AN Ada/CS COMPILER FOR THE E-MACHINE

by

David Keith Poole

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

of

Master of Science

in

Computer Science

Montana State University
Bozeman, Montana

July 1994

junk

APPROVAL

of a thesis submitted by

David Keith Poole

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]. As this thesis is being completed, Craig Pratt and Chris Boroni are developing DYNALAB animators, and Tory Eneboe is implementing a C comppiler for the project.

I would like to take this opportunity to thank my graduate committee members, Dr. Rockford Ross, Dr. Gary Harkin, and Dr. Robert Cimikowski, 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, 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

Contents—Continued

Contents—Continued

List of Tables

List of Figures

ABSTRACT

This thesis is part of the third phase in the development of a program animation system called DYNALAB (DYNAmic LABoratory). DYNALAB is an interactive software system that demonstrates programming and computer science concepts at an introductory level. The first DYNALAB development phase was the design of a virtual computer—the E-machine (Education Machine). The E-machine was designed by Samuel D. Patton and is presented in his Master's thesis, *The E-machine: Supporting the Teaching of Program Execution Dynamics*. In order to facilitate the support of program animation activities, the E-machine has many unique features, notably the ability to execute in reverse. The second phase in the development of DYNALAB was the design and implementation of an E-machine emulator, which is presented in Michael L. Birch's Master's thesis, *An Emulator for the E-machine*. The third, ongoing phase of the DYNALAB project is the development of compilers generating E-Machine code, the first of which is the miniPascal compiler by Frances Goosey, described in her Master's thesis, *A miniPascal Compiler for the E-Machine*. This thesis presents the design and implementation of the second compiler for the E-machine. The compiler's source language is *Ada/CS*, which is a slightly modified subset of Ada.

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

CHAPTER 1

INTRODUCTION

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, 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]. During the miniPascal compiler development, the E-machine and its emulator were again modified as practical considerations uncovered new design issues.

Continuing DYNALAB's third phase, the Ada/CS compiler and this thesis were developed. Ada/CS is a modified subset of Ada, a large and complex programming language designed for software engineering and reliability. The Ada/CS compiler was developed using C++, and takes advantage of its object oriented features. As with the previous development stages, during the course of the Ada/CS compiler de-

velopment, the E-Machine and its emulator were modified and updated as problems were uncovered.

The fourth phase of the DYNALAB project, currently in progress, is the design and implementation of program animators that will drive the E-machine and display programs in dynamic, animated fashion under control of the user. Once the animators are complete, the first functional version of DYNALAB will be ready for use in introductory computer science laboratory and lecture courses by students and instructors alike. An animator for Unix using Motif and another for Microsoft Windows are currently under development.

The DYNALAB project will not end at this point. Compilers for C and C++ are in the initial stages of development. Also, work will continue on the Ada/CS compiler to make it suitable for use in teaching. Algorithm animation (as opposed to program animation—-see for example, [Brown 88-1, Brown 88-2]) is also a planned extension to DYNALAB. In fact, the DYNALAB project will likely never be finished, as new ideas and pedagogical conveniences are incorporated as they become apparent.

Preview

The thesis consists of eight 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, and Goosey theses. During the Ada/CS 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 an introduction to the modified Ada subset Ada/CS, including a description of the Ada features of Ada/CS and differences in common language features. Chapter 4 is an introduction to the Purdue Compiler Construction Tool Set (PCCTS), the parser/scanner tool used in the development of Ada/CS. Chapter 5 describes the Ada/CS compiler symbol table, Chapter 6 describes the Ada/CS type mechanisms, and Chapter 7 describes Ada/CS E-Machine code generation considerations. The current status of the Ada/CS compiler is discussed in chapter 8.

Since there are many E-code examples used throughout the thesis, appendices A and B are included for completeness. Appendix A describes the E-machine instruction set and appendix B describes the E-machine addressing modes. Both of these appendices are adapted from chapter 2 of Birch's thesis.

CHAPTER 2

THE E-MACHINE

This chapter is included to provide a description of the E-machine and is adapted from chapter 5 of Patton's thesis [Patton 89], chapters 1, 2, and 3 of Birch's thesis [Birch 90], and chapters 2 and 3 of Goosey's thesis [Goosey 93]. This chapter is a summary and update of information from those three theses (much of the material is taken verbatim). New E-machine features that have been added as a result of this thesis are noted by a leading asterisk (*).

The E-machine is a virtual computer with its own machine language, called E-code. The E-code instructions are described in appendix A; these instructions may reference various E-machine addressing modes, which are described in appendix B. The E-machine's 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 is the second. The real purpose of the E-machine is to support the DYNALAB program animation system, as described more fully in [Ross 91], [Birch 90], [Ross 93] and in Patton's thesis [Patton 89], where it was called a "dynamic display system."

E-machine Design Considerations

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

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

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

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

One other aspect of program animation substantially influenced the design of the reversing mechanism of the E-machine. Since the animator is meant to animate high level language programs, the E-machine actually has to be able to effect reversal only through high level language animation units in one reversal step, not each low level E-machine instruction in the packet that is the translation of an animation unit. This observation led to further efficiencies in the design of the E-machine and the incorporation of two classes of E-machine code instructions, critical and 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.

E-machine Architecture

Figure 1 shows the logical structure of the E-machine. A stack-based architecture was chosen for the E-machine; however, a number of components that are not found in real stack-based computers were included.

Program memory contains the E-code program currently being executed by the E-machine. Program memory is loaded with the instruction stream found in the CODESECTION of the E-machine object code file, which is described later in this chapter. The *program counter* contains the address in program memory of the next E-code instruction to be executed. The *previous program counter*, needed for reverse execution, contains the address in program memory of the most recently executed E-code instruction.

Packet memory contains information about the translated E-code packets and their corresponding source language animation units. Packet memory, which is

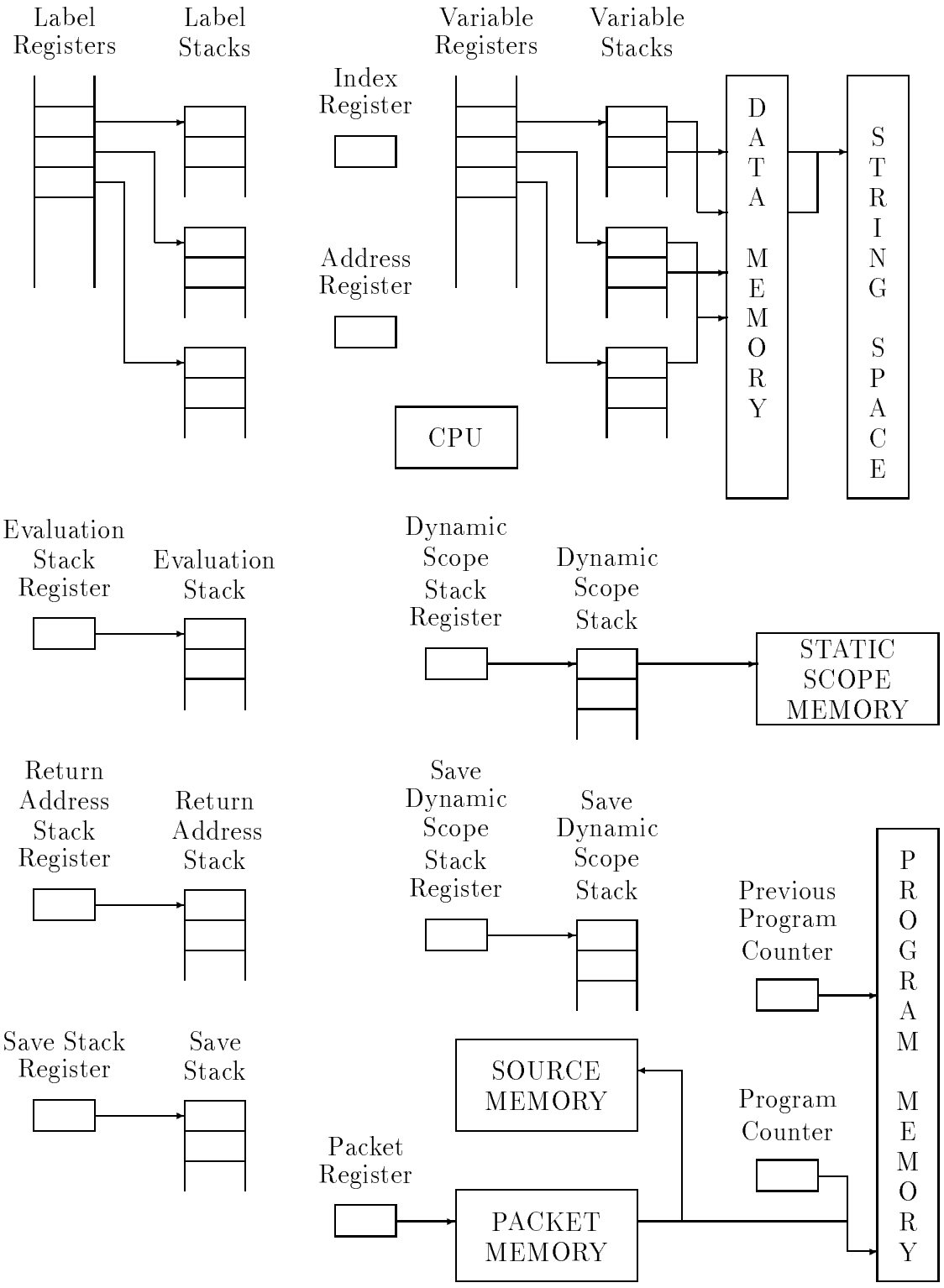

Figure 1: The E-machine

loaded with the information found in the PACKETSECTION of the E-machine object code file, essentially effects the "packetization" of the E-code program found in source memory. Packet information includes the starting and ending line and column numbers of the original source program animation unit (e.g, an entire assignment statement, or just the conditional expression in an if statement) whose translation is the packet of E-code instructions about to be executed. Other packet information includes the starting and ending program memory addresses for the E-code packet, which are used internally to determine when execution of the packet is complete. The *packet register* contains the packet memory address of the packet information corresponding to either the next packet to be executed, or the packet that is currently being executed.

Source memory 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 an Ada/CS program, a variable named `Result` might be declared in the current program scope and another variable—also named `Result`—might be declared in another enclosing procedure scope. The compiler will assign a unique variable register to each of these two variables. Once a variable is assigned a variable register, the register remains associated with the variable for the duration of the program's compilation and subsequent execution, regardless of whether the variable is currently active or not.

The information held in a variable register consists of the corresponding variable's size (e.g., number of bytes) as well as a pointer to a corresponding *variable stack*. Each variable stack entry, in turn, holds a pointer into *data memory*, where the actual variable values are stored. The variable stacks are necessary because a particular variable may have multiple associated instances due to its being declared in recursive procedures or functions. In such instances, the top of a particular variable's register stack points to the value of the current instance of the associated variable in data memory; the second stack element points to the value of the previous instantiation of the variable, and so on. The E-machine's data memory represents the usual random access memory found on real computers. The E-machine, however, uses data memory only to hold data values (it does not hold any of the program instructions).

The *string space* component of the E-machine's architecture contains the values of all string literals and enumerated constant names encountered during the compilation of a program. The string space is loaded with the information contained in the STRINGSECTION of the E-machine object file. Currently, this string space is used only by the animator when displaying string constant and enumerated constant values. 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 Pascal is a variable that contains a data address. Data at that address can be accessed using the address register via the appropriate E-machine addressing mode. The address register can be used in place of variable registers for any of the addressing modes.

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

The *return address stack* (or *call stack*) is the E-machine's mechanism for implementing procedure and function calls. When a subroutine call is made, the program counter plus one is pushed onto the return address stack. Then, when the E-machine executes a return from subroutine instruction, all it has to do is load the program counter with the top of the return address stack. A pointer to the top of the return

address stack is kept in the *return address stack register*.

The *save stack* contains information necessary for reverse execution. Whenever some critical information (as determined by the execution of a critical instruction) is about to be destroyed, the required information is pushed onto the save stack. This ensures that when backing up, the instruction that most recently destroyed some critical information can be reversed by retrieving that critical information from the save stack. The *save stack register* points to the top of the save stack.

The *dynamic scope stack* allows the animator to determine all currently active scopes for memory display. The animator must be able to display variable values associated with the execution of a packet both from within the current invocation of a procedure (or function) and from within the calling scope(s). That is, the animator must have the ability to illustrate a program's run time stack during execution. The Static Scope Table, which is loaded into *static scope memory* from the E-machine object file's STATSCOPESECTION, provides the animator with the information relevant to the static nature of a program (e.g., information pertaining to variable names local to a given procedure). However, the specific calling sequence resulting in a particular invocation of a procedure (or function) is obviously not available in the static scope memory.

To keep track of the set of active scopes at any point during program execution, the dynamic scope stack provides the dynamic chain as found in the run time stack of activation records generated by most conventional compilers. (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, a procedure name, or a function name—contains the index of the Static Scope Table entry describing that name (i.e., a static scope name). Once these

indices are available, the animator can then use the Static Scope Table information to determine the variables whose values must be displayed following the execution of a packet. The animator needs access to the entire dynamic scope stack in order to display all pertinent data memory information following the execution of any given packet. The *dynamic scope stack register* points to the top of the dynamic scope stack.

In order to handle reverse execution, a *save dynamic scope stack* was added to the E-machine architecture. This stack records the history of procedures and/or functions that have been called and subsequently returned from. The *save dynamic stack register* points to the top of this stack.

Finally, the CPU is what executes E-Machine instructions. It is the E-Machine emulator originally programmed by Birch and is described in the next section.

E-machine Emulator

The E-machine emulator was designed and written by Michael Birch and is described in his thesis [Birch 90]. The emulator's design essentially follows the design of the E-machine presented the previous sections of this chapter. The emulator was written in ANSI Standard C for portability and has been compiled on a variety of hardware platforms from a 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. During the course of Ada/CS development, a Unix Motif animator was also being

developed. A pre-release version of this animator was used to test compiled Ada/CS programs.

E-machine Object File Sections

The E-machine emulator defines the object file format that must be generated by a compiler. As a result of the Ada/CS compiler development, several changes were made to the E-machine object file definition and are denoted with a leading asterisk (*) in the following discussion. A single E-code object file ready for execution on the E-machine consists of 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.

***The HEADERSECTION**

The HEADERSECTION was created during Ada/CS development and 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.

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.

***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 (discussed later in this chapter);
- *a variable register number that will hold the result of the execution of a conditional expression.

The VARIABLESECTION

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

The LABELSECTION

The LABELSECTION consists of label structures describing the label numbers generated by the compiled program. A label structure consists of a single field that contains the program address at which the corresponding label is defined. This information is used to initialize the label program address information held in the E-machine's label registers.

The SOURCESECTION

The SOURCESECTION contains a copy of the source program being executed. Each record in this section corresponds to a line of original source code, and is loaded

into the E-machine's source memory at run time. Source memory is referenced only by the animator for display purposes. The animator references source memory via packet memory information that describes correlations between the currently executing E-code packet and the corresponding source program animation unit. The animator references the packet structure fields that hold starting and ending line and column numbers in source memory to determine the animation unit to highlight.

***The STATSCOPESECTION**

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

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 procedure name);
- upper and lower bounds (for array variables);
- the index of the Static Scope Table entry containing the next array index bounds (for multidimensional arrays);
- the offset value (for record fields);
- an enumerated value indicating the data type (e.g., INTEGER, RECORD, or STRING);

- the record size (for arrays of records);
- a pointer to this entry's parent Static Scope Entry;
- a pointer to the child of this entry (e.g., if this static scope entry describes a procedure, this field would hold the index of the first entry in the static scope block describing the variables declared local to the procedure);
- a variable register number (for variable names);
- a number statically assigned to procedure and functions entries; this number is used in determining the dynamic scoping level at execution time;
- *a value denoting whether a variable name is an array, and if so, whether it is static or dynamic;
- *a value describing the index type of an array variable (e.g. integer, enumerated, or character).

The 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.

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.

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 Ada/CS compiler accomplishes these two tasks.

Identifying Program Animation Units

```
 0  Package Body Samp1 Is
 1
 2  I,J,K : Integer;
 3  N : Float;
 4
 5  Procedure Init( X,Y : in out Integer ) Is
 6  t : integer;
 7  Begin
 8    X := 1;
 9    Y := 2;
10  End;
11
12  Begin
13    Init(I,J);
14    If I < 10 Then
15        K := 100;
16    else
17        K := 0;
18    end if;
19    N := K + I*J;
20  end;
```

Figure 2: Source Code for Program Samp1

The compiler identifies individual animation units as it is parsing the high level language source code. Consider the Ada/CS program in figure 2 (the numbers on the left correspond to line numbers in the source program file). For this program, the Ada/CS compiler identifies the twenty animation units shown in figure 3 (the numbers on the left correspond to each animation unit's associated packet structure, as discussed in the next section). These animation units will be successively highlighted (in the original source program of figure 2) by the 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

```
 0  Package Body Samp1 Is
 1  I,J,K : Integer;
 2  N : Float;
 3  Procedure Init( X,Y : in out Integer ) Is
 4  t : integer;
 5  Begin
 6  X := 1;
 7  Y := 2;
 8  End;
 9  Begin
10  Init(I,J);
11  If
12  I < 10
13  Then
14  K := 100;
15  else
16  K := 0;
17  end if;
18  N := K + I*J;
19  end;
```

Figure 3: Animation Units Identified in Program Samp1

on subjective esthetics of program animation. As can be seen from this example, an animation unit can correspond to "chunks" of source code representing a single keyword, an entire program statement, the conditional part of an if statement, and so forth.

Translating Program Animation Units into E-code Packets

Once the compiler has identified an animation unit, it must then translate this unit into a corresponding packet of E-code instructions along with an associated descriptive packet structure. Thus, compilation of the example given in figure 2 would result in the generation of twenty-three E-code packets and twenty-three corresponding packet structures. Three of these packets have no corresponding source and are explained later so there are twenty packets with an associated source "chunk". Figure 4 shows the pseudo assembly language representation of the E-code

instructions generated for the Ada/CS program shown in figure 2.

The numbers shown on the left in figure 4 correspond to program memory addresses (instruction numbers). The individual packets, corresponding to the animation units of figure 3, are shown separated by blank lines in figure 4.

Table 1 shows the array of packet structures—called the Packet Table—describing the individual packets resulting from the translation of the program of figure 2. The PacketNumber field (column) is included for clarity—it is not part of the Packet Table. The first two fields in the Packet Table (StartAddr and EndAddr) give the starting and ending addresses in program memory of the E-code packet. The next four fields (StartLine, StartCol, EndLine, and EndCol) demark the physical location of the packet's corresponding program animation unit in the source program array. The ScopeIndex field in the Packet Table is discussed in the next section of this chapter. The final two fields (DisplayPacket and TestResultVar) provide additional information necessary for animating an animation unit.

As might be guessed by the fact that there are twenty source "chunks" and twenty-three 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 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.

```
 0  pushd c, I, C13
 1  inst c, V0
 2  inst c, V1
 3  push c, I, C-2147483648
 4  pop c, I,V0
 5  push c, I, C2147483647
 6  pop c, I,V1
 7  inst c, V2
 8  inst c, V3
 9  push c, R, C-100.000000
10  pop c, R,V2
11  push c, R, C100.000000
12  pop c, R,V3
13  inst c, V4
14  push c, I, C0
15  pop c, I,V4
16  inst c, V5
17  push c, I, C65536
18  pop c, I,V5
19  nop c

20  inst c, V6
21  inst c, V7
22  inst c, V8

23  inst c, V9

24  br c, L0

25  label c, L1
26  pushd c, I, C10
27  inst c, V11
28  pop c, I,V11
29  inst c, V10
30  pop c, I,V10

31  inst c, V12

32  nop c

33  push c, I, C1
34  pop c, I,V10

35  push c, I, C2
36  pop c, I,V11

37  label c, L3
38  push c, I,V10
39  push c, I,V11
40  uninst c, V11
41  uninst c, V12
42  uninst c, V10
43  popd c

44  return c

45  label c, L0

46  push c, I,V6
47  nop c
48  push c, I,V7
49  nop c
50  call c, L1
51  label c, L8
52  pop c, I,V7
53  pop c, I,V6

54  nop c

55  push c, I,V6
56  push c, I, C10
57  less c, I
58  brf c, L9

59  nop c

60  push c, I, C100
61  pop c, I,V8

62  br c, L10

63  label c, L9

64  push c, I, C0
65  pop c, I,V8

66  label c, L10

67  push c, I,V8
68  push c, I,V6
69  push c, I,V7
70  mult c, I
71  add c, I
72  cast c, I,R
73  pop c, R,V9

74  uninst c, V9
75  uninst c, V6
76  uninst c, V8
77  uninst c, V7
78  popd c
```

Figure 4: E-code Instructions Resulting from Compilation of Program Samp1

Packet Number	Start Addr	End Addr	Start Line	Start Col	End Line	End Col	Scope Index	Display Packet	Test ResultVar
0	0	19	0	0	0	20	0	07	-1
1	20	22	2	0	2	15	3	07	-1
2	23	23	3	0	3	9	4	07	-1
3	24	24	-1	-1	-1	-1	4	00	-1
3	25	30	5	0	5	40	4	07	-1
4	31	31	6	0	6	11	3	07	-1
5	32	32	7	0	7	4	3	07	-1
6	33	34	8	2	8	8	3	07	-1
7	35	36	9	2	9	8	3	07	-1
8	37	43	10	0	10	3	3	07	-1
9	44	44	10	0	10	3	4	07	-1
10	45	45	12	0	12	4	4	01	-1
11	46	53	13	2	13	11	4	07	-1
12	54	54	14	2	14	3	4	07	-1
13	55	58	14	5	14	10	4	07	10
14	59	59	14	12	14	15	4	07	-1
15	60	61	15	4	15	12	4	07	-1
16	62	62	-1	-1	-1	-1	4	07	-1
17	63	63	16	2	16	5	4	00	-1
18	64	65	17	4	17	10	4	07	-1
19	66	66	18	2	18	8	4	07	-1
20	67	73	19	2	19	14	4	07	-1
21	74	78	20	0	20	3	4	07	-1

Table 1: Packet Table Resulting from Compilation of Program Samp1

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 4 for the source code in figure 2, packets 24 and 62 are invisible. Packet 24 branches from the variable instantiations over the subroutines to the start of the program. Packet 62 is the branch over the "else" part of the if statement. Neither of these two packets has a corresponding animation unit in the source code. Ada/CS uses invisible packets on occasions such as popping return parameters after a subroutine call and copying function return values into their destinations.

TestResultVar was added during the development of the Ada/CS compiler 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 and not k>5 or flag then
```

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, making understanding the program simpler.

Generation of the Static Scope Table

The compiler writer must also provide information describing all of the data memory variables that the animator must display. This information is provided in the Static Scope Table, a linear array which is, in turn, logically divided into numerous scope blocks. Each scope block describes the identifiers (e.g., variable names and procedure names) declared in a single static scope in a program. Even though this information

is obtained from the compiler's symbol table, the generation of the Static Scope Table is not a straightforward task due to scope nesting characteristics of many high level languages.

Table 2 shows the Static Scope Table that is generated as a result of compiling the Ada/CS program given in figure 2. The Entry (entry number) column, or field, is included for clarity—it is not part of the Static Scope Table. This Static Scope Table consists of three scope blocks—a block describing the identifiers declared within the scope of procedure Init (entries 0–4), a block describing the identifiers declared within the scope of program Samp1 (entries 5–11), and a "bootstrap" block describing the main program entry (entries 12–14).

Entry	Id Name	Array Type	Upr Bnd	Lwr Bnd	Nxt Idx	Off set	Type	Rec Siz	Par ent	Ch ild	Var Reg	Proc Num	Index Type
Scope block describing procedure Init													
0		-	-	-	-	-	HEADER	-	5	-	-	-	-
1	X	-	-	-	-	-	INTEGER	-	-	-	5	-	-
2	Y	-	-	-	-	-	INTEGER	-	-	-	4	-	-
3	t	-	-	-	-	-	INTEGER	-	-	-	6	-	-
4		-	-	-	-	-	END	-	-	-	-	-	-
Scope block describing program Samp1													
5		-	-	-	-	-	HEADER	-	12	-	-	-	-
6	I	-	-	-	-	-	INTEGER	-	-	-	2	-	-
7	J	-	-	-	-	-	INTEGER	-	-	-	1	-	-
8	K	-	-	-	-	-	INTEGER	-	-	-	0	-	-
9	N	-	-	-	-	-	REAL	-	-	-	3	-	-
10	Init	-	-	-	-	-	PROCEDURE	-	-	0	-	1	-
11		-	-	-	-	-	END	-	-	-	-	-	-
Bootstrap scope block													
12		-	-	-	-	-	HEADER	-	-	-	-	-	-
13	Samp1	-	-	-	-	-	PROGRAM	-	-	5	-	0	-
14		-	-	-	-	-	END	-	-	-	-	-	-

Table 2: Static Scope Table Resulting from Compilation of Program Samp1

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

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

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

Simple variables such as integer, float, and boolean, 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

```
type matrix is array( 1..10 ) of integer;
a : matrix;
```

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 3.

Entry	Id Name	Array Type	Upr Bnd	Lwr Bnd	Nxt Idx	Off set	Type	Rec Siz	Par ent	Ch ild	Var Reg	Proc Num	Index Type
0	ProcName	-	-	-	-	-	HEADER	-	-	-	-	1	-
1	a	STATIC	10	1	-	-	INTEGER	-	-	-	3	-	INTEGER
2	-	-	-	-	-	-	END	-	-	-	-	-	-

Table 3: Static Scope Block for One Dimensional Array

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

The IndexType of the array uses the same type elements that appear in the "type" field of the static scope structure. High level languages such as Pascal and Ada allow arrays to be indexed by integers, enumerated names, characters, so the IndexType may be INTEGER, ENUMINT, CHARACTER, or BOOLEAN.

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 4.

```
type Matrix is array ( 1..10, FALSE..TRUE ) of integer;
a : Matrix;
```

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, which is indexed by a boolean type range, is described in the static scope block from 0 to 2. The array bounds are the integer representation of the boolean range.

Entry	Id Name	Array Type	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num	Index Type
0	Matrix	-	-	-	-	-	HEADER	-	-	-	-	-	-
1	-	STATIC	1	0	-	-	-	-	-	-	-	-	BOOLEAN
2	-	-	-	-	-	-	END	-	-	-	-	-	-
3	ProcName	-	-	-	-	-	HEADER	-	-	-	-	1	-
4	a	STATIC	10	1	0	-	INTEGER	-	-	-	3	-	INTEGER
5	-	-	-	-	-	-	END	-	-	-	-	-	-

Table 4: Static Scope Block for Two Dimensional Array

In the case of an array using an enumerated type as the index, the upper and lower ranges will be the indices into the string section (described later in this chapter).

As implemented, Ada/CS arrays are handled differently due to their dynamic nature—array bounds may be specified by variables that cannot be determined until run time. To this end, ArrayType would be DYNAMIC which causes the animator to interpret the upper and lower bounds as registers that contain the ranges.

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 type. By following these child indices, the animator can find all necessary record type information. For example, suppose we have the record

```
type complex is record
   real_part : float;
   imag_part : float;
end record;
c1, c2 : complex;
```

The static scope table for the record and its variables is shown in table 5. 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	Array Type	Upr Bnd	Lwr Bnd	Nxt Idx	Off set	Type	Rec Siz	Par ent	Ch ild	Var Reg	Proc Num	Index Type
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 5: Scope Block of Record Complex

The ProcNum Field

As each program, procedure, and function name identifier 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 an Ada/CS program name 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 5 contains a recursively called function (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 6.

```
Package Body Ftrl Is

num, nfact : Integer;

Function Fact(n:Integer) Return Integer Is
Begin
  If n = 0 Then
    Return 1;
  Else
    Return n * Fact(n-1);
  End If;
End;

Begin
  num := 3;
  nfact := Fact(n);
End;
```

Figure 5: Source Code for Program Ftrl

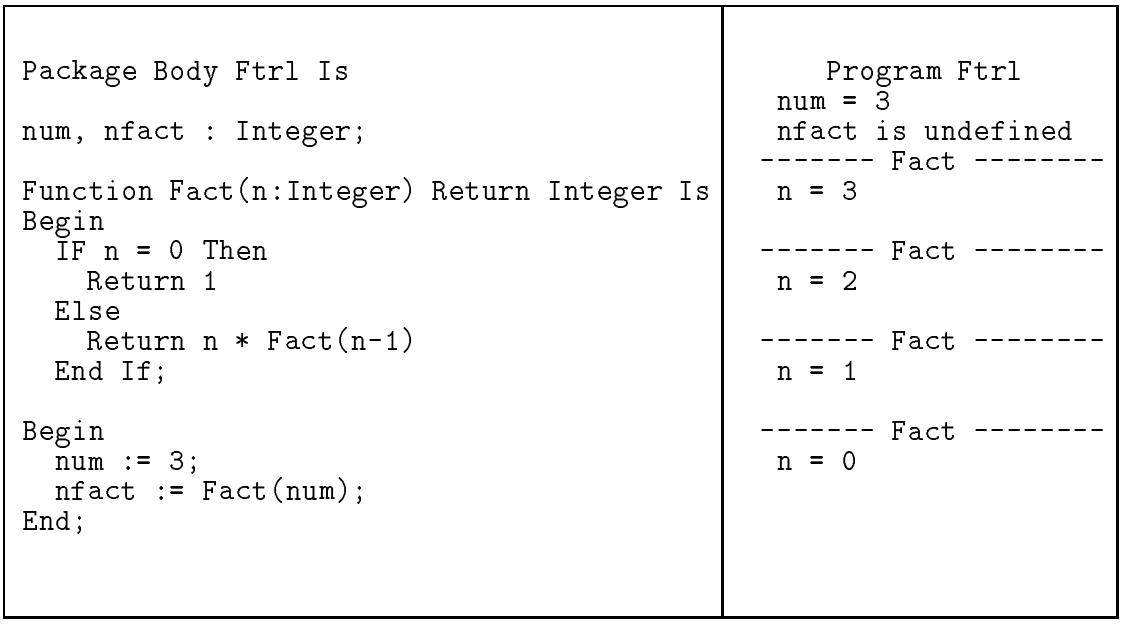

Figure 6: Animation Display After Final Recursive Call of Function Fact

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 5 is shown in table 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 5, this array has two entries. Entry 0 corresponds to the program scope and entry 1 corresponds to function Fact. 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 procedure or 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.

Entry	Id Name	Array Type	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num	Index Type
Scope block describing function Fact													
0		-	-	-	-	-	HEADER	-	4	-	-	-	-
1	n	-	-	-	-	-	INTEGER	-	-	-	2	-	-
2	Fact	-	-	-	-	-	INTEGER	-	-	-	3	-	-
3		-	-	-	-	-	END	-	-	-	-	-	-
Scope block describing program Ftrl													
4		-	-	-	-	-	HEADER	-	9	-	-	-	-
5	num	-	-	-	-	-	INTEGER	-	-	-	1	-	-
6	nfact	-	-	-	-	-	INTEGER	-	-	-	0	-	-
7	Fact	-	-	-	-	-	FUNCTION	-	-	0	-	1	-
8		-	-	-	-	-	END	-	-	-	-	-	-
Bootstrap scope block													
9		-	-	-	-	-	HEADER	-	-	-	-	-	-
10	Ftrl	-	-	-	-	-	PROGRAM	-	-	4	-	0	-
11		-	-	-	-	-	END	-	-	-	-	-	-

Table 6: Static Scope Table Resulting from Compilation of Program Ftrl

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

translating the animation unit `Fact:=1`, the procedure count array and the dynamic scope stack are in the state shown in figure 7. The values in the procedure count array indicate that the program Ftrl has one active "call" and that function Fact has four active calls. In this example, the animator begins its retrieval of data memory values by examining the value at the *bottom* of the dynamic scope stack. The bottom stack value is 10, which means that the animator now examines the tenth 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.

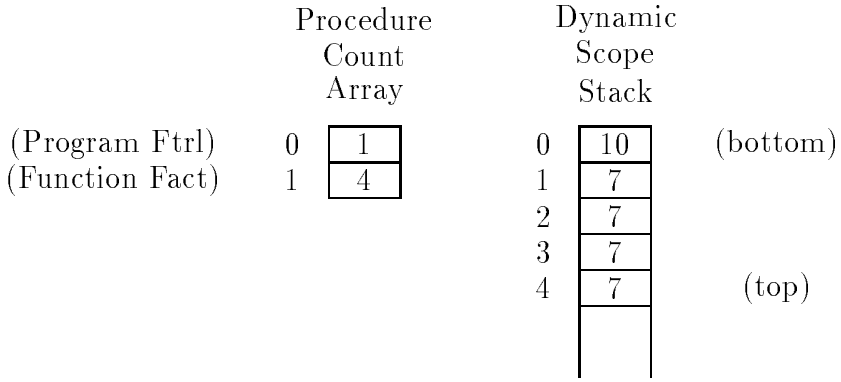

Figure 7: Procedure Count Array and Dynamic Scope Stack

Next, the animator examines the value in entry 1 in the dynamic scope stack. This value is 7, corresponding to the seventh entry in the Static Scope Table. This entry, whose ProcNum field has the value 1, describes function Fact. The animator then examines entry 1 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 so that the next iteration will result in displaying the values associated with the first recursive call to function Fact. The animator continues this process until the dynamic scope stack is exhausted, resulting in the display shown in figure 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

```
X := 1;
```

in procedure Init of figure 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 two entries, each containing an index into the Static Scope Table shown in Table 1. The top entry contains the value 10 and the bottom entry contains the value 13. These values indicate to the animator that procedure Init (Static Scope Table entry number 10) is the most current active scope and that program Samp1 (entry number 13) is the calling scope. By using the child pointers associated with these two Static Scope Table entries, the animator can now determine the appropriate data memory values to be displayed. Figure 8 shows a possible animation resulting from the execution of this animation unit. The arrow (`==>`) pointing to the instruction `Y := 2;` indicates where animation proceeds.

```
Package Body Samp1 is

  I,J,K : Integer;
  N : Integer;

  Procedure Init(X,Y:in out Integer) Is
    Begin
      X := 1;
  ==> Y := 2;
    End;

  Begin
    Init(I,J);
    If I < 10 Then
      K := 100;
    Else
      K := 0;
    End If;
    N := K + I*J
  End;
```

```
  Program Samp1
I is undefined
J is undefined
K is undefined
N is undefined
------------------
  Procedure Init
X = 1
Y is undefined
```

Figure 8: Animation Display After Execution of `X := 1;`

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

```
I,J,K:INTEGER;
```

and has returned control to the animator. The animator, via a query of the dynamic scope stack, now determines that only the values of the variables contained in the outer program scope should be displayed. The variables listed in the block describing this scope's variables are `I`, `J`, `K`, and `N`. However, at this point in the program's execution, variable `N` has not yet been declared, and thus should not be displayed. The ScopeIndex field of the packet structure associated with the above animation unit contains the value 3. This value indicates to the animator that it should only display data memory values for entries numbered 0, 1, 2, and 3 in the window associated with the *most current* active scope block which in this case starts at scope table index 5. Hence, the animator will display the values of the variables `I`, `J`, and `K` (0 stands for the HEADER entry). In this case, all of these variables would have the value "undefined," as they have only just been declared and have not yet had values assigned to them.

Translating Enumerated Type Variables

Ordinarily, only the internal numeric value of an enumerated type variable is required in translated object code. It is desirable, however, for program animation purposes to have the animator display the enumerated constant name rather than just the internal numeric value of a variable of an enumerated type. Thus, when translating an enumerated type variable, the compiler must provide a way for the animator to relate the variable's internal numeric value to its corresponding constant name. This task 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 an Ada/CS program.

The String Section consists of a statically allocated character array containing all of the enumerated constant names defined in an Ada/CS 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 9, the animator can display the enumerated type variable values as shown in figure 10.

Figure 11 illustrates the relationship of the E-machine's string space with the variable registers and data memory. This illustration assumes that a variable register associated with an enumerated type variable represents 32-bits of data memory. The 16 high-order bits of this data memory location contain the dynamically determined internal numeric value of the enumerated constant associated with this variable; the 16 low-order bits contain an index into the string space where the associated enumerated constant name can be found. As can be seen in figure 11, the index into the string space is always that of the first constant name of the enumerated type. This is due to the fact that the compiler can statically generate code to increment or decrement the numeric value of an enumerated type variable (e.g., for an enumerated type control variable in a for loop). The compiler cannot, however, statically determine in advance the absolute string space index of the enumerated constant name associated with an enumerated type variable at any given time. Instead, the animator has the capability to retrieve the variable's numeric value and the starting string space index. The animator can then step sequentially through

the string space until the name corresponding to the numeric value is found; the names are null-terminated, thus allowing such a search.

```
Package Body Payroll Is

type Days = (MON,TUES,WED,THURS,FRI);
type Frequency = (WEEK,MONTH);

OffDay, PayDay : Days;
PayFreq : Frequency;

Begin
  OffDay := WED;
  PayDay := FRI;
  PayFreq := WEEK;
End Payroll;
```

Figure 9: Source Code for Program Payroll1

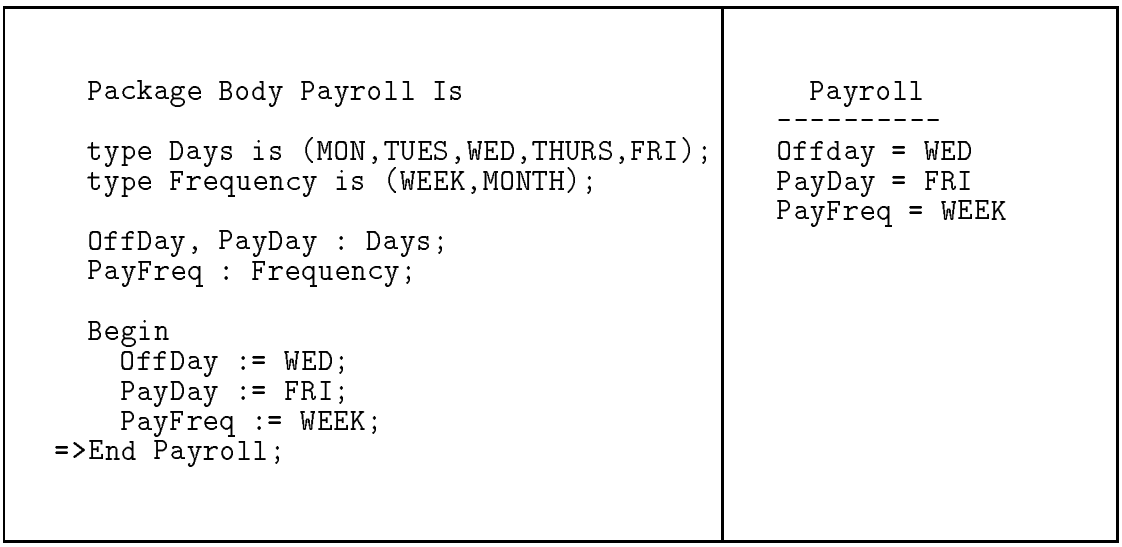

Figure 10: Animation Display After Execution of Program Payroll1

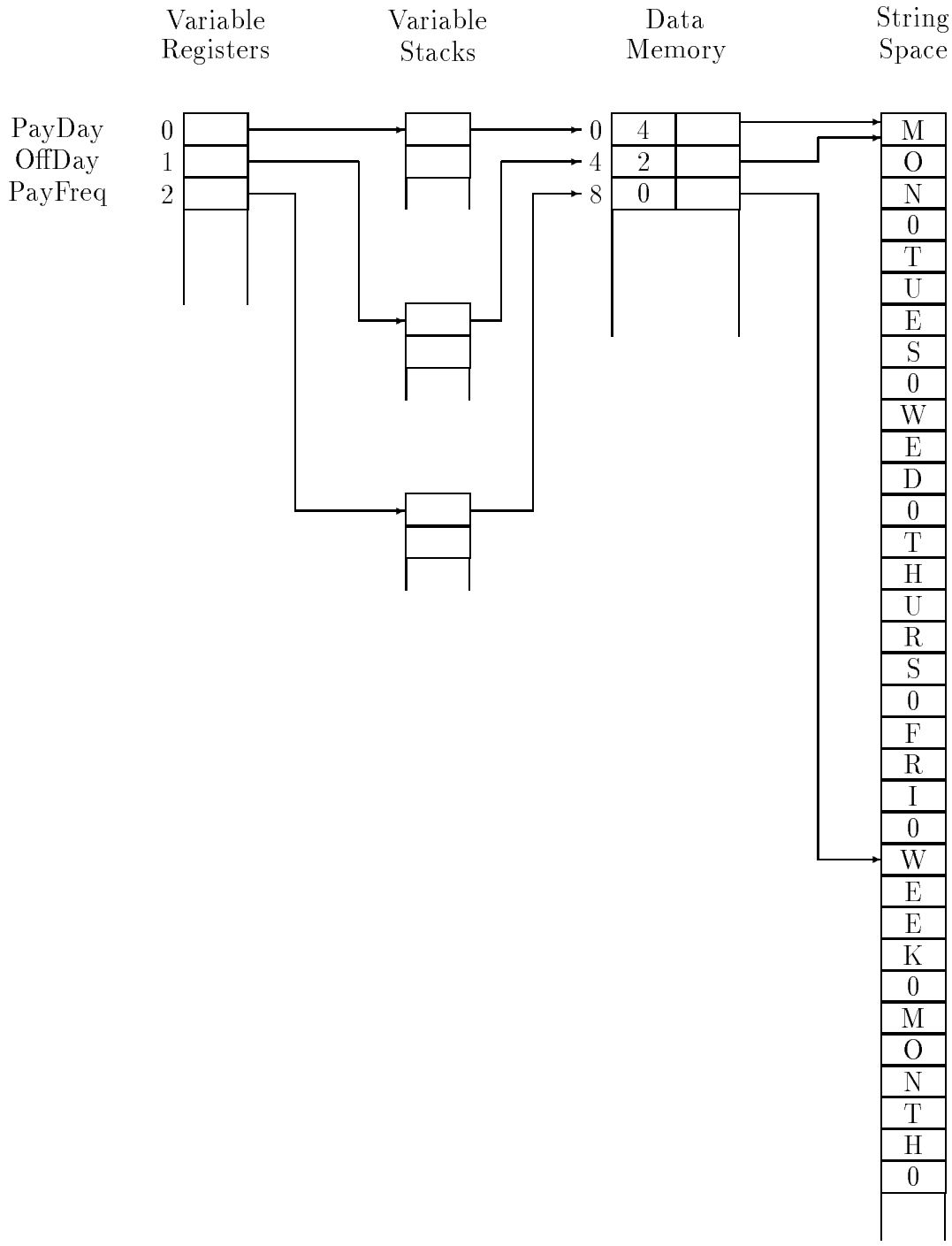

Figure 11: String Space's Relationship with Variable Registers and Data Memory

```
Package Body Payroll2 is

Type Days is (Mon,Tues,Wed,Thurs,Fri);
Type Daylist is Array (Mon..Fri) Of Integer;

DayCode : Daylist:
Day : Days;

Begin
  For Day in Mon .. Fri loop
    DayCode[Day] := 0;
  End Loop;
End Payroll2;
```

Figure 12: Source Code for Program Payroll2

The animator also accesses the string space when displaying enumerated type array indices. In this case, the animator retrieves the values of the enumerated type indices through information stored in the Static Scope Table. In this example, the Static Scope Table entry for the variable DayCode in figure 12 contains the following information:

- Identifier Name: DayCode
- Upper array bound: 19
- Lower array bound: 0
- Type: INTEGER
- Index type: ENUMINT
- Variable Reg: 0

Type INTEGER and index type ENUMINT mean that the variable DayCode is an array with integer elements and an enumerated index type. This indicates to the animator that the array bounds are indices into the string space rather than absolute numbers.

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 Ada/CS compiler was to produce a usable compiler, the current version of the compiler treats all E-code instructions as critical. For example, the animation unit

```
N := K + I*J;
```

in figure 2 corresponds to the packet of E-code instructions numbered 67 through 73 in figure 4. All of these instructions are marked critical via the "c" operand. Only instruction number 73 is actually critical, however, as only it results in critical information being destroyed. That is, the old value of `N` is being destroyed by popping a new value into it in instruction 73; for reverse execution, this old value of `N` must be saved. Thus, the packet of E-code instructions corresponding to this animation unit could be generated as shown in figure 13, where the operand "n" indicates a non-critical instruction.

```
67  push n, I, V8
68  push n, I, V6
69  push n, I, V7
70  mult c, I
71  add  c, I
72  cast c, I, R
73  pop  c, R, V9
```

Figure 13: E-code Instructions Translating `N := K + I*J`

CHAPTER 3

INTRODUCTION TO Ada/CS

Ada/CS is a language originally described in [Fischer 88] as a modified Ada subset used for compiler construction. The text's code was written in Ada/CS to take advantage of Ada's high readability. In the text's next edition, [Fischer 91], the sample code was C, but the compiled language in the text was Ada/CS.

The E-Machine Ada/CS compiler is a one-pass compiler using a recursive descent LL(1) parser. The compiler itself is written in C++ and developed successively on a Dos IBM PC with Borland's Turbo C++ 3.0, a Silicon Graphics workstation with GNU C++, and lastly, a PC running the FreeBSD Unix system, also with GNU C++. During the course of this compiler's development, a Unix Motif animator was also being developed by Craig Pratt, which was used for actual generated code testing. This animator will be discussed in an upcoming thesis.

Differences between Ada/CS and Ada

Ada is such a vast and complicated language that describing all the Ada features not included in Ada/CS would be a monumental task. There are a few of Ada's advanced features that Ada/CS does support, such as packages, that it defines in

different ways. For the purposes of this thesis, the Ada 83 standard is assumed as defined in [USDOD 93], although certain restrictions of Ada 83 that Ada/CS does not share have been loosened with the new Ada 9X draft standard.

Ada/CS Types

Ada/CS types are more like Pascal than Ada. Ada/CS has simple integer and float built-in types and doesn't support the extensive set of Ada types. Ada/CS supports arrays (including unconstrained arrays and array subtypes), records (including variant records), access types (i.e., pointers), and enumerated types and subtypes. Ada/CS does not support derived types or type renaming. Enumerated name overloading is not supported.

Ada/CS Packages

There are a few extensions to Ada used by Ada/CS, the greatest of which is in the package mechanism. Ada/CS packages are like Ada packages with a few additional features. While Ada packages require the package specification and package body to be in separate files and further require only one specification or body per file, both Ada 9X and Ada/CS allow specification and body within one file and even multiple packages within a file.

Ada packages may be brought into scope by the *with* and *use* statements. The *with* brings the package name into scope, as shown in figure 14.

```
with text_io;
...
begin
  text_io.put_line( "hello, world" );
end;
```

Figure 14: The Ada *with* Statement

The *use* statement makes using the package name redundant, as shown in figure 15.

```
with text_io;
use text_io;
...
begin
  put_line( "hello, world" );
end;
```

Figure 15: The Ada *use* Statement

Ada/CS only has the *use* statement, which performs both functions of Ada's *with* and *use*. Using the package name as part of a name expression, as in figure 14, is allowed but not necessary.

Ada/CS Exceptions

Ada supports five predefined exceptions:

- CONSTRAINT_ERROR: Raised when array bounds or numeric ranges are violated, or an invalid pointer is used.
- NUMERIC_ERROR: Raised when a numeric expression cannot deliver a correct value due to various conditions.
- PROGRAM_ERROR: Raised in a number of cases if the execution of the program is in error (attempt to call an unelaborated subprogram, the end of a function is reached with no *return* statement, etc.)

- TASKING_ERROR: Raised when exceptions arise during intertask communications.
- STORAGE_ERROR: Raised when dynamic storage available to the program is exhausted.

These are predefined exceptions, meaning they are always part of every Ada program. Other exceptions are defined within Ada packages, such as package Sequential_IO's STATUS_ERROR and NAME_ERROR, which are raised during I/O errors.

Ada/CS supports three of the above exceptions and three others. Of the standard five, TASKING_ERROR is not supported (and would have no meaning in any case since Ada/CS doesn't support tasking). PROGRAM_ERROR also is not supported. The three new exceptions are:

- TIME_LIMIT. Raised if an execution time limit is reached.
- EOF_ERROR. Raised by an attempt to read past end of file.
- INVALID_INPUT. Raised by an attempt to read an invalid input item.

Changes to Ada/CS for this Project

Certain parts of the Ada/CS language as described in [Fischer 91] are breaks with Ada and as such are contrary to one of the goals of the E-Machine project: to allow, with minimal modification, any simple Ada source to be compiled. The differences in some areas are so great that extensions to Ada/CS are necessary to make it closer to Ada.

- planned implementation of with/use as in Ada. The lack of the *with* statement is an unnecessary impediment.
- planned addition of character type and implementation of the string type as a standard array of characters.

- no multiple packages within a file. In the currently implemented Ada/CS grammar, multiple packages within a file are not allowed simply as a convenience.

Ada/CS has no predefined I/O; therefore, I/O will be implemented as close to the Ada standard as possible (without using generic packages).

Wherever there was an ambiguity or confusion in the definition of Ada/CS, the definition of Ada was used.

Ada/CS Exceptions

As discussed previously, there are six predefined exceptions in Ada/CS. Of those six, only CONSTRAINT_ERROR has been partially implemented. EOF_ERROR and INVALID_INPUT would more properly be defined in the IO packages. TIME_LIMIT is not supported. NUMERIC_ERROR is only partially supported due to limitations with the E-Machine which intercepts numeric problems such as divide by zero and has no way for a program to assume control in the event of such an error.

STORAGE_ERROR is supported in the case of dynamic memory, but other cases, such as recursion consuming all available E-Machine variable memory, are handled by the E-Machine. As with numeric errors, there is no way for a program to assume control in such cases.

Forcing the Ada/CS program to clean up the E-Machine run-time environment would be a monumental task and unnecessary for the scope of this project. Ada is a language designed for reliability and needs to be able to recover from such run-time disasters. On the other hand, the E-Machine is designed for education. If a catastrophic error occurs, all it must do is inform the user about the error then cleanly die.

Currently Unimplemented Ada/CS Features

The following features of Ada/CS have not been implemented in this compiler.

- separate compilation (packages), including:
  - private package members;
  - a "bind" program.
- input/output through standard packages text_io, integer_io, and float_io;
- arrays;
- exceptions;
- default/named subroutine arguments;
- function return type overloading. Currently, overloading is allowed for subroutines that differ by type and number of parameters only. Function return type overloading is nontrivial in a one-pass compiler.
- case statement;
- local blocks;
- unnamed types. Variable declarations such as "vector : array(1..10)" are not yet supported
- access types and dynamic memory.

Overview of the Ada/CS Compiler

As this compiler was written in C++, it makes sense to discuss its overall structure in terms of the C++ classes used. There are currently twenty-three classes within the compiler: three for symbol table management (described in Chapter 5), ten for the type management (described in Chapter 6), eight for code generation

(described in Chapter 7), and two miscellaneous classes: id list management and an error message builder.

The parser and scanner were created using a tool called the Purdue Compiler Construction Tool Set (PCCTS), which is described in detail in Chapter 4. The parser and scanner are not classes themselves, being generated C code. However, a new version (1.20) of PCCTS does create C++ parser/scanner classes. This project was done using version 1.10 of PCCTS.

The symbol table is managed by three classes:

- class NameTable which manages all names in each scope;
- class TypeTable which manages the Ada/CS and user types;
- class SymbolTable which manages and provides a program interface to the first two.

Ada/CS type management is done by a set of classes, one for each intrinsic type in the language. When the user defines a record, for example, a RecordType class is instantiated and all information about that record type stored within it. If the parser needs information concerning that record type, it queries the type class. Each type in the system is represented by an instantiation of one of the following classes:

- IntegerType which manages integer types and subtypes;
- FloatType which manages float types (float types cannot have subtypes so there is only one instantiation of the FloatType class);
- ArrayType which manages array types and subtypes;
- EnumeratedType which manages enumerated types and subtypes (boolean is implemented as an enumerated type);
- RecordType which manages records;
- AccessType which manages access (pointer) types;
- ProcedureType and FunctionType, which manages procedures and functions, are not truly types, but are part of the type system.

Code generation functions are done by the seven classes named for the E-code sections they create. These classes are:

- LabelSection;
- SourceSection;
- VariableSection;
- StatScopeSection;
- PacketSection;
- CodeSection.

Control of compilation is centered in the C routines of the parser which controls the various C++ classes throughout the compiler.

Error Detection and Recovery

Like the miniPascal compiler, there are no provisions for error recovery in the Ada/CS compiler. When the compiler detects an error, a descriptive message is displayed, and the compiler is halted. However, unlike Yacc, the PCCTS parser generates very descriptive error messages and has excellent error recovery. There are provisions in the symbol table for error recovery, but no use is yet made of them.

Optimization

Originally, it was felt that there was no real need for E-Code optimization. However, new developments in the E-Machine, such as an input/output system, generate tremendous amounts of code. While Ada/CS exceptions are not fully implemented, compiling exceptions will produce large code files due to the need to generate E-code to perform exception tests. Such executable files will have such large and complex packets that their run-time speed will suffer. An intelligent optimizer would reduce that burden.

CHAPTER 4

PARSING Ada/CS

Ada/CS was developed using a fairly new tool called the Purdue Compiler Construction Tool Set (PCCTS), which was developed by Terrence Parr and Gary Cohen [Parr 93]. PCCTS generates an integrated scanner and parser from an Extended BNF specification describing the Ada/CS tokens and grammar. Unlike other contemporary bottom up compiler tools, such as yacc [Mason 90], the PCCTS parser is an LL(k) recursive descent parser, which allows increased flexibility in the development of Ada/CS.

Unlike yacc, PCCTS isn't yet a standard tool under an operating system, but is freely available with no restrictions. It is written in ANSI C and has been used under numerous platforms, such as BSD and SysV Unix, Dos, and OS/2. The generated code is compilable as C++ (Ada/CS was written entirely in C++).

Since PCCTS is a fairly new tool and doesn't yet have the assumed familiarity of yacc, the next few sections will describe in part using PCCTS to scan and parse Ada/CS. A few items of note: PCCTS doesn't use two specification files—one for scanner and one for 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 is recursive descent, for a large grammar

(such as for Ada/CS), the resulting code will be large, much larger than an LR parser generated by yacc, for example.

The PCCTS Scanner

Unlike lex, a popular scanner tool, the PCCTS scanner specification is an integrated part of the parser specification. Further unlike lex, if a case insensitive scanner is needed, PCCTS will do the work, which reduces the size of the scanner specification.

To describe a lexical element, C preprocessor-like definitions are used, as shown in figure 16.

```
#token Package   "package"
#token Procedure "procedure"
```

Figure 16: Example PCCTS Reserved Word Scanner Specification

Both Packet and Procedure will act as terminals within the parser's grammar. More complex lexical elements are described using regular expressions, such as the definition for Ada's integer numbers described in figure 17. (Ada integer constants are just like Pascal and C integers except that, for readability, underscores may be placed between the digits; no leading or trailing underscores and no adjacent underscores are allowed.)

```
#token Uint            "([0-9]{_})* [0-9]"
```

Figure 17: PCCTS Scanner Specification for the Ada/CS Integer

```
rule_1 : rule_2 Terminal ;
```

Figure 18: A General PCCTS parser rule

Lexclasses

An interesting feature of the PCCTS scanner are the lexclasses. With separate lexclasses, under different circumstances the scanner will have entirely different behavior, as if there were multiple scanners that trade control.

Two lexclasses are used in Ada/CS—one for scanning all parts of a program except for comments and another for scanning comments. Ada uses line comments—everything from a $--$ token to the end of line is ignored. When the language lexclass finds a start of comment token, it switches control to the comment lexclass, which ignores all input characters until it encounters an end of line token, whereupon it switches back to the scanner lexclass. In this manner, instead of a parser handling the comments, they may be eliminated directly in the scanner.

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 18. By convention, terminals are capitalized, nonterminals are all lowercase. Tokens can be directly inserted into the parser specification without having to create a separate named scanner token.

As an example, the rule specifications to recognize a variable declaration and an identifier list are shown in figure 19.

In figure 19, characters in double quotes are unnamed tokens. Being able to place tokens directly into the parser specification makes the specification more readable.

```
variable_dec : id_list "\:"
               Id { { Constant } "\:\=" expression } "\;" ;

id_list : Id ( "\," Id )* ;
```

Figure 19: PCCTS Rule for Ada/CS Variable Declaration

```
id_list : << printf( "just entered rule id_list\n" ); >>

          Id

          << printf( "found an id!\n" ); >>

             ( "," Id

               << printf( "found another id!\n" ); >>
             )*

          << printf( "leaving rule id_list\n" ); >>
        ;
```

Figure 20: PCCTS Grammar Rule with Actions

Id and Constant are named tokens, and expression is a nonterminal that is described in an excruciating set of rules later in the specification.

Grammar Actions and Token Attributes

What use would a parser be if there was no way to perform a rule action? Just as yacc uses {} to denote actions, PCCTS uses <<>>. Some simple actions are shown in figure 20, which involves generating actions during the parse of id_list of figure 19.

```
variable_dec : << IdList var_list;
               >>
               id_list[var_list] "\:" Id
                 << symbol_table.add_variable( var_list, $3.text );
                 >>
               { { Constant } "\:\=" expression } "\;"
             ;

id_list[ IdList& new_list ] : Id << new_list.add($1.text); >>
                                 >>
                               ( "\," Id
                                 << new_list.add($2.text); >>
                               )*
                            ;
```

Figure 21: PCCTS Grammar Rule with Parameters

Actions may appear anywhere within a rule (even before the first rule element). An action may even appear outside a rule, where it is inserted directly into the generated source code. In that case, the action's code would be at the mercy of the compiler.

Semantic Stack and Rule Parameters

Communication between rules is performed in two ways—a traditional semantic stack and a new method: rule parameters. Because PCCTS generates recursive descent compilers, each rule is a C subroutine, and PCCTS allows the user to add C/C++ parameters to the rules, as shown in figure 21. Processing of the variable_dec rule in turn requires processing of the id_list rule, which has a C++ pass-by-reference parameter of var_list, which is an instantiation of a class that manages a list of Ids.

PCCTS also maintains a semantic stack, which behaves as yacc's does. However, because of the EBNF rules of PCCTS, there are a few more complicated stack access

```
simple_expression : { ( "\-" | "\+" ) } << {} expression is $1 >>
                    term   << this is $2 >>
                      (
                        ( "\+" | "\-" | "\&" ) << this is $1 or $3.1 >>
                        term  << this is $2 or $3.2 >>
                      )*   << inside the ()'s is $3 >>
```

Figure 22: PCCTS Semantic Stack

expressions. Consider, for example, the grammar fragment shown in figure 22. The unary operators within the {} expression are represented by $1; the first "term" nonterminal by $2, and the ()* expression by $3. However, inside the ()* expression, $1 would be the binary additive operator expression. Suppose access to the first term expression is desired inside that expression. In that case, $1.2 would be the semantic attribute of that nonterminal; the $1 is the first semantic stack block (the very outermost), and the .2 represents the second element. The grammar expressions thus act like scopes, with the outermost visible scope being $1, the next inner visible scope being $2, etc. Continuing this idea further, if the grammar is complex enough, expressions such as $1.3.5 and $1.2.1.3 are possible.

The elements of the semantic stack are a user defined structure called Attrib. The Ada/CS Attrib structure contains information about the element name, type position, variable, whether or not it is a constant, and (most useful of all), the start and end line and column of the token, which is used to create packets.

For example, the variable_dec rule in figure 23 has an action that creates a new packet from the attributes of its rules. $1.start_line and $1.start_col refer to the starting line and column of the id_list rule and $5.end_line, $5.end_col refer to the ending line and column of the ";" token.

```
variable_dec :
           << IdList var_list; >>
        id_list[var_list] "\:" Id
           << symbol_table.add_variable( var_list, $3.text );>>
        { { Constant } "\:\=" expression } "\;"
           << PacketS.new_packet( startAddress, endAddress,
                     $1.start_line, $5.end_line,
                     $1.start_col,  $5.end_col );
           >>
      ;

id_list[ IdList& new_list ] :
         Id << new_list.add($1.text); >>
         ( "\," Id
           << new_list.add($2.text); >>
         )*
           << $0.start_line = $1.start_line;
              $0.start_col  = $1.start_col;
              $0.end_line   = $2.end_line;
              $0.end_col    = $2.end_col;
           >>
      ;
```

Figure 23: PCCTS Rule for Variable Declaration with Semantic Stack Actions

The id_list rule also has an action that demonstrates setting the semantic stack structure members. $0, like yacc, is the semantic structure that will be returned on the stack. Nonterminals must have their $0 members set by terminals' semantic structures.

CHAPTER 5

THE Ada/CS SYMBOL TABLE

Ada/CS, as a subset of Ada, shares some of its parent's complex symbol structures—subtypes, subroutine and operator overloading, unconstrained arrays, and packages, to name a few. The symbol table of Ada/CS thus had to be flexible enough to allow for these structures and yet strong enough to support Ada's firm type checking.

All the structures of the the symbol table were built around the desire to simplify the Ada/CS separate compilation package mechanism. As such, almost all the structures used are arrays whose size can grow through the C library routine realloc(). The overall structure of the symbol table contains three elements—the name table, the type table, and the symbol table itself, which contains and manages the first two.

The name table contains all the active names currently in a particular scope (variables, types, subroutines, etc.). The names are held as a chained hash table. The symbol table maintains a linked list of name tables, one table for each scope. Since the symbol table needs so much control over how scopes are searched, it was decided that instead of making one name table that would manage all scopes, the symbol table would manage a scope list of name tables. The type table contains information describing all active types. There is only one type table during the run of

the compiler. Type information is stored in a linear array so that type comparisons are simply a check to see if type table positions match.

The Name Table

At its heart, the name table is a chained hash table of nameEntry structures (shown in figure 24), one node per name.

```
struct nameEntry {
  id_name name;
  int entryType;
  int typePointer;
  int varreg;
  int ssChild;
  int ssPushd;
  int position;
  int passMode;
  boolean constant;

  struct nameEntry *next;
};
```

Figure 24: The nameEntry Structure

The elements of the nameEntry are as follows:

- name: An array of characters storing the name of the symbol.
- entryType: The "type" of the name. The name may be one of
  - VARIABLE_NAME: The name of a variable.
  - TYPE_NAME: The name of a type.
  - ENUMERATED_NAME: The name of an enumerated type's element.
  - SUBPROG_NAME: A procedure or function name.

  - PACKAGE_NAME: A package name brought into scope by the "with" statement.

- typePointer: Index into the type table array of the type of this name.
- varreg: The variable register that holds the data for that name. This field isn't used for package or type names. For subroutines, the field holds the label register of the subroutine.
- ssChild: Used only with subroutines as part of the building of the static scope table entry for the subroutine. The static scope table contains an entry for the subroutine name whose child is set to this field so that the information on the internals of the subroutine may be found.
- ssPushd: Used only with subroutines. When a subroutine is entered, it must inform the E-Machine of the starting position of the static scope table block that contains the subroutine's variables (which is done through the *pushd* instruction). After the subroutine is completely parsed and its scope is collapsed, the starting position of its static scope table entry is obtained (through a StatScopeSection method—described later) and stored in this field. When the scope containing the subroutine name is collapsed, as part of saving the subroutine the "pushd" for the subroutine is modified to push this field.
- position: For animation purposes, variables must be added to the static scope table in the order in which they were declared. When a scope is collapsed and before the name table information is saved to the static scope table, all the nameEntry structures are stored in a linear array and sorted by this field.
- passMode: This field is only used for subroutine parameter name entries indicating the mode of the parameter (in only, out only, or in out). Certain rules apply to the use of Ada subroutine parameters. In-only parameters may not appear on the left hand side of an expression; out-only parameters may not appear on the right. Further, special code generation considerations must be taken for each of the three Ada parameter modes.
- constant: Ada constants may not appear on the left hand side of an expression nor may they be passed in-out or out-only to a subroutine. This boolean field indicates whether or not the variable name is a constant.
- next: If a hash collision has occurred during the adding of a new name, chained resolution is used. This is a pointer to the next element in the chain, if any.

Access to the name data is managed by the nameTable class shown in figure 25.

```
class nameTable {
  protected:
    id_name scope_name;

    int num_symbols;
    int name_count;
    nameEntry *table;

    void save_sscope_type( nameEntry save_me );
    void save_sscope_var( nameEntry save_me );
  public:
    int ScopeIndex;

    nameTable( );
    nameTable( int size );
    ~nameTable();

    void add( nameEntry *new_entry, int hash_pos );
    void add( nameEntry *new_entry ) {...}

    nameEntry& find( char *name,
                     int hash_pos, boolean *found );
    nameEntry& find( char *name, boolean *found ) {...}

    int size(void) { return num_symbols; }
    void Print( TypeTable *ttable );
    void pop_scope( void );

    void set_ss_data( AdacsType atype,
                      int child_pos, int pushd_pos );

    void save( FILE *i_file );
};
```

Figure 25: The nameTable Class

The members of the nameTable class are as follows:

- scope_name: Name of this scope.
- num_symbols: Size of the hash table. Unlike other data structures in this compiler, this size is fixed so the hash function has a constant modulus for each scope.
- name_count: Number of names in the current scope.
- table: Array of nameEntry structures as a chained hash table.
- ScopeIndex: A special consideration for E-Machine animation. During the course of animation, the animator needs to know how deep into the current scope table to display variables (since all variables are not declared at once).
- save_sscope_type, save_sscope_var: These methods save the name table types and variables to the static scope table and are fully explained in the section on collapsing a name table scope.
- nameTable: There are two constructors, one of which (the parameterless one) is not used and is only included because, as the default constructor, it is required by some compilers. The constructor is responsible for creating its hash table.
- ~nameTable: The nameTable destructor. This is not used during the run of the compiler since no nameTable is ever deleted. After a new nameTable is created (by a stack overflow) and a scope is collapsed, the nameTable persists and is reused. The destructor is called upon program termination.
- add: There are two methods for adding names to the nameTable. This routine is responsible for maintaining the hash chains (if necessary).
- find: Finds a method in the nameTable, returning a reference to a nameEntry structure. A boolean parameter is set to indicate whether the name is found or not.
- size: Returns the current number of names within the table.
- Print: This is a debugging routine that prints the nameEntry contents of the name table.
- pop_scope: Because nameTables are not deleted when a scope is collapsed, they must be cleared and their allocated chain memory deleted. When the symbol table pops a scope, this method is called to perform the clean up.

- set_ss_data: This is an unusual method created to help build the static scope table. The method is called by the parser after a subroutine call has been completely parsed and its scope collapsed to set the ssPushd and ssChild nameEntry members.
- save: For future compiler development. This method will save the contents of the name table to a package file.

Adding a New Name

Adding a name to the table is done through one method, even though there are two "add" methods. The second add method, *add(nameEntry *new_entry)* is a C++ class inline call to the first, *add(nameEntry *new_entry, int hash_pos)* with a call to the hashing routine to get the hash_pos parameter.

Two methods are used to avoid hashing a name as much as possible. During the add process, the name is first hashed and then looked up in the table. If the search is successful, a duplicate name exits and an error is reported. If the search is not successful, the add routine is called with the hash value, thus avoiding rehashing the name. The inline routine is used as a convenience (it doesn't require the explicit calling of the hash function) and is used on the rare occasion when an added name is guaranteed not to encounter a collision. One such occasion is during the parse of a subroutine; the parameter names have already been checked for uniqueness and must be added to the new scope's symbol table.

The add method checks the table position at the hash location, and if the slot is currently filled, adds the new name at the end of the chain. At this time, the position field of the nameEntry structure is filled with the name_count counter.

Finding a Name

Like the methods for adding a name, there are two methods for finding names. The second of these two functions is simply an inline call to the first, with a call to the

hash routine to get the hash value of the name.

Calling the first find method is only done when adding a new symbol to the table. First the name is hashed and that value is used both with the find and the subsequent add. If the name is found, the "found" parameter is set to TRUE and the nameEntry data is returned; otherwise, "found" is set to FALSE and the return value is undefined.

Creating a New Scope

The name table is essentially a scope-ignorant data structure—it knows nothing of other scopes. When a new scope is being created, the name of the new scope is set through the call to the method *new_scope(char *name)*; no further actions are taken.

Popping a Scope

Unlike creating a new scope, the process of collapsing a scope has a large number of responsibilities. The routine to collapse a name table scope is *pop_scope(void)*. In spite of its name, the pop_scope() routine doesn't pop anything—the name was used to maintain name similarity across the symbol management classes (both the symbol table and type table classes also use pop_scope() to collapse scopes and both actually pop something).

When a variable is declared, an *inst* instruction is generated to instantiate it. Inside pop_scope(), an *uninst* instruction is generated to destroy the variable. As the hash table is traversed to generate the *uninst* instructions, each nameEntry structure is saved in a linear array for later sorting. Also, the hash table is traversed, chains are deleted, and each slot of the hash table is set to empty. The table itself is never actually deleted. If the scope again increases, the hash table will be empty and ready to be filled.

After the traversal of the hash table, the nameEntry array is sorted by the order in which the names were added to the table. The animator must display variable values only after the declaration has been animated. By using an offset from the start of the current static scope block, the animator will only display those variables that have already been instantiated. So, the compiler is responsible to put the variables into the static scope table in the order in which they were declared; this ordering is saved in the "position" field of the nameEntry structure.

Next, the variable and type elements of the name table are saved to the static scope table. The types are added first through the method *save_sscope_type(nameEntry save_me)*. The types are saved first because certain static scope members (records, arrays, and subroutines) will need to know the location of their types in the static scope table. The variables are added next through the method *save_sscope_var(nameEntry save_me)*. The responsibility of creating the static scope entries rests with the StatScopeSection code generation class (described in Chapter 7). The save_sscope_type() and save_sscope_var() routines call methods of the StatScopeSection class.

The Type Table

The type table is a stack of Ada/CS type descriptor structures called AdacsType. Like the name table, the type table is hidden from the rest of the program and is accessed through methods in the symbol table. Unlike the name table, there is only one type table per compilation.

Ada/CS is a strongly typed language, so much attention must be paid towards good, fast, type checks. Since each type currently in scope has a unique position in the type table, it was natural to use that position as the basis for the type comparison.

The type table stack manages the C++ classes that describe all the Ada/CS types—integer, float, record, array, and so forth. Each type has its own particular class. The type table actually stacks AdacsType structures, as shown in figure 26.

```
struct AdacsType {
  id_name type_name;
  int typeOf;
  int flags;
  void *element;
};
```

Figure 26: The AdacsType Structure

The structure members of AdacsType are as follows:

- type_name: The name of the type.
- typeOf: In order to allow ALL possible type Classes to be held in the type stack even though they are of different types, the AdacsType uses a void pointer, which of course introduces the problem of how to tell what the pointer actually points to.
  - ADACS_DUMMY: Originally planned as a type for use with error recovery. If a variable was declared of a nonexistent type, the type would be set to ADACS_DUMMY and compilation would continue. As yet unused.
  - ADACS_INTEGER: Built in integer type or subtype. Type Class is AIntegerType.
  - ADACS_FLOAT: Built in float type or subtype. Type Class is AFloatType.
  - ADACS_ENUMERATED: Enumerated type or subtype. Type class is EnumeratedType.
  - ADACS_BOOLEAN: Built in boolean type or subtype. Type class is implemented as an instantiation of EnumeratedType.
  - ADACS_ARRAY: Array type or subtype. Type class is ArrayType.
  - ADACS_ACCESS: Pointer type whose type class is AccessType.

  - ADACS_PROCEDURE, ADACS_FUNCTION: class is ProcedureType and FunctionType. Technically, procedures and functions are not types and in Ada/CS cannot be used as such (unlike Pascal), their information (type and number of parameters, location, etc.) is stored in the type class.
  - ADACS_RECORD: Record type whose type class is RecordType.
  - ADACS_INCOMPLETE: Incomplete type declaration (also known as forward declarations). The type is added to the table as an incomplete type and the details filled in later during the compile. No variables may be declared of this type.
  - ADACS_PRIVATE: Both Ada and Ada/CS support private package members, type declarations whose name is known but whose structure is hidden. Since packages are as yet unimplemented, this is unused.
  - ADACS_PACKAGE: When a package is brought into scope with the "with" statement, information about the package (file system location, etc.) would have been stored within this type class. Packages are currently unimplemented.

- flags: There are a few certain special cases in the Ada/CS type system that require slightly different symbol table behavior. The type flags indicate when such a situation is in effect. The flags are as follows:

  - FORWARD_DEC: Indicates this type entry is a forward declaration and the type descriptor has no real type information. No variables may be declared of this type.
  - PRIVATE_DEC: This flag is for future development of Ada/CS packages. Private types are types whose internal description are not known outside the package in which the type is declared. Variables can be declared of private types but, since the type's internal structure is not known, no access to part of the type can be performed.

- element. A void pointer which points to the type class for the particular type indicated by the typeOf member.

The type table is essentially a container class. It does no actual work upon the AdacsType structures it contains. The type management responsibility is part of the Symbol Table class.

Adding a New Type

Adding a new type is very easy and is done through the method *add(AdacsType &new_type)*. When a type declaration is found, the parser builds an AdacsType structure describing the new type, including creating a class instantiation for that type (RecordType for records, ArrayType for arrays, and so forth). The type classes are described in Chapter 7. The structure is passed to the symbol table routine, which performs some checking described later in this chapter, then passes it to the type table. The type structure is put onto the stack (which is enlarged if necessary) and the top of stack index is increased.

Creating a New Scope

The method used to track scopes in the type table is completely different than the separate data structure per scope method used with the type table. The TypeTable class contains a stack of integer indices into the table. When a new scope is created, the type stack top is pushed onto this stack. When the scope is popped, the type table elements from the top of the type stack are popped down to this marker.

Popping a Scope

As described in the previous section, popping a scope is simply a matter of running from the top of the type stack down to the highest saved marker. One of the functions performed during this method is the destruction of the type classes. The type stack is a structure that contains pointers to the type classes; when a type is no longer needed, the memory must be freed. On occasion, the type classes' destructors have certain necessary functionality. For example, the RecordType class must set its variables' static scope table structure child indices to its own static scope table entry. (The animator needs to know the type description of record variables to

display them correctly. The child index of a record variable's static scope table entry is a scope block describing the record type. The RecordType class knows its scope block position and that of its variables.)

The Symbol Table

The symbol table class is the most complex class in the Ada/CS compiler and is shown in figure 27.

The members of the symbol table class are as follows:

- curr_scope: Top of the name table stack. All operations upon the symbol table start at this name table array position and work downward toward zero.
- num_scopes: Maximum size of the array of name tables.
- scope: Array of pointers to nameTable classes, one element per scope, created during SymbolTable class construction.
- types: Pointer to the type table class instantiation, created during SymbolTable class construction.
- SymbolTable: Symbol table constructor which allocates memory for the name table stack and creates a type table instantiation. Because the symbol table remains for the entirety of the compiler's run, no destructor is necessary.
- ScopeIndex: Inline method that returns the current scope's name table's scope index and used by the parser as it builds packets.
- add_object, add_objects: Adds variables to the symbol table. These are fully discussed later in this section.
- add_nameless_type: Adds a nameless type to the type table. If, for example, the parser encounters "a : array(1..10) of integer", the array definition will be added to the type table, even though the definition can be used in no other place in the source.
- add_type: Add as type to the symbol table. First, the routine checks whether the name is already defined in the current scope. If not, it adds the name to the current scope's name table, then adds the AdacsType structure describing the type to the type table.

```
class SymbolTable {
  private:
    int curr_scope, num_scopes;
    nameTable **scope;
    TypeTable *types;
  public:
    SymbolTable( );
    int ScopeIndex(void);
    void add_objects( IdList *name_list, char *type_name, boolean is_const,
                      boolean expr, Attrib expr_attrib );
    void add_objects( IdList *name_list, int ttable_pos );
    void add_object( nameEntry *new_name, int inc_by );

    int add_type( AdacsType& new_type );
    void replace_type( AdacsType new_type, int type_pos ) {...}
    int add_nameless_type( AdacsType& new_type );

    int add_procedure( AdacsType& new_type );
    nameEntry& find_procedure( char *name, int num_params, Attrib *params );
    nameEntry& find_function( char *name, int num_params, Attrib *params,
                              int return_type );

    boolean match_procedure( AdacsType find_me, AdacsType& atype );
    boolean match_procedure( AdacsType find_me, int num_params, Attrib *params );
    boolean match_function( AdacsType find_me, AdacsType& atype );
    boolean match_function( AdacsType find_me, int num_params,
                            Attrib *params, int return_type );

    void new_scope( char *scope_name );
    void pop_scope( void );

    nameEntry& find_name( char *name, boolean *found );
    AdacsType& get_type( int pos ) { return types->get(pos); }

    void set_ss_data( AdacsType atype, int child_pos, int pushd_pos ) {...}

    void save( char *pack_name, int source_type, FILE *i_file );

    void Print( void );
    void print_ttable( void ) { types->Print(); }
};
```

Figure 27: The Symbol Table Class

- replace_type: This method replaces a type table element with a new descriptor. This is necessary for filling out the details of forward types when they are fully described.
- add_procedure: Adds a subroutine to the symbol table. Since the functionality of adding a procedure and adding a function is so close, it made sense to consolidate both into one method. The full details, including overload resolution, are discussed further in this section.
- find_procedure, find_function: Attempt to resolve a subroutine name and parameter list into a subroutine reference. These methods handle Ada/CS's overloading by calling the match_procedure and match_function methods.
- match_procedure, match_function: Compare two symbol table descriptions of a procedure or function and return a boolean value denoting whether the two represent an overload match.
- new_scope: Creates a new scope (discussed in detail later in this section).
- pop_scope: Collapses a scope (also discussed in further detail later in this section).
- find_name: Finds a name in the symbol table (discussed later in this section).
- get_type: Variables reference their type by an integer index into the type table. If it becomes necessary to perform some function based upon detailed information about the type, get_type is an inline call to the type table that returns the type descriptor at that table position.
- set_ss_data: This method is simply an inline call to the current scope's name table's set_ss_data method. See the section on the Name Table for further detail.
- save: Method to save the current symbol table to a package file. This method has not been completely implemented.
- Print: Debugging routine that prints the contents of the symbol table.
- print_ttable: Debugging routine that asks the type table to print itself.

Adding a Variable

There are a number of methods for adding variables to the symbol table. The symbol table add routine is responsible for confirming name uniqueness within the current

scope, confirming that a variable's type exists, adding a variable to the name table, and generating code to instantiate and, if necessary, initialize the variable.

There are several forms of Ada/CS variable declaration, the simplest of which are the standard, Pascal-like declarations, such as "a, b, c : integer". The parser adds the variable names to an instantiation of the IdentifierList class and passes that list and the type name to the symbol table variable add routine add_objects.

Initialization of variables at declaration time is a feature of Ada/CS that allows code such as "a, b, c : integer := 0". The symbol table add_objects is passed a parser Attrib structure that describes the expression on the right hand side of the assignment. The add routine confirms that the expression result and the variable declaration are of compatible types and generates initialization code as well as the variable instantiation code, so that when the corresponding declaration packet is animated, a, b, and c will not be undefined, but zero.

Ada/CS allows variables to be declared of an unnamed type description such as "a, b, c : array(1..10) of integer". The array type has no formal type declaration so the compiler has a dual responsibility. It first builds an identifier list of a, b, and c, and then builds a type descriptor describing the array. A different add method, add_nameless_type, is used to add the type to the symbol table. This method accepts an id_list and a type descriptor. The type descriptor is added to the type table and its type table position is saved to the name table structures of its variables. Because such a type descriptor has no name in the name table, it can not be directly found by any means other than through its variables's type position.

The final variable adding method is a result of the development of the for loop. The loop variable need not be declared, and in such an event, it will be implicitly declared as the type of the loop's range. As the parser goes through a for loop, it builds a nameEntry structure for the loop variable. The add_object method will accept that structure and add it to the name table.

Adding a Type

Adding a type to the symbol table is a simple matter. The parser builds an AdacsType structure and an instantiation of a type class fully describing the new type. The symbol table type add routine checks whether the name is unique within the scope and if so, adds the name to the name table and the type to the type table (through the methods of the type table).

Ada/CS supports forward type references. For example, the type declaration shown in figure 28 sets up a record and record pointer suitable for a linked list.

```
0 type data_rec;
1 type rec_ptr is access data_rec;
2 type data_rec is record
3   name : string(10);
4   next_rec : rec_ptr;
5 end record;
```

Figure 28: Example of a Forward Declaration

Line 0 is a forward declaration, which is necessary due to the circular reference of the access type rec_ptr (which points to the record type data_rec) and the record declaration (which contains a variable of type rec_ptr).

Forward declarations are added to the symbol table with the same method as full type declarations. The difference is that the type descriptor for the forward type has its type field set to FORWARD_TYPE. When the record declaration on line 2 is parsed and type data_rec is ready to be added, the symbol table checks to make sure the name is unique and finds the previously added name. A further check is done to find that data_rec is a forward type and already has a descriptor in the type table. A new descriptor cannot be added to the top of the type table since the current type descriptor's table position might already be used as a reference (as

in figure 28 where rec_ptr on line 1 uses the type table position of forward declared data_rec). So, the type descriptor in the table is replaced by a full description of the type with the symbol table method replace_type, which is simply an inline call to the type table's own replace_type method.

Unnamed types are added to the symbol table through the variable declaration methods, as was described earlier.

Creating a New Scope

When the parser detects a new scope (starts parsing a subroutine or a local block), it informs the symbol table. The symbol table increases the current name table scope index and creates a new NameTable class if necessary. The type table is informed of the new scope so it can place the marker enabling it to return to the state it was during the previous scope.

Popping a Scope

The symbol table is informed about the end of a scope by the parser. The type table is informed and the disappearing scope's name table is also informed. The symbol table does not delete the top name table when it pops a scope; instead, the table is saved for possible future use. The name table pop_scope method reinitializes itself to an empty scope.

Subroutine Overloading

Ada/CS supports Ada's subroutine name overloading. When a procedure or function call is discovered, the parser builds a complete description of the call with the name of the subroutine and type and number of parameters. (The return type of a function should also be part of this information, but is currently unimplemented.)

The parser then calls upon the symbol table to return it a reference to the subroutine that matches the description.

The symbol table first searches by name. When it discovers the sought after name, it calls the match_procedure or match_function method to confirm the type and number of parameters. If this check fails, the name search will continue until the end of the outermost scope.

CHAPTER 6

THE Ada/CS TYPE SYSTEM

Like its parent language, Ada/CS has a rich type system, repleat with subtypes, variant records, unconstrained arrays, and enumerated types. Because of the complexity of Ada/CS types, a simple symbol table management routine did not suffice and the type system was instead implemented as its own C++ class hierarchy.

The Ada/CS Type Classes

Every Ada/CS type is represented by an instantiation of a C++ class which describes the type and its basic operations. Whenever a new type or subtype is defined, a new type class is created with information about the type and the class added to the type stack.

Base Class BasicType

BasicType is the base class of the hierarchy; all type classes used descend directly from it. Common functionality among all the types was put into this base type. Data such as type size, subtype ancestors, and methods to access them, are part of *BasicType* since all the types share (at least most of) this basic functionality.

Ada/CS Subtypes

Like Ada, Ada/CS supports subtypes upon ordinal types (integer, enumerated, and array types). Also like Ada, these subtypes might have to be determined at run time. Consider the code fragment in figure 29. The compiler can not assume it knows the size of "x", since "x" could be a value input from the user at some point. Since Ada/CS also shares Ada's strong type checking, the full range of the x_integer subtype would have to be known so that code for a range check could be created to cause evaluation of an expression such as num:=x+1 to generate a CONSTRAINT_ERROR exception.

```
0  procedure something( x : integer ) is
1
2  subtype x_integer is integer range -x..x;
3  num : x_integer;
```

Figure 29: Ada/CS Run Time Subtype

Another interesting problem with Ada/CS subtypes concerns their range expressions. For example, refer to figure 30.

```
0 subtype short is integer range 1..1000;
1 subtype short is range 1..1000;
2
3 subtype shorter is short range 100..200;
4 subtype shorterer is shorter range 150..175;
5 subtype shortest is shorter range 160..170;
6
7 subtype regetni is integer;
```

Figure 30: Ada/CS Subtype Declarations

The first two declarations are equivalent. The compiler must determine the base type of the subtype from the type of the range elements, which in this case is integer. Line 7 is an unranged subtype, which is a simple renaming of a type.

Lines 3, 4, and 5 illustrate a symbol table challenge. Because the range of a subtype must be less than or equal to that of the parent type, run-time range checks must be generated by the compiler. For this reason, subtype information not only needs to contain the range of the subtype, but also the range of its immediate parent (transitivity will assure that the ranges of all children are no greater that the ranges of their ancestors). Another consideration is type compatibility. All subtypes are assignment compatible with their parent types. The compiler must therefore have some way of checking for compatibility beyond the usual method of comparing type table positions.

The BasicType base class has two integer fields, parent and root. Parent is the type table position of a subtype's immediate parent; root is the root of the subtype hierarchy (which may get extremely complex) and is used for compile time type compatibility checks. For example, again using the subtypes in figure 30, the parent and root of subtype "short" would be the type table position of integer; the parent of subtype "shorter" would be the position of subtype "short" and its root would be integer; the parent of "shorterer" would be "shorter" and its root would again be integer.

Even though only integer, enumerated, and array types can be subtyped, all types share the parent/root data elements because they are members of BasicType. A better object oriented method would be to make another descendant of BasicType with the parent/root elements and descend the Integer, Enumerated, and Array classes from that type. This will probably become part of a future version of the Ada/CS compiler.

Array Types

Arrays have been handled earlier in the Pascal compiler for the E-Machine. However, Ada/CS arrays are much more flexible than Pascal arrays and therefore required changes to the E-Machine and special compilation considerations. Pascal arrays are created at compile time since all information concerning the array is known at compile time. Ada/CS array types are similar to Ada arrays in that they can be dynamically sized. For example, suppose we had the code fragment shown in figure 31.

```
0  procedure something( x : integer ) is
1  type vector is array( 1..x ) of integer;
2  v1, v2, v3 : vector;
3
4  for i in 1..x loop
5    v1(i) := i;
6  end loop;
```

Figure 31: Ada/CS Array Declaration

Until Ada/CS, the E-Machine required that all information about arrays be generated at compile time. For arrays such as in figure 31, the Ada/CS compiler has no idea how much space to set aside for v1, v2, and v3 since the variable x will not have a value assigned to it until run time. For this reason, the ArrayType field was added to the static scope table for the E-Machine object code file.. This field can be NONARRAY, DYNAMIC_ARRAY, or STATIC_ARRAY. A variable of type STATIC_ARRAY is the traditional E-Machine array in which the variable size is known at compile time and is listed in the variable section of the E-Machine object code file. Variables of type DYNAMIC_ARRAY are those whose size is unknown and is created at run-time. In this case, the array is translated into an E-Machine

variable that is merely a pointer to a block of memory. The third type, NONARRAY, is used in those special cases where the upper and lower bounds of the array are equal (i.e., for arrays of just one element). E-Machine considerations aside, the compiler must still generate run time array offset calculations for references similar to those on line 5 of figure 31, instead of being able to calculate them at compile time.

Things get even more complicated. Like Ada, Ada/CS generates run-time array bounds checking. If the for loop in figure 31 attempted to access v1(x+1), a CONSTRAINT_ERROR exception would have to be raised. So all the ranges of the array must be known in order that bounds checks be performed.

Unconstrained arrays are another concern. Consider the program fragment in figure 32. Unconstrained arrays have no size until a variable of that type is actually declared. The Ada/CS compiler must take this into account.

```
.  type vector is array( integer range <> ) of integer;
.  . . .
0  procedure something( x : integer ) is
1  v1 : vector(x);
2  v2 : vector(x+10);
3  v3 : vector(x+20);
4
5  for i in 1..x loop
6    v1(i) := i;
7  end loop;
```

Figure 32: Ada/CS Unconstrained Array

As a result of these aspects of Ada/CS, there are now several pieces of array information that must be determined at run time. For offset calculations, the total size of each dimension of an array must be known since it is possible to declare an

array as complex as "array(1..x, -x..x, -2*x..x+1)." The upper and lower range values of each array dimension must also be computed and stored so run time bounds checking can be performed. In the E-Machine, these values are stored in separate integer registers that can be initialized to the proper values at run time. When the parser needs to generate code to allocate space or generate code to do an offset calculation, it can query the type class for the appropriate registers.

An Ada/CS E-Machine array is actually a pointer to a block of memory created with the e-code instruction *alloc*. Ada/CS array entries in the Static Scope table look much like Pascal array entries except that the ArrayType is DYNAMIC_ARRAY, and the array ranges do not represent actual bounds, but registers that contain the bounds.

Having the ArrayType class manage a set of registers for dynamic dimension bounds is fine for regular arrays, but unconstrained arrays present a further problem. With regular arrays, the class instantiation for some array type can know and use the registers that will contain the ranges and array size. Notice, however, the situation presented in figure 32. There are three variables of array type "vector", each of a different size. The "set of registers" technique fails in this case since each variable of that type may be of a different size.

The solution is to allow the ArrayType class to know its variables, something no other type class may do. Whenever an array variable is added to the symbol table, the variable is also added to its particular type class. The responsibility of offset calculations and memory allocation and freeing then falls on the ArrayType class. For example, when the parser encounters something like line 6 of figure 32, it finds the type of v1 and asks that type class to generate an offset calculation of i into that array.

Enumerated Types

Ada/CS enumerated types are exactly like Pascal enumerated types. However they are slightly different than Ada subtypes, which allow name overloading (a name may be used in more than one enumerated type in which case the type name must be used to resolve conflicts). Except for subroutines, Ada/CS allows no name overloading.

The EnumeratedType class receives a list of identifiers and is responsible for adding those identifiers to the StringSection of the object code file and the current scope's name table. It is further responsible to return the StringSection location of any of its members, as would be necessary for the code generation of the Ada/CS successor and predecessor type attributes.

Record Types

Ada/CS records are just like Pascal records and also include provisions for variant records (although these are not yet implemented in the current compiler). When the parser encounters a record declaration, it creates an instantiation of a RecordType class. As the parse continues, each set of variable declarations is bundled up and sent to the type class where it is checked for validity (name uniqueness, existence of the type, etc.) and added to a list. The members of a record type are stored in an array of structures that holds each element's name, its type's type table position, and its offset position in the record. As the static scope table element for the record is built, the offset position is stored in the "offset" field. The animator uses this field to find record members.

There is a special consideration for compiling records for the E-Machine. The static scope table requires that each variable have an associated description of its type. Integers, floats, and booleans are all base types and need no further description. Record and array variables, however, need to have some description of their

type for the animator. For this purpose, records and arrays use the child field of the static scope table structure as an integer index into the static scope table where their descriptive types are stored. The name table section of Chapter 5 describes in detail how this field is managed.

Access Types

Ada/CS access types (pointers) have not yet been implemented. The AccessType class therefore has no functionality as of yet.

Procedures and Functions

Even though procedures and functions are not types, *per se*, their data class is part of the type table simply because placing them there works well in practice. Procedure and Function type classes manage the parameter lists and, in the case of functions, the return type.

When a subroutine is encountered, the parser builds a list of parameter names, types, and pass modes, and passes this list to the type class, which checks the list for validity (variable name uniqueness, whether the type name actually exists, etc.). When the subroutine code is generated, the ProcedureType class directly adds its variables to the name table. Because the variable list has already be verified to be correct (no duplicate names, types exist, etc.), bypassing the checks done by the symbol table saves time. Since the procedure/function type class is a C++ friend of the symbol table class, it has direct access to the symbol table data elements (i.e., the name table).

As far as the symbol table is concerned, procedures and functions share many of the same properties. In fact, the only difference between the two is that functions return a value and so the return type must be known. Taking advantage of C++, FunctionType is actually a descendant of ProcedureType, adding one method for the

return type, and inheriting the parameter list management from the ProcedureType class.

CHAPTER 7

E-MACHINE CODE GENERATION

The Code Generation Classes

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 source section, the static scope section, the label section, the variable section, the packet section, the code section, and the string section. In the Ada/CS compiler, code generation is handled by seven classes, one for each code file section (with the exception of the Header Section). Each class is responsible to save 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. Like the symbol table and type table classes, there is only one instantiation of each of these section classes.

Base Class Section

All the code generation classes descend from the base class Section. The Section class currently has no responsibilities and only serves as an abstract base class.

Class StringSection

The StringSection class manages the list of strings that are encountered in the source program by the compiler: enumerated names, string literals, run-time error messages (like "unhandled exception"), and so forth. This is a very simple class whose only responsibility is to accept and store string literals.

Class SourceSection

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 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.

Class LabelSection

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 in one of two ways. The first label allocation method accepts an address, associates a label with that address, and returns the new label. The second method is a little more complex. On occasion, the compiler needs a label, but doesn't yet know the associated address of the label. For example, in parsing an "if" statement the compiler needs to generate a destination label for creating a branch around the else part to the end of the if. However, the destination address cannot be determined until later, when the else part is completely parsed. The parser requests a label through the method *alloc_label* and when it eventually determines the address of the destination (when it has finished parsing the else part, for example), it tells

the LabelSection to set that label number to that known address.

Class VariableSection

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 the CodeSection's methods. Most E-machine instructions require a type (integer, real, boolean, etc.). By making the type a part of each variable, the CodeSection need only query the type from the VariableSection when necessary.

Class CodeSection

The CodeSection class generates the E-machine instructions and has a method for each instruction of the E-machine, as well as a few general purpose methods. The CodeSection class is responsible for the "bootstrap" code, the *pushd* instruction at the start of the executable code that pushes onto the dynamic scope stack the location in static scope table of the outermost scope. The CodeSection class is also responsible for adjusting out-only subroutine parameters (discussed in detail later in this chapter).

Class PacketSection

The PacketSection class is responsible for the executable packets defined by the compiler. This class has one workhorse method called *new_packet*. The new_packet method is straightforward. It accepts as parameters the starting and ending line and column numbers of the new packet, the scope index, and the flags telling how the packet should be displayed. It then generates the proper packet information for the packet section.

Class StatScopeSection

The StatScopeSection class, which creates the E-machine Static Scope section, is probably the most complex of the code generation classes because of the parent/child relationships. Record variables must have their child index set to the static scope block of their record type. Like record variables, the child index of a subroutine static scope structure would point to a static scope block describing all the variables and types in that subroutine. One-dimensional arrays are described in the variable's own static scope structure, but two or more dimensions require a static scope block for each dimension. These new scope blocks are indexed by the child of the previous dimension. For instance, consider a three dimensional array variable. The first array dimension would be described by the variable's own static scope structure and the child index of that structure would point to a static scope block that describes the second dimension. The second dimension's child index would point to a static scope block that describes the third dimension.

When a scope is collapsed, the name table method pop_scope first saves all the user defined types to the static scope table through calls to its method save_sscope_type. There are two types that require special static scope table consideration: records and arrays. Arrays are not yet fully implemented and save_sscope_type has no array functionality. However, records are fully implemented.

A record static scope block contains its members and looks much like a subroutine static scope block with one exception. The "offset" static scope structure member is used to indicate the byte offset of each record member from the start of the record. Because of this difference, the StatScopeSection class has separate methods to save record members rather than use the methods which save variables. The save_sscope_type loops through the members of a record type and calls the appropriate static scope table member to save the variable. Srecinteger creates an integer

record member static scope structure, Srecfloat creates a float record, Srecboolean creates a boolean record, and so forth.

After saving the user defined types, the name table saves all the variables in the collapsing scope through calls to StatScopeSection methods named for the types they save. The methods are as follows:

- Sinteger, Sfloat, Sboolean: These methods save variables of the built in E-machine types.
- Senumerated: This method saves a variable of an enumerated type to the static scope table.
- Sprocedure, Sfunction: Saves a procedure or function name to a static scope entry. The name of the subroutine is stored in its enclosing scope. For example, if procedure print_data is defined within function test_record, print_data's name will be in test_record's static scope block. The child index of print_data's static scope structure will point to a scope block describing the variables, types, and subroutines in print_data.
- Srecord, Sarray: Creates a record and array variable static scope entry, respectively. Both of these methods return the table location where the record or array was added. This value is stored in the type class for later use when the variable's child indices are set to their type's static scope block.
- Sprogram: Creates the "bootstrap" block at the end of the static scope table. The bootstrap static scope block contains one entry, the program name.

Another responsibility of the StatScopeSection class concerns the *pushd* instruction. When the compiler encounters a subroutine, it generates a *pushd* instruction which pushes the static scope table position of a new subroutine onto the static scope stack. However, this index is not known until the subroutine is saved to the static scope table so the *pushd* instruction cannot be completed yet. After the parser completes the subroutine and the scope is collapsed (so that the subroutine is now in the static scope table), the parser queries the StatScopeSection class to get the location of the newly added subroutine. This position is saved to the symbol table entry of the subroutine in the enclosing scope and is used by the StatScopeSection methods Sprocedure and Sfunction to complete their *pushd* instructions.

Special Ada/CS Code Generation Considerations

Ada/CS, even though a small subset of Ada, shares some of its parent's complex code generation challenges. Other challenges arise in the code generation because of the special needs of the E-machine.

Out-Only Subroutine Parameters

Out-only subroutine parameters are an interesting feature of Ada and are supported by Ada/CS. Out-only subroutine parameters are only allowed on the left hand side of an assignment statement and cannot be used in an expression. An implementation difficulty comes from the fact that the E-machine doesn't appreciate an attempt to push an undefined variable onto the stack and sets an UNDEFINED_DATA fault if a program attempts to do so. However, it is entirely valid to pass an undefined variable to an out only parameter (passing an undefined variable through an in-only or in-out parameter would correctly cause an UNDEFINED_DATA fault). The compiler would only know a parameter is out-only after it has been parsed and a *push* instruction has already been generated.

A clean solution to this problem would have required that a complex set of special cases be handled through the expression grammar; therefore, a dirty solution was implemented. When the compiler discovers it has pushed a value is shouldn't have, it looks back at the instructions generated (which are saved in an array until the entire source program is compiled) and modifies the instructions in such a way that a constant is pushed instead. Since a number of *push* instructions could have been generated (such as in passing 3+5), a marker instruction (a *nop*) is placed after each subroutine parameter. The compiler then searches for the appropriate instruction

by counting *nop* instructions and replaces the instruction as necessary. To avoid stack corruption, upon subroutine entry the bogus value that has been pushed is popped into a garbage variable and forgotten.

Functions Returning Records or Arrays

The usual way to return a value from a function is to push the return value onto the stack. The E-machine has eight addressing modes, none of which allow the program to access elements on the stack. Further, since only E-machine base elements (integer, float, etc.) may be pushed onto the stack, records cannot be pushed onto the stack anyway. This poses a problem in parsing Ada/CS functions, which can return records or arrays.

As per the Ada definition, passing records and arrays as parameters is done by pointer (and copied to a local if an in-only parameter). However, functions returning aggregate data structures are a little more complex to implement. The solution was to allocate a record or array for the return value in the function and return a single pointer to this value. The code that uses the returned record or array (perhaps an assignment statement or a record reference such as funct(10).a) is responsible for retrieving what it needs from the returned structure and then freeing the memory.

The For Loop

The for loop, while a simple construct, presented some difficulty with respect to the E-machine. In both Ada and Ada/CS, the loop variable need not be explicitly declared and, in such a case, must be implicitly declared to be the same type as the loop's range. But variables must be in the static scope table for animation purposes. How can a loop variable be placed into the static scope table?

Since the loop variable is only valid within the scope of the loop body, it can not be added to the end of the current static scope table. Consider the code shown in figure 33.

```
package body prog is

begin
  for i in 1..10 loop
    null;
  end loop;

  for j in true..false loop
    null;
  end loop;
end
```

Figure 33: For Loop

The animator determines which variables of the static scope table to display through two indices. The first index tells the position in the static scope table of the subroutine information currently in scope and is set through the E-code instruction *pushd*. The second index is an offset from the first index and is one of the members of the packet table structure (scope index). The animator will start at the top of static scope block set by the *pushd* instruction and display the number of variables indicated by the scope index. The animator only knows an offset. If the scope index is 5, the animator cannot ignore the 3rd variable in the list; it must display all five.

To display a certain variable in the static scope table, the animator must display all the variables before that variable in the static scope block. Now, assume that i is added to the end of the current static scope table entry. This means that the loop variable's scope index must be incremented by one so the animator will reach deep enough into the static scope entry to display it. However, when the compiler

reaches the second loop, j would be need to be added in a similar fashion after i at the end of the static scope table entry. In order for the animator to display j's values, the scope index will have to be increased by one again, thus also displaying i's values, even though it is no longer in scope.

A solution to this problem was to create an entire new static scope entry for each loop, making the loops look like small one variable subroutines. A *pushd* would push the static scope entry position of the loop onto the dynamic scope stack, and a *popd* pops this entry after the completions of the loop. A static scope entry for figure 33 is shown in figure 7.

Entry	Id Name	Array Type	Upr Bnd	Lwr Bnd	Nxt Idx	Offset	Type	Rec Siz	Parent	Child	Var Reg	Proc Num	Index Type
Scope block describing loop i													
0	loop i	-	-	-	-	-	HEADER	-	6	-	-	-	-
1	i	-	-	-	-	-	INTEGER	-	-	-	2	6	-
2		-	-	-	-	-	END	-	-	-	-	-	-
Scope block describing loop j													
3	loop j	-	-	-	-	-	HEADER	-	6	-	-	-	-
4	j	-	-	-	-	-	INTEGER	-	-	-	2	10	-
5		-	-	-	-	-	END	-	-	-	-	-	-
Scope block describing program Prog													
6	Prog	-	-	-	-	-	HEADER	-	10	-	-	-	-
7	loop i	-	-	-	-	-	FUNCTION	-	-	0	-	-	-
8	loop j	-	-	-	-	-	FUNCTION	-	-	3	-	-	-
9		-	-	-	-	-	END	-	-	-	-	-	-
Bootstrap scope block													
10		-	-	-	-	-	HEADER	-	-	-	-	-	-
11	Prog	-	-	-	-	-	PROGRAM	-	-	6	-	0	-
12		-	-	-	-	-	END	-	-	-	-	-	-

Table 7: Static Scope Table for a For Loop

CHAPTER 8

CONCLUSIONS AND FUTURE ENHANCEMENTS

Conclusions

An Ada/CS compiler for the E-machine has been designed and partially implemented. The Ada/CS compiler is a one-pass compiler written in C++ and was developed using the parser development tool PCCTS 1.10 (Purdue Compiler Construction Tool Set) [Parr 93]. PCCTS generates an integrated recursive descent LL(k) parser and DFSA based scanner. Development started under MS-DOS, moved to OS/2 when the compiler became too large for 16-bit DOS, then to a Silicon Graphics workstation, and was finally completed on an IBM-PC clone running the FreeBSD Unix. A number of Ada/CS programs have been successfully compiled and animated with the preliminary Unix Motif animator.

Future Enhancements

There are several standard Ada/CS features that are not yet finished. Implementation of arrays, pointers, and exceptions (reversal through exceptions will be an interesting challenge) is necessary to make this compiler meet the needs of the Dynalab project. The Ada/CS compiler must match as closely as possible the fundamental features of Ada so that simple Ada programs will be readily compilable by Ada/CS and students using Ada/CS will have no difficulty making a transition to full Ada.

In order to emphasize the advantages of Ada, the Ada/CS compiler will need to support separate compilation (packages), which has yet never been attempted in an E-Machine compiler. The "bind" program would have to take into account the E-Machine code file sections and be able to correctly merge those sections. Further, the "bind" would have to take into account that the source for certain library packages should not be animated (I/O will be implemented as a package; animating through put_line would be unnecessary).

Additional features beyond those of Ada/CS are needed to make the language more closely match Ada. For instance, the addition of a character type (strangely left out of Ada/CS) and implementation of a *with* statement would make Ada/CS more closely match Ada.

This compiler was written in C++ and takes advantage of several object oriented features. By the time the compiler development was moved to the Unix based platforms, it had grown complex enough that another C++ feature, templates, would have been extremely useful. However, as of this writing, the GNU C++ compiler used on those platforms does not fully support templates. Templates are a

feature of C++ much like generics in Ada. They allow the programmer to describe a general set of code to perform on an unknown data type. For instance, the template declaration in figure 34 defines a generic stack class.

```
template <class Type>
class Stack {
  private:
     Type *stk;    // array
     int max_size; // current maximum size of stack array
     int top;      // top of stack
     // next push location is top+1

  public:
     Stack( int size );

     Type& pop( void );
     void push( const Type& elem );
     Type& operator []( int pos );
};
```

Figure 34: C++ Stack Template

Wherever "Type" appears in the definition, a real type is substituted when the template class is instantiated. A class declaration would look something like Stack<int>, Stack<struct AdacsType>, or Stack<NameTable> for a stack of integers, AdacsType structures, and NameTables, respectively. This template was created using Borland C++ under OS/2, but was never made a full part of Ada/CS because of GNU C++'s lack of template support (although full template support is expected shortly).

PCCTS is still under development and periodically has updates and new features. The newest version (1.20) released during the final stages of this project, generates C++ parser classes which would fit in well with the object oriented nature of this

compiler. Another feature of PCCTS is its ability to generate Abstract Syntax Trees (ASTs), which would be useful if this compiler were converted to a two-pass compiler. In that case, the Ada/CS source program would be compiled into an AST and a second pass could resolve certain difficult problems (such as function return type overloading).

The PCCTS parser has very good error reporting and recovery. However, the compiler does not take advantage of the parser's recovery features although preliminary steps are included. A future improvement would be better error recovery so that the student will receive more than one error message before the compiler terminates.

REFERENCES

References

[Birch 90] M. L. Birch. *An Emulator for the E-machine.* Master's thesis. Computer Science Department, Montana State University. June 1990.

[Brown 88-1] M. Brown. *Algorithm Animation.* The MIT Press, Cambridge, Massachusetts. 1988.

[Brown 88-2] M. Brown. 'Exploring Algorithms Using Balsa-II", *Computer* Volume 21, Number 5. May 1988.

[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.

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

[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.

[Ross 91] R. J. Ross. "Experience with the DYNAMOD Program Animator", *Proceedings of the Twenty-second Symposium on Computer Science Education, SIGCSE Bulletin*, 23(1):35–42. 1991.

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

[USDOD 93] United States Department of Defense. *Reference Manual for the Ada Programming Language*. Springer-Verlag, New York Berling Heidelberg. 1993.

APPENDICES

APPENDIX A

THE E-MACHINE INSTRUCTION SET

This appendix, which is adapted from chapter 2 of Birch's thesis and appendix A of Goosey's thesis, lists all of the instructions in the instruction set of the E-machine. A pseudo assembly language format is used to describe the instructions, however the instruction stream itself is actually an array of structures loaded from the CODESECTION portion of the E-machine object file at run time. 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 critical or noncritical (CFLAG);
- a field denoting the data type to be used in 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 others 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 # 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: Arbitrarily 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 favorably 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 for 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 the # 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 the # onto the dynamic scope stack (where # is the index of a program, procedure, or function entry in the Static Scope Table)

Forward: Pushes # onto the dynamic scope stack.

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

read CFLAG, TYPE, ADDR

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

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

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

return CFLAG

Returns to the appropriate program address following a call instruction.

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

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

shl CFLAG, TYPE, #

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

Forward-Critical: Pops the top value of the evaluation stack, pushes it onto the save stack, then shift 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 shift 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 and appendix B of Goosey's thesis, 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:

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

variable mode - V#:

variable register # ⟶ top of variable stack ⟶ data memory

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

Example:

$B := 1;$

could be translated into:

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

variable indirect - (V#):

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

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

Example:

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

could be translated into:

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

variable offset mode - V#{offset}:

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

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

Example:

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

could be translated into:

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

address offset mode - A{offset}:

address register + IR ⟶ data memory

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

Example:

I := H↑.Data

could be translated into:

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

variable indexed mode - V#[index]:

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

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

Example:

B := L[3];

could be translated into:

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

address indexed mode - A[index]:

*address register + IR * data size ⟶ data memory*

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

Example:

B := S↑[4];

could be translated into:

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