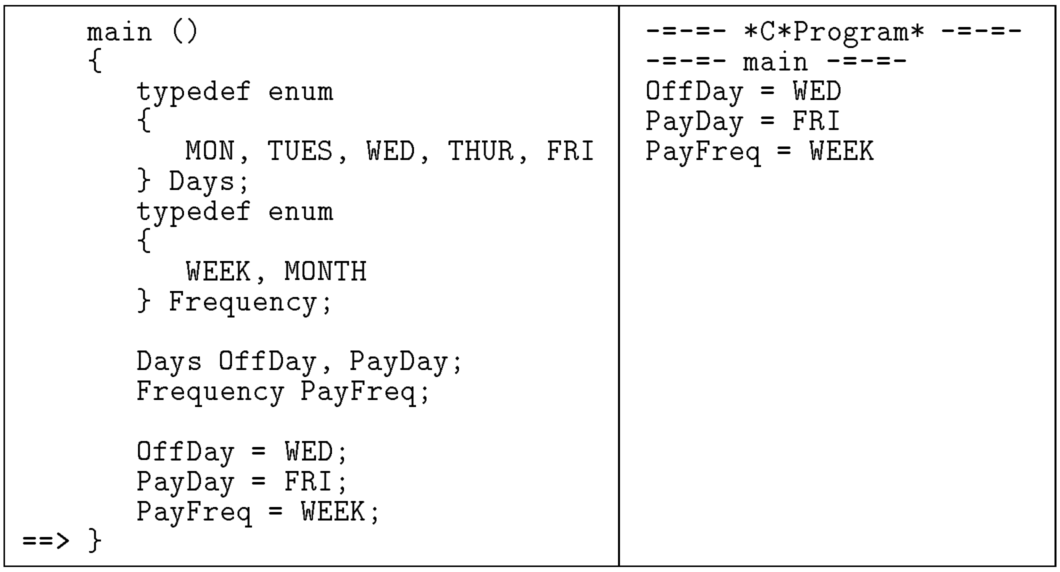
When a program is animated, the String Section portion of the E-code file is loaded into the E-machine's string space. The string space is then accessed by the animator for displaying string constants and enumerated variable values. For example, upon completion of execution of the program in figure 2.9, the animator can display the enumerated type variable values as shown in figure 2.10. The arrow (`==>`) pointing to the statement `}` indicates where animation proceeds.

```
main ()
{
   typedef enum
   {
      MON, TUES, WED, THUR, FRI
   } Days;
   typedef enum
   {
      WEEK, MONTH
   } Frequency;

   Days OffDay, PayDay;
   Frequency PayFreq;

   OffDay = WED;
   PayDay = FRI;
   PayFreq = WEEK;
}
```

Figure 2.9: Source Code for Program Payroll



```
    main ()
    {
       typedef enum
       {
          MON, TUES, WED, THUR, FRI
       } Days;
       typedef enum
       {
          WEEK, MONTH
       } Frequency;

       Days OffDay, PayDay;
       Frequency PayFreq;

       OffDay = WED;
       PayDay = FRI;
       PayFreq = WEEK;
==> }
```

```
-=-=- *C*Program* -=-=-
-=-=- main -=-=-
OffDay = WED
PayDay = FRI
PayFreq = WEEK
```

Figure 2.10: Animation Display After Execution of Program Payroll

2.5.8 Identifying Critical and Non-critical E-code Instructions

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

```
a -= b--;
```

in figure 2.2 corresponds to the packet of E-code instructions numbered 108 through 119 in figure 2.4. All of these instructions are marked critical via the "c" operand. Only instructions 109, 113, 118, and 119 are actually critical, however, as only they result in critical information being destroyed. That is, the old value of those variable registers is being destroyed by popping a new value into them. For reverse execution, the old values of these variable registers must be saved before a new value is popped into them. Thus, the packet of E-code instructions corresponding to this animation unit could be generated as shown in figure 2.11, where the operand "n" indicates a non-critical instruction.

```
108  push     n, I, V10
109  pop      c, I, V0
110  push     n, I, V4
111  push     n, I, V0
112  sub      n, I
113  pop      c, I, V4
114  push     n, I, V4
115  push     n, I, V10
116  push     n, I, CI1
117  sub      n, I
118  pop      c, I, V10
119  pop      c, I, V0
```

Figure 2.11: E-code Instructions Translating `a -= b--;`

Chapter 3

Introduction to the ANSI C Compiler

The ANSI C programming language was established as a standard by the American National Standards Institute in 1983. It was originally developed and distributed freely with the UNIX operating system. It has fast become one of the most popular programming languages because of its simplicity and power. Nowadays, with C compilers available for almost every platform, and with the extension of C to C++, the popularity of the language will no doubt continue to rise.

The grammar I used for implementing the ANSI C compiler was taken from [Kernighan 88]. This book was co-authored by Brian Kernighan and Dennis Ritchie, two of the orignal "inventors" of the C programming language. It is an excellent reference and has served me well.

3.1 Unimplemented Features of ANSI C

Because of the complexity involved in designing a compiler, especially one for the E-machine, only part of the ANSI C language was implemented in this thesis. The following are the ANSI C features that have not been implemented yet:

- Enumerated Types
- Structures
- Unions
- Bit-fields
- Pointers
- Function Prototypes
- Arrays
- Strings
- Typedefs
- Gotos
- Preprocessor Directives
- Standard C Library Functions
- The following storage class specifiers, type specifiers, and type qualifiers:
  - register
  - extern
  - short
  - long
  - unsigned
  - signed
  - const
  - volatile

Of course, when one of these features is implemented, it will have to be integrated into the current compiler so that it works with all other features that are currently present. For example, when structures are implemented, they will have to be made to work with features such as `sizeof` operator, explicit casts, and function return types, all of which are part of the current compiler.

3.2 Overview of the ANSI C Compiler

The ANSI C compiler was developed using C++. There are currently twenty-three classes that make up the ANSI C compiler. The following is a breakdown of where these classes fit into the overall picture. Each of the classes are discussed in depth in subsequent chapters.

There is one class that is used for describing the ANSI C parser, which is covered in chapter 4. The class is:

- ANSICParser

There are seven classes that are used for symbol table management, which is covered in chapter 5. The classes are:

- Symbol
- SymbolTable
- CrossLinks
- HashTable
- Specifier
- Declarator
- BaseSpecOrDecl

There are two classes that are used for handling declarations, which are covered in chapter 6. The classes are:

- GlobalVarBranch
- StaticVariable

There are two classes that are used for handling statements, which are covered in chapter 7. The classes are:

- SwitchStatement

- JumpStatements

There are three classes that are used for handling expressions, which are covered in chapter 8. The classes are:

- PostIncDec
- ImplicitFuncCall
- EvaluationStack

There are eight classes that are used for E-machine code generation, which is covered in chapter 9. The classes are:

- HeaderSection
- StringSection
- SourceSection
- LabelSection
- VariableSection
- CodeSection
- PacketSection
- StatScopeSection

Chapter 4

Parsing ANSI C Using PCCTS

4.1 Overview of PCCTS

The ANSI C compiler discussed in this thesis was developed using a relatively new compiler construction tool called the Purdue Compiler Construction Tool Set (PCCTS), which was developed by Terrence Parr and Will Cohen [Parr 93]. PCCTS generates an integrated scanner and parser from an extended BNF specification describing the ANSI C tokens and grammar. Unlike other contemporary bottom up compiler tools, such as yacc [Mason 90], PCCTS generates an LL(k) recursive descent parser. Also unlike yacc, PCCTS isn't yet a standard compiler construction tool, but it is freely available with no restrictions. It is written in ANSI C and has been used with numerous operating systems, including Unix, DOS, and OS/2.

Since PCCTS is relatively new and doesn't yet have the assumed familiarity of yacc, the next few sections will describe, in part, using PCCTS to scan and parse ANSI C. There are a few things to note. PCCTS doesn't use two specification files—one for the scanner and one for the parser—as does lex/yacc. Both specifications are within one file, although PCCTS does use two separate programs to generate the scanner and the parser. Because the parser generated by PCCTS is recursive descent, for a large grammar (such as for ANSI C), the resulting parser will be large, much larger than an LR parser generated by yacc, for example.

4.1.1 The PCCTS Scanner

Unlike the case with lex, a popular scanner tool, the PCCTS scanner specification is an integrated part of the parser specification. To describe a token, C preprocessor-like definitions are used, as shown in figure 4.1.

```
#token WHILE         "while"
#token SHIFTLEFT     "\<\<"
```

Figure 4.1: Example PCCTS Token Specifications

Both `WHILE` and `SHIFTLEFT` will act as tokens, or terminals, within the parser's grammar. More complex tokens are described using regular expressions, as is the case for ANSI C decimal integers, shown in figure 4.2.

```
#token DECIMALINT      "[1-9][0-9]*"
```

Figure 4.2: PCCTS Token Specification for ANSI C Integers

Lexclasses

An interesting feature of the PCCTS scanner is the lexclasses. With separate lexclasses, under different circumstances the scanner will have entirely different behavior, as if there were multiple scanners that trade control. For example, the following four lexclasses are used by the ANSI C scanner in this thesis:

- A lex class for scanning ANSI C comments (e.g. – /* comment */).

- A lex class for scanning ANSI C strings (e.g. – "string").
- A lex class for scanning ANSI C characters (e.g. – 'c').
- A lex class for scanning everything else in ANSI C (known as the language lexclass).

The language lexclass starts out with control. When this lexclass finds the start of a comment, string, or character, it passes control to the appropriate lexclass described above. After the comment, string, or character has been scanned by that lexclass, control is passed back to the language lexclass.

4.1.2 The PCCTS Parser

PCCTS uses extended BNF rules to generate a parser to recognize a language. A general form of a PCCTS rule is shown in figure 4.3. By convention, terminals (tokens) are capitalized and nonterminals are all lowercase. Tokens can be directly inserted into the parser specification without having to create a separate named scanner token (as in figure 4.1).

```
rule_1 : rule_2 Terminal ;
```

Figure 4.3: A General PCCTS Grammar Rule

As an example, the rule specifications to recognize `parameter_list` and `parameter_declaration_list` are shown in figure 4.4. In figure 4.4, the ... in double quotes is an unnamed token. Being able to place tokens directly into the parser specification makes the specification more readable. `COMMA` is a named token. Non-terminals include `parameter_list`, `parameter_declaration_list`, and `parameter_declaration`. Each non-terminal used in a rule must have its own rule definition somewhere else in the grammar.

```
parameter_list : parameter_declaration_list { COMMA "..." } ;

parameter_declaration_list : parameter_declaration
                             (
                                COMMA parameter_declaration
                             )* ;
```

Figure 4.4: Example PCCTS Grammar Rules

Grammar Actions

What use would a parser be if there were no way to perform a rule action? Just as yacc uses {} to delimit actions, PCCTS uses <<>>. Some simple actions are shown in figure 4.5, which involves generating actions during the parse of the `parameter_declaration_list` rule of figure 4.4.

```
parameter_declaration_list :
   << printf ("entering parameter_declaration_list."); >>

   parameter_declaration
   << printf ("found a parameter_declaration!"); >>

   (
      COMMA parameter_declaration
      << printf ("found another parameter_declaration!"); >>
   )*

   << printf ("leaving parameter_declaration_list."); >>
   ;
```

Figure 4.5: PCCTS Grammar Rule with Actions

Actions may appear anywhere within a rule (even before the first rule element). An action may even appear outside a rule.

Rule Parameters

Communication between rules is performed in two ways—by way of a traditional semantic stack and by a new method: rule parameters. Because PCCTS generates recursive descent compilers, each rule is a C++ method of the parser class (described later), and PCCTS allows the user to pass parameters to the rules (methods), as shown in figure 4.6. Processing of the `postfix_expression` rule of figure 4.6 in turn requires processing of the `argument_expression_list` rule, which has a pass-by-reference parameter—namely `num_parameters`.

```
postfix_expression :
   << int num_parameters = 0; >>
   IDENTIFIER LPAREN
   {
      argument_expression_list[&num_parameters]
   }
   RPAREN
   ;

argument_expression_list[int *num_parameters] :
   assignment_expression
   << num_parameters++; >>
   (
      COMMA assignment_expression
      << num_parameters++; >>
   )*
```

Figure 4.6: PCCTS Grammar Rule with Parameters

Accessing Token Information

Of course, during the parsing phase, it is often necessary to retrieve information about tokens. PCCTS offers the following methods to retrieve information about a particular token:

- getText()—retrieves the text (lexeme) of the specified token.
- line()—retrieves the line number that the specified token is on.
- begcol()—retrieves the column number that the first character of the specified token is on.
- endcol()—retrieves the column number that the last character of the specified token is on.

A token is specified by `LT(x)`, where "x" tells the scanner how many tokens ahead to look into the token buffer. Thus, `LT(1)->line()` would get the line number associated with the next look-ahead token in the token buffer, while `LT(3)->getText()` would get the text associated with the third look-ahead token in the token buffer.

Syntactic Predicates

Often, when designing an LL(k) compiler for a large language, it is impossible to remove all of the conflicts from the grammar. PCCTS offers a tool known as a *syntactic predicate* to help deal with just this problem. A syntactic predicate, in essence, allows an unbounded look-ahead for certain rules. This differs from, say, a standard LL(2) parser, which allows a look-head of at most two tokens at any one time. Syntactic predicates are often used on rules where it is very difficult, if not impossible, to remove an LL conflict from the rule. For example, in figure 4.7 a syntactic predicate is used to remove an ambiguity from the rule `external_declaration`. The syntactic predicate (enclosed in the `()?` structure) is evaluated to determine whether the `function_definition` rule or `global_declaration` rule should be expanded. The synactic predicate can look ahead an unbounded number of tokens to determine which rule to expand. Without the syntactic predicate, this rule would be ambiguous in a standard LL(2) parser because, with just two tokens of look-ahead, the parser could not determine which rule it should expand. For example, consider the token sequence

```
unsigned long int
```

With only two tokens of look-ahead to work with, the parser could not determine if this token sequence is the start of a global variable declaration, as in

```
unsigned long int a, b, c;
```

or the start of a function definition, as in

```
unsigned long int func1 ()
{
   ...
}
```

Thus, the parser would not be able to determine which of the two rules (`function_definition` or `global_declaration`) to expand.

For efficiency reasons, PCCTS uses the unbounded look-ahead approach only for rules with syntactic predicates in them. Otherwise, it uses standard LL(k) parser techniques, where "k" is defined ahead of time by the compiler writer.

```
external_declaration :
     ({ declaration_specifiers } declarator LBRACE)?
          function_definition
        | global_declaration
   ;
```

Figure 4.7: Example PCCTS Syntactic Predicate

The Parser Class

Older versions of PCCTS produced straight C code. Recently, newer versions of PCCTS have been released that produce C++ code. These newer versions require that a parser class be created to hold the rules of the grammar. Figure 4.8 shows an example of a parser class. The rule `start` will become a public method of the class `ExampleParserClass`. The variable `aftera` will become a private member of the class `ExampleParserClass`, while the method `inBetween` will become a protected member of the same class. Thus, through the parser class, the compiler writer is able to take advantage of C++ niceties such as information hiding and encapsulation.

The ANSI C compiler described in this thesis was developed with one of the newer versions of PCCTS (1.31) that produces C++ code, and thus required a parser class. The parser class of this compiler is named ANSICParser.

There is one final point to note about PCCTS. A compiler requires a lot of semantic processing. If all of the code to handle the semantic processing were put into the file containing the grammar, this grammar file would become messy very quickly. Thus, much of the semantic processing code of the ANSI C compiler resides in separate files, with member methods of the parser class accessing this code, as needed, from within the grammar. For example, in figure 4.8, the method `inBetween` would contain some semantic processing code associated with the rule `start`. Furthermore, this code (the method `inBetween`) would reside in a separate file.

4.2 Changes Made to PCCTS

Since PCCTS is so new, it is not yet entirely stable. In developing the ANSI C compiler described in this thesis, a few changes had to be made to the PCCTS

```
class ExampleParserClass
{
   <<
      private :
         int aftera = 0;
      protected :
         int inBetween (int);
   >>

   start : ("a")*
           << aftera = 1;  inBetween (aftera); >>
           ("b")*
           ;
};
```

Figure 4.8: Example PCCTS Parser Class

code to get things to work properly. Following is a list of these changes:

- In the file generic.h, the value of the constant ZZLEXBUFSIZE was changed from 2000 to 10000. This was done because more member functions had to be declared in the parser class than the buffer could hold, causing a warning message to be generated by PCCTS.

- In the file config.h, the name of the constant WORDSIZE was changed to AWORDSIZE. This was done because the E-machine has a variable of the same name, causing the warning message to be generated by the compiler.

- In the files AToken.h, DLexerBase.C, and DLexerBase.h, code was added to keep track of column information (beginning and ending) used for highlighting purposes during program animation. This was done by adding the methods begCol and endCol to the classes in these files. The methods begCol and endCol mimic the code that is already present in PCCTS for keeping track of line information for tokens.

Chapter 5

The ANSI C Symbol Table

The symbol table is the heart of a compiler. Not surprisingly, then, it is also one of the most complicated parts of a compiler. Most of the ideas for the ANSI C symbol table were taken from [Holub 90]. One major difference, though, is that Holub implemented his symbol table in C, whereas the ANSI C compiler's symbol table is implemented in C++. The following seven sections detail the seven classes used to implement the ANSI C symbol table.

5.1 The Symbol Class

The ANSI C symbol table can be thought of as a collection of symbols (objects) of type Symbol. Each symbol describes a single declaration found while parsing a C program—such as a variable or a function. Because there is a lot of information associated with each symbol, the Symbol class is quite large. The following is a detailed description of the Symbol class.

The data members of the Symbol class are as follows:

- name—name of the symbol. For example, if the symbol describes a function, name would contain the name of the function.
- name_space—name space that the symbol belongs to. This is needed because C has several different name spaces (that do not interfere with each other) to which an identifier can belong. For a detailed discussion of C name spaces, see appendix A, section 11.1 of [Kernighan 88].

- next_bucket_element—a pointer to the next symbol in the symbol's bucket. This data member is closely tied in with the HashTable class (discussed below).

- prev_bucket_element—a pointer to the previous symbol in the symbol's bucket. This data member is closely tied in with the HashTable class (discussed below).

- next_cross_link—a pointer to the next symbol in the symbol's cross link. This data member is closely tied in with the CrossLinks class (discussed below).

- first_type—a pointer to the first element in a linked list of types that describes the type of the symbol. This data member is closely tied in with the Specifier, Declarator, and BaseSpecOrDecl classes (all discussed below).

- return_type—if the symbol is a function symbol, return_type points to another symbol that describes the return type of the function.

- first_parameter—if the symbol is a function symbol, first_parameter points another symbol (the first one in a linked list of parameters) that describes the first parameter of the function.

- next_parameter—if the symbol is a parameter symbol, next_parameter points to a symbol that describes the next parameter.

- var_reg—if the symbol is a variable symbol, var_reg holds the E-machine variable register number associated with the variable.

- pushd_pos—if the symbol is a function or unnamed block symbol, then pushd_pos holds the position in the code section where the PUSHD instruction associated with the function or unnamed block is located.

- label_num—if the symbol is a function symbol, label_num holds the label number associated with the start of the function in E-code.

The methods of the Symbol class are as follows:

- Copy—makes a copy of all of the data members and puts them into a new object (symbol).

- SetName—sets the contents of the name data member.

- GetName—gets the contents of the name data member.

- SetNameSpace—sets the contents of the name_space data member.
- GetNameSpace—gets the contents of the name_space data member.
- SetNextBucketElement—sets the contents of the next_bucket_element data member.
- GetNextBucketElement—gets the contents of the next_bucket_element data member.
- SetPrevBucketElement—sets the contents of the prev_bucket_element data member.
- GetPrevBucketElement—gets the contents of the prev_bucket_element data member.
- SetNextCrossLink—sets the contents of the next_cross_link data member.
- GetNextCrossLink—gets the contents of the next_cross_link data member.
- AddTypeToBegOfList—adds a new type to the beginning of the linked list of types. The new type is an object of type BaseSpecOrDecl. Thus, either a specifier or a declarator is added to the linked list of types.
- AddTypeToEndOfList—adds a new type to the end of the linked list of types. The new type is an object of type BaseSpecOrDecl. Thus, either a specifier or declarator is added to the linked list of types.
- GetFirstType—gets the contents of the first_type data member. In other words, this method gets the first element in the linked list of types.
- GetLastType—gets the last element in the linked list of types.
- SetReturnType—sets the contents of the return_type data member.
- GetReturnType—gets the contents of the return_type data member.
- AddParameterToEndOfList–adds a new parameter symbol to the end of the linked list of parameters.
- GetFirstParameter—gets the contents of the first_parameter data member. In other words, this method gets the first element in the linked list of parameters.

- GetNextParameter—gets the contents of the next_parameter data member.
- GetNumOfParameters–gets the number of parameters in the linked list of parameters.
- AddParametersToSymbolTable–adds each parameter in the linked list of parameters to the symbol table. It does this by making a copy of the parameter and then adding that copy to the current scope. This allows the parameters of a function to be treated as if they were declared as local variables.
- SetVarReg—sets the contents of the var_reg data member.
- GetVarReg—gets the contents of the var_reg data member.
- SetPushdPos—sets the contents of the pushd_pos data member.
- GetPushdPos—gets the contents of the pushd_pos data member.
- SetLabelNum—sets the contents of the label_num data member.
- GetLabelNum—gets the contents of the label_num data member.
- GetSize—gets the E-machine size associated with the type of the symbol. For example, for a symbol being used to describe an integer variable, GetSize would return EM_INTEGERSIZE.
- GetType—gets the E-machine type associated with the type of the symbol. For example, for a symbol being used to describe an integer variable, GetType would return EM_INTEGER.
- DumpContents—prints the contents of all of the data members.

Figure 5.2 shows what the symbol for the function declaration on line 2 of figure 5.1 would look like. For the symbol with the name `Anderson`, the values for the fields next_bucket_element, prev_bucket_element, and next_cross_link are shown in figure 5.3. Any other blank field in figure 5.2 is either unused or assigned a default value.

```
int Todd;

float Anderson (unsigned char BuzzCut, long int *Stewart[50])
{
   int Beavis;
   if (expression)
      {
         int BuzzCut, McVicker;
      }
}
```

Figure 5.1: Source Code for Partial C Program `Cartoon`

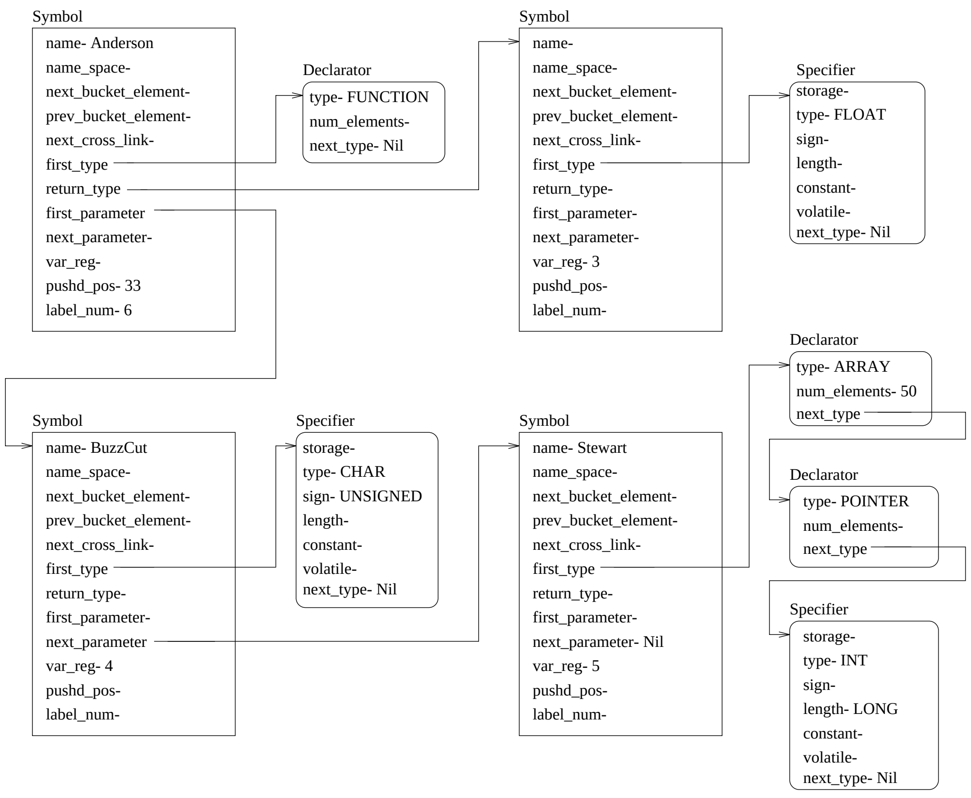


Figure 5.2: Symbol Structure of Function Declaration `Anderson`

5.2 The SymbolTable Class

As previously discussed, the ANSI C symbol table is a collection of symbols of type Symbol. The symbols are organized within the symbol table using a combination of two different structures—a hash table (implemented by the HashTable class discussed below) and a set of cross link structures (implemented by the CrossLinks class discussed below). The hash table provides fast access to symbols in the symbol table. The cross links provide access to symbols in the symbol table based on their scoping level. The cross links are needed because the hash table is oblivious to scoping rules. Yet, if the symbol table were implemented with just cross links, searching for a symbol would be inefficient, since each cross link structure is a linked list. Thus, the hash table solves the major problem associated with cross links and visa-versa. Figure 5.3 shows what the ANSI C compiler's symbol table would look like for the partial C program found in figure 5.1. All fields except for name, next_bucket_element, prev_bucket_element, and next_cross_link are omitted from the symbols.

The SymbolTable class provides the interface to the ANSI C symbol table. Most of the actual work is done by methods in the Symbol, CrossLinks, and HashTable classes. The following is a list of the methods associated with the SymbolTable class:

- PushScope—creates a new scope in the symbol table.
- FakePushScope–fakes the creation of a new scope in the symbol table. This method is needed by the SwitchStatement class so that it can readjust the dynamic scope of a `case` or `default` label. The reason for this readjusting is discussed in chapter 7, where the SwitchStatement class is covered in detail.
- PopScope—removes the current scope (including all of the symbols in that scope) from the symbol table.
- FakePopScope–fakes the removal of the current scope from the symbol table. This method is needed by the JumpStatements class so that

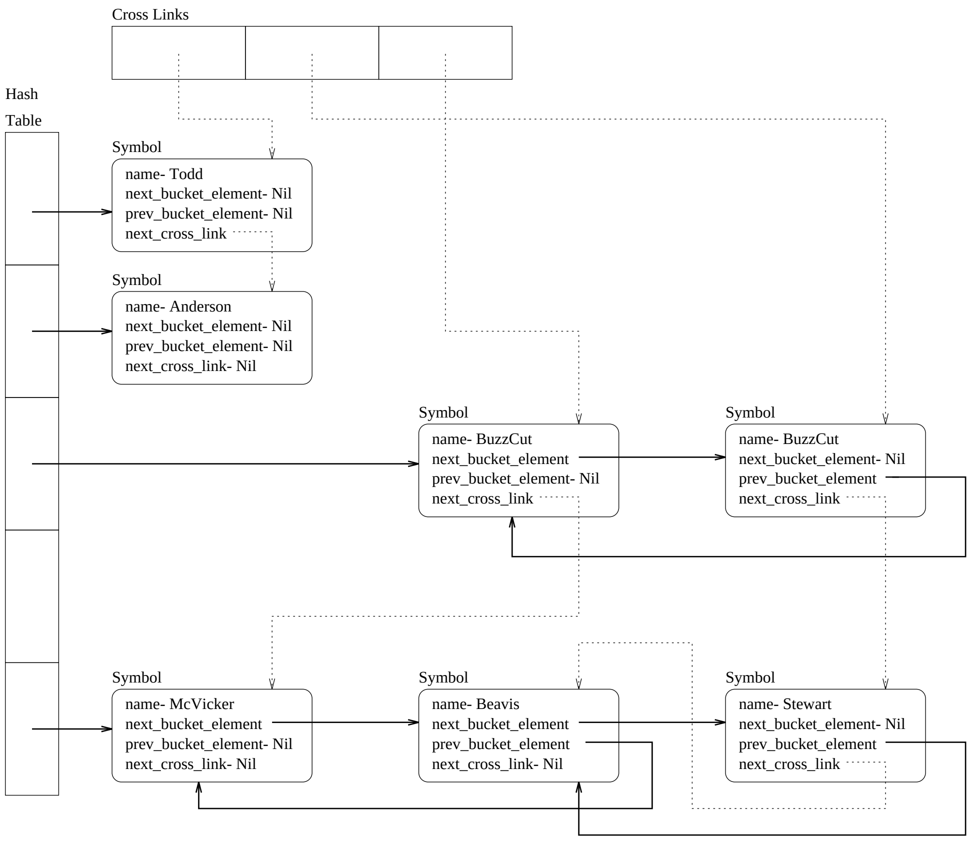


Figure 5.3: Symbol Table Structure of Partial C Program `Cartoon`

it can readjust the dynamic scope of a `break`, `continue`, or `return` statement. The reason for this readjusting is discussed in chapter 7, where the JumpStatements class is covered in detail.

- GetScope—gets the number associated with the current scope of the symbol table.
- SetScope—sets the number associated with the current scope of the symbol table.
- GetScopeIndex—gets the number of symbols in the current scope of the symbol table.
- AddSymbol—adds a symbol to the current scope of the symbol table.
- SymbolInCurrScope—checks to see if a duplicate symbol is already present in the current scope of the symbol table.
- FindVariableSymbol—searches for a variable symbol in the symbol table.
- FindFunctionSymbol—searches for a function symbol in the symbol table.
- DumpContents—prints the contents of all symbols in the symbol table.

5.3 The CrossLinks Class

As previously discussed, the ANSI C symbol table is a combination of a hash table and a set of cross link structures. The CrossLinks class, as its name implies, provides the implementation for the cross links part of the ANSI C symbol table. The following is a list of the methods associated with the CrossLinks class:

- PushScope—creates a new cross link structure.
- FakePushScope–fakes the creation of a new cross link structure by laying down INSTs and PUSHDs. The reason for this method will become clear when the SwitchStatement class is discussed in chapter 7.
- PopScope—removes the current cross link structure by removing every symbol in the structure. As each symbol is encountered, the HashTable class method DeleteSymbol is called so that the symbol is also removed from the hash table.

- FakePopScope–fakes the removal of the current cross link structure by laying down UNINSTs and POPDs. The reason for this method will become clear when the JumpStatements class is discussed in chapter 7.
- GetScope—gets the current number of cross link structures.
- SetScope—sets the number of cross link structures.
- GetScopeIndex—gets the number of symbols in the current cross link structure.
- AddSymbol—adds a symbol to the current cross link structure. This method then calls the HashTable class method InsertSymbol so that the symbol also gets added to the hash table.
- SymbolInCurrScope—checks to see if a symbol is in the current cross link structure.
- FindVariableSymbol—first, this method calls HashTable class method FindSymbol. This performs a fast check of whether or not the variable symbol is in the symbol table. If the symbol is in the symbol table, it searches for the symbol using the cross link structures, starting at the current cross link structure and working out towards the outer-most cross link structure.
- FindFunctionSymbol—searches through the outer-most cross link structure for a function symbol.
- DumpContents—prints the contents of the every symbol in every cross link structure.

5.4 The HashTable Class

As previously discussed, the ANSI C symbol table is a combination of a hash table and a set of cross links. The HashTable class, as its name implies, provides the implementation for the hash table part of the ANSI C symbol table. The following is a list of the methods associated with the HashTable class:

- HashIt—used to obtain a hash value from a key by applying a hash function to it.

- InsertSymbol—inserts a symbol into the hash table.
- FindSymbol—finds a symbol in the hash table.
- DeleteSymbol—removes a symbol from the hash table.

The hash table implemented by the HashTable class uses chaining as its collision resolution strategy. Thus, the method HashIt is used by the methods InsertSymbol, FindSymbol, and DeleteSymbol to determine which hash table bucket a particular symbol belongs to.

5.5 The Specifier Class

A variable declaration in C can be thought of as having two parts: a specifier part (discussed here) and a declarator part (discussed in the next section). The specifier part of a C variable declaration is a combination of the various keywords used in describing the variable. For example, on line 2 of figure 5.1, *long* and *int* are part of the specifier for the parameter `Stewart`. C keywords associated with specifiers can be broken down into six classes—storage, type, sign, size, constant (or not), and volatile (or not). All six classes share one common trait—the attributes within a single class are mutually exclusive. In other words, a specifier for a given variable declaration can contain only one attribute from each of the six classes. It should be noted that specifier part for a given variable declaration is constrained—there are only a limited number of legal combinations of keywords that can be used. Thus, the specifier part of a variable declaration can be represented by a single structure. The Specifier class is used to control the attributes associated with this structure. Below is a discussion of the methods that are part of the Specifier class. The methods are grouped by use.

The following methods are used to set the storage of the specifier:

- SetAuto
- SetRegister
- SetStatic
- SetExern
- SetTypedef

The following methods are used to set the type of the specifier:

- SetInt
- SetFloat
- SetDouble
- SetChar
- SetVoid
- SetStruct
- SetUnion
- SetEnum
- SetLabel

The following methods are used to set the sign of the specifier:

- SetSigned
- SetUnsigned

The following methods are used to set the size of the specifier:

- SetShort
- SetLong

The following method is used to set the specifier to be constant:

- SetConstant

The following method is used to set the specifier to be volatile:

- SetVolatile

The following methods can be used to retrieve the settings of the six attributes of the specifier:

- GetStorage
- GetType
- GetSign
- GetLength
- GetConstant
- GetVolatile

The following are general methods:

- Copy—makes a copy of the specifier.
- DumpContents—prints the settings of the six attributes of the specifier.

5.6 The Declarator Class

The declarator part of a C variable declaration is a combination of a variable number of stars, brackets, and parentheses used in describing the variable. For example, on line 2 of figure 5.1, `*` and `[50]` are part of the declarator for the parameter `Stewart`. C has three types of declarators—pointers (identified by a star), arrays (identified by brackets), and functions (identified by parentheses). It should be noted that the declarator part for a given variable declaration is not constrained—any number of stars, brackets, and parentheses are permitted in any combination. Thus, the declarator part of a variable declaration can be represented by a multitude of structures—one for each star, set of brackets, and set of parentheses. The Declarator class is used to control the attributes associated with one of these structures. Below is a discussion of the methods that are part of the Declarator class. The methods are grouped by use.

The following methods are used to set the type of the declarator:

- SetArray
- SetNumElements (used only if the declarator type is array).
- SetFunction
- SetPointer

The following methods are used to retrieve the type of the declarator:

- GetType
- GetNumElements (can be used only if the declarator type is array).

The following are general methods:

- Copy—makes a copy of the declarator.
- DumpContents—prints the type of the declarator.

5.7 The BaseSpecOrDecl Class

Remember, from our discussion above, that the Symbol class has two methods named AddTypeToBegOfList and AddTypeToEndOfList. These methods are used to create a linked list structure that describes the type of a variable symbol. Also, remember that the type of variable declaration has two parts—a specifier part and a declarator part. Thus, the linked list structure describing the type of a variable must contain one object of type Specifier and can optionally contain one or more objects of type Declarator.

The BaseSpecOrDecl class serves as a base class for the Specifier and Declarator classes described in the previous two sections. The reason for its existence is quite simple—there needed to be a way to enable two completely different structures (namely, specifiers and declarators) to exist in the same linked list. For example, in figure 5.2, the linked list of types for the parameter

`Stewart` has two declarator structures and one specifier structure, which are all part of the same list. [Holub 90] allows specifier and declarator structures to exist in the same class through the use of C unions. C++ offers a cleaner solution—inheritance.

The BaseSpecOrDecl class contains code to implement a linked list. The Specifier class contains code specific to specifiers while the Declarator class contains code specific to declarators. Yet, since the Specifier and Declarator classes are both derived from the BaseSpecOrDecl class, they both inherit the same linked list code (from the BaseSpecOrDecl class). This allows objects of the Specifier *and* the Declarator classes to exist in the same linked list.

The BaseSpecOrDecl class contains two methods to implement a linked list—SetNextType and GetNextType. The BaseSpecOrDecl class also contains one virtual function declaration for every method that is a member of the Specifier or Declarator classes. These virtual function declarations are needed because of the way that inheritance works.

Chapter 6

Declarations

The following C declarations were implemented in this compiler:

- variables (global and local) with:
  - initializers
  - type specifiers (`int`, `float`, `double`, `char`)
  - storage classes (`auto`, `static`)
- functions with:
  - parameters (`int`, `float`, `double`, `char`)
  - return types (`void`, `int`, `float`, `double`, `char`)
- type names

Two declarations, in particular, needed special attention paid to them. One was global variable declarations. The other was static variable declarations. The following two sections discuss the two classes, GlobalVarBranch and StaticVariable, that were developed to help deal with the problems posed by these two declarations.

6.1 The GlobalVarBranch Class

In C, global variables are variables that are declared outside functions. For example, in figure 6.1, the variables `abc`, `def`, `ghi`, and `jkl` are all global variables. A global variable is accessible only to functions declared after it. For example, the variables `abc` and `def` are accessible to both functions `main` and `func1`. On the other hand, the variables `ghi` and `jkl` are accessible only to the function `func1`, not to `main`.

```
 0  int abc, def;
 1
 2  int main ()
 3  {
 4  }
 5
 6  float ghi;
 7  float jkl;
 8
 9  void func1 ()
10  {
11  }
```

Figure 6.1: Source Code for Program `Globals`

While developing the ANSI C compiler, it was decided that each statement declaring a global variable(s) should be highlighted, one at a time, before the `main` function is called. For example, in figure 6.1, the statement on line 0 should be highlighted first, followed by the statement on line 6, followed by the statement on line 7, and finally the function `main` called (thus highlighting the statement on line 2). The GlobalVarBranch class was developed to help generate the code necessary to perform these jumps.

The GlobalVarBranch class has eight methods, which can be divided into the following three groups based on use:

- Methods used to jump between global variable declarations:
  - SetNewLabel

  - SetPrevBranch
  - PatchPrevBranch
- Methods used to jump to where `main` is called:
  - SetCallToMainLabel
  - PatchCallToMainBranch
- Methods used to jump to the last packet:
  - SetLastPacketLabel
  - SetLastPacketBranch
  - PatchLastPacketBranch

The following three steps summarize how the methods within each of these groups interact with each other. First, a branch instruction to a garbage label number is generated. The position of this branch instruction is "remembered" by calling the method Set...Branch. Second, when the label number that the branch instruction is supposed to branch to is known, the method Set...Label is called, thus "remembering" the label number. Finally, the Patch...Branch method is called to patch up the branch instruction's garbage label number with the appropriate label number.

6.1.1 Methods Used to Jump Between Global Variable Declarations

The following three methods provide the code necessary to jump from one global variable declaration to the next.

The method SetNewLabel sets the label number associated with the top of a global variable declaration. In figure 6.2, these labels are found on lines 12, 28, and 34.

The method SetPrevBranch sets the position of the branch instruction that will branch to the above mentioned label. In figure 6.2, these branches are found on lines 7, 19, and 33.

The method PatchPrevBranch patches the above mentioned branch instruction so that it branches to the label set in SetNewLabel.

6.1.2 Methods Used to Jump to Where `main` is Called

The following two methods (along with the SetPrevBranch method described above) provide the code necessary to jump from the last global variable declaration to where `main` is called.

The method SetCallToMainLabel sets the label number associated with the call to `main`. In figure 6.2, this label is found on line 8. Notice that the following line is the call to `main`.

Here, there is no need for a method to set the branch position as the method SetPrevBranch discussed in the previous section does. This is because the branch that branches to the label set in the method SetCallToMainLabel should immediately follow the *last* variable declaration. Since the method PatchCallToMainBranch (discussed below) is not called until after the entire program has been parsed, this branch position is identical to the position set by the *last* call to SetPrevBranch. In figure 6.2, the branch that branches to the label associated with the call to `main` is found on line 39. Notice the code associated with the last global variable declaration is directly above it (lines 34-38).

The method PatchCallToMainBranch patches the above mentioned branch instruction so that it branches to the label set in SetCallToMainLabel.

6.1.3 Methods Used to Jump to the Last Packet

The following three methods provide the code necessary to jump from the position returned to after `main` has finished executing to the last packet. This needs to be done because `main` does *not* have to be declared at the end of a C source code file.

The method SetLastPacketLabel sets the label number associated with the last packet. This is the packet that will be executed after the entire `main` function has been executed. In figure 6.2, this label is found on line 56.

The method SetLastPacketBranch sets the position of the branch instruction that will branch to the above mentioned label. In figure 6.2, this branch is found on line 11. Notice that this instruction immediately follows the code associated with the call to `main` (lines 8-10). Thus, this jump does not occur until after `main` has finished executing.

The method PatchLastPacketBranch patches the above mentioned branch instruction so that is branches to the label set in SetLastPacketLabel.

6.2 The StaticVariable Class

In C, a variable with a static storage class has the following two properties:

- Its memory is initialized to zero, unless specified otherwise by an initializer in the variable's declaration.
- The memory for the variable is created once, when the program begins execution, and is not destroyed until the program has completed execution.

There are two ways that a C variable can obtain a static storage class. First, all global variables are, by default, given a static storage class. Second, a variable declared anywhere inside a function can obtain a static storage class by prefixing the variable declaration with the reserved word `static`. For example, in figure 6.3, the variables `outer_static` and `inner_static` both have a static storage class, while the variable `i` does not.

Normally, the memory for a variable is created (with the E-machine INST instruction) when the variable comes into scope and is destroyed (with the

```
 0  pushd    c, DS13
 1  inst     c, V0
 2  inst     c, V1
 3  inst     c, V2
 4  inst     c, V3
 5  br       c, L8
 6  label    c, L0
 7  br       c, L3
 8  label    c, L1
 9  call     c, L4
10  label    c, L2
11  br       c, L9
12  label    c, L3
13  inst     c, V4
14  push     c, I, CI0
15  pop      c, I, V4
16  inst     c, V5
17  push     c, I, CI0
18  pop      c, I, V5
19  br       c, L5
20  label    c, L4
21  pushd    c, DS7
22  nop      c
23  push     c, I, CI0
24  cast     c, I, C
25  pop      c, C, V6
26  popd     c
27  return   c
28  label    c, L5
29  inst     c, V7
30  push     c, I, CI0
31  cast     c, I, R
32  pop      c, R, V7
33  br       c, L6
34  label    c, L6
35  inst     c, V8
36  push     c, I, CI0
37  cast     c, I, R
38  pop      c, R, V8
39  br       c, L1
40  label    c, L7
41  pushd    c, DS10
42  nop      c
43  push     c, I, CI0
44  pop      c, I, V9
45  popd     c
46  return   c
47  label    c, L8
48  inst     c, V9
49  push     c, I, CI0
50  pop      c, I, V9
51  inst     c, V6
52  push     c, I, CI0
53  cast     c, I, C
54  pop      c, C, V6
55  br       c, L0
56  label    c, L9
57  uninst   c, V0
58  uninst   c, V1
59  uninst   c, V2
60  uninst   c, V3
61  uninst   c, V9
62  uninst   c, V6
63  uninst   c, V4
64  uninst   c, V5
65  uninst   c, V7
66  uninst   c, V8
67  popd     c
```

Figure 6.2: E-code Instructions Resulting from Compilation of Program `Globals`

```
int outer_static;

int func1 ()
{
   static int inner_static = 10;

   inner_static++;
}

void main ()
{
   int i;

   for (i = 0; i < 5; i++)
      func1 ();
}
```

Figure 6.3: Source Code for Program `Statics`

E-machine UNINST instruction) when the variable goes out of scope. This method works fine, as long as the variable is not a static variable declared inside of a function. The problem with using this method on static variables declared inside functions is as follows—the memory for the variable is created every time the function is entered and is destroyed every time the function is exited. Obviously, this violates the property of static variables that says the variable's memory is set aside once, when the program begins execution, and is destroyed only after the program has completed execution.

This is where the StaticVariable class comes into play. If a static variable declared inside a function is found during the parsing of a program, a method named AddStaticVariable is called. This method adds the static variable to a linked list. Later, after the entire program has been parsed, two methods named InstStaticVariables and UninstStaticVariables are called. The method

InstStaticVariables runs through the linked list, generating an INST instruction and initializing code for each variable. The method UninstStaticVariables runs through the linked list, laying down an UNINST instruction for each variable. Of course, branches and labels are added in the appropriate places so that the INSTs are done before any program code is executed. For example, in figure 6.4, line 5 branches to where the static variable `inner_static`, found on line 4 of figure 6.3, is INSTed and initialized. Line 82 branches back to line 6, where execution of the program code commences. Finally, after the program code has finished executing, line 89 (part of the last packet) UNISTs the static variable `inner_static`. Notice the INST instruction associated with `inner_static` is executed just once (before the rest of the program code is executed) and the UNINST instruction associated with the same variable also is executed just once (after the program code is executed).

It should be noted that the StaticVariable class deals only with static variables declared inside functions. Global static variables are automatically handled correctly because, in part, of the way the GlobalVarBranch class does things. Thus, global static variables do not need the help of the StaticVariable class to work correctly.

```
 0  pushd    c, DS12
 1  inst     c, V0
 2  inst     c, V1
 3  inst     c, V2
 4  inst     c, V3
 5  br       c, L13
 6  label    c, L0
 7  br       c, L3
 8  label    c, L1
 9  call     c, L5
10  label    c, L2
11  br       c, L14
12  label    c, L3
13  inst     c, V4
14  push     c, I, CI0
15  pop      c, I, V4
16  br       c, L1
17  label    c, L4
//  CODE FOR ``func1'' OMITTED!!!
30  return   c
31  label    c, L5
//  CODE FOR ``main'' OMITTED!!!
71  return   c
72  label    c, L13
//  CODE FOR INSTANTIATING THE VARIABLE REGISTER ASSOCIATED WITH THE RETURN
//  VALUE OF ``func1'' OMITTED!!!
76  inst     c, V6
77  push     c, I, CI10
78  pop      c, I, V6
//  CODE FOR INSTANTIATING THE VARIABLE REGISTER ASSOCIATED WITH THE RETURN
//  VALUE OF ``main'' OMITTED!!!
82  br       c, L0
83  label    c, L14
84  uninst   c, V0
85  uninst   c, V1
86  uninst   c, V2
87  uninst   c, V3
88  uninst   c, V7
89  uninst   c, V6
90  uninst   c, V5
91  uninst   c, V4
92  popd     c
```

Figure 6.4: E-code Instructions Resulting from Compilation of Program `Statics`

Chapter 7

Statements

The following C statements were implemented in this compiler:

- compound
- expression
- null
- `if` without `else` part
- `if` with `else` part
- `switch` (including `case` and `default` labels)
- `for` loop
- `while` loop
- `do-while` loop
- `break`
- `continue`
- `return`

Although sometimes long, the code implementing many of these statements was fairly straightforward. There were, however, a few statements that posed some unique problems. In particular, the `switch` statement and the jump statements (`break`, `continue`, and `return`) were much harder to implement than the rest of the statements. The following two sections discuss the two classes, SwitchStatement and JumpStatements, that were developed to help deal with the problems posed by these statements.

7.1 The SwitchStatement Class

The `switch` statement was the most difficult statement implemented in the ANSI C compiler. It posed three major problems. One was ensuring that all `case` and `default` labels are matched with the proper `switch` statement. Another was making sure that the program jumps to the proper `case` or `default` label when a `switch` statement is executed. Finally, a last problem was readjusting the dynamic scope (by generating INSTs and PUSHDs) when jumping to a `case` or `default` label. This readjusting needs to be done because jumping to a `case` or `default` label can cause a jump out of one scope and into another, as explained later.

7.1.1 Matching a `case` or `default` Label with the Proper `switch` Statement

The NewSwitchLevel method is used to tell the SwitchStatement class that a new level that can possibly contain `case` and `default` labels has been encountered. Thus, this method is called at the beginning of every `switch` statement encountered while parsing a program. In figure 7.1, this method would be called just before parsing line 4.

The AddCaseLabel method sets the label number and constant value associated with a particular `case` label. For example, in figure 7.2, the labels on lines 29, 34, and 80 correspond respectively to the `case` labels found on lines 6, 7, and 17 of figure 7.1. The level set by the method NewSwitchLevel helps this method to associate the `case` label with the proper `switch` statement.

Of course, an error message is displayed if there is no `switch` statement to associate the `case` label with.

The AddDefaultLabel method sets the label number associated with a particular `default` label. For example, in figure 7.2, the label on line 85 corresponds to the `default` label found on line 18 of figure 7.1. The level set by the method NewSwitchLevel helps this method to associate the `default` label with the proper `switch` statement. Of course, an error message is displayed if there is no `switch` statement to associate the `default` label with.

7.1.2 Jumping to the Proper `case` or `default` Label

The SaveSwitchExpr method INSTs a variable register and pops the value of the expression the `switch` statement is "switching on" into that variable register. This variable register is used by the PatchSwitch method, which will be discussed shortly. For example, lines 24-25 of figure 7.2 save the `switch` expression found on line 4 of figure 7.1. Notice the code to evaluate the expression that the `switch` statement is "switching on" (lines 19-23) resides directly above these two lines.

The PatchSwitch method is called after the entire `switch` statement has been parsed. In figure 7.1, this method would be called just after parsing line 20. The PatchSwitch method is used to generate the code necessary to determine which label (`case` or `default`), if any, the `switch` statement should jump to. It does this using two separate steps. First, it generates code that performs the following steps for every `case` label :

- Push the variable register associated with the expression that the `switch` statement is "switching on" (saved by SaveSwitchExpr).
- Push the constant value of the `case` label (saved by AddCaseLabel).
- Check to see if these two values are equal.

- If they are, branch to the label associated with the `case` label (saved by AddCaseLabel).

For example, in figure 7.2, lines 100-103, 104-107, and 108-111 contain this code, which correspond to the `case` labels found on lines 6, 7, and 17 of figure 7.1. Second, after the `case` labels have been processed, an unconditional branch to the label associated with the `default` label (saved by AddDefaultLabel) is generated. Of course, if the `switch` statement has no `default` label, this step is skipped. Line 112 of figure 7.2 contains this code, which corresponds to the `default` label found on line 18 of figure 7.1.

The UnsaveSwitchExpr method has two important jobs. One is to UNINST the variable register INSTed by the method SaveSwitchExpr. An example of this can be found on line 114 of figure 7.2. Since UnsaveSwitchExpr is called at the end of each `switch` statement, this method's other job is to tell the JumpStatements class that the new switch level set by the NewSwitchLevel method no longer exists.

7.1.3 Readjusting the Dynamic Scope of a `case` or `default` Label

A jump to a `case` label occurs if the expression that the `switch` statement is being "switched on" has the same value as the `case` label. Otherwise, a jump to the `default` label occurs (if the `switch` statement has one). In either case, it is possible to jump from one scope (the one containing the expression the `switch` statement is "switching on") to another scope (the one containing the `case` or `default` label). For example, in figure 7.1, when the expression on line 4 is executed, a jump to line 7 will occur. Yet, the statements on these two lines are *not* at the same scoping level (the one on line 7 is one deeper). The AdjustCaseOrDefaultScope method is used to generate INSTs and PUSHDs so that the dynamic scope is correct when the program jumps to a `case` or `default` label that is not at the same scoping level as the `switch` expression. For example, in figure 7.2, line 35 corresponds to the code necessary to readjust the dynamic scope for the `case` label found on line 7 of figure 7.1. As you can see, after this code is executed, the dynamic scope will be correct.

```
 0  main ()
 1  {
 2     int a = 10;
 3
 4     switch (10 + 'b' - 10)
 5     {
 6        case 0 :
 7        case 'b' :
 8        {
 9           int d=2, e=3;
10           {
11              int c = 5;
12
13              a *= c ? d : e;
14              break;
15           }
16        }
17        case (8 || 10) << (4 && !(2>50)) :
18        default :
19           a %= 6;
20     }
21  }
```

Figure 7.1: Source Code for Program `Switch`

7.2 The JumpStatements Class

In C, jump statements include the following:

```
// CODE BEFORE ``switch'' STATEMENT OMITTED.
 19  push     c, I, CI10
 20  push     c, I, CI98
 21  add      c, I
 22  push     c, I, CI10
 23  sub      c, I
 24  inst     c, V6
 25  pop      c, I, V6
 26  br       c, L14
 27  pushd    c, DS13
 28  br       c, L5
 29  label    c, L4
 30  pushd    c, DS13
 31  nop      c
 32  label    c, L5
 33  br       c, L7
 34  label    c, L6
 35  pushd    c, DS13
 36  nop      c
 37  label    c, L7
 38  pushd    c, DS9
 39  inst     c, V7
 40  push     c, I, CI2
 41  pop      c, I, V7
 42  inst     c, V8
 43  push     c, I, CI3
 44  pop      c, I, V8
 45  pushd    c, DS6
 46  inst     c, V9
 47  push     c, I, CI5
 48  pop      c, I, V9
 49  push     c, I, V9
 50  push     c, I, CI0
 51  eql      c, I
 52  brt      c, L8
 53  push     c, I, V7
 54  cast     c, I, I
 55  br       c, L9
 56  label    c, L8
 57  push     c, I, V8
 58  cast     c, I, I
 59  label    c, L9
 60  pop      c, I, V0
 61  push     c, I, V5
 62  push     c, I, V0
 63  mult     c, I
 64  pop      c, I, V5
 65  push     c, I, V5
 66  pop      c, I, V0
 67  uninst   c, V9
 68  popd     c
 69  uninst   c, V7
 70  uninst   c, V8
 71  popd     c
 72  popd     c
 73  br       c, L15
 74  uninst   c, V9
 75  popd     c
 76  uninst   c, V7
 77  uninst   c, V8
 78  popd     c
 79  br       c, L11
 80  label    c, L10
 81  pushd    c, DS13
 82  nop      c
 83  label    c, L11
 84  br       c, L13
 85  label    c, L12
 86  pushd    c, DS13
 87  nop      c
 88  label    c, L13
 89  push     c, I, CI6
 90  pop      c, I, V0
 91  push     c, I, V5
 92  push     c, I, V0
 93  mod      c, I
 94  pop      c, I, V5
 95  push     c, I, V5
 96  pop      c, I, V0
 97  popd     c
 98  br       c, L15
 99  label    c, L14
100  push     c, I, V6
101  push     c, I, CI0
102  eql      c, I
103  brt      c, L4
104  push     c, I, V6
105  push     c, I, CI98
106  eql      c, I
107  brt      c, L6
108  push     c, I, V6
109  push     c, I, CI2
110  eql      c, I
111  brt      c, L10
112  br       c, L12
113  label    c, L15
114  uninst   c, V6
// CODE AFTER ``switch'' STATEMENT OMITTED.
```

Figure 7.2: E-code Instructions Resulting from Compilation of Program `Switch`

- `break`—used to jump to the statement immediately following the loop or `switch` statement that the `break` statement is located in.
- `continue`—used to jump to the top of the loop that the `continue` statement is located in.
- `return`—used to jump out of a function (optionally returning a value).
- `goto`—not implemented by the ANSI C compiler.

Implementing these statements presented a couple of different problems. One was ensuring that a `break` statement is matched with the proper loop/`switch` statement and that a `continue` statement is matched with the proper loop. Another problem was readjusting the dynamic scope (by generating UNINSTs and POPDs) when a `break`, `continue`, or `return` statement is encountered. This readjusting needs to be done because these three statements can all jump out of one scope and into another, as explained later.

7.2.1 Matching a `break` or `continue` Statement with the Proper Loop or `switch` Statement

The NewBreakLevel method is used to tell the JumpStatements class that a new level that can possibly contain `break` statements has been encountered. Thus, this method is called at the beginning of every loop or `switch` statement encountered while parsing a program. In figure 7.3, this method would be called just before parsing line 4.

The method AddBreakLabel sets the label number associated with the bottom of a loop or `switch` statement. For example, in figure 7.4, the label on line 77 corresponds to the bottom of the `for` loop found in figure 7.3.

The method AddBreakBranch sets the position of a branch instruction that will branch to the label set in AddBreakLabel. In figure 7.4, these branches are on lines 34, and 71. The reason for the branch instruction on line 34 is discussed shortly. The branch instruction on line 71 is a consequence of the `break` statement found on line 12 of figure 7.3.

The method PatchBreaks patches up all of the branches added by AddBreakBranch so that they branch to the label set in AddBreakLabel. Also, since this method is called at the end of each loop and `switch` statement, it is used to tell the JumpStatements class that the new break level set by the NewBreakLevel method no longer exists. In figure 7.3, this method would be called just after parsing line 14 of figure 7.3.

The methods NewContinueLevel, AddContinueLabel, AddContinueLabel, and PatchContinues are all very similar to the above mentioned methods. The major differences are that the method AddContinueLabel is used to mark the top (not the bottom) of a loop and that all three methods are used for the `continue` statement as opposed to the `break` statement.

Now, back to the branch instruction on line 34 of figure 7.4. Every loop contains one invisible `break` statement. When the test expression to determine if the loop should be iterated through one more time fails, this invisible `break` statement is executed. The branch instruction on line 34 of figure 7.4 is the invisible `break` statement for the `for` loop found on line 4 of figure 7.3. Notice that the code to test if the `for` loop should be exited is directly above line 34 (lines 28-33). Similarly, every loop also has one invisible `continue` statement. When the bottom of a loop is reached, the invisible `continue` statement is executed. This causes a jump to the top of the loop to occur. The branch instruction on line 76 of figure 7.4 is the invisible `continue` statement for the `for` loop found on line 4 of figure 7.3.

7.2.2 Readjusting the Dynamic Scope of a `break`, `continue`, or `return` Statement

When a `break` statement is executed, the loop or `switch` statement that the `break` statement is located in is exited. In C, this means that a jump from one scope (the one containing the `break` statement) to another scope (the one containing the statement just after the loop or `switch` statement) is possible. For example, in figure 7.3, when the `break` statement on line 12 is executed, a jump to line 15 will occur. Yet, the statements on these two lines are *not* at the same scoping level (the one line 12 is two deeper). The AdjustBreakScope method is used to generate UNINSTs and POPDs so that the dynamic scope after the `break` statement is executed is the same as it was before the loop or `switch` statement was entered. For example, in figure 7.4, lines 68-70 correspond to the code necessary to readjust the dynamic scope after the `break` statement on line 12 of figure 7.3 is executed. As you can see, after this code is executed, the dynamic scope will be correct.

The AdjustContinueScope and AdjustReturnScope methods are identical to the AdjustBreakScope method, except for the fact that they are used for the `continue` and `return` statements, respectively.

```
void main ()
{
   int i, temp = 2;

   for (i = 0; i < 10; i++)
   {
      int a = 2;

      if (i < 5)
         temp *= a;
         else
         {
            break;
         }
   }
}
```

Figure 7.3: Source Code for Program `Jumps`

```
 0  pushd    c, DS15
 1  inst     c, V0
 2  inst     c, V1
 3  inst     c, V2
 4  inst     c, V3
 5  br       c, L12
 6  label    c, L0
 7  br       c, L1
 8  label    c, L1
 9  call     c, L3
10  label    c, L2
11  br       c, L13
12  label    c, L3
13  pushd    c, DS12
14  nop      c
15  inst     c, V5
16  inst     c, V6
17  push     c, I, CI2
18  pop      c, I, V6
19  br       c, L5
20  label    c, L4
21  br       c, L7
22  label    c, L5
23  push     c, I, CI0
24  pop      c, I, V5
25  push     c, I, V5
26  pop      c, I, V0
27  label    c, L6
28  push     c, I, V5
29  push     c, I, CI10
30  less     c, I
31  cast     c, B, I
32  push     c, I, CI0
33  eql      c, I
34  brt      c, L11
35  br       c, L8
36  label    c, L7
37  push     c, I, V5
38  push     c, I, V5
39  push     c, I, CI1
40  add      c, I
41  pop      c, I, V5
42  pop      c, I, V0
43  br       c, L6
44  label    c, L8
45  pushd    c, DS9
46  inst     c, V7
47  push     c, I, CI2
48  pop      c, I, V7
49  nop      c
50  push     c, I, V5
51  push     c, I, CI5
52  less     c, I
53  cast     c, B, I
54  push     c, I, CI0
55  eql      c, I
56  brt      c, L9
57  push     c, I, V7
58  pop      c, I, V0
59  push     c, I, V6
60  push     c, I, V0
61  mult     c, I
62  pop      c, I, V6
63  push     c, I, V6
64  pop      c, I, V0
65  br       c, L10
66  label    c, L9
67  pushd    c, DS4
68  popd     c
69  uninst   c, V7
70  popd     c
71  br       c, L11
72  popd     c
73  label    c, L10
74  uninst   c, V7
75  popd     c
76  br       c, L4
77  label    c, L11
78  push     c, I, CI0
79  cast     c, I, C
80  pop      c, C, V4
81  uninst   c, V5
82  uninst   c, V6
83  popd     c
84  return   c
85  label    c, L12
86  inst     c, V4
87  push     c, I, CI0
88  cast     c, I, C
89  pop      c, C, V4
90  br       c, L0
91  label    c, L13
92  uninst   c, V0
93  uninst   c, V1
94  uninst   c, V2
95  uninst   c, V3
96  uninst   c, V4
97  popd     c
```

Figure 7.4: E-code Instructions Resulting from Compilation of Program `Jumps`

Chapter 8

Expressions

The following C expressions and operators were implemented in this compiler:

- comma operator
- assignment operators (`=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, and `|=`)
- conditional operator (`?:`)
- constant expressions
- logical operators (`&&`, `||`, and `!`)
- bitwise operators (`|`, `^`, `&`, `~`, `<<`, and `>>`)
- relational operators (`==`, `!=`, `<`, `>`, `<=`, and `>=`)
- arithmetic operators (`unary and binary +`, `unary and binary -`, `*`, `/`, and `%`)
- explicit casts
- pre increment and decrement operators
- `sizeof` operator
- post increment and decrement operators
- function calls
- identifiers
- integer constants

- floating constants
- character constants

The implementation of many of these expressions was trivial. There were, however, a few problems that were encountered along the way. Those problems arose during the implementation of the post increment/decrement operators and implicit function calls. The PostIncDec and ImplicitFuncCall classes discussed in the two sections below were developed to solve the problems presented during implementation of these expressions. One other class, EvaluationStack, is also discussed in a section below. This class is used by all expressions and is quite complex.

8.1 The PostIncDec Class

In C, the post increment/decrement operators add/subtract one from a variable *after* the entire expression that the variable appears in has been evaluated. For example, in figure 8.1, after the execution of the statement

```
a = b++ * b++;
```

`a` would have the value 25 while `b` would have the value 7. Notice the value of `b` was not incremented until after the entire expression was evaluated. Thus, `a` was assigned the value equivalent to *5 * 5*, and not something like *5 * 6* or *6 * 7*.

The post increment and decrement operators posed the following problem—how to increment or decrement the variable *after* the entire expression it appears in has been evaluated. This is where the PostIncDec class comes into play. Every time a post increment or decrement operator is found, a method named AddPostIncDec is called. This method adds the variable that is to be incremented/decremented to a linked list. Then, after the entire expression that the variable appears in has been evaluated, a method named DoPostIncDecs is called. This method runs through the linked list and actually generates the code for each increment/decrement operator found in the list. Thus, the code for the post increment/decrement operator is not executed until after the code for the rest of the expression is executed. For example, the statement `a = b++ * b++;` in figure 8.1 translates into the code given in figure 8.2. Notice that the code for both post increments of `b` (lines 24-27 and 28-31) come after the code for evaluating the entire expression `a = b++ * b++;` (lines 19-23).

```
void main ()
{
   int a, b = 5;

   a = b++ * b++;
}
```

Figure 8.1: Source Code for Program `Posts`

8.2 The ImplicitFuncCall Class

In C, an implicit function call is a call that is made to a function that has not been declared yet. For example, in figure 8.3, line 2 contains an implicit call to the function `f1`. The call is implicit because it precedes the definition of the function (line 5).

When a function call is found while parsing a program, the E-machine code instruction CALL must be generated. A crucial piece of information that must be supplied to the CALL instruction is the label number corresponding to the

```
// CODE BEFORE STATEMENT ``a = b++ * b++;'' OMITTED.
19  push     c, I, V6
20  push     c, I, V6
21  mult     c, I
22  pop      c, I, V5
23  push     c, I, V5
24  push     c, I, V6   // START OF CODE FOR FIRST ``b++''.
25  push     c, I, CI1
26  add      c, I
27  pop      c, I, V6   // END OF CODE FOR FIRST ``b++''.
28  push     c, I, V6   // START OF CODE FOR SECOND ``b++''.
29  push     c, I, CI1
30  add      c, I
31  pop      c, I, V6   // END OF CODE FOR SECOND ``b++''.
32  pop      c, I, V0
// CODE AFTER STATEMENT ``a = b++ * b++;'' OMITTED.
```

Figure 8.2: E-code Instructions Resulting from Compilation of Program `Posts`

top of the function that is being called. The problem with implicit function calls is as follows—at the time the CALL instruction is to be generated, this label number is not known. The reason for this is quite simple—the function has not been declared yet. For example, in figure 8.3, the compiler must generate a CALL instruction for the call to function `f1` on line 2. Yet, the label corresponding to the function `f1` will not be known until line 5.

The ImplicitFuncCall class was developed to solve exactly this problem. The way it works is as follows. When an implicit function call is found, a CALL instruction with a garbage label number is generated. At this time, a method named AddImplicitCall is also called. This method adds the information such as the name of the function being called and the position that the CALL instruction was generated to a linked list. After the entire program has been parsed, a method named PatchImplicitCalls is called. This method runs through the linked list and patches up the CALL instructions by replacing the garbage label numbers with the label numbers corresponding to the functions that by now presumably have been declared (since we are at the end of the program). Of course, if an implicit call to a function is made and that function is *not* later declared, an appropriate error message is displayed.

It should be noted that the unseen function call to `main` that gets every C program started can be thought of as an implicit call. Thus, this call is also added to the implicit call linked list by way of the AddImplicitCall method.

```
void main ()
{
   f1 ();
}

int f1 ()
{
}
```

Figure 8.3: Source Code for Program `Implicit`

8.3 The EvaluationStack Class

In one way or another, the EvaluationStack class is used by every expression. Because of this fact, the EvaluationStack class is one of the most complicated classes in the ANSI C compiler. The EvaluationStack class has the following two responsibilities:

- Knowing when and when not to generate E-code.
- Tracking the types of all expressions.

8.3.1 Knowing When and When Not to Generate E-code

Most expressions require that E-code be generated that can be evaluated at *run time* using the E-machine's evaluation stack. There are, however, a few expressions that require that no E-code be generated. Instead, these expressions are evaluated at *compile time* using a private evaluation stack. For example, E-code is *not* generated for constant expressions or expressions that are operands to the `sizeof` operator. The following is the method of the EvaluationStack class used to deal with code generation:

- GenerateCode—this method is used to determine whether or not E-code should be generated.

Constant Expressions

In C, constant expressions are expressions that *always* evaluate to a constant value. Thus, they cannot contain assignments, increment and decrement operators, function calls, or the comma operator. The value of a constant expression must be of integral type. Integral types include integer, character, and enumerated types. Constant expressions are required in several contexts—after the `case` reserved word, as array bounds and bit-field lengths, as the value of an enumeration constant, and in static variable initializers. For further discussion on constant expressions, see appendix A, section 7.19 of [Kernighan 88].

Only two contexts of constant expressions were used in the development of the ANSI C compiler—after the `case` reserved word and in static variable initializers. The following are the methods of the EvaluationStack class used to deal with constant expressions:

- InsideConstExpr—this method is used to tell the EvaluationStack class that it should start (or stop) tracking the value of a constant expression.
- GetCaseConstExpr—this method is used to get the value of a constant expression used in a `case` label.
- GetInitializerConstExpr—this method is used to get the value of a constant expression used in a static variable initializer.

- IsValidConstExpr—this method is used to tell whether or not the constant expression currently being tracked is valid (does not contain assignments, function calls, etc.).
- SetValidConstExpr—this method is used to set whether or not the constant expression currently being tracked is valid.

The Sizeof Operator

The `sizeof` operator yields the number of bytes required to store an object of the type of its operand. If the operand is an expression, the expression is not evaluated. Instead, the number of bytes required to store an object of the type of that expression is calculated. Thus, the `sizeof` operator is very similar to constant expressions in that no actual E-code is generated for evaluating the expression that is an operand of a `sizeof` operator. Instead, the type of the expression is tracked (see section below). Afterwards, knowing the type of the expression, it is trivial to determine how many bytes it will take to store an object of that type. The following is the method of the EvaluationStack class used to deal with the `sizeof` operator:

- InsideSizeof—this method is used to tell the EvaluationStack class that the current expression is (or is not) an operand to a `sizeof` operator.

8.3.2 Tracking the Types of Expressions

There are a couple of different reasons why the types of the values on the E-machine's evaluation stack need to be kept track of. One reason is that some C operators require operands to be of integral type. Thus, the types of the operands on the E-machine's evaluation stack must be known so that compile time error messages can be displayed if this rule is broken. For example, in figure 8.4, the statement on line 4 is illegal. The problem is that both operands of the mod operator (`%`) must be of integral type. The 8.0 violates this rule, since it is not an integral type. Thus, this program should emit an error message when compiled.

The other reason that the type of the values on the E-machine's evaluation stack need to be kept track of is that C has automatic type conversions. This means that before an expression that contains two operands of different types is evaluated, one of the operands is automatically converted (casted) to the type of the other operand based on a set of conversion rules. For example, in figure 8.4, the expression on line 5 would be evaluated as follows:

- Push the integer `5` onto the evaluation stack.
- Push the float `8.0` onto the evaluation stack.
- Cast the integer `5` on the evaluation stack to a float.
- Pop the top two values on the evaluation stack, multiply them, and push the result back on top of the evaluation stack.
- Cast the float `40.0` on top of the evaluation stack to an integer.
- Pop the top value on the evaluation stack into the variable register associated with the variable `a`.

Notice that it is necessary to keep track of the types of the values on the E-machine's evaluation stack so that the compiler can determine when the proper casts are to be performed.

The following is a list of the EvaluationStack methods that were developed to implement C expressions. If an expression is not supposed to generate E-code (see discussions on constant expressions and the `sizeof` operator above), "the stack" in the discussion below refers to the EvaluationStack class's private stack. In this case, the expression is evaluated at compile time using this private stack. Yet, if E-code is to be generated for an expression, "the stack" in the discussion below refers to the E-machine's evaluation stack. In other words, E-code is generated that is evaluated at run time using the

```
main ()
{
   int a;

   a = 5 % 8.0;
   a = 5 * 8.0;
}
```

Figure 8.4: Source Code for Program `Evaluation`

E-machine's evaluation stack. In either case, the type of the values on both the EvaluationStack class's private stack and the E-machine's evaluation stack are always kept track of.

- PushInt—pushes a constant integer onto the stack.
- PushFloat—pushes a constant float onto the stack.
- Push—pushes the variable register associated with a symbol onto the stack.
- PushReg—pushes a variable register onto the stack.
- Reverse—reverses the top two elements on the stack.
- Assign—pops the top value off the stack and places it into a specified variable register (by calling Pop). Pushes the specified variable register onto the stack (by calling PushReg).
- Pop—performs any necessary casting (by calling PopCast). Pops the top value off the stack and places it into a specified variable register.
- ThrowAwayPop—pops the top value off the stack and places it into a garbage variable register.
- UnaryIntegral—pops the top element off the stack. Makes sure the element is of integral type (by calling TopElementIntegral). Pushes the result of the unary integral operator applied to the popped element onto the stack. The only unary integral operator is bitwise not.

- BinaryIntegral—pops the top two elements off the stack. Makes sure both elements are of integral type (by calling TopTwoElementsIntegral). Pushes the result of the binary integral operator applied to the two popped elements onto the stack. Binary integral operators include: bitwise or, bitwise xor, bitwise and, shift-left, shift-right, and mod.

- UnaryArithmetic—pops the top element off the stack. Pushes the result of the unary arithmetic operator applied to the popped element. Unary arithmetic operators include: unary plus, unary minus, and logical not.

- BinaryArithmeticLogical—pops the top two elements off the stack. Performs any necessary casting (by calling BinaryArithmeticCast). Pushes the result of the binary logical operator applied to the two popped elements onto the stack. Binary logical operators include: logical or and logical and.

- BinaryArithmeticRelational—pops the top two elements off the stack. Performs any necessary casting (by calling BinaryArithmeticCast). Pushes the result of the binary relational operator applied to the two popped elements onto the stack. Binary relational operators include: less than, greater than, less than or equal to, greater than or equal to, not equal, and equal.

- BinaryArithmeticMath—pops the top two elements off the stack. Performs any necessary casting (by calling BinaryArithmeticCast). Pushes the result of the binary math operator applied to the two popped elements onto the stack. Binary math operators include: binary plus, binary minus, times, and divide.

- Cast—casts the top element on the stack from one type to another. This method is a result of the C explicit cast expression.

- PopCast—checks to make sure that the top element on the stack is of the same type as the variable register it is about to be popped into. If it isn't, this method performs the casting necessary so that it is the same type.

- TopElementIntegral—checks to make sure that the top element on the stack is of integral type. If it isn't, an error message is displayed.

- TopTwoElementsIntegral—checks to make sure that the top two elements on the stack are of integral type. If they aren't, an error message is displayed.

- BinaryArithmeticCast—checks to make sure that the top two elements on the stack are of the same type. If they aren't, this method performs the casting necessary to make them have the same type.

- PopTop—pops the top value off the EvaluationStack's private stack.

Chapter 9

E-machine Code Generation

Translation of a source program for the E-machine requires the generation of the eight components of the E-machine object code file—the header section, the string section, the source section, the label section, the variable section, the code section, the packet section, and the static scope section. In the ANSI C compiler, code generation is handled by eight classes, one for each code file section. Each class is responsible for saving the structures it generates in the proper section of the E-code object file. Beyond that, there are very few similarities between the classes. Part of the E-machine is a set of support routines that write the various E-code sections. These routines are used by the section classes to save their data to the E-code file.

9.1 The HeaderSection Class

The HeaderSection class does nothing but write an empty section to the E-machine object file. This section is always empty because the DYNALAB group members have not decided upon all of its contents yet.

9.2 The StringSection Class

The StringSection class manages the list of strings that are encountered in the source program by the compiler, mainly enumerated type names and string literals. This is a very simple class whose only responsibility is to accept and store string literals. Because the ANSI C compiler did not implement enumerated types or strings, the StringSection class always writes out an empty section. However, when the time comes that it is needed, the StringSection does have the capabilities to accept and store strings.

9.3 The SourceSection Class

The SourceSection class has no responsibility beyond reading the program source file and saving it to the source section of the E-machine object file. Unlike many of the other section classes, the SourceSection instantiation lies dormant until the end of program compilation when the save methods of all the section classes are called. At that point, this class reads the source file into a string array and calls upon the E-machine library to write the source program into the source section.

9.4 The LabelSection Class

The LabelSection class manages program labels and their addresses. When the compiler needs to generate a label during compilation, it must request one from the LabelSection. When a label is requested, the LabelSection class accepts an address, associates a label with that address, and returns the new label.

9.5 The VariableSection Class

The VariableSection class manages the list of variable registers used by the compiler. Whenever the compiler needs a new register, it tells the VariableSection the size and type of the variable it needs. The size becomes part of the E-code file variable section. The type, however, is an extension used by, among other things, the CodeSection class methods. Many E-machine instructions require a type (integer, real, etc.). By making the type a part of each variable, the CodeSection need only query the type from the VariableSection when necessary.

9.6 The CodeSection Class

The CodeSection class generates the E-machine instructions and has a method for each instruction of the E-machine. Many of these methods have a corresponding patch method. This allows the class to generate an instruction before all of the information for that instruction is known. Later, after the information for the instruction is known, the patch routine can be called and the instruction's missing information filled in.

9.7 The PacketSection Class

The PacketSection class is responsible for the executable packets defined by the compiler. To add a packet to the packet section, the following three methods must be called:

- MarkStart
- MarkEnd
- AddPacket

MarkStart is used to mark the starting address, line, and column numbers of a packet. MarkEnd is used to mark the ending address, line, and column numbers of a packet. AddPacket is used to actually add the packet to the packet section. AddPacket uses the information set in the MarkStart and MarkEnd methods, as well as parameters that describe the rest of the information, to determine the contents of the packet that is to be added to the packet section. This means that the MarkStart and MarkEnd methods must be called *before* the AddPacket method is called.

9.8 The StatScopeSection Class

The StatScopeSection class, which is used to create the static scope section of the E-code file, is probably the most complex of the code generation classes because of the parent/child relationships. This class has six methods for adding entries to the static scope table. They are:

- SaveProgram
- SaveHeader
- SaveEnd
- SaveVariable
- SaveFunction
- SaveUnnamedBlock

The methods SaveHeader, SaveEnd, SaveVariable, SaveFunction, and SaveUnnamedBlock are called from within the PopScope method of the CrossLinks class. When a scope is popped, the PopScope method first marks the beginning of a new block in the static scope table by calling the method SaveHeader. Then each variable, function, and unnamed block in the scope that is being popped is added to the static scope table (by calling

SaveVariable, SaveFunction, or SaveUnnamedBlock). Finally, the PopScope routine marks the end of the new block in the static scope table by calling the method SaveEnd.

The SaveProgram method is called after the parsing of the program is complete. This method adds the bootstrap block to the static scope table. At end of the SaveProgram method, a method called SetParentAndChildFields is called. This method sets the parent and child fields of all of the appropriate entries in the static scope table.

Another responsibility of the StatScopeSection class concerns the PUSHD instruction. When the compiler encounters a function or unnamed block, it adds a PUSHD instruction to the CodeSection class. This instruction pushes the position in the static scope table of the function or unnamed block. However, this position is not known until the function or unnamed block is actually added to the static scope table. Thus, it is the responsibility of the StatScopeSection class to patch the PUSHD instruction associated with each function or unnamed block as they are added to the static scope table. In other words, for each function or unnamed block, the CodeSection class generates a PUSHD instruction that is later patched up by the StatScopeSection class.

Chapter 10

Conclusions and Future Enhancements

10.1 Conclusions

An ANSI C compiler for the E-machine has been designed and partially implemented. The ANSI C compiler is a one-pass compiler written in C++ and was developed using the parser development tool PCCTS 1.31 (Purdue Compiler Construction Tool Set) [Parr 93]. PCCTS generates an integrated recursive descent LL(k) parser and DFSA based scanner. Development was done on an IBM-PC clone running the FreeBSD Unix Operating System. A number of ANSI C programs have been successfully compiled and animated using both the OSF/Motif and Microsoft Windows animators.

10.2 Future Enhancements

Many future enhancements to the current ANSI C compiler come to mind. Here are just a few of them:

- There are many ANSI C features that have not been implemented yet (see chapter 3). Implementation of these features is necessary to make this compiler meet the needs of the DYNALAB project.

- A few parts of the current complier use static allocation of memory. Although it is unlikely that this static allocation of memory could be a problem, it would be nice if the entire compiler used dynamic memory allocation.

- The PCCTS parser has very good error reporting and recovery. However, the ANSI C compiler does not fully take advantage of these features. A future improvement would be better error recovery so that a user receives more than one error message before the compiler terminates. Also, some error messages could be a little more descriptive.

- New versions of PCCTS are constantly being released. Whenever possible, the ANSI C compiler should use these new versions of PCCTS so that it can take advantage of bug fixes and new features present in them.

Bibliography

[Birch 90]	M. L. Birch. *An Emulator for the E-machine.* Master's thesis. Computer Science Department, Montana State University. June 1990.
[Birch, *et al* 95]	M. R. Birch, C. M. Boroni, F. W. Goosey, S. D. Patton, D. K. Poole, C. M. Pratt, R. J. Ross. "A Dynamic Computer Science Laboratory Infrastructure Featuring Program Animation", *SIGCSE Bulletin*, Volume 27, Number 1, pp 29–33. March 1995.
[Brown 88-1]	M. Brown. *Algorithm Animation.* The MIT Press, Cambridge, Massachusetts. 1988.
[Brown 88-2]	M. Brown. "Exploring Algorithms Using Balsa-II", *Computer*, Volume 21, Number 5. May 1988.
[Fischer 88]	C. N. Fischer and R. J. LeBlanc, Jr. *Crafting a Compiler.* Benjamin/Cummings Publishing Company, Menlo Park, California. 1988.
[Fischer 91]	C. N. Fischer and R. J. LeBlanc, Jr. *Crafting a Compiler in C.* Benjamin/Cummings Publishing Company, Menlo Park, California. 1991.
[Foster 91]	L. S. Foster. *C By Discovery.* Scott/Jones Inc. Publishers, El Granada, California. 1991.
[Goosey 93]	F. W. Goosey. *A miniPascal Compiler for the E-machine.* Master's thesis. Computer Science Department, Montana State University. April 1993.
[Holub 90]	A. I. Holub. *Compiler Design in C.* Prentice-Hall Inc., Englewood Cliffs, New Jersey. 1990.

[Kernighan 88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall Inc., Englewood Cliffs, New Jersey. 1988.

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

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

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

[Parr 93] T. Parr, W. Cohen, and H. Dietz. *PCCTS Reference Manual.* On-line reference manual.

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

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

[Pratt 95] C. M. Pratt. *An OSF/Motif Program Animator for the DYNALAB System.* Master's thesis. Computer Science Department, Montana State University. May 1995.

[Ross 91] R. J. Ross. "Experience with the DYNAMOD Program Animator", *SIGCSE Bulletin*, Volume 23, Number 1, pp 35–42. March 1991.

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

[Ross 95] R. J. Ross. "Using Visual Demonstrations to Teach Computer Science", Invited DYNALAB presentation, *SIGCSE Bulletin*, Volume 27, Number 1, pp 370–371. March 1995.

[Stevens 93] A. Stevens. *teach yourself...C++.* MIS:Press, New York, New York. 1993.

Appendix A

The E-machine Instruction Set

This appendix, which is adapted from chapter 2 of Birch's thesis [Birch 90], appendix A of Goosey's thesis [Goosey 93], and appendix A of Poole's thesis [Poole 94], lists all of the instructions in the instruction set of the E-machine. A pseudo assembly language format is used to describe the instructions and closely resembles the format used in the CODESECTION of the E-machine object file. The object file is described in detail in chapter 2 of this thesis.

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

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

The mnemonic field is separated from the other fields by one or more spaces, and the remaining fields are separated by commas. The CFLAG field must be either *c* or *n* to designate whether the instruction is to be treated as critical (*c*) or noncritical (*n*). The TYPE field holds a single capital letter, I, R, B, C, or A, referring to the data types *integer, real, boolean, character*, or *address*, respectively. The # refers to a constant specifying the number of an E-code label, a constant numeric value, or an E-machine variable register number. If the ADDR argument is used for the fourth field, it refers to any of the addressing modes described in appendix B.

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

add CFLAG, TYPE

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

alloc CFLAG, ADDR

Allocates a block of memory of positive integer size.

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

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

and CFLAG, TYPE

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

br CFLAG, #

Unconditionally branches to label #.

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

Reverse: No operation.

brt, brf CFLAG, #

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

call CFLAG, #

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

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

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

cast CFLAG, TYPE, TYPE

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

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

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

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

Reverse-Noncritical: Nothing happens.

div CFLAG, TYPE

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

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

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

inst CFLAG, #

Creates an instance of the variable register #.

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

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

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

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

label CFLAG, #

Marks the location to which a branch may be made.

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

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

link CFLAG, #

Associates one variable register with the value of another.

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

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

loadar CFLAG, ADDR

Places the address ADDR in the address register.

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

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

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

Reverse-Noncritical: Nothing happens.

loadir CFLAG, #

Places # into the index register.

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

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

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

Reverse-Noncritical: Nothing happens.

mod CFLAG, TYPE

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

mult CFLAG, TYPE

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

neg CFLAG, TYPE

Negates the top value on the evaluation stack.

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

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

nop CFLAG

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

not CFLAG, TYPE

Bitwise complements the top value of the evaluation stack.

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

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

or CFLAG, TYPE

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

pop CFLAG, TYPE, ADDR

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

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

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

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

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

popar CFLAG

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

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

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

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

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

popd CFLAG

Pops the top value from the dynamic scope stack.

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

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

popir CFLAG

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

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

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

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

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

push CFLAG, TYPE, ADDR

Pushes the value in ADDR onto the evaluation stack.

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

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

pusha CFLAG, ADDR

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

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

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

pushd CFLAG, #

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

Forward: Pushes # onto the dynamic scope stack.

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

pushstr CFLAG

Pushes a string onto the evaluation stack. Immediately prior to generating this instruction, the compiler must have first pushed the index into the string space of the string onto the evaluation stack, and then pushed the length of the string onto the evaluation stack.

Forward: Pops the top two values of the evaluation stack. Retrieves the specified number of characters (indicated by the value that was on the top of the evaluation stack) from the string space, starting at the string space index (indicated by the second value from the top of the evaluation stack) plus the specified number characters (i.e., the desired string length). That is, the string is retrieved in reverse order. As the characters are being retrieved, they are placed onto the evaluation stack, resulting in the string in normal order on the evaluation stack. The string length is then pushed onto the save stack.

Reverse: Pops the string length from the save stack and then pops and discards the corresponding number of characters from the evaluation stack. It then pushes two DUMMY values onto the evaluation stack.

read CFLAG, TYPE, ADDR

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

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

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

return CFLAG

Returns to the appropriate program address following a call instruction.

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

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

shl CFLAG, TYPE, #

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

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

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

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

Reverse-Noncritical: Nothing happens.

shr CFLAG, TYPE, #

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

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

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

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

Reverse-Noncritical: Nothing happens.

sub CFLAG, TYPE

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

unalloc CFLAG, #

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

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

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

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

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

uninst CFLAG, #

Dispose of an instance of variable register #.

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

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

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

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

unlink CFLAG, #

Disassociates a variable register from another.

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

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

write CFLAG, TYPE, ADDR

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

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

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

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

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

xor CFLAG, TYPE

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

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

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

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

Reverse-Noncritical: Pushes DUMMY onto the evaluation stack.

Appendix B

The E-machine Addressing Modes

This appendix, which is adapted from chapter 2 of Birch's thesis [Birch 90], appendix B of Goosey's thesis [Goosey 93], and appendix B of Poole's thesis [Poole 94] describes the various addressing modes allowed in E-machine instructions. Quite a few modes are defined in order to accommodate standard high level language data structures more conveniently. Note that each addressing mode refers to either the data at the computed address or the computed address itself, depending on the instruction. That is, for those instructions that need a data value, such as **push**, the data value at the address computed from the addressing mode is used. For instructions that need an address, such as **pop**, the address that was computed from the addressing mode is used.

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

constant mode - C#

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

Example:

$A := 1.5;$

could be translated into the following, where V1 is the variable register associated with A:

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

variable mode - V#:

variable register # $\longrightarrow$ top of variable stack $\longrightarrow$ data memory

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

Example:

$B := 1;$

could be translated into:

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

variable indirect - (V#):

variable register # $\longrightarrow$ top of variable stack $\longrightarrow$ data memory $\longrightarrow$ data memory

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

Example:

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

could be translated into:

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

variable offset mode - V#{offset}:

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

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

Example:

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

address indirect - (A):

address register ⟶ data memory

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

Example:

```
c = *(*g);
```

could be translated into:

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

address offset mode - A{offset}:

address register + IR $\longrightarrow$ *data memory*

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

Example:

```
I := H↑.Data
```

could be translated into:

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

variable indexed mode - V#[index]:

variable register # $\longrightarrow$ *top of variable stack + IR * data size* $\longrightarrow$ *data memory*

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

Example:

```
B := L[3];
```

could be translated into:

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

address indexed mode - A[index]:

*address register + IR * data size* $\longrightarrow$ *data memory*

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

Example:

B := S↑[4];

could be translated into:

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