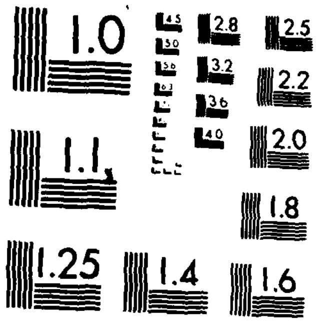
AD-A138 027   ADA1: AN ADA SUBSET COMPILER FOR THE AFIT SYNTAX   1/4
DIRECTED PROGRAMMING ENV..(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..   M L MCCRACKEN
UNCLASSIFIED   05 DEC 83 AFIT/GCS/MA/83D-4   F/G 9/2   NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

ADA1
AN ADA SUBSET COMPILER FOR THE AFIT SYNTAX DIRECTED PROGRAMMING ENVIRONMENT

THESIS

AFIT/GCS/MA/83D-4 Michael L. McCracken
Capt. USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/MA/83D-4

ADA1
AN ADA SUBSET COMPILER FOR THE AFIT SYNTAX DIRECTED PROGRAMMING ENVIRONMENT

THESIS

AFIT/GCS/MA/83D-4 Michael L. McCracken
Capt. USAF

Approved for public release; distribution unlimited.

AFIT/GCS/MA/83D-4

ADA1

AN ADA SUBSET COMPILER FOR THE AFIT SYNTAX DIRECTED PROGRAMMING ENVIRONMENT

THESIS

PRESENTED TO THE FACULTY OF THE SCHOOL OF ENGINEERING
OF THE AIR FORCE INSTITUTE OF TECHNOLOGY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTERS OF SCIENCE IN COMPUTER SYSTEMS

by

Michael L. McCracken
Capt. USAF

Graduate Computer Science

5 December 1983

Preface

I came to AFIT interested in compiler writing and programming languages. This thesis effort allowed me to combine those interests into a very interesting and self-satisfying project, a compiler and interpreter for a subset of the Ada programming.

I would like to thank Maj. Roie Black for proposing this project to me and introducing me to the Ada programming language. I would also like to thank my advisor, Capt. Pat Lawlis, for all her help and guidance during this effort.

Finally I must thank my loving wife for her patience and support and my children for just being children and making me laugh. Without their support and love this project might not have been completed.

Table of Contents

Table of Contents

Table of Contents

List of Figures

Abstract

This document describes the effort involved in moving the Ada0 compiler and interpreter developed by Capt. Scott E. Ferguson as part of the AFIT syntax directed editor environment from a microcomputer to the VAX 11/780.

As part of this effort the compiler and interpreter were expanded to accept a larger suset of Ada. The compiler and interpreter work with an abstract syntax representation of a computer program produced by the syntax directed editor. This abstract representation, which is guaranteed to be syntactically correct, makes the compiler much easier to write and understand. The compiler in a top-down compiler but no backtracking is needed since the program is known to be syntactically correct. The interpreter is able to use the abstract representation to give the user an interactive display of the program during execution.

Designs to allow overloading of names and operators, and passing parameters to subprograms are also presented.

INTRODUCTION

1.0 INTRODUCTION

Ada is the new computer programming language developed by the Department of Defense. Motivated by desires for increased productivity and lower costs, requirements for a programming support environment were also developed (Ref 8). This thesis is one of a pair in a continuing effort to develop such an environment. This effort is based on a prototype environment developed by Capt Scott Ferguson (Ref 9). His effort is centered around a syntax directed editor which creates an abstract syntax representation of a program which the other tools in the support environment work with. This effort continues the development of the compiler and interpreter within this environment.

1.1 PRIMARY OBJECTIVE

The principle objective of this thesis effort was to continue the development of the AFIT Ada programming support environment originally developed by Scott E. Ferguson. This involved an analysis of the Ada programming language and the work done by Ferguson. This was done with the intent of moving the support environment from the microprocessor it was developed on to the VAX 11/780 and expanding the subset of Ada that it would accept and compile.

INTRODUCTION

Since the support environment included several basic tools, the effort was divided into two separate but related thesis efforts. The basic tools included in the original system are a syntax directed editor, a program lister, a compiler code generator, and an interpreter to run the compiled program. This document describes the effort involved in getting the code generator and interpreter tools of the support environment working on the VAX 11/780 and the expansion of the Ada subset the compiler would accept. Also included is brief discussion of the other tools in the support environment and how they interface with the compiler.

1.1.1 SECONDARY OBJECTIVES

The main secondary objective was to enlarge the subset of Ada the compiler would accept. The features to be added are new data types, functions and some new statement forms. Since the original effort was named the Ada0 compiler by Ferguson, this effort was called the Ada1 compiler.

The other secondary objective was to design algorithms and data structures to support a later implementation of overloading of names and operators, parameters for functions and procedures, packages, and user defined data types into the expanded Ada1 compiler. This objective was left as designing the algorithms and data structures since many problems had to be solved before any of the algorithms could be implemented. Time restrictions placed on the thesis effort rather than complexity of the algorithms was the main consideration.

1.2 BACKGROUND OF THE ADA0 COMPILER

The current Ada0 compiler accepts only a small subset of the full Ada language. Included in this subset are integer variables, parameterless procedures, integer arithmetic, Boolean expressions involving integer relations, and many of the statement forms of Ada. The statement forms included are assignment, procedure calls, if statement and while loops. The Boolean expressions included are integer comparisons and the Boolean operators and, or , and not.

1.3 OVERVIEW OF THE THESIS

To be able to properly discuss improvements to the Ada0 compiler and support environment a discussion of the features of Ada itself is required. Chapter 2 provides this discussion and a short discussion of the Ada1 subset. Chapter 3 is a discussion of the support environment and the tools as currently implemented. These tools include a syntax directed editor, a program lister, the compiler, and an interpreter/run time mechanism. Chapter 4 provides a more detailed discussion of the current Ada1 compiler and how the extensions were implemented. Chapter 5 provides the details of the designs done for this thesis effort. These include a semantic analyzer to handle the resolution of overloaded names and operators, a way to handle passing of parameters that will allow in and out parameters and not be limited to simple types, and the data structures to implement user defined data types and allow for package elaboration and inclusion. Chapter 6 describes the ultimate goals and environment of the Ada compiler. Chapter 7 is a discussion of my conclusions from this thesis effort and my recommendations for future work.

ADA

2.1 BACKGROUND OF ADA

Ada is the new computer programming language developed for the Department of Defense. The driving force behind its development is the the rapidly increasing cost of software, both new and modified. These increasing costs are further amplified by the fact that many different and incompatible computers are used within the Department of Defense. These differing computers have led to the use of a plethora of special purpose languages to program them.

This causes problems in training since a programmer may need to learn a new language or system when he changes jobs. This also causes the programmer to be less efficient until he comes up to speed under the new system. In fact if the languages are different enough the programmer may have to begin almost at square one.

This also means that a routine developed for one system must be rewritten, if it is needed on a different system. Even programs written in high level languages, like FORTRAN, can require changes to work on different computers. These changes can be quite extensive even if both computers have FORTRAN compilers. A standardized language and working environment, as Ada is intended to provide, could help to reduce or even eliminate these problems. One study estimates the savings to be in excess of one billion dollars per year (Ref 10).

ADA

For these reasons and others that will be brought out later, the Department of Defense developed criteria for a standard and universal language. The search for this new language began with existing computer languages. Several of the more popular languages were examined and each was found to be deficient in one respect or another. Since no existing language could be used, the Department of Defense held a four contractor competitive design effort to develop the new language. This competition resulted in the development of Ada.

2.2 GOALS OF ADA

The main goal of Ada is to help reduce the costs involved in writing and modifying software. This goal is to be attained in two separate but interacting ways. The first is through the use of a standard language that has many features to aid the programmer. The other is through a standardized working environment. The requirements for this environment are laid out in "STONEMAN, Requirements for Ada Programming Support Environments" Feb 1980. The purpose stated for the support environment is to "support the development and maintenance of Ada applications software through its life cycle" (Ref 8). The syntax directed editor mentioned earlier could be one of these tools.

2.3 FEATURES OF ADA

The syntax directed environment will help make Ada more popular among computer programmers but the many features of the language will do more to stimulate its use. Ada has many interesting features several of which should be discussed. These features are packages, overloading, tasks, separate compilation, and universality.

2.3.1 PACKAGES

Packages are one of the more interesting features included in Ada. They are one form of Ada program units. Their intent is to allow the specification of groups of logically related entities. Packages are allowed to contain their own data structures and types, and the subroutines to manipulate them. The pieces of a package can be visible to an outside program, invisible to it, or a combination of the two.

Their intent is to allow another programmer to use the data structures and types, and the manipulating routines declared in the package without knowing or caring how they are implemented. Keeping the structure of a data type invisible to the programmer, even though the type itself is visible, is intended to prevent the programmer from directly manipulating the data which can create problems for other routines in the package.

One benefit of this is a programmer can write and compile the package once and make it available for use by other unrelated programs. After the package is written, its users need not care how the data is stored or manipulated but only that the results obtained from the package are correct. Since a user does not care how a package is implemented, the package body can be changed and if done properly the change will not affect any of the programs using the package.

```
package complex_arithmetic is

  type complex_number is private;

  function "+"(A,B : complex_number)
                     return complex_number;
  function "-"(A,B : complex_number)
                     return complex_number;
  function create_complex(R,I : integer)
                     return complex_number;

  private

    type complex_number is
      record
        real_part : integer;
        complex_part : integer;
      end record;

end complex;
```

Figure 2-1. Sample Package Declaration

An example is a package to do arithmetic on complex numbers. The data type used for the complex numbers could be a record, an array, or even a linked list. This type could be changed from one to the other without any affect, except possibly execution speed, being seen by any user. A sample package is declaration is shown in figure 2-1.

This interface of the package is all the user needs to see and know about the package. Declaring the type complex_number as private means the user can declare variables to be of type complex but must use the functions provided to manipulate them. The user is allowed to do assignment and membership tests (Ref 6: 7-6). This package exhibits another feature of Ada, overloading.

2.3.2 OVERLOADING

Overloading of names and operators another feature of interest included in Ada. Overloading, as defined by the Ada standard, is a relatively new concept. Unlike Pascal and other block structured languages which hide previous declarations of a name when the name is redeclared, a redeclaration of a name in Ada only hides previous declarations of the name which are of the same type. The idea is to allow one name or symbol to have several different meanings depending on the context of its use. To be valid the context must make the use unambiguous. If two or more meanings are consistent with the context then the use is erroneous and must be modified in some way.

In the previous example the operators "+" and "-" are overloaded to allow their use with complex_numbers. An overloaded operator can be used in the usual manner. For example the following code fragment will cause two complex_numbers to be added using the "+" operator and the result assigned to another complex_number.

```
     a,b,c : complex_number;
      a := b + c;
```

The idea is to make programs more readable and understandable for other programmers. In other languages the addition of two complex numbers would have to be done using a function or procedure with a name like complex_addition. This would make the program readable but some programmers would shorten the name to someless meaningful mnemonic like comp_add.

The designers' of Ada felt that the "+" operator could have meanings other than standard addition with respect to non-standard types and that it should be allowed to express that meaning. They also allow a user to overload and hide the standard meaning of these operators if they wish to define their own functions to do addition.

2.3.3 TASKS

Tasks are the next feature of interest included in Ada. Since Ada is intended to be used mainly in the area of embedded computers by the Department of Defense, this feature is very important. These embedded computer systems usually require several functions or processes to be performed simultaneously. Tasks allow the programmer to define several processes and allow them to execute in parallel either on a single processor or on multiple processors.

Tasks are allowed to communicate or synchronize through the rendezvous feature. Since tasking can be implemented in several different ways the Ada standard says that any program that depends on the implementation is *erroneous*.

An interesting feature of tasks is that they can be defined as a type and used in data declarations. This allows the user to define arrays of tasks, records that contain tasks, or any other legal use of a data type. The task type is limited so assignment and predefined comparison are not allowed. Examples of tasks can be found in the Ada Reference Manual (Ref 7: 9-20).

2.3.4 SEPARATE COMPILATION

Separate compilation is another feature of interest. Separate compilation is a term that has been misused in the past. Many languages have claimed to allow separate compilation when what they truly did was independent compilation. The difference lies in the semantic checking that is done.

Independent compilation allows a function or procedure to be compiled independently of the rest of the program. The idea was to allow a function or procedure to be used by several different programs and eliminate the need for each to compile the common parts. Though this saves time, it causes other problems since a using program can misuse an external function and still compile correctly. The error generated on execution may show no relation to that misuse and cause the programmer to waste valuable time debugging the problem.

Separate compilation requires that all semantic checks be done as if the separate pieces were all compiled together as one unit. This requires the compiler to have the specifications of those pieces available when they are needed. This difference is pointed out quite nicely in the preliminary reference manual (Ref 7: 10-1).

2.3.5 UNIVERSALITY AND STANDARDIZATION

Universality is the final feature and perhaps the one with the most promise for helping reduce the costs of software. This feature is being enforced by the Department of Defense. Any Ada compiler must pass a series of validation tests to ensure it is neither a subset compiler nor a superset compiler. Only after it passes these tests is it allowed to be called an Ada compiler. An exception to this is allowed for any subset compiler that is still in the development stage and eventually will become a full Ada compiler.

This standardization should help reduce the retraining problem encountered when a programmer changes jobs. Since it is intended that most tools in use in the Ada support environment be written in Ada, they will be fully transportable. This will further reduce the retraining problem.

Another benefit of standardization is the incentive it gives programmers to write useful programs and packages, since the potential market is much larger than for a similar program written for one machine or family of machines. Hopefully this will lead to the emergence of large software houses and catalogs from which a programmer can order the packages and subprograms needed to do a project. The packages and subprograms can then be compiled with new code to produce a new program much more quickly.

2.4 SCOPE OF THE THESIS

This section outlines the scope of the thesis effort. The scope can be broken into three parts, the features of Ada that were implemented, the features of Ada that designs were done for, and the features of Ada that were ignored for this thesis effort. Since Ada is such a large language and time was limited a workable subset of the it had to be chosen and some of the features had to be left out.

2.4.1 FEATURES IMPLEMENTED

The features of Ada implemented in the Ada1 compiler included the Ada0 subset, new data types, functions, the remaining operators, and some of the remaining statement forms. These features are explained in more detail below.

2.4.1.1 FEATURES OF THE ADA0 SUBSET

The first things to be included in the Ada1 subset were any features already implemented through the Ada0 subset. Since this thesis effort is based on the Ada0 compiler, developed by Scott Ferguson, all features of it were included. The code from the Ada0 compiler required some modification to work for the expanded Ada1 subset.

2.4.1.2 DATA TYPES

The next feature added to the Ada1 subset was two new data types, booleans and characters. These were added since the Ada0 subset was integer only. Unfortunately, the addition of new data types required most of the code written for the Ada0 compiler to be modified. The modifications to the code were made to allow the compiler to do type checking and report typing errors.

2.4.1.3 FUNCTIONS AND PROCEDURES

The next feature added to the Ada1 subset was functions. The Ada0 subset allowed only procedures. Since a function is essentially a procedure that returns a value functions were easy to include.

2.4.1.4 NEW OPERATORS

The Ada0 subset left out some of the operators defined in the Ada grammar. These operators are included in the Ada1 subset. The new operators are the boolean operators, and then, or else, and xor, and the integer operators rem, mod, abs, and the exponentiation operator, **.

2.4.1.5 ADA STATEMENT FORMS

Most of the Ada statement and expression forms are included. The statement forms included are the assignment, if-then-elsif-else, iterative loops, while loops, return, and null statements. The statement forms missing from the compiler are mostly associated with tasking. Since tasking was not implemented these could not be included.

2.4.1.6 COMMENTS

Comments were the final feature added to the Ada1 grammar. The comments are a limited subset of the Ada comment feature. The limitation was added to make comments easier to handle for the compiler. Ada allows comments to occur at any point in a program. The Ada1 subset limits comments to be used as a program header, as a regular statement, and to follow statements and variable declarations. These places were thought to be the most useful and also are the places within a program that a user usually puts comments.

2.4.2 FEATURES DESIGNED

Designs were done to allow several of the more interesting features of Ada to be implemented at a later time. These features are overloading of names and operators, packages, user defined data types, and passing parameters to subprograms.

2.4.2.1 OVERLOADING OF NAMES AND OPERATORS

The first design done was an algorithm to handle overloading of names and operators. Several good algorithms exist (Ref 2,14,16) to handle the problems of operator and name identification. All that should be needed is a "black box" implementation of one of them to do a prewalk of the abstract syntax tree and hang the correct types onto the various structures.

The code involved in such a tool is long and very involved, even though several of the references actually give most of the code or pseudo-code for their algorithm. The time needed to include this tool is beyond the scope of this thesis effort.

Also overloading is not really essential to getting a subset in which these tools could be rewritten. The only change to the compiler that would be needed to include this semantic analysis tool is the call to it when an expression is encountered and the deletion of calls to the symbol table routines when a name is finally found since the type and symbol table information will already be attached to the node. A design of the proposed semantic analyzer is presented in chapter 5 and Appendix D.

2.4.2.2 PACKAGES

The second feature designed were data structures to allow implementation packages. The design is presented in chapter 5. The current implementation in no way precludes or limits the inclusion of the design.

2.4.2.3 DATA TYPES

The data structures needed to allow user defined data types were designed next. These were only designed because the code needed to do semantic verification was not written. Even without overloading the code to do type verification of expressions was difficult to implement correctly. Since a semantic analyzer will eliminate the need for this code, I felt writing extra code that would later need to be eliminated was wasteful of time and effort. The data structures needed to implement user defined data types were designed and are presented in chapter 5.

2.4.2.4 PASSING PARAMETERS TO SUBPROGRAMS

The final design done was of an algorithm to allow parameters to be passed to subprograms. The algorithm designed is presented in chapter 5 and Appendix C.

2.4.3 FEATURES NOT INCLUDED IN THE SUBSET

Since time was limited certain features had to be excluded from the subset. Several papers (Ref 5,15,17) influenced my decisions regarding what features to exclude. These papers pointed out problems with the Ada grammar, problems with a feature or the fact that a feature was extraneous to the language. One of these papers quoted Niklaus Wirth who said

> The choice of what is to be omitted from a new language is in practice much more critical than the choice of what is to be included. The decision to omit a feature requires not only familiarity with this feature (and knowledge of how to live without it) but the courage to face the inevitable criticism of its absence in the new language in spite of its presence in another existing language (Ref 15).

Interestingly this quote was taken from the Green Reference Manual which was the original Ada design but it was omitted from later versions of the manual. Wirth was referring to the fact that DOD was attempting to include too many features into Ada to be popular. He was afraid Ada would follow the same path as PL/1 which has essentially died in spite of the backing given it by IBM (Ref 17). Most of the features excluded from this subset are features that the user can do without and in fact many of them can be simulated using the features contained by the language subset that was implemented.

2.4.3.1 TASKING

The first feature excluded from the subset was tasking. Many problems with this feature are pointed out in a technical note from the Defense Communications Engineering Center (Ref 5). Since these problems complicate the understanding of exactly how tasking is supposed to work and therefore how it must be implemented, tasking was excluded.

Another factor which helped allow the exclusion of tasking is that a tasking Ada subset compiler already existed at AFIT (Ref 11) and it could be used or rewritten for later integration into the syntax directed editor environment. This rewrite would probably have taken most of the time allotted to the development of this subset compiler and thus the other features would not have been implemented.

2.4.3.2 GENERIC PACKAGES

The next feature to be eliminated was generic packages. The main reason for not implementing generic packages was that packages themselves were not implemented and generic packages cannot be implemented until such time.

2.4.3.3 SEPARATE COMPILATION

The next feature eliminated was the separate compilation of Ada sub-units. This feature only complicates the design of the how the generated code is stored and written out. The current design does not preclude the insertion of code from a later compile but it is not easily extended to allow such an insertion.

One method that will work is to generate a jump statement as the specification is compiled. When the sub-unit body is compiled fix that statement to jump to where the actual code is placed. This would allow the code to be placed anywhere in the code array that space permits.

One problem comes up when this type of separate compilation is done. This is the problem of what the symbol table looks like when the procedure should have been encountered. The solution to this problem is not trivial since the entire symbol table up to the point where the specification is found must be saved for use when the subunit is compiled. Due to the limitation of time and the fact that this feature is not needed to attain the goal of writing the tools into Ada it was not included.

2.4.3.4 THE GOTO STATEMENT

The next feature eliminated was the goto statement. This feature is not needed due to the numerous control structures already available in Ada. Since Ada is such an otherwise structured language the rationale for including a goto statement is hard to figure out. Its inclusion, no matter how structured, can only lead to misuse or very complicated compiler restraints to ensure proper usage. Since the arguments against the inclusion of the goto outweigh any possible gain from its inclusion, the goto was not included in the subset.

2.4.3.5 OTHER FEATURES LEFT OUT

Several other features were not implemented. Most of these were excluded for reasons of time or not being needed for the chosen goal. Although these features are of interest they tend to add little to the capabilities of the subset. These include private types, access types, named parameters, and input/output of enumeration types.

The design of the compiler in no way precludes the later inclusion of any of these features and in fact many of the data structures used by the compiler were designed with the excluded factors being considered. The goal behind the design was to allow an easier extension than the Ada0 subset compiler did. Several fields of the data structures are not even used but were included to handle the analysis of these features.

2.5 PROBLEMS WITH THE ADA GRAMMAR

The development of this subset and the compiler for it also brought out several problems with the Ada grammar as presented in the Ada reference manual (Ref 7). These problems were mostly in areas where the Ada syntax specification allows the use of a non-terminal with a pre-fixed italicized modifier. The italicized modifier creates a variety of the non-terminal, but syntactically the modifier is ignored (Ref 7,17). For example procedure_name and package_name are varieties of name.

Usually these qualified non-terminals are used in such restricted situations that the italicized modifier can be retained without causing the grammar to become ambiguous. Since the non-terminal can be restricted in this way, the Ada1 subset incorporated these restrictions.

Incorporating these restrictions into the Ada1 subset accomplished two things. First, it simplified the user's job when writing a program with the syntax directed editor This is because some choices that would otherwise be available to the user are eliminated. This also eliminates the need for the compiler to do some of the semantic checks that would be necessary if the restrictions were not incorporated. This is because the restrictions eliminate choices that would be syntactically correct but semantically incorrect.

For example, in the full Ada grammar procedure_name is interpreted simply as name. The legal choices for name are shown in figure 2-2. Of these choices only identifier is semantically correct. Thus in the Ada1 grammar procedure_name was replaced with identifier (see Appendix A). This greatly simplifies not only the user's job when entering programs but also the compiler since the illegal choices need not be considered.

```
     name ::= identifier
             | indexed_component
             | selected_component
             | function_call
             | slice
             | attribute
             | operator_symbol
```

Figure 2-2 Ada Name Production

Most of the other problems with the grammar are resolved by the user as he walks through the parse and chooses the type of production or construct he wants to use next. This means the compiler always knows what it is working on and does not need to do any backtracking.

2.6 SIMPLIFICATIONS TO THE ADA1 GRAMMAR

After the Ada1 grammar was developed, it was input to the META program (Ref 9). The META program is used to convert a grammar from its external, English form to an internal representation that the syntax directed editor and the other tools can use.

META does some checking of the input grammar. It ensures that the grammar is complete in the sense that no undefined non-terminals exist. It also checks to see if the grammar could be simplified by either eliminating unused productions or by combining two or more productions into a single production.

Unused productions are automatically eliminated. Unfortunately they are not tagged as such in anyway. Only a thorough analysis of the output from META would reveal those productions which were eliminated. A production is unused if its non-terminal is not used on the right-hand side of any other production. There is one exception to this rule. That is the first production in the grammar. This production is considered to be the goal or start symbol for the grammar and as such does not have to appear on the right side of a production.

META also points out several potential simplifications to the input grammar. These are pointed out as a single unconditional term, a single alternative, and an alternation alternative. Each of these potential simplifications is the result of the subset nature of the Ada1 grammar, and the causes are explained below.

2.6.1 A SINGLE UNCONDITIONAL TERM

The first potential simplification that META points out is a single unconditional term. This is a non-terminal that is replaced by a single terminal or non-terminal. For example A ::= B;. META is suggesting the grammar could be simplified by eliminating the production and replacing all occurences of the non-terminal A with B.

Several productions of this form appear in the Ada1 grammar. These productions are due to the way the Ada1 subset was developed. These productions are actually concatenations or alternations in the full Ada grammar. A decision was made to leave the subset as is since this makes future expansion of the subset and the compiler easier and it does not make the syntax directed editor any more difficult to use.

An example of a production of this type in the Ada1 grammar is the production for decimal_number shown in figure 2-3.

```
Ada1

    decimal_number = integer ;

Full Ada

    decimal_number = integer [decimal_part]
[exponent] ;
```

Figure 2-3 Decimal number

2.6.2 A SINGLE ALTERNATIVE

The second potential simplification pointed out by META is a single alternative. This is similar to the single unconditional term except that the production involved is an alternation rather than a concatenation. For example B = < C > ;. By pointing this out META is suggesting the grammar could be made simpler by making this production into a concatenation as B = C ; or by replacing all occurences of B with C and eliminating this production.

Several productions of this form appear in the Ada1 subset. These productions are again a result of the way the Ada1 subset was developed. These productions are actually multi-alternatives in the full Ada grammar. Since simplifying the grammar does not make the user's job any easier, the grammar was left unsimplified. This also makes expansion of the subset grammar and the compiler easier.

An example production of this type in the Ada1 grammar is the production for a designator, shown in figure 2-4.

```
Ada1

    designator = < identifier > ;

Full Ada

    designator = < identifier
                   op_symbol > ;
```

Figure 2-4 Designator Productions

2.6.3 AN ALTERNATION ALTERNATIVE

The third potential simplification that META points out is an alternation alternative. This results when a choice of an alternation is an alternation in its own right. An example is shown in figure 2-5a. META is suggesting that these two productions can be combined and the grammar simplified as shown in figure 2-5b. If the non-terminal C only appears in other alternations it can be eliminated from the grammar since after the combinations are done it will not appear on the right hand side of any production.

```
(a)
    A =  B
       | C ;

    C =  D
       | E ;
(b)
    A =  B
       | D
       | E ;

    C =  D
       | E ;
```

Figure 2-5 Simplification of an Alternation Alternative

Several productions of this form appear in the Ada1 grammar. These do not result from the way the Ada1 subset was developed but are actually caused by the Ada grammar itself and the way the productions must be formed for input to META. In this case simplifying the grammar probably would not make expansion of the subset grammar or the compiler any more difficult, but in most cases the simplification does not result in any true simplification since no productions are eliminated. Doing the simplifications can make the user's job somewhat more difficult since he must choose from a larger list of alternatives when a choice must be made.

The Ada1 grammar was left unsimplified to avoid potentially overloading the user with too many choices at any one time. Another factor that influenced this decision is that in some cases the extra information gained through the extra decision was very useful in the semantic analysis and code generation for a program.

An example of this type of production in the Ada1 grammar is the productions for primary and boolean_value and is shown in figure 2-6.

```
primary = < decimal_number
            name
            nested_exp
            char_lit
            boolean_value
            func_call > ;

boolean_value = < "true"
                  "false" >
```

Figure 2-6 Primary and Boolean Value Productions

3. TOOLS IN THE SUPPORT ENVIRONMENT

The tools of the support environment work together to accomplish a common goal. This goal is to simplify the program development process for the programmer. To do this efficiently the tools must communicate with each other. Since the tools work independently, the communication is done through a common data structure that is retained throughout the development cycle. This data structure is the abstract syntax tree representation of the program that the syntax directed editor creates as the program is entered.

This abstract representation of the program is used or manipulated by each of the tools in the support environment. The tools currently implemented are a syntax directed editor, a compiler, an interpreter/debugger and a program lister. Many other tools can be written to use the abstract representation of the program. These tools include code optimizers, cross reference routines, semantic analyzers, and execution analyzers. The nature of these tools is limited only by the imagination and skill of a user or group of users.

3.1 SYNTAX DIRECTED EDITOR

The syntax directed editor is the first tool encountered by a programmer. The syntax directed editor is used to enter a program into the environment. In this sense a syntax directed editor is a text editor. It differs from a standard text editor in its use and ultimate goal. A standard text editor allows the user to enter any text desired. The output of a standard text editor is the text that was entered. A syntax directed editor allows the user to enter text that is limited by the syntax or grammar of the language the program is written in. The output of the syntax directed editor is a syntactically correct program that the other tools can work with.

The syntax directed editor is not part of this thesis effort. It is an integral part of the programming environment and is used to create the abstract syntax tree the other tools of the environment use. Its use is necessary to be able to use the compiler and interpreter being implemented for this thesis effort. The syntax directed editor was originally developed by Scott E. Ferguson (Ref 9) and was moved to the VAX 11/780 by John Koslow.

3.1.1 USING THE SYNTAX DIRECTED EDITOR

To use the syntax directed editor the programmer must tell it the name for the new program and what language the program is to be written in. The syntax directed editor then creates a template of the syntactically legal constructs for a program in that language. The user must then "walk" around that template and choose the elements to be included in the program. With each choice the syntax directed editor replaces the object chosen with its own template. This process of replacement continues until the user is left with no more choices and the program is fully written.

For example in the Ada1 language a program is a compilation_unit. Since a compilation_unit is an alternation the user must choose whether to write a function or a procedure. After the choice is made the syntax directed editor displays the template that was chosen. It is this template that the user sees when he starts to enter a program. Suppose a procedure_body was chosen, the user descends the tree into the procedure_body and must satisfy the requirements of a procedure_specification. This is simply an identifier which is entered simply by typing in the name.

This "walk"/selection process continues until the program has been fully entered. The programmer can then exit the syntax directed editor or call the compiler, interpreter, or program lister. The abstract syntax tree representation of the program is stored by the syntax directed editor before an exit is allowed. A more complete explanation of the syntax directed editor can be found in the thesis written by Scott Ferguson (Ref 9).

The use of a syntax directed editor can have several advantages and disadvantages. These are outlined below.

3.1.2 ADVANTAGES OF A SYNTAX DIRECTED EDITOR

The use of a syntax directed editor leads to several advantages that can help later in the program development cycle. The first and most important advantage is the compiler does not have to do any syntactic analysis. This can dramatically speed up the compilation process since the compiler does not have to recreate the syntax tree whenever the program is recompiled. It also allows the compiler to concentrate on other aspects of the compilation process. These aspects include semantic checking, error recovery, and code optimization. The compiler essentially becomes a semantic analyzer and a code generator.

A second advantage of the syntax directed editor is the fact that it is independent of the language the programmer is using. This means it does not have to be rewritten for each new language a programmer wants to use. Only the language's syntax need be created and input to the META preprocessor described earlier. This also means a programmer does not have to remember how many different editors work since the same editor will be used for each project. This has one other advantage since only one editor is needed less secondary storage is needed to keep editors online.

A third advantage of a syntax directed editor is in the more efficient use of the computer. Several aspects of this have already been pointed out. The first is the compiler does not have to reanalyze the program each time it is recompiled. Another is that the programmer does not have to wait for a printout to be able to go in and fix an error. This can save not only time but other resources such as paper or wear and tear on a printer.

Another more efficient use of the computer is achieved by a syntax directed editor since the CPU has functions other than waiting for a user to make inputs to perform. Since most CPU's can handle data much faster than a user can enter it, the CPU tends to sit idle when it could be doing other tasks. One of these tasks is the creation of the various syntax nodes being put to use by the programmer.

Another task the CPU could be doing is background compilation. The compiler or some portion of it could be running as a background task to the syntax directed editor. A major problem must be overcome before this is attempted. That problem is how the editor and the compiler communicate with each other to avoid the editor changing the program that the compiler has already generated code for. Also the editor must ensure it does not destroy a subtree the compiler is working while the compiler is analyzing it. For these reasons and a lack of time this feature was not implemented.

3.1.3 DISADVANTAGES OF A SYNTAX DIRECTED EDITOR

A syntax directed editor can have disadvantages. The first of these is the user must have much more knowledge of the grammar and syntactic structure of the language the program is written in. This is due to the way the user must interact with the syntax directed editor to indicate what template to use next. This aspect will be further discussed in the next chapter. This interaction may also force a user to receive extra training and it may take some users longer to learn how to properly use the new tool.

The other main disadvantage to a syntax directed editor is the syntax tree can take up to eight times more secondary storage than its text based counter part. This is due in large part to the amount of information contained in the syntax tree that is not needed or maintained by a text based system. (Ref 9). This is becoming less and less of a problem as the cost of hardware and secondary storage in particular decreases.

3.2 COMPILER

The compiler in this environment becomes a simple tree walking routine. It systematically walks around the tree hanging code fragments for the node it is currently on. The code fragments are generally hung only on the leaf nodes of the tree.

A function is developed for each type of node that the tree might contain. This makes the compiler somewhat easier to modify or expand since only the functions for the productions that changed must be modified. An exception to this occurs when a change like adding data types to the language is made. All functions must be examined to see if type checking is needed. One way around this is to write a separate semantic analyzer that pre-walks the tree hanging type and symbol table information onto the nodes. The current implementation of the compiler is explained in more detail in the next chapter.

3.2.1 INCREMENTAL COMPILATION

The compiler can also use this structure to save information that can be used when a program is recompiled. If the nodes are marked to indicate a change was made that affected that node then the compiler can reuse any information that is associated with unchanged nodes.

Since ADA is such a structured, type oriented language this feature was not implemented. This was to ensure the semantic correctness of the program being compiled, since changes in declarations and external packages can have unknown affects. Such changes can even affect areas of a program that have not been changed for a long time.

3.2.2 ERROR DETECTION AND RECOVERY

Using the abstract syntax representation of the program and knowing that the program is syntactically correct makes error detection and recovery done by the compiler much easier. Since the compiler knows at all times what construct it is working on, no error recovery in the usual sense is needed. All the compiler needs to do is ascend the tree to a node above the error and continue as if the subtree with the error is ok. This allows the compiler to do more extensive error checking during the initial compile and also helps eliminate the usual stream of false errors that are caused while the compiler tries to resynchronize itself.

The only limitation on this error detection is the maximum number of errors a user wants to allow before the compiler quits. Since error flags are stored in the syntax tree itself or in a similar structure, the iimiting factor is how much storage is available for these errors. These error flags have pointers into the syntax tree at the node with the error. The syntax directed editor can then be invoked and will point to the first erroneous node. The editor can also have the error message available which relieves the programmer from having to wait for a printout. This will save the programmer time and allow programs to be written and debugged faster.

3.3 INTERPRETER

The interpreter or run time mechanism can use the information stored in the syntax tree to allow the input and output of enumeration types, to do range checking, and any other run time checks that might be desired. In many cases these checks can be done dynamically by the interpreter without the need for extra code being generated by the compiler. This can save time in the compile process and also means that it will be easier to override these run time checks since they only need be turned off in the interpreter and no recompilation is needed.

3.4 A DEBUGGING TOOL

An interactive debugging tool can be used to trace the execution of a computer program. Debugging tools exist but they are usually limited in their capabilities. Using information contained in the syntax tree, the debugger can show the programmer exactly what is happening during execution. This can help pinpoint an error and show the programmer exactly where it is. This is especially helpful if the error is in the middle of a complex statement or if the error does not cause the program to terminate. The current implementation combines the interpreter and debugging tools into a single tool. This is not the only way it could be done, but due to the simple nature of the debugging tool it was the easiest way to get them working.

3.5 PROGRAM LISTER

A program lister uses the information stored in the syntax tree and the syntax description file to produce nicely formatted listings. Using other information in the syntax tree a lister could be written to generate cross reference listing and other useful outputs. If done properly these listings are more detailed than the usual listings produced by a compiler.

An added feature of this lister is that it is usually independent of the programming language in use. This is also true of the syntax directed editor and means that only the compiler and grammar need to be rewritten for the syntax directed editor to work on another programming language.

Like the syntax directed editor, the lister was not directly part of this thesis effort. It is part of the effort done by Scott Ferguson and moved to the VAX 11/780 by John Koslow. It was used during this thesis effort to produce listings of the programs used to test the compiler.

3.6 CODE LISTER

The code lister is a simple tool to extract the code generated by the compiler and transform it into a readable format. The code lister is newly implemented as part of this thesis effort. It is dependent only on the set of "executable" instructions being used by the compiler and interpreter. It is the only tool of the support environment that does not use the abstract syntax representation of the program.

4.0 THE ADA1 COMPILER

The Ada1 compiler can be split into three interacting but separate parts. These are the semantic analyzer, the symbol table routines, and the code generator. The semantic analyzer is used to hang the types onto expressions and variables. The symbol table routines are used to insert and look up names in the symbol table. The code generator is used to produce code for the program.

These three parts work together to compile an Ada1 program. The code generator is the controlling program and it calls the other two as needed. The semantic analyzer is used to preview an Ada1 expression and determine its type. This preview makes the generation of code much easier. The symbol table routines are called by both the code generator and the semantic analyzer as necessary.

This chapter will discuss the implementation of the code generator and the symbol table routines. A proposed design for the semantic analyzer will be presented in the next chapter.

Since no means currently exists for specifying semantic actions in the META syntax description, the code generator must be coded separately. Fortunately the code generator merely needs to walk the tree building the symbol table and generating code for the pseudo-machine.

4.1 PROGRAM TREE WALK

The need for a parsing step is eliminated since the structure and syntactic correctness of the program tree is assured. Walking the tree provides access to the syntactic elements of the program. Since the code to walk the tree models the syntactic structure of the language, each non-terminal in the grammar maps into a function to evaluate and validate its subtree. The function is passed a single argument which is the root node of the subtree. Thus to start the compilation process the root node of the tree is passed to the function goal.

Since a non-terminal can be either a concatenation or an alternation, two basic function structures are used.

4.1.1 CONCATENATION NODES

For a concatenation non-terminal, the function consists of a series statements to analyze and validate each child node in turn. Non-terminals are processed by calling subroutines to analyze and validate their subtrees. Terminal strings and sets are processed for their value. Terminal strings present in the grammar are not present in the abstract syntax tree so they present no real problem. For example the syntax for a function body is given in figure 4-1. Sample code to accept this syntax is shown in figure 4-2.

```
func_body =
        func_spec  "is"
                { decl }
                { program_component }
        "begin"
                seq_of_stmts
        "end"  [ identifier ] ";"
```

Figure 4-1 Function Body Syntax

```
procedure  FUNC_BODY(node : tree_node);

    var  child : tree_node;

begin
    child := first_child(node);
    FUNC_SPEC(child);
    child := right_sibling(child);
    while (node_type(child) = "decl")
        begin
            DECL(child);
            child := right_sibling(child);
        end;
    while (node_type(child) = "program_component)
        begin
            PROG_COMP(child);
            child := right_sibling(child);
        end;
    SEQ_OF_STMTS(child);
    child := right_sibling(child);
    if (node_type(child) = "identifier")
        old_ident(child);
end;
```

Figure 4-2 Function Body Accepting Procedure

Required non-terminals like func_spec and seq_of_stmts are processed by their own functions. The decl and prog_comp repeaters are processed in while loops for as many such nodes as exist in the func_body. The optional identifier at the end of the func_body is processed by an if statement. Unestablished optionals and repeaters are ignored since the program is correct without them.

4.1.2 ALTERNATION NODES

An alternation non-terminal is processed by a case or switch construct. The case is based on the type of the non-terminal's only child. One case is used for each possible alternative. If no child exists due to an unestablished alternative, the incomplete program fragment is reported as an error.

4.1.3 SIMPLE NON-TERMINALS

A third type of non-terminal exists in the subset. This is a non-terminal that is replaced by a single non-terminal. These non-terminals are processed by a simple call to the function that processes the second non-terminal. For example

```
stmt =
        simp_stmt ;
```

This is processed by the function

```
function stmt(node : tree_node) return integer;

begin
 stmt := simp_stmt(son(node));
end;
```

This has several causes. In the above example the subset eliminated an optional label from the production. The production was left as is to allow easier expansion of the subset to the full Ada grammar. The full Ada stmt is

```
    stmt =
          [ label ]  simp_stmt  ;
```

Another cause is shown in the definition of a proc_decl.

```
    proc_decl =
            proc_spec_semi  ;
```

This is actually an alternation in the full Ada grammar. Once again no reduction was made to allow for easier expansion of the compiler. The full production is

```
    proc_decl =
          < proc_spec_semi
            generic_proc_decl
            generic_proc_instant >  ;
```

4.1.4 COMPILATION IN PIECES

Since the entire tree is available during the complete compilation process, it need not be accessed in a strict linear fashion just described. It could be compiled in pieces with each completed subtree being pass by the editor to the appropriate subroutine for processing. This would allow the compiler to run background to the editor, which would improve the efficiency of the entire system.

4.2 ERROR HANDLING

The syntax directed editor ensures the syntactic correctness of the program . It is still possible for the program to contain semantic errors. It is the compiler's responsibility to detect and report these errors. Two examples of semantic errors are an undeclared identifier and an identifier of the wrong type in an expression.

4.2.1 ERROR RECOVERY

Error recovery is usually a difficult process for a compiler, since it is trying to check the syntax as well as the semantics of a program. When it encounters what it thinks is an error it must check if using a different syntax would eliminate the error. Also the recovery process itself may cause new errors to be detected since the compiler must guess where to restart the compiling process.

Due to the assured syntactic correctness of the program, the error recovery function is all but eliminated. Semantic errors are easy to recover from, since they tend to only affect a relatively small part of the entire program. The recovery process involves patching the code that is generated, reporting the error to the user, and marking the erroneous node. Marking the node allows the syntax directed editor to detect it as erroneous and move the focus to that node to allow the programmer to make the necessary correction.

The fact that an error was encountered is also passed back up the tree through the returns from subroutine calls until a subroutine is found that can continue in spite of the error. Thus error recovery is built right into the functions themselves and poses no particular problems except how to determine when an error no longer has an affect.

For example if the symbol table is searched for an identifier and it is not found, the error undeclared identifier is generated. This error only affects the expression in which the identifier is being used. The compiler reports the error and then generates code to load a 0 value instead of the real value and processing is allowed to continue. By generating this code, execution of the program could actually take place although the results would probably be invalid. This example might result in the expression being of the wrong type and further errors reported.

4.3 SYMBOL TABLE

The symbol table is used by the compiler to store information about the names in the program. As implemented the names are left in the abstract syntax tree and a pointer to the identifier node is stored in the symbol table. The characters of the name are distributed as children of the identifier node. Comparison of two identifiers is done by pattern matching the two identifier subtrees. This leaves the storage for names in the syntax tree and conserves memory since an identifier is stored in only one place.

4.3.1 SYMBOL TABLE STRUCTURE

The symbol table structure is one area that required major revisions to do the desired expansions. The symbol table data structure was revised to hold much more information about a symbol. This was required since new types were introduced and functions were allowed. Also fields were added to allow parameters for functions and procedures and to allow overloading to be implemented at a later date.

The symbol table is implemented as an array. This is a simple method that allowed for easy implementation of packages. An array of integers is very easy to read and write to disk to allow the visible part of a package to be saved when it is compiled and then read back in when it is used. This saving of the symbol table is necessary to avoid the necessity of recompiling the package each time it is used. This reading and writing of the symbol table is the main extension to the symbol table that was implemented.

The new data structure is shown in Appendix D. A discussion detailing the use of each field will also be found there.

4.3.2 SYMBOL TABLE ROUTINES

The symbol table routines did not require any major modifications since overloading was not implemented. Overloading would require the lookup routine to find all visible occurrences of an identifier. These would be linked together and passed back to the compiler. Since overloading is not allowed only the first occurrence of an identifier is found and the symbol table index is returned.

4.4 CODE GENERATION

The code generated by the compiler is for a pseudo-machine similar to the PL/0 interpreter written by Niklaus Wirth (Ref 18). Code is generated as the compiler walks the tree. Each instruction generated contains a pointer to the abstract syntax tree node which is thought to be responsible for the instruction. This pointer is used by the interpreter to dynamically show the programmer what part of the program is being executed.

4.5 THE INTERPRETER

The interpreter can be called by the syntax directed editor or executed independently. The interpreter calls the compiler to compile the program and if no errors are detected the program is executed. Using the information in the code element, the interpreter is able to highlight the program tree display to show where in the program execution is currently occurring. The highlighted portion of the tree moves around as instructions are executed to trace program execution. The interpreter also displays the top few elements of the run time stack and the next instruction to be executed.

4.5.1 MODIFICATIONS TO THE INTERPRETER

The interpreter required several revisions to get it to run on the Vax 11/780. These revisions were due mainly to differences in the way the micro-computer operating system required space to be allocated and the way the Vax required it.

Code for some of the pseudo-instructions of the run time machine had to be modified to accommodate differences in the way the two computers and their C compilers handled expressions. For example the _SUB instruction was coded

```
     push(-pop() + pop())
```

This seems to be correct but the result of execution as shown by running a program with a subtraction in it was incorrect on the Vax. The code was changed to

```
i = -pop();
j = pop();
push(i+j);
```

This code seems to do the exact same function but the results of the two are different. The first set of code actually gives the negative of the correct answer. The second set of code gives the correct answer. The reason is unknown but probably lies in a different implementation of the C compilers used on the two machines. The code to evaluate the relational instructions; LES, LEQ, GRT, GEQ; were also changed in this manner.

They were changed in this manner rather than directly manipulating the stack in order to ensure stack integrity by using the stack manipulation functions, push and pop. Another method that could have been used was to change the operator being used in the interpreter. Thus SUB instruction would have become

```
(push(pop() - pop());
```

This would not be portable to other machines nor would it be clear as to why the apparent order of evaluation was changed. To avoid this ambiguity, this method was not used.

The other changes that were made to the interpreter involved the addition of instructions to handle the new statement forms added to the subset.

4.6 CURRENT IMPLEMENTATION AND EXTENSIONS

The compiler as implemented by Scott Ferguson was for a limited subset of Ada. Various extensions and modifications were made to this subset. The extensions include new predefined operators and functions. The modifications to the current compiler were done to allow predefined data types other than integer to be used.

4.6.1 BOOLEAN OPERATORS

The new predefined boolean operators added to the compiler are the AND_THEN, OR_ELSE, and XOR. The AND_THEN and OR_ELSE presented several problems in their implementation. Though the implementation is similar to the other Boolean operators in form, they had to generate code much differently since their intent is as short-circuit operators.

The AND_THEN operator must evaluate its left operand and if it has a value of true the right operand is evaluated. If the value is false then the right operand is not evaluated.

Similarly the OR_ELSE operator must evaluate its left operand and if the value is false it must evaluate the right operand. If the value is true then the right operand is not evaluated.

locations of the instructions be saved until the entire expression is evaluated so that the code can be fixed to jump to the correct location.

This was done by creating two new instructions for the interpreter, _AND_THEN and _OR_ELSE. The code was also generated one node higher in the tree than for the other Boolean operators. This allowed the locations for the branching instructions to be linked together and then fixed after the expression was fully analyzed. See Appendix E for the code.

4.6.2 INTEGER OPERATORS

The other new predefined operators added were REM and MOD. These were rather easy to include in the compiler since they only involved adding new case values to existing functions. They did cause some problems for inclusion in the interpreter since C does not have a REM function. This meant code for the REM function had to be written. After some experimentation using the examples in the Ada reference manual (Ref 7), a formula was devised to calculate the correct value. See Appendix E for the code.

4.6.3 SUBPROGRAMS

The Ada0 subset included only procedures. The Ada1 subset was expanded to include functions as well. This expansion required the code used to analyze and compile procedures to be changed because the Ada0 subset did not have a return statement. Since Ada requires all functions are required to have at least one return statement, the return statement had to be added to the Ada1 grammar. Since the code required to handle functions and procedures is very similar they are discussed together.

Functions were rather easy to include in the extended compiler because a function is essentially a procedure which returns a value. Procedures were already included, so the extension amounted to modifying existing code to fit the function syntax. It is the returning of a value that requires the use of a return statement.

The code to compile a procedure also had to be changed to accommodate the return statement. Two changes were made. The first was to indicate a procedure was being compiled. The other was to fix the code generated by the return statements. Since procedures do not require a return statement no code was written to verify the existence of a return statement only to fix the code generated if any exist.

The return statement caused most of the problems, mainly because it could appear anywhere in the body of a function or procedure. These problems included determining the type of subprogram currently being compiled, where to put the value being returned, how to verify its type, and how to link the return statements to the end of the subprogram so that the stack clean up steps could be done properly.

4.6.3.1 SUBPROGRAM RETURN

When the return statement is compiled it is necessary to know if the return is from a function or a procedure since a function return must include an expression while the procedure return return cannot.

Since the return statement can only appear within the statement body and the type of the statement body cannot change until the body is complete, an external pointer can be used. An external symbol table pointer is set to point to the name of the current function or procedure being compiled. The pointer is set just before the body is compiled. When a return statement is encountered the subprogram type of this name is checked to see if a function or procedure is being compiled. Appropriate actions are taken in each case. The code to handle a return statement is shown in Appendix E.

4.6.3.2 VALUE RETURN FROM A FUNCTION

The problem of how to save a value being returned from a function was also rather easy to solve. The calling routine expects the value to be on top of the stack when the return is made therefore code was generated to ensure the value being returned appeared where it was expected. The solution involved a two step process and is limited to handling simple data types. The solution can be easily expanded to handle more complex data types but since they were not in the subset the extensions were not made.

The first step was to create a stack entry for the return value before the function was actually called. This ensured the value would appear on top of the stack upon return and it allows the function to do its normal stack clean up with out worrying about what to do with the returned value. This is done by loading a 0 value onto the stack. To expand this to data structures a 0 could be loaded for each element of the structure.

The second step was to store the value into this location. This location is a -4 offset from the function's stack base. Thus a store instruction with offset -4 is generated just before the stack clean up and subprogram return instructions are generated. To expand this to handle data structures a similar store instruction could be issued for each element of the structure.

Several problems with this method exist. These are a function return statement that does not return a value, return to the operating system, and return type validation. These problems were handled rather easily.

A function return statement that does not return a value is in error and must be reported as such. This is handled quite easily but the user may want to allow execution to occur in spite of the errors. Partial execution is possible and if the statement which caused the errors is not executed the program may actually give correct results. To allow partial execution to occur an instruction is generated to put a value onto the stack in place of the value that is expected. The value used is 0 but to cause termination the undefined value could be used. Another solution would be to generate an abort statement with an appropriate error message.

To allow return to the "operating system" the interpreter was modified to push an extra 0 onto the stack to account for the value being returned by a function. This can occur when a function is being written as a separate entity to be used by several programs. Though no method of separate compilation is implemented, a function could be written and tested independently with this change.

The third problem is type validation of the value being returned. The type must be checked when the expression is compiled and its type is known. The solution is discussed in the next section.

4.6.3.3 RETURN TYPE VALIDATION

The type validation problem is also rather easy to solve. It is actually a two part problem. The first part is handled by the compiler to check that the expression is of the correct type. The second must be handled by the interpreter to verify the value being returned is in the correct range for a subtype.

The compiler can do its checking using the symbol table pointer discussed earlier. This pointer gives access not only to the functions name but also t the type it must return. Using the pointer the expected and actual types are compared. If they are the same the type is returned. If not an error message is generated and error is returned. This problem will be handled by the semantic analyzer when it is implemented.

The problem of range checking is handled by the interpreter and was not implemented since subranges were not in the subset. The solution is rather easy and is given now. The compiler must generate code to load the upper and lower bounds of the range onto the stack and an instruction to cause the value to be checked. The interpreter must then execute these instructions and leave the value on the stack if it is in the range or generate a run time error if it is not in the range.

4.6.3.4 LINKING THE RETURN TO THE END OF THE SUBPROGRAM

Linking the return statements to the end of the subprogram was the most difficult problem to solve. Two potential solutions were investigated.

The first potential solution is to generate the code to do the stack clean up, subprogram return and value store each time a return statement is compiled. This is easy to do but requires the symbol table to be modified to hold the number of names declared in the subprogram. It also requires 2 or 3 instructions to be generated for each return statement.

The other method is to generate a jump instruction to transfer control to the end of the subprogram where stack clean up, subprogram return, and value store will be done. This requires the compiler to do some extra work since the jump instructions must be linked together to allow them to be fixed when the subprogram end is found. This requires the compiler to keep a list of the jump instructions. The jump instructions themselves can be used for this purpose since the operand fields are not used until the instruction is fixed. Only a single new variable is needed to keep track of the return list. This variable is needed anyways to indicate whether a subprogram has a return statement or not.

Since the second method was rather easy to implement and required less memory for a subprogram with more than one return statement it was chosen. See Appendix E for the code written to implement these features.

4.6.4 DATA TYPES

The modifications to the Ada0 compiler to allow new types to be added were quite extensive. The Ada0 compiler uses functions that return one of four possible values, SUCCESS, ERROR, a symbol table index, and a value. Only functions that returned SUCCESS had to be changed, to return a type value. Code also had to be added to verify the returned type was alright in the context it appeared.

For example in the if_statement the expression must be a boolean expression. The syntax for expression allows any type expression to be entered so the expression analyzer had to be changed to return a type and the if_statement analyzer had to test that type. A full list of the modified functions is shown in Appendix xx.

A simple semantic analyzer was embedded in the compiler. It allowed type checking but could not handle the problems of overloading. This approach was used since a separate semantic analyzer will be almost as large as the compiler itself and when it is written it should take overloading into account and solve the problems overloading causes. Since overloading was not allowed in the Ada1 subset, this problem was not addressed except to design a mechanism to handle overload resolution. That design is presented in the next chapter.

4.6.5 OTHER MODIFICATIONS

Other modifications had to be made to the compiler. These involved differences in how the micro-computer which Ferguson used and the VAX 11/780 do things. These changes were minor but difficult to pin down since they did not appear until the compiler was actually being tested. The main area of concern was how memory is dynamically allocated to a program. The way Ferguson had allocated memory to the Ada0 compiler should have worked on both systems but it did not. The compiler did not catch the problem since it was not a syntax or semantic error but a difference in how the functions involved were implemented. This is a prime example of how a standardized language and implementation, like Ada, would have saved time and effort.

The second change was in how the various tools of the environment called each other. This again was not caught until the system was tested as a whole. The problem was in the function, EXECL, which used its arguments differently on the two systems. Again the change was minor but still a change that should not have been necessary. To complicate matters neither system documented their usage well and the changes had to be made through experimentation.

The third area of change was due to changes in the Ada syntax as presented in the Ada standard (Ref 6). The Ada grammar was changed in several subtle areas between the preliminary Ada reference grammar (Ref 7) which Scott Ferguson used, and the Ada standard which I had to follow. Most of the changes were in areas that did not affect this thesis effort, but one change did.

This change is the introduction of a new level of precedence for operators, the highest precedence operators. The newly defined highest precedence operators are an exponentiation operator, the NOT operator, and an absolute value operator, ABS. In the preliminary reference grammar the NOT operator was defined as a unary operator, while the other two were not defined at all. The NOT and ABS operators are unary operators but they are given a higher precedence than the other unary operators defined in the Ada grammar. This change in precedence required several changes to the grammar and changes to the compiler to handle the new syntax.

5.0 DESIGN OF THE NEW TOOLS

Several new tools were designed as a result of this thesis effort. The original intent was to also do the implementation of these tools but time limitations made this impossible.

The new tools designed are a semantic analyzer, a method for passing parameters to subprograms, and data structures to allow user defined data types.

5.1 SEMANTIC ANALYZER

A semantic analyzer is needed to resolve overloading of names in Ada. Languages like Pascal do not need a separate semantic analyzer because names cannot be overloaded as they can in Ada. A simple symbol table search is all a Pascal compiler needs to do since only one instance of a name is visible at anytime. Either a name is in the symbol table or it is not and the compiler does its analysis accordingly. An Ada compiler must also consider the context in which a name appears because all instances of a name are potentially visible.

Since this analysis can be quite complicated a separate analyzer is proposed and designed.

5.1.1 THE BASIC SEMANTIC ANALYSIS ALGORITHM

The semantic analyzer is called by the compiler when the compiler needs to know the type of an expression. The compiler passes the root node of the expression to the semantic analyzer. The semantic analyzer uses this node as the root of the tree it must analyze. The semantic analyzer walks this tree much like the compiler would and determines the type of each component of the tree.

The analysis takes place in three phases or passes. This is all the analyzer requires to analyze and completely type an expression (Ref 2). This analysis will result in either a valid expression with each node of the tree being typed or an invalid expression. An invalid expression is one that either does not have a valid interpretation or has more than one valid interpretation.

5.1.1.1 PASS ONE

The first pass is a top down pass that hangs the desired types onto each node of the tree. This is done by analyzing expression beginning with the operator that will be executed last. This operator is the rightmost operator with the lowest precedence in the expression. Since the tree being analyzed is an operator precedence tree the rightmost operator at this level in the tree is chosen. The operator is analyzed and right and left operand lists are developed for it. If either of these lists is empty an error is reported and analysis can stop. If both lists are non-empty they are passed to recursive calls to pass one to analyze the operands as expressions themselves. This process continues until a leaf node is reached at which time pass two is called.

5.1.1.2 PASS TWO

The second pass is a bottom up pass that delivers the available types for an expression based on the types available in the tree below. These available types are compared with the list of desired types for the node and a new list is created. This new list consists of the types that are on both the available list and the desired list. If this list is empty an error is reported and analysis con stop. If the list is non-empty, it is hung on the node replacing the desired list and it is passed back up the tree to the expression above. This process continues until the root node of the expression is reached at which time two possible results can occur, a single valid type or the expression has multiple valid types.

If the expression has a single valid type pass three is run to resolve any ambiguities that still appear in the tree. If the expression has more than one valid interpretation an error is reported and analysis is stopped.

This is the only point at which multiple valid interpretations cause an error to be reported.

5.1.1.3 PASS THREE

The third pass is another top down pass that must resolve any remaining ambiguities. Pass two could be written to indicate that no ambiguities appear below a given node thus making analysis by pass three unnecessary. If at any time during this pass an ambiguity cannot be resolved the expression is invalid and an error must be reported. No further analysis will resolve the ambiguity since previous analysis has shown the interpretations to be valid and no new restrictions have been introduced.

5.1.2 DATA STRUCTURES USED

The semantic analyzer requires two different data structures to do its analysis. Both structures are linked lists but their contents are somewhat different.

5.1.2.1 OPERAND LIST

The first structure is the operand list. This list contains the currently valid type or types for the operand or name in question. The structure consists of a type and a pointer to the next element of the list.

5.1.2.2 OPERATOR LIST

The other structure is the operator list. This list contains the currently valid interpretations of the operator. This structure consists of five elements. The valid type for the left operand, the corresponding valid type for the right operand, the type this operator will return, the symbol table index of the operator, and a pointer to the next element in the list.

The symbol table pointer for a predefined operator is set to "-1". This is because the operators are not in the symbol table but will be recognized by the code generator from the types of the operands. This will also save time during execution since no function call is made to evaluate the operator.

5.2 PARAMETER PASSING

Parameter passing to functions and procedures in Ada can be extremely complicated. The complications result from the many diverse methods of passing parameters that are allowed in Ada. Ada allows parameters to be in, out, or in out parameters. They can be passed positionally, by name, by default value, or any combination of the three.

Due to the complex nature, a full parameter passing mechanism was not designed. Only positional parameter passing is handled but the mechanism does not preclude enhancement to handle the other methods. Positional parameter passing is complicated because of the various modes a parameter is allowed to have.

5.2.1 PARAMETER MODES

Parameters in Ada can be given one of three modes. These modes are IN, OUT, and IN OUT. If no mode is given in the declaration of a parameter it defaults to the IN mode. A short explanation of each of these modes is given below.

5.2.1.1 IN PARAMETERS

In parameters are treated as local constants inside the subprogram. They have a value associated with them when the subprogram is called and they cannot be updated by the subprogram.

5.2.1.2 OUT PARAMETERS

Out parameters are treated as local variables inside the subprogram. They do not have a value associated with them when the subprogram is called but they are allowed to be assigned a value. In fact the subprogram is considered to be in error if an assignment is not made. The value associated with an out parameter when the subprogram returns is copied back into the location of the actual parameter that was used in the call. For this reason an out parameter must be a variable name and not an expression.

5.2.1.3 IN OUT PARAMETERS

In out parameters are also treated as local variables inside the subprogram. They do have an initial value associated with them when the subprogram is called and they are allowed to have a new value assigned to them during the execution of the subprogram. In out parameters do not have to be updated by the subprogram. The value of an in out parameter is also copied back when the subprogram returns. In out parameters must also be variables and not expressions.

5.2.2 PARAMETERS TO FUNCTIONS

Since functions are not allowed to produce any side effects other than returning a value, out and in out parameters are not allowed. This makes functions relatively easy to handle since each parameter must have an initial value and that value is then pushed onto the stack. The compiler must make certain that the formal parameters are not updated or used as out or in out parameters to another procedure. This is relatively easy since the parameters will be marked as constants and the compiler ensures that a constant is not updated. Also since the values associated with the parameters are not needed after the function returns the stack can be cleaned up by the function return and the returned value(s) left on top of the stack.

5.2.3 PARAMETERS TO PROCEDURES

It is procedures that cause the majority of problems since they are allowed to have out and in out parameters. In parameters are treated the same as functions. Their value is pushed onto the stack and then the procedure references it from there. This method was chosen for in parameters since the value is known when the function is called and the actual parameter can be an expression which the run time machine will evaluate and leave the value for it on top of the stack. Since that is where we want the value to be, no special treatment is needed. Also if default values are allowed they can be pushed onto the stack if the actual parameter is not found for the call.

Out parameters cause many problems of their own. Since they must receive a value during execution of the subprogram the run time mechanism must be changed to verify all out parameters were updated. Perhaps the easiest way to do this is to reserve one bit pattern as an undefined value. This value could then be detected by the run time mechanism if it was ever used as a valid value.

The value that is the easiest to use is the smallest negative number or the largest positive number. This reduces the values a program can use but the restriction is easy to live with since numbers of that magnitude seldom get used in a program.

5.2.4 DATA STRUCTURES AS PARAMETERS

Structures like arrays and records can cause problems. They can be handled in two ways. The first is to simply push all the values onto the stack and reference them from there. The other idea is to push a pointer at the first element of the array or record and reference them indirectly. Both methods have their advantages and disadvantages.

Pushing the structure onto the stack has the advantage that only that procedure or function can address that copy of the variable. This is especially useful if the structure is an in parameter and it can be addressed globally as well. The standard says that any program that does that is in error since a "constant" (the formal parameter) was updated during execution of the subprogram. The handling of this problem seems to be beyond the scope of this thesis effort and was not addressed.

The other problem that can arise is if the actual parameter is visible to more than one concurrently executing task. In this case the indirect method could produce varying results depending on how quickly each task executed. In this case the program is also in error. Since tasks were not included in the Ada1 subset, this problem was not addressed.

Pushing all values onto the stack has its drawbacks as well since it "wastes" stack space. With a limited stack this could present a problem for a large program. Access through indirection does not have this problem but it does have the problems discussed in the previously. Since the former problem is much more difficult to solve than running out of stack space, the method of pushing all values onto the stack was chosen. If programs are written that exhaust the stack space, the user must simply recompile the interpreter and increase the stack space or rewrite the stack handler to allow linked stacks.

5.2.5 PROPOSED SOLUTIONS

The Ada standard (Ref 6) does not specify how out and in out parameters are to be handled. They can either be passed by value (undefined) and then update the actual parameter when the procedure is complete or they can be passed by reference and be updated as the procedure executes. To avoid some of the other problems, pass by value and update on return was chosen.

Structures and simple names present no real problems and are handled easily. This is because their addresses are known at compile time and the code to do the load and update can be generated correctly. The parameters that cause problems are the indexed names like a single array element. Since the index is allowed to be an expression whose value is not known until execution, the location to update is not known until run time. This is no problem for the call but only for the return.

5.2.5.1 INDEX RE-EVALUATION

The first method that comes to mind is to evaluate the expression and push the value onto the stack then load the value using the index to get the value to be passed. For the return the same expression can be re-evaluated and the index used for the store instruction. That looks good but what if value of the expression for the index has changed. This can happen if one of the variables in the expression is used as an out or in out parameter to the procedure or if it is used globally by the procedure. This method was not used.

5.2.5.2 INDEX LEFT ON STACK

The second method that came to mind was to leave the value of the index on the stack after the load and then use it when the procedure exits and the new value is saved. A minor problem with this is that all parameters below this one are now offset one too many and the references to them will be incorrect. A solution seemed to be to change the offset value in the symbol table but this prevents recursion since the second call has no way of saving the offset values from the first call. This method was not used.

5.2.5.3 MODIFY SYMBOL TABLE

A third method that was investigated was to add the new elements to the symbol table and then have the run time mechanism reference them through the symbol table. This would work but would require a major rewrite of the run time machine and the symbol table handler. For these reasons this method was not used.

5.2.5.4 USE A SECOND STACK

The fourth method that was investigated involved the use of a secondary stack by the run time machine. The second stack is used to save the index values for out parameters. It is only used when a procedure is invoked that has an out or in out parameter(s) and the actual parameter used in the call is an indexed_component. This required a minor rewrite of the run time machine but since several new instructions were being added, the rewrite was inevitable. The rewrite does not change any of the instructions currently implemented but was limited to adding code to interpret the new instructions. Since no major problems were found with this solution, it was designed. The basic algorithm is given in Appendix C.

This is a workable design and it does not preclude an extension to allow named parameter passing or default parameter values to be used. This design also allows recursion and seems to solve many of the Ada related problems. One change seems necessary to this algorithm. That is to handle overloading of names. This problem is solved by using the semantic analyzer to resolve the ambiguities and only use this algorithm as a code generator after all ambiguities have been resolved. The steps that verify correct types and number of parameters can then be deleted since the procedure will be semantically correct before this algorithm is used.

5.3 DATA STRUCTURES

The data structures to implement user defined data types and allow for package elaboration are shown in figure 5-1 and 5-2. A short explanation of how these structures were developed and how they are used is given below.

5.3.1 USER DATA TYPES

The user_data_type structure, shown in figure 5-1, is used to store information about a user defined data type. Eventually all data types will be "user defined", since the package standard will be used to pre-load the symbol table with the predefined types and functions. In the current implementation the predefined types and functions are handled as special cases by the compiler. They are recognized and handled properly.

```
struct  user_data_types
 {
  int   name,         /* pointer to name in tree */
        base_type,    /* pointer to base type    */
        first,        /* pointers to first and   */
        last,         /* last enumeration lits   */
                      /* for enumeration types,  */
                      /* fields for records,     */
                      /* indices for arrays      */
        index_type,   /* type of index elements  */
        num;          /* number of values for    */
                      /* enumeration types,      */
                      /* size of an array or     */
                      /* record                  */
   unsigned flags;    /* binary flags explained  */
                      /* below                   */

#define IS_ARRAY   0x01  /* type is an array    */
#define IS_RECORD  0x02  /* type is a record    */
#define IS_ENUM    0x04  /* enumeration type    */
#define INDEX_INT  0x08  /* indices are integer */
```

Figure 5-1 User Data Type

5.3.1.1 NAME FIELD

The name field is a pointer back to the node that created the new type. The children of this node are the name of the type. This is done to allow to data type names to be compared in a manner similar to the one used to compare to identifiers.

5.3.1.2 BASE TYPE FIELD

The base_type field is used for arrays and subtypes to indicate the type of the elements of the array or the base type of the subtype. Since this is a pointer into the abstract syntax tree, it is possible to define an array of arrays. This construct is left as illegal in the Ada1 subset but since this is how Ada defines multiply subscripted arrays, extending this data structure should be easy.

5.3.1.3 FIRST AND LAST FIELDS

First and last have two uses. For enumeration types they point into the abstract syntax tree to indicate the first and last elements of an enumeration type. For arrays they point into the abstract syntax tree to indicate the first and last indices of an array. This is unless the indices are integers then they are simply the first and last indices.

5.3.1.4 INDEX TYPE FIELD

Index_type is used for arrays to indicate the type of indices the array uses. This is also a pointer into the abstract syntax tree to allow two types to be compared. If the indices are integers this field is set to -1.

5.3.1.5 NUM FIELD

Num also has two uses. For enumeration data types and subtypes it is the number of elements in the type. For structures it is the size in words of the structure.

5.3.1.6 FLAGS FIELD

The flags field is used to indicate type the new type is. The is_array flag is set if it is an array. The is_record is set for records. The is_enum bit is set for enumeration types. The int_index is set if the indices for an array is a range of integers.

5.3.2 WITH AND USE DATA STRUCTURES

The with and use data structures, shown in figure 5-2, are used for package elaboration. When a with_clause is found in the abstract syntax tree the symbol table and code for the package must be read in from secondary storage. Before reading them in a search of the with_list is done to ensure a package is only elaborated once. The current environment is also searched to ensure the package has been previously compiled and the symbol table and code for it exist. As the symbol table is read in the addresses of all procedures and functions must be updated to indicate the new address currently being used. This is very easy to do since the new address will be the old address plus the current offset in the code array. The new package name is added to the with_list and processing continues.

```
struct with_list
 {
  int   name,   /* pointer to package name in   */
                /* syntax tree                  */
        first,  /* index into symbol table to   */
                /* first and last symbol table  */
        last;   /* entries for the package.     */

  struct with_list  *next;  /* pointer to next  */
                            /* package name     */
 }

struct use_list
 {
  int   name;   /* pointer to package name in   */
                /* the syntax tree              */
  struct use_list  *next;   /* pointer to next  */
                            /* use list         */
 }
```

Figure 5-2 With and Use data structures

When a use_clause is encountered the with_list is checked for the name of the package. If it is not found an error is reported since a with_clause for that package must precede the use_clause. If found the current use_list is searched to see if the package is currently visible. This is done to avoid having two or more occurrences of the same package name in the use_list. This speeds up the search process for the symbol table routines since they only have to search a package's symbol table once. If not found in the use_list the name is added to the use_list and processing continues.

5.3.2.1 CHANGES TO SYMBOL TABLE ROUTINES

When packages are added to the subset some changes are needed in the symbol table routines. The first change is the lookup routine must be changed to search not only the current symbol table but also the symbol tables of any visible packages. The packages that are visible are the ones that have been added to the use_list. A consequence of this is that when the symbol table is peeled back to a previous level any use clauses that are no longer visible must be removed from the use_list.

The symbol table lookup routine must also be able to search for qualified names like A.X where A is a package name and X is a name declared within the package. This simply means the with_list must be searched for package A and the the symbol table for package A is searched for X.

The fields of the two structures are explained well enough in Appendix E so no discussion of the fields is included here.

6. ULTIMATE GOALS AND ENVIRONMENT

The ultimate goal, of course, is to get a full Ada compiler implemented. This should be done with certain other goals in mind. To blindly implement the full Ada language without any thought to the remaining development environment would be foolish. These other goals concern the environment in which the programmer must function. Each of these goals is aimed at improving programmer productivity and reducing the time spent testing and fixing a program. These goals are incremental compilation, development of a multitasked environment, and semantic specification of the language in use.

6.1 INCREMENTAL COMPILATION

The aim of incremental compilation is to further reduce programmer idle time during the development cycle. This is accomplished by reducing the time needed to compile a program. The syntax directed editor already does some of this by maintaining the abstract syntax tree between compiles. This eliminates the need to reparse the program each time it is compiled.

Another aspect of this is the fact that the source code usually changes very little when a correction or addition to it is made. This means that most of the code the compiler generated on the previous run is still valid and could be reused if some method were devised to tell the compiler which nodes had changed.

One method of doing this is to have the syntax directed editor mark each node in the abstract syntax tree with a flag to indicate no change, a change in this node or a change occurred at some point below this node. Links between the generated code and the abstract syntax tree must also be maintained since the compiler must be able to change the code if necessary. Since the compiler would only have to generate code for those parts of the program which had changed, significant time could be saved.

Two types of changes can cause problems for an incremental compiler. The first is any change in the declaration of variables or subprograms. Since these changes may have far reaching effects even on sections of the program which have not changed a complete recompile might be warranted. This is especially true in Ada because of the problem of overloading and the fact that no automatic type conversions are allowed.

The other problem is more specific to Ada. It is caused when the environment or context of a program is changed through new or different use and with clauses. Once again a total recompile might be warranted. Since a programmer would know when these changes had been made, he could control the compiler through a pragma to turn the incremental feature on or off.

6.2 MULTITASKED ENVIRONMENT

Multitasking, which Ada supports, is another means of reducing unproductive time. Since the CPU tends to be relatively idle while it waits further user inputs, an incremental compiler and/or a semantic analyzer could be run as separate yet parallel tasks to the editor. The syntax directed editor could trigger either one to analyze a section of code that the user had just entered or changed. A specialized interpreter/debugger might also be written to allow the programmer to test small sections of code independent of the rest of the program.

When this happens the tools seem to merge into one multipurpose tool which the programmer can use to assist in the development of programs. Using this tool properly the programmer sees faster turnaround and a program that is ready to test almost as fast as it can be entered into the system. The programmer may also feel more confident that the program is correct since sections of the code can be tested as they are written.

The multitasked environment does present some problems to be solved. These are mainly in the interfaces between the tools and how the data structures in use are protected. The implementation cannot allow two or more tools to work on the same limbs of the tree unless the two tools are compatible. A typical example that must be disallowed is having the editor delete a subtree while another tool is working on that same subtree.

6.3 SEMANTIC SPECIFICATION

New means have been developed to extend the syntax description of a language to include its semantic specification (Ref 9,13). These extensions could be incorporated in a way that would allow the syntax directed editor to use the semantic information to prevent semantic as well as syntactic errors. This is no trivial task since a change in a declaration may have far reaching effects. To be used by a programmer, such a tool must not make the programmer wait after such a change has been made. This probably means that the semantic checks would be done in background and errors reported to the user giving the user the option of correcting the error immediately or continuing with the current effort and returning to fix the errors later.

7 RECOMMENDATIONS

Extensions to this thesis effort could continue in several directions. Several major extensions were presented in the previous chapter. Since they are the ultimate goals of this effort, any extensions should be done with them in mind.

Perhaps the most important continuation would be the implementation of the tools designed and described in Chapter 5. The extensions that were designed are the semantic analyzer, the data structures necessary for data types and packages, and a method to allow parameter passing to functions and procedures.

7.1 ADA SUBSET EXPANSION

The mos obvious continuation from this point would be to continue expanding the implemented subset until a full Ada compiler is available. This would be not only a good academic exercise but would also be useful to AFIT since it would provide the school with a new tool to use. The easiest direction to take in this effort would probably be to adapt the run time machine developed by Alan Garlington (Ref 11). Since his run time machine already allows tasking, The design of such a mechanism would be avoided.

7.2 REWRITE TOOLS INTO ADA

The other direction to take with this effort is to quit expanding the subset and to rewrite the entire system into Ada. This becomes especially attractive since several Ada compilers have been validated and should become available for use soon. Using a full Ada compiler avoids the problems of doing the rewrite with the current subset.

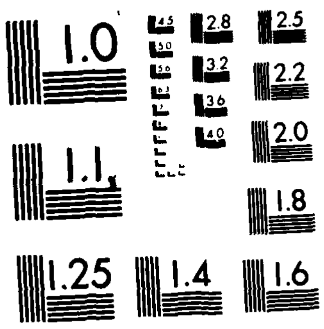
The first of these problems is speed. Since the current implementation compiles to an intermediate code that is interpreted the compilation of a package as large as the current support environment would be extremely time consuming. This speed problem would also show up during execution since interpreting code is much slower than running the equivalent machine code.

The other drawback is memory usage. Currently the abstract syntax tree representation of a program takes about eight times more memory than the equivalent text representation. Another problem is the symbol table and code array are limited due to the way they are implemented. This could be solved by implementing them in a more dynamic fashion such as a linked list.

AD-A138 027 ADA1: AN ADA SUBSET COMPILER FOR THE AFIT SYNTAX DIRECTED PROGRAMMING ENV..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. M L MCCRACKEN

UNCLASSIFIED 05 DEC 83 AFIT/GCS/MA/83D-4 F/G 9/2 NL

2/4


1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

7.3 SUPPORT ENVIRONMENT

Other directions related research might take involve the tools in the support environment. Either the current tools could be enhanced or new tools written. Several useful new tools come to mind. These include a code lister, a code optimizer, online help facility, and a program converter.

7.3.1 CODE LISTER

The code lister, as currently implemented, is a simple tool used only to extract the code generated by the compiler and print it for the programmer. The power of the code lister could be expanded to include the source lines that generated the code. Since the code is already linked to the abstract syntax tree, the source code is available and could easily be included in the listing. Shown with the generated code, the source code could prove useful in showing the programmer how he could have written a fragment of code more efficiently. Another use this could have is as a debugging aid. The programmer could study a fragment of code and see exactly how the compiler interpreted the source code. This might help him spot an error that was caused by how an expression is evaluated.

7.3.2 CODE OPTIMIZER

A code optimizer could work on the generated code to optimize the code with regard to some characteristic the programmer wanted to improve. This characteristic could be execution speed, more efficient memory usage, or any other measure the programmer desired. The optimized code could be either machine language code or code for the interpreter.

7.3.3 ONLINE HELP TOOLS

Online help tools would be useful for training or as a ready reference during the development stage. Several tools come to mind. The first is a tool that would tell the user what command the syntax directed editor will accept at that point and exactly what the command would do. This could be done either through a query by the user or intelligently by the editor as it realizes the user is having problems. A second tool would allow the programmer to check what constructs the syntax would currently accept and what each of them are.

7.3.4 PROGRAM CONVERTERS

Program converters fall into two basic categories. The first would be a tool to convert a program written in one subset of Ada into an expanded subset. This would probably have to be done with programmer intervention where a choice has changed or a newly required component is needed but it would save time since programs would become upwardly compatible. This would be especially useful if the support environment were rewritten into the Ada subset before a full Ada compiler is available.

Another type of program converter would translate the abstract syntax tree representation into some other intermediate form. This could be used to transfer programs from one machine to another. Since the Ada standard has proposed a new language, Diana, be used for this purpose, it could serve as the target language. This would allow programs such as the editor to be more easily transferred from one computer to another.

7.3.5 INTERPRETER

The current interpreter could be improved to run faster, use less memory or a combination of the two. It could also be changed to do dynamic type and range checking. Since tasking is a feature of Ada, a tasking interpreter could be written either from Garlington's design (Ref 11) or a design of the user's choosing.

7.3.6 PROGRAM LISTER

The program lister could be improved to give the programmer a more detailed listing of the program. One such improvement could be the generation of a cross reference listing for the program.

7.3.7 DEBUGGING TOOLS

The debugging tools could be improved to do more run time checks and some execution analysis. The execution analysis could include listing how many times a statement was executed, detecting that a variable is used before it is initialized, or detecting that certain fragments of code are unreachable and are therefore can be eliminated. Another debugging tool that could be written is a tool to automatically generate test data based on an analysis of the program being tested.

Bibliography

1. Aho, Alfred V. and Jeffrey D. Ullman. Principles of Compiler Design. Addison-Wesley 1979.

2. Baker, T.P. "A One Pass Algorithm for Overload Resolution in Ada." ACM Transactions on Programming Languages and Systems. 4 (4):601 (October 1982).

3. Barnes, J.G.P. Programming in Ada. Addison-Wesley 1982.

4. Barrett, William A. and John D. Couch. Compiler Construction: Theory and Practice Science Research Associates Inc. 1979.

5. Defense Communications Engineering Center. "Problems with the Multitasking Facilities in the Ada Programming Language." May 1981, (AD-A105229).

6. Department of Defense. Military Standard Ada Programming Language. Washington, D.C. January 1983, (ANSI/MIL-STD-1815A).

7. Department of Defense. Reference Manual for the Ada Programming Language. Washington, D.C. July 1980, (AD-A090709).

8. Department of Defense. Requirements for ADA Programming Support Environments - Stoneman. Washington, D.C. 1980.

9. Ferguson, Scott E. A Syntax-Directed Programming Environment for the Ada Programming Language. Master's Thesis Air Force Institute of Technology. December 1982, (AD-A124843).

10. Fox, Joseph M. "Benefit Model for High Order Languages." Decisions and Designs Inc. McLean, Virginia. 1978, (AD-A053032).

11. Garlington, Alan R. Preliminary Design and Implementation of an Ada Pseudo-Machine. Masters Thesis Air Force Institute of Technology. March 1981, (AD-A100796).

12. Gaudino, Richard L. Analysis and Design of Interactive Debugging for the ADA Programming Environment. Masters Thesis Air Force Institute of Technology. November 1981, (AD-A115636).

13. Gordon, Michael J.C. The Denotational Description Of Programming Languages - An Introduction. Springer-Verlag 1979.

14. Janas, Jurgen M. "A Comment on Operator Identification in Ada." ACM SIGPLAN Notices 15 (9):39 (September 1980).

15. Ledgard, Henry F. and Andrew Singer. "Scaling Down Ada (Or Towards a Standard Ada Subset)." Communications of the ACM. 20 (2):121 (February 1982).

16. Pennello, Tom et.al. "A Simplified Operator Identification Scheme for Ada." ACM SIGPLAN Notices. 15 (7&8):82 (July-August 1980).

17. Wetherell, C.S. "Problems with the Ada Reference Grammar." ACM SIGPLAN Notices. 16 (9):90 (September 1981).

18. Wirth, Niklaus. Algorithms + Data Structures = Programs. Prentice-Hall 1976.

APPENDIX A

APPENDIX A. META DESCRIPTION FOR THE ADA1 SUBSET

The following is the META description of the ADA1 subset implemented by the ADA1 compiler.

```
compilation =
        compilation_unit ;

compilation_unit = <
        func_body
        proc_body >;

func_body =
        [ program_header ]
        @ func_spec ^ "is"
                { + decl }
                { + program_component }
        @ "begin"
                + seq_of_stmts
        @ "end" [ ^ designator ] ";" ;

func_spec =
        "function" ^ designator ^ "return"
                            ^ subtype_indication ;

designator = <
        identifier >;


proc_body =
        [ program_header ]
        @ proc_spec ^ "is"
                { + decl }
                { + program_component }
        @ "begin"
                + seq_of_stmts
        @ "end" [ ^ identifier ] ";" ;

program_header =
        comment
        { @ comment } ;

proc_spec =
        "procedure" ^ identifier ;

decl =
        object_decl  [ ^ comment ] ;
```

APPENDIX A

```
program_component = <
        func_body
        proc_body >;

seq_of_stmts =
        stmt
        { @ stmt } ;

identifier =
        'AZ|az' { '09|AZ|_|az' } ;

object_decl =
        id_list ^  ":" [ ^ "constant" ] ^ object_type
                                        [ ^ initial ] ";" ;

stmt =
        simple_stmt [ ^ comment ] ;

id_list =
        identifier { identifiers } ;

object_type = <
        subtype_indication >;

initial =
        ":=" ^ expression ;

simple_stmt = <
        assignment_stmt
        if_stmt
        loop_stmt
        return_stmt
        proc_call
        comment
        null_stmt >;

identifiers =
        "," ^ identifier ;

subtype_indication = <
        "integer"
        "boolean"
        "char" >;

expression = <
        relation
        and_comp
        or_comp
        and_then_comp
        or_else_comp
        xor_comp >;
```

APPENDIX A

```
assignment_stmt =
        name ^ ":=" ^ expression ";" ;

if_stmt =
        "if" ^ expression ^ "then"
                + seq_of_stmts
        { @ elsif_part }
        [ @ else_part ]
        @ "end" ^ "if" ";" ;

loop_stmt =
        [ iteration_clause ^ ] "loop"
                + seq_of_stmts
        @ "end" ^ "loop" ";" ;

proc_call =
        name ";" ;

return_stmt = <
        procedure_return
        function_return > ;

comment =
        "-- " { ' }' } ;

null_stmt =
        "null;" ;

procedure_return =
        "return;" ;

function_return =
        "return" ^ expression ";" ;

name = <
        identifier >;r

relation =
        simple_exp [ ^ relation_part ! ] ;

and_comp =
        relation { ^ and_relation } ;

or_comp =
        relation { ^ or_relation } ;

and_then_comp =
        relation { ^ and_then_relation } ;

or_else_comp =
        relation { ^ or_else_relation } ;
```

APPENDIX A

```
xor_comp =
        relation { ^ xor_relation } ;

elsif_part =
        "elsif" ^ expression ^ "then"
                + seq_of_stmts ;

else_part =
        "else"
                + seq_of_stmts ;

iteration_clause = <
        while_clause >;

simple_exp =
        [ unary_operator ! ] term { ^ terms } ;

relation_part = <
        relational >;

and_relation =
        "and" ^ relation ;

or_relation =
        "or" ^ relation ;

and_then_relation =
        "and then" ^ relation ;

or_else_relation =
        "or else" ^ relation ;

xor_relation =
        "xor" ^ relation ;

while_clause =
        "while" ^ expression ;

unary_operator = <
        "+"
        "-" >;

term =
        factor { ^ factors } ;

terms =
        add_op ^ term ;

relational =
        rel_op ^ simple_exp ;
```

APPENDIX A

```
factor = <
        exp_primary
        abs_primary
        not_primary >;

factors =
        mul_op ^ factor ;

exp_primary =
        primary [ ^ expon_part ! ] ;

expon_part =
        "**" ^ primary ;

abs_primary =
        "abs" ^ primary ;

not_primary =
        "not" ^ primary ;

add_op = <
        "+"
        "-" >;

rel_op = <
        "="
        "/="
        "< "
        "<="
        "> "
        ">=" >;

primary = <
        decimal_number
        name
        nested_exp
        char_lit
        boolean_value
        func_call >;

func_call =
        designator "(" ")" ;

mul_op = <
        "*"
        "/"
        "mod"
        "rem" >;

decimal_number =
        integer ;
```

APPENDIX A

```
nested_exp =
        "(" expression ")" ;

integer =
        '09' { '09|_' } ;

char_lit =
        "'" ' }' "'" ;

boolean_value = <
        "true"
        "false" >;
```

APPENDIX B. OUTPUT OF THE META PROGRAM

The following is the output produced by META when the ADA1 grammar was used as input.

```
META 10/15/82

Source input = ada1x.syn.
Output file = ada1x.sdf.
Goal symbol: compilation.

item name......................... used.. term/non..
alt/cat..
 }                                 2      set
'                                  2      terminal
(                                  2      terminal
)                                  2      terminal
*                                  1      terminal
**                                 1      terminal
+                                  2      terminal
,                                  1      terminal
-                                  2      terminal
--                                 1      terminal
/                                  1      terminal
/=                                 1      terminal
09                                 1      set
09|AZ|_|az                         1      set
09|_                               1      set
:                                  1      terminal
:=                                 2      terminal
;                                  8      terminal
<                                  1      terminal
<=                                 1      terminal
=                                  1      terminal
>                                  1      terminal
>=                                 1      terminal
AZ|az                              1      set
abs                                1      terminal
abs_primary                        1      non-term   cat
add_op                             1      non-term   alt
and                                1      terminal
and then                           1      terminal
and_comp                           1      non-term   cat
and_relation                       1      non-term   cat
```

APPENDIX B

```
and_then_comp                    1      non-term   cat
and_then_relation                1      non-term   cat
assignment_stmt                  1      non-term   cat
begin                            2      terminal
boolean                          1      terminal
boolean_value                    1      non-term   alt
char                             1      terminal
char_lit                         1      non-term   cat
comment                          5      non-term   cat
compilation                      1      non-term   cat
        a single unconditional term.
compilation_unit                 1      non-term   alt
constant                         1      terminal
decimal_number                   1      non-term   cat
        a single unconditional term.
decl                             2      non-term   cat
designator                       3      non-term   alt
        a single alternative.
else                             1      terminal
else_part                        1      non-term   cat
elsif                            1      terminal
elsif_part                       1      non-term   cat
end                              4      terminal
exp_primary                      1      non-term   cat
expon_part                       1      non-term   cat
expression                       7      non-term   alt
factor                           2      non-term   alt
factors                          1      non-term   cat
false                            1      terminal
func_body                        2      non-term   cat
func_call                        1      non-term   cat
func_spec                        1      non-term   cat
function                         1      terminal
function_return                  1      non-term   cat
id_list                          1      non-term   cat
identifier                       6      non-term   cat
identifiers                      1      non-term   cat
if                               2      terminal
if_stmt                          1      non-term   cat
initial                          1      non-term   cat
integer                          1      terminal
integer                          1      non-term   cat
is                               2      terminal
iteration_clause                 1      non-term   alt
        a single alternative.
loop                             2      terminal
loop_stmt                        1      non-term   cat
```

```
mod                                    1      terminal
mul_op                                 1      non-term   alt
name                                   3      non-term   alt
        a single alternative.
nested_exp                             1      non-term   cat
not                                    1      terminal
not_primary                            1      non-term   cat
null;                                  1      terminal
null_stmt                              1      non-term   cat
        a single unconditional term.
object_decl                            1      non-term   cat
object_type                            1      non-term   alt
        alternation alternative: subtype_indication.
        a single alternative.
or                                     1      terminal
or else                                1      terminal
or_comp                                1      non-term   cat
or_else_comp                           1      non-term   cat
or_else_relation                       1      non-term   cat
or_relation                            1      non-term   cat
primary                                4      non-term   alt
        alternation alternative: name.
        alternation alternative: boolean_value.
proc_body                              2      non-term   cat
proc_call                              1      non-term   cat
proc_spec                              1      non-term   cat
procedure                              1      terminal
procedure_return                       1      non-term   cat
        a single unconditional term.
program_component                      2      non-term   alt
program_header                         2      non-term   cat
rel_op                                 1      non-term   alt
relation                               11     non-term   cat
relation_part                          1      non-term   alt
        a single alternative.
relational                             1      non-term   cat
rem                                    1      terminal
return                                 2      terminal
return;                                1      terminal
return_stmt                            1      non-term   alt
seq_of_stmts                           6      non-term   cat
simple_exp                             2      non-term   cat
simple_stmt                            1      non-term   alt
        alternation alternative: return_stmt.
stmt                                   2      non-term   cat
subtype_indication                     2      non-term   alt
term                                   2      non-term   cat
terms                                  1      non-term   cat
then                                   2      terminal
true                                   1      terminal
unary_operator                         1      non-term   alt
while                                  1      terminal
```

```
while_clause                        1      non-term   cat
xor                                 1      terminal
xor_comp                            1      non-term   cat
xor_relation                        1      non-term   cat

121 elements described.
312 nodes in syntax description file.

0 errors detected.
META processing complete.
```

APPENDIX C. DESIGN FOR PASSING PARAMETERS TO SUBPROGRAMS

1. Find the subprogram name in the symbol table.

2. Get the next parameter from the associated parameter list.

3. Determine the type parameter it is. If it is an in parameter goto step 4. if it is an out or an in out parameter goto step 5.

4. Call expression to evaluate the parameter. A type will be returned verify the type is correct. Generate an instruction to pop the stack. This last instruction is put into a list so that it can be appended to the code array after all parameters are evaluated. Goto step .

5. Verify the actual parameter is a variable name that has a location associated with it. If not generate an error that says the parameter must be a name and not an expression.

6. Check if the actual parameter is an indexed component. If it is then goto step 9.

7. If the formal parameter is an out parameter generate an instruction to load the undefined value. If it is an in out parameter generate the code to load the value.

8. Generate an instruction to store the value back and prepend this instruction to the list of instructions that will be added to the code array after all parameters are processed. Goto step .

9. Generate code to evaluate the index. If the parameter is an out parameter goto step 10. If it is an in out parameter generate an instruction to copy the index to the index stack. This instruction will leave the index on the run time stack. Generate an instruction to load the value onto the stack using the index on top of stack. Goto step 11.

10. Generate an instruction to move the index from the run time stack to the index stack. This instruction removes the index from the run time stack. Generate an instruction to load the undefined value onto the run time stack.

11. Generate an instruction to save the value when the procedure returns. Prepend this instruction to the list of instructions to be added to the code array after all parameters are evaluated.

12. Check if another parameter is needed for this procedure. Check if a parameter is available. If both are true goto step 2. If not generate an error saying either a parameter is missing or an extra parameter was passed.

13. Append the list of instructions being held at bay to the code array.

14. Return control up the tree.

APPENDIX D

APPENDIX D. A DESIGN FOR THE SEMANTIC ANALYZER

1. If the expression is a function call goto step 34.

2. If the expression is a procedure call goto step 23.

3. If the expression is a name goto step step 19.

4. (Expression still has an operator) Find the controlling operator. It is the rightmost operator at this level in the tree.

5. If the operator is a unary operator goto step 18

6. (Binary operator) Find all occurrences of the operator in the symbol table. If none return error.

7. Create an operator list entry for each occurrence.

8. If a type list was passed in eliminate any entries in the operator list that cannot return one of the desired types.

9. If the operator list is empty return error.

10. Create the left operand list from the operator list.

11. Call the semantic analyzer with the left operand list and the left hand operand.

12. If error is returned return error.

13. Compare the returned type list with the operator list. Eliminate elements from operator list whose left hand operand type does not appear in the returned type list.

14. Repeat steps 9-13 for right hand operator.

15. If the list is empty return error.

16. If the operator contains a single entry goto step call pass3. If pass3 returns error return error.

17. Create operand list from the return types of the operator list. Return this list.

18. (Unary operator) Same as a binary operator (steps 6-17) except only done for a single right hand operator.

19. (Name) Find all occurrences of the name in the symbol

table. If none return error.

20. Create an operand list element for each entry.

21. If a type list was passed in eliminate the unavailable types from it by comparing it to the operand list created in step 20. Return the modified type list.

22. No list passed in return the operand list generated in step 20.

23. (Procedure call) Find all occurrences of the procedure name in the symbol table. If none return error.

24. Count the number of actual parameters used in the procedure call.

25. Create a list of the procedures found in the symbol table which need that number of parameters. If the list is empty return error.

26. If no parameters are needed and more than one procedure is possible return error. Else mark the syntax tree node with the symbol table entry of the procedure name and return success.

27. Repeat steps 28-32 for each parameter and procedure in turn.

28. Check if the first actual parameter is a name or an expression. If it is an expression eliminate all procedures that require an out or in out parameter. If the list is now empty return error.

29. Create an operand list from the procedure list for this parameter.

30. Call the semantic analyzer with the operand list and the first parameter.

31. If error is returned return error.

32. Compare the returned type list with the procedure list and eliminate any procedures which do not have this parameter available.

33. If two or more procedures remain in the procedure list return error. If only one remains call pass 3 with each parameter and its type in turn. Return success.

34. (Function call) Same as procedure except as noted below.

35. Add step 25a. If a type list is passed in compare it to the types that can be returned. Eliminate any functions which cannot return a valid type.

36. Eliminate step 28 since only in parameters are allowed.

37. Change step 33 to If two or more functions remain in the list create the type list from their return types and return the list. If a single function is in the list call pass 3 with each parameter and its type in turn. If pass 3 returns error return error. Otherwise return the type of the function.

PASS 3

Pass 3 is very similar to the actual semantic analyzer except that no multiple types are allowed to be returned. If at any time during this pass an ambiguity cannot be resolved the expression is in error and error is returned. As each node of the syntax tree is correctly typed it is marked with the appropriate symbol table pointer and a flag is raised to indicate the expression is ok below this node. Pass 3 must check this flag before it descends the tree further. This is only to avoid resolving ambiguities more than one time.

APPENDIX E - SYSTEMS USERS MANUAL

This appendix describes how the compiler, interpreter, and code lister are invoked, the inputs they expect and the outputs they produce.

Before any of these tools can be used a program must be entered using the syntax directed editor. The syntax directed editor is invoked with the command, synde TEST ada1x. This command will produce a program file named TEST using the language ADA1X. For more information on using the syntax directed editor refer to the syntax directed editor users manual (Ref 9).

E.1 INVOKING THE COMPILER

The compiler can be invoked in three ways, from the syntax directed editor, directly by the user, or by the interpreter.

E.1.1 FROM THE SYNTAX DIRECTED EDITOR

To invoke the compiler from the syntax directed editor simply enter the invoke compiler command. This command is selected by the user when the editor is configured using the CONFIG program (Ref 9). The compiler can be used in this way to do a preliminary semantic check of the program.

The compiler will flag up to five errors using program tree markers. The compiler will halt processing when either all error markers are used or the compilation is complete. If invoked in this manner the compiler will return control to the syntax directed editor with the focus set at the first error in the program or at the root of the program tree if no errors were detected. The user can then correct any errors and continue entering the program.

E.1.2 DIRECTLY BY THE USER

The second way to invoke the compiler is with the command ada1xC TEST, where TEST is the program the user wants to compile. The compiler will still flag up to five errors and will return control to the command level when five errors are found or compilation is completed. To fix an error the user need only invoke the editor with the command, synde TEST. The editor will put the focus at the first error or at the root of the tree if no errors were detected.

E.1.3 FROM THE INTERPRETER

The third way to invoke the compiler is through the interpreter. The interpreter calls the compiler to generate the code it is to interpret and to link that code to the abstract syntax representation of the program. The interpreter is invoked as shown below.

E.1.4 COMPILER INPUTS AND OUTPUTS

The compiler expects no special inputs other than the program file given on the command line. The compiler produces a file called TEST.cod where TEST is the name of the input program file.

E.2 INVOKING THE INTERPRETER

The interpreter can be invoked in two ways, from the syntax directed editor, or directly by the user.

E.2.1 FROM THE SYNTAX DIRECTED EDITOR

To invoke the interpreter from the syntax directed editor simply enter the invoke interpreter command. This command is selected by the user when the editor is configured for the terminal in use (Ref 9). The interpreter can be used in this way to do a preliminary semantic and logic check of the program. This can be especially useful if the user is not certain what a specific instruction will do.

The interpreter first calls the compiler to compile the program. The compiler works as described above. When control is returned to the interpreter, it checks if any errors were detected. If some were the interpreter asks the user if execution is to continue. If not control is returned to the syntax directed editor as described above with the focus at the first error. If execution is allowed or the program had no errors, the execution routine is called. Operation of the execution routine is explained below.

E.2.2 DIRECTLY BY THE USER

The other way to invoke the interpreter is with the command adalxI TEST. This will cause the interpreter to compile and execute the program TEST. The interpreter functions as explained above except that control will be returned to the command level rather than to the syntax directed editor.

E.3 PROGRAM EXECUTION

When a program is executed control is returned to the user. The user is given a display showing the program, the top few elements of the program stack, and the next instruction to be executed. The portion of the program thought responsible for that instruction is highlighted in reverse video. The user is also given a choice of four commands to input. The four commands are single step, continue execution, restart, and exit.

Any command not described below is ignored and the user is prompted to enter a new command.

E.3.1 SINGLE STEP

The single step command, invoked by 'S' or ' ', causes execution of the displayed instruction. The display is updated and the user is shown the new program stack, the next instruction, and the next program fragment is highlighted. If the instruction executed was the last instruction of the program or an INVALID instruction, the interpreter is reset to reexecute the program from the beginning, and the display is reset to its initial state.

E.3.2 CONTINUE EXECUTION

The continue execution command, invoked by 'C', causes the interpreter to execute the program in a continuous fashion until the end of program is found. The displays are updated as if the user was single stepping the program very rapidly. When the end of the program is found the interpreter is reset as before and the user is prompted for further inputs.

E.3.3 RESTART

The restart command, invoked by 'R', causes the interpreter to reset itself to its initial state. Program execution will continue at the start of the program.

E.3.4 EXIT

The exit command, invoked by 'E', causes the interpreter to halt execution and return control either to the syntax directed editor or to the command level depending on how the interpreter was invoked.

E.3.5 INTERPRETER INPUTS AND OUTPUTS

The interpreter expects only those inputs described above. Any other inputs are ignored. The interpreter produces no outputs of its own other than the displays described above. The compiler does produce the file TEST.cod where TEST is the input program file name.

E.4 THE CODE LISTER

The code lister is invoked by the command, codelist TEST, where TEST is the name of the program whose code the user wants to list. The file TEST.cod must exist. This is the file produced by the compiler when the program is compiled. The code lister produces an output file called TEST.codlst. This file can then be read and displayed by any program which uses a standard text file as input such as cat, more, vpr.

APPENDIX F

```
/************************************************************************
*               00000   00    00   00   00   00000    00000             *
*               00       00  00    000  00   00  00   00                *
*               00000     0000     00 0 00   00  00   0000              *
*                  00      00      00  000   00  00   00                *
*               00000      00      00   00   00000    00000             *
*                                                                       *
*           SYNtax-Directed Editor (c) Copyright December 1982          *
*             CAPT. Scott Edward Ferguson, USAF, AFIT GCS-82D           *
*                          Modified October 1983                        *
*             CAPT. Michael L. McCracken, USAF, AFIT GCS-83D            *
*                                                                       *
*                               SYNDE.H                                 *
*       Define global information types and values for SYNDE data       *
*       structures.                                                     *
************************************************************************/

#include <stdio.h>

#include "ctype.h"

#define SUCCESS         1       /* success return value                 */
#define ERROR           (-1)    /* error return value                   */

/* special defined types                                                */

#define REG     register int    /* type for register optimization       */

#define BOOL    int             /* type declaration for boolean values  */
#define TRUE    1               /* boolean TRUE constant                */
#define FALSE   0               /* boolean FALSE constant               */

/* file action status "enumeration" values                              */

#define TO_LIST 1               /* vector execution to lister           */
#define TO_COMP 2               /* vector execution to compiler         */
#define TO_INT  3               /* vector execution to interpreter      */
#define TO_SYN  4               /* vector execution to synde            */
```

APPENDIX F

```
/*********************************************************************
*       Abstract Syntax Tree (AST) node description.                *
*********************************************************************/

struct  ast_node {              /* AST node structure                */
        char
                a_flags,        /* AST node flags (described below)  */
                a_value;        /* contains character (or 0) for a node */
                                /*      corresponding to a "set" element*/
        unsigned
                a_prod,         /* "pointer" to syntax element       */
                                /*      corresponding to this node   */
                a_right,        /* "pointer" to right sibling; if RTMOST*/
                                /*      is set, this points to father  */
                a_son;          /* "pointer" to node's leftmost son or */
                                /*      NIL if node is a leaf node   */
        };

#define NIL     0               /* NIL pointer to "no" AST node      */

/* a_flags masks for bit-field values                                */

#define RTMOST  0x01            /* node is rightmost son             */
#define LTMOST  0x02            /* node is leftmost son              */
#define ONLY    0x03            /* node is only child (left and right) */
#define ROOT    0x04            /* node is an AST tree root          */
#define OPEN    0x10            /* node is an unsatisfied conditional */
#define ELIDE   0x20            /* mark to suppress display of subtree */
#define MARK    0x40            /* node is marked in f_mark list     */
#define FREE    0x80            /* node is free (unallocated)        */
```

APPENDIX F

```
/**************************************************************************
*       Syntax Tree node description.                                     *
**************************************************************************/

struct  syn_node {
        unsigned
                s_flags,                /* node flags (described below) */
                s_link;                 /* if header:                   */
                                        /*      "pointer" to name string*/
                                        /* if element:                  */
        };                              /*      "pointer" to header     */

#define MAX_ALT 20                      /* max number of alternatives   */
                                        /*      in an alternation       */

/* s_flags masks for bit-field values                                   */
/*      for a definition header node---                                 */

#define ALT     0x0001                  /* alternation rule header      */
#define CAT     0x0002                  /* concatenation rule header    */
#define RULE    0x0003                  /* rule header                  */
#define STR     0x0004                  /* string header                */
#define SET     0x0008                  /* set header                   */
#define HEAD    0x000F                  /* node is a header node        */

/*      for a rule element node---                                      */

#define OPTION  0x0010                  /* optional rule element        */
#define REPEAT  0x0020                  /* repeated rule element        */
#define COND    0x0030                  /* conditional OPTION or REPEAT */
#define HIDE    0x0040                  /* hidden conditional           */
                                        /* output formats:              */
#define NEWLINE 0x0100                  /* newline                      */
#define INDENT  0x0200                  /* indent (and newline)         */
#define LINE    0x0300                  /* generates new line           */
#define PRESP   0x0400                  /* space precedes node          */
#define POSTSP  0x0800                  /* space follows node           */

/**************************************************************************
*       Image Generation parameters.                                      *
**************************************************************************/

#define WIDE    132             /* max screen width in characters       */
#define DEEP    30              /* max screen depth in lines            */
#define HILITE  0x80            /* highlight bit for display            */
```

APPENDIX F

```
/*********************************************************************
*       Source File information block.                               *
*********************************************************************/

struct  file_info {
        int     f_buf;
        char
                f_name[16],      /* creation file name                   */
                f_lang[16],      /* associated language grammar name     */
                f_creat[16],     /* creation date                        */
                f_last[16],      /* last access date                     */
                f_conf[34];      /* configuration control information    */
        int
                f_edit,          /* return control to SYNDE editor       */
                f_update,        /* version update count                 */
                f_avail,         /* "pointer" to available list head     */
                f_root,          /* "pointer" to program tree root       */
                f_clip,          /* "pointer" to clipping tree root      */
                f_mark[10];      /* place markers, 5 thru 9 reserved as  */
                                 /*      error markers                   */
        };

/*********************************************************************
*       Terminal Description File characteristics block.             *
*********************************************************************/

/* input command "enumeration" values                                */

#define _RIGHT          1       /* move right                           */
#define _LEFT           2       /* move left                            */
#define _LEAF           3       /* move to leaf node                    */
#define _LAST           4       /* move back to last focus              */
#define _UP             5       /* move up                              */
#define _DOWN           6       /* move down                            */
#define _LUP            7       /* long up (skip only child nodes)      */
#define _LDOWN          8       /* long down (skip only child nodes)    */
#define _RINS           9       /* insert conditional right             */
#define _LINS           10      /* insert conditional left              */
#define _CLIP           11      /* clip subtree                         */
#define _DEL            12      /* delete subtree (clip and kill)       */
#define _KILL           13      /* kill subtree                         */
#define _COPY           14      /* copy subtree                         */
#define _ELIDE          15      /* elide subtree (suppress display)     */
#define _HELP           16      /* help                                 */
#define _WINDOW         17      /* open/close second window             */
#define _SEARCH         18      /* search for literal string            */
#define _AGAIN          19      /* search again                         */
#define _MARK           20      /* set/clear node marker                */
#define _GO             21      /* go to node marker                    */
#define _COMP           22      /* invoke compiler                      */
#define _INT            23      /* invoke interpreter                   */
#define _LIST           24      /* invoke lister                        */
```

APPENDIX F

```
#define _SYNDE          25      /* invoke synde                         */
#define _LRIGHT         26      /* move right past identical siblings   */
#define _LLEFT          27      /* move left past identical siblings    */
#define _REDRAW         28      /* clear screen and redraw it           */
#define _SAVE           29      /* save all current changes             */
#define _WRITE          30      /* write to any given file              */
#define _EXIT           31      /* exit edit session (last command)     */

struct term_info {
        int     lines,          /* lines per screen                     */
                chars,          /* characters per line                  */
                wsize,          /* window size                          */
                xxx[5];         /* (reserved for expansion)             */

        char    cmds[32][8],    /* command input sequence strings       */
                                /* terminal control output sequences:   */
                init[16],       /* terminal initialization              */
                tab[8],         /* tab display string                   */
                elide[8],       /* elision display string               */
                div[8],         /* window divider                       */
                clr[8],         /* clear screen                         */
                pos[8],         /* position cursor                      */
                eol[8],         /* erase to end of line                 */
                dc[8],          /* delete character                     */
                rev[8],         /* enter reverse video mode             */
                norm[8],        /* exit reverse video mode              */
                ion[8],         /* enter insert character mode          */
                ioff[8],        /* exit insert character mode           */
                il[8],          /* insert line                          */
                dl[8],          /* delete line                          */
                fini[16];       /* terminal termination                 */
        };
```

APPENDIX F

```
/****************************************************************/
/*                                                              */
/*                          SYMBOL.H                            */
/*         Symbol table structure for compiler generation.      */
/*                                                              */
/****************************************************************/

struct  sym_table {             /* symbol table                 */
        int
                sym_node,       /* node "pointer" to identifier */
                sym_level,      /* lex level                    */
                sym_addr,       /* offset address               */
                sym_type;       /* symbol type                  */
        unsigned    sym_flags;  /* symbol table flags.  described */
                                /*  below.                      */
        };

#define SYMBOLS 50              /* symbol table size            */


/****************************************************************/
/*                                                              */
/*      symbol flags - used to give more information about a   */
/*      symbol name.                                            */
/*                                                              */
/****************************************************************/

#define  TYPE_VAR      0x0001         /* symbol is a variable    */
#define  TYPE_CONST    0x0002         /* symbol is a constant    */
#define  TYPE_PROC     0x0004         /* symbol is a procedure   */
#define  TYPE_INIT     0x0008         /* symbol was initialized  */
#define  TYPE_TYPE     0x0010         /* symbol is a type name   */
#define  TYPE_FUNC     0x0020         /* symbol is a function    */


/****************************************************************/
/*                                                              */
/*      variable types - predefined                             */
/*                                                              */
/****************************************************************/

#define  TYPE_INT      1
#define  TYPE_CHAR     2
#define  TYPE_BOOL     3
```

APPENDIX F

```
/*******************************************************************/
/*                                                                 */
/*      ADA1.H                                                     */
/*                                                                 */
/*******************************************************************/

#include "synde.h"          /* system global information structures */

#include "code.h"              /* code generation structures       */

#include "symbol.h"            /* symbol table structures          */

#include "types.h"             /* production type definitions      */

extern
struct  code_word *code;       /* pseudo-code                      */

extern
struct  sym_table *symbol;     /* symbol table                     */

extern
int     code_ptr,              /* index of next code cell          */
        sym_ptr,               /* top of symbol table              */
        level,                 /* current lexical level            */
        offset,                /* current offset in level          */
        ret_lst,               /* return list for sub-programs     */
        ret_type,              /* return type for a function       */
        body_type;             /* current sub-program type         */


extern
char    *types[LAST_PROD_],          /* production types            */
        *noson,*bool_exp,*int_exp;   /* frequent message strings    */
```

APPENDIX F

```
/**********************************************************************/
/*                                                                    */
/*      types.h - production type definitions                         */
/*                                                                    */
/**********************************************************************/

#define MUL_            0
#define PLUS_           1
#define MINUS_          2
#define DIV_            3
#define NEQ_            4
#define LES_            5
#define LEQ_            6
#define EQU_            7
#define GRT_            8
#define GEQ_            9
#define AND_COMP_       10
#define A_STMT_         12
#define CONST_          15
#define DEC_NUM_        16
#define DECL_           17
#define ELSIF_P_        18
#define IF_STMT_        21
#define I_CLAUSE_       23
#define LP_STMT_        24
#define NAME_           26
#define N_EXP_          27
#define OR_COMP_        28
#define P_CALL_         31
#define P_COMP_         32
#define RELATION_       33
#define U_OP_           36
#define REM_            34
#define MOD_            25
#define AND_THEN_COMP_  11
#define OR_ELSE_COMP_   29
#define XOR_COMP_       37
#define INTEGER_        22
#define BOOLEAN_        13
#define CHAR_LIT_       14
#define IDENT_          20
#define TRUE_           35
#define FALSE_          19
#define F_BODY_         38
#define F_CALL_         39
#define P_BODY_         30
#define RET_STMT_       40
#define BOOLEAN_VALUE_  41
#define EXPON_PART_     42
#define ABS_PRIMARY_    43
#define NOT_PRIMARY_    44
```

APPENDIX F

```
#define EXP_PRIMARY_    45
#define CHAR_           46
#define PROC_RET_       47
#define FUNC_RET_       48
#define COMMENT_        49
#define NULL_STMT_      50
#define LAST_PROD_      51
```

```
/*****************************************************************/
/*      pseudo-code word description                             */
/*****************************************************************/

struct  code_word {
        char
                c_opcode,               /* operation code        */
                c_label;                /* label indicator       */
        int
                c_op1,                  /* first operand         */
                c_op2,                  /* second operand        */
                c_node;                 /* "pointer" to AST node */
                                        /* which generated the   */
                                        /* the code word.        */
        };

#define WORDS   512                     /* code table size       */

/*****************************************************************/
/*      pseudo-code instruction operation codes                  */
/*              TOP = top of stack                               */
/*****************************************************************/

#define _LAB    0       /* mark code word for label (no code)    */
#define _JMP    1       /* jump to op1                           */
#define _CPY    2       /* copy TOP                              */
#define _STO    3       /* [ level op1, offset op2 ] = TOP       */
#define _CAL    4       /* call subroutine op2 at level op1      */
#define _JPC    5       /* jump if TOP false to op1              */
#define _EQU    6       /* set TOP = (TOP-1 == TOP)              */
#define _NEQ    7       /* set TOP = (TOP-1 != TOP)              */
#define _LES    8       /* set TOP = (TOP-1 < TOP)               */
#define _LEQ    9       /* set TOP = (TOP-1 <= TOP)              */
#define _GRT    10      /* set TOP = (TOP-1 > TOP)               */
#define _GEQ    11      /* set TOP = (TOP-1 >= TOP)              */
#define _NEG    12      /* set TOP = -(TOP)                      */
#define _ADD    13      /* set TOP = (TOP-1 + TOP)               */
#define _SUB    14      /* set TOP = (TOP-1 - TOP)               */
#define _MUL    15      /* set TOP = (TOP-1 * TOP)               */
#define _DIV    16      /* set TOP = (TOP-1 / TOP)               */
#define _LIT    17      /* set TOP = op1                         */
#define _LOD    18      /* set TOP = [ level op1, offset op2 ]   */
#define _RET    19      /* return from subroutine                */
#define _AND    20      /* set TOP = (TOP-1 & TOP)               */
#define _OR     21      /* set TOP = (TOP-1 | TOP)               */
#define _NOT    22      /* set TOP = !(TOP)                      */
#define _DCS    23      /* stack pointer -= op1                  */
#define _MOD    24      /* set TOP = (TOP-1 mod TOP)             */
#define _REM    25      /* set TOP = (TOP-1 rem TOP)             */
#define _AND_THEN  26   /* jump if TOP false to op1              */
#define _OR_ELSE   27   /* jump if TOP true to op1               */
```

APPENDIX F

```
#define _XOR    28      /* set TOP = (!TOP-1 & TOP) | (TOP-1 & !TOP)*/
#define _ABS    29      /* set TOP = abs(TOP)                       */
#define _EXP    30      /* set TOP = (TOP-1 ** TOP)                 */
#define _NOOP   31      /* no operation                             */
```

APPENDIX F

```
/*********************************************************************/
/*                                                                   */
/*      ctype.h - global defines and one line functions              */
/*               replaces the UNIX file ctype.h                      */
/*                                                                   */
/*********************************************************************/

#define _U      01
#define _L      02
#define _N      04
#define _S      010
#define _P      020
#define _C      040
#define _X      0100

extern  char    _ctype_[];

#define isalpha(c)      ((_ctype_+1)[c]&(_U|_L))
#define isupper(c)      ((_ctype_+1)[c]&_U)
#define islower(c)      ((_ctype_+1)[c]&_L)
#define isdigit(c)      ((_ctype_+1)[c]&_N)
#define isxdigit(c)     ((_ctype_+1)[c]&(_N|_X))
#define isspace(c)      ((_ctype_+1)[c]&_S)
#define ispunct(c)      ((_ctype_+1)[c]&_P)
#define isalnum(c)      ((_ctype_+1)[c]&(_U|_L|_N))
#define isprint(c)      ((_ctype_+1)[c]&(_P|_U|_L|_N))
#define iscntrl(c)      ((_ctype_+1)[c]&_C)
#define isascii(c)      ((unsigned)(c)<=0177)
#define toascii(c)      ((c)&0177)
```

APPENDIX F

```
/*********************************************************************/
/*                                                                   */
/*           00000   00    00   00   00   00000    00000            */
/*           00       00  00    000  00   00  00   00               */
/*           00000     0000     00 0 00   00  00   0000             */
/*              00      00      00  000   00  00   00               */
/*           00000      00      00   00   00000    00000            */
/*                                                                   */
/*       SYNtax-Directed Editor (c) Copyright December 1982         */
/*         CAPT. Scott Edward Ferguson, USAF, AFIT GCS-82D          */
/*                     Modified October 1983                         */
/*         CAPT. Michael L. McCracken, USAF,AFIT GCS-83D            */
/*                                                                   */
/*                           COMPILER.C                              */
/*    SYNDE system compiler entry point.  To be linked with COMPILE.C */
/*    and appropriate language specific compiler routines to produce  */
/*    a language specific compiler.                                  */
/*                                                                   */
/*********************************************************************/

#include "synde.h"            /* system global information structures */

#include "types.h"            /* production type definitions          */

extern                        /* source file information              */
struct  file_info src_info;   /*      in AST.C                        */
char    *noson,*bool_exp,*int_exp;
char    *types [LAST_PROD_];
```

APPENDIX F

```
/************************************************************************/
/*                                                                      */
/*      main                                                            */
/*              Entry point and driver for compiler.                    */
/*                                                                      */
/************************************************************************/

main(argc,argv)
        int     argc;                   /* input arguement count        */
        char    *argv[];                /* input arguement ptrs         */
{
 puts("COMPILER 11/11/83");

 if (argc < 2)                          /* check for source file name   */
  {
   puts("Unspecified source file.");
   exit();
  }

/* initialize                                                           */
 if ((a_init(argv[1]) == ERROR) || (s_init(src_info.f_lang) == ERROR))
   exit();

 compile();                     /* generate pseudo-code for interpreter */

 a_wrap();
 s_wrap();

 if (src_info.f_edit)
  {
   execl("synde","synde",argv[1],2);
   puts("Cannot access SYNDE.");
  }
}
```

APPENDIX F

```
/*************************************************************************/
/*                                                                       */
/*          00000   00    00   00   00   00000    00000                 */
/*          00       00  00    000  00   00  00   00                    */
/*          00000     0000     00 0 00   00  00   0000                  */
/*             00      00      00  000   00  00   00                    */
/*          00000      00      00   00   00000    00000                 */
/*                                                                       */
/*       SYNtax-Directed Editor (c) Copyright December 1982             */
/*         CAPT. Scott Edward Ferguson, USAF, AFIT GCS-82D              */
/*                      Modified October 1983                           */
/*         CAPT. Michael L. McCracken, USAF, AFIT GCS-83D               */
/*                                                                       */
/*                             COMPILE.C                                 */
/*    SYNDE system generic entry point and utility routines for         */
/*    compiler use.                                                      */
/*                                                                       */
/*************************************************************************/

#include "synde.h"              /* system global information structures */

#include "code.h"               /* code generation structures          */
struct  code_word *code;        /* pseudo-code                         */

#include "symbol.h"             /* symbol table structures             */
struct  sym_table *symbol;      /* symbol table                        */

extern                          /* source file data                    */
struct  file_info src_info;     /*      in AST.C                       */

extern                          /* node production types               */
char    *types[];               /*      in language specific compiler  */

int     code_ptr,               /* index of next code cell             */
        sym_ptr,                /* top of symbol table                 */
        errors,                 /* error count                         */
        level,                  /* current lexical level               */
        offset,                 /* current offset in level             */
        body_type,              /* current sub-program type            */
        ret_type,               /* return type from a function         */
        ret_lst;                /* return list from a sub-program      */

BOOL    gerror,                 /* code table overflow error           */
        serror;                 /* symbol table overflow error         */
```

APPENDIX F

```
/*************************************************************************/
/*                                                                       */
/*      compile                                                          */
/*              Initialize compilation and generate code.                */
/*                                                                       */
/*************************************************************************/

compile()
{
        REG     size;                   /* code size in bytes           */
        char    code_name[20];          /* code file name               */
        FILE    *code_file;             /* code file descriptor         */
        int     i;

errors = level = sym_ptr = code_ptr = 0;
gerror = serror = FALSE;

/* allocate a clear space for code generation memory                    */
if (!(code = malloc(size = sizeof(struct code_word)*WORDS)))
        {
        puts("Insufficient memory for code.");
        return ERROR;
        }

/* allocate a clear space for symbol table memory                      */
if (!(symbol = malloc(sizeof(struct sym_table)*SYMBOLS)))
        {
        puts("Insufficient memory for symbols.");
        return ERROR;
        }

for (i = 5; i <= 9; ++i)                /* clear all error markers      */
        if (src_info.f_mark[i])
                clr_mark(src_info.f_mark[i]);

goal(src_info.f_root);                  /* goal                         */

/* delete old code file and save new one                               */
strcpy(code_name,src_info.f_name);
strcat(code_name,".cod");
puts("Code file = "); puts(code_name); puts(".");
unlink(code_name);
if ((code_file = creat(code_name,0644)) == ERROR ||
                write(code_file,code,size) != size ||
                close(code_file) == ERROR) {
        ioerr(code_file);
        puts("Cannot generate code file.");
        return ERROR;
        }
free(symbol);                   /* free symbol table (leave code)       */
return errors;
}
```

APPENDIX F

```
/****************************************************************************/
/*                                                                          */
/*      collect                                                             */
/*              Collect the 'set' symbols in a sibling list into the        */
/*              supplied string.  Return ERROR if an unsatisfied and        */
/*              unconditional set element node is encountered.             */
/*                                                                          */
/****************************************************************************/

int collect(node,str)
        int     node;
        char    *str;
{
        REG     next;

next = son(node);
for ( ; ; ) {
        if (value(next))
                *str++ = value(next);  /* transfer character          */
        else if (!chk_flag(next,OPEN)) {
                *str = 0;
                return ERROR;           /* unsatisfied uncond set      */
                }
        if (chk_flag(next,RTMOST)) {
                *str = 0;
                return SUCCESS;         /* done                        */
                }
        next = right(next);
        }
}
```

APPENDIX F

```
/************************************************************************/
/*                                                                      */
/*      compare                                                         */
/*              Compare the terminal 'set' children of two identifier   */
/*              nodes.  Return TRUE if equal.  Return ERROR if an       */
/*              unsatisfied and unconditional set element node is       */
/*              encountered.                                            */
/*                                                                      */
/************************************************************************/

int compare(node1,node2)
        int     node1,                  /* node "pointers"              */
                node2;
{
        char    id1[40],
                id2[40];

if (collect(node1,id1) == ERROR || collect(node2,id2) == ERROR)
        return ERROR;
if (strcmp(id1,id2))
        return FALSE;
return TRUE;
}

/************************************************************************/
/*                                                                      */
/*      number                                                          */
/*           Convert the terminal 'set' children numeric values from    */
/*           an integer node and return the unsigned numeric value.     */
/*           Return ERROR if an unsatisfied and unconditional set       */
/*           element node is encountered.                               */
/*                                                                      */
/************************************************************************/

number(node,base)
        int     node,                   /* "pointer" to subtree         */
                base;                   /* numeric base                 */
{
        REG     accum;                  /* accumulated value            */
        char    ch;                     /* digit from set               */
        int     next;                   /* sibling chain "pointer"      */

next = son(node);
accum = 0;                              /* accumulate digits            */
for ( ; ; ) {
        if (!(ch = value(next)) && !chk_flag(next,OPEN))
                return ERROR;  /* exit if unsatisfied unconditional set*/
        if (isnumeric(toupper(ch),base))        /* test base validity   */
                accum = accum * base + ((ch<='9') ? ch-'0' : ch-55);
        if (chk_flag(next,RTMOST))
                return accum;
        next = right(next);
        }
}

/**********************************************************************/
/*                                                                    */
/*      gen                                                           */
/*              Generate a pseudo-code instruction or label.  Return  */
/*              the address of the generated word.                    */
/*                                                                    */
/**********************************************************************/

gen(opcode,op1,op2,node)
        int     opcode,         /* pseudo-instruction op-code         */
                op1,            /* operand 1                          */
                op2,            /* operand 2                          */
                node;           /* node responsible for code          */
{
switch (opcode) {
        case _LAB:      code[code_ptr].c_label = TRUE;

        case _NOOP:     return code_ptr;

        default:        if (code_ptr >= WORDS) {
                                if (!gerror) {
                                        puts("Code table overflow.");
                                        ++errors;
                                        gerror = TRUE;
                                        }
                                return code_ptr;
                                }
                        code[code_ptr].c_opcode = opcode;
                        code[code_ptr].c_op1 = op1;
                        code[code_ptr].c_op2 = op2;
                        code[code_ptr].c_node = node;
                        return code_ptr++;
        }
}

/**********************************************************************/
/*                                                                    */
/*      fix                                                           */
/*              Fix a forward reference by setting op1 of the jump    */
/*              instruction at the specified address to the current   */
/*              code_ptr.                                             */
/*                                                                    */
/**********************************************************************/

fix(addr)
        int     addr;
{   /* fix jump and mark label                                        */
code[code[addr].c_op1 = code_ptr].c_label = TRUE;
}
```

APPENDIX F

```
/**********************************************************************/
/*                                                                    */
/*      find                                                          */
/*              Find a symbol in the symbol table.  Return the symbol */
/*              table index or ERROR if not found.                    */
/*                                                                    */
/**********************************************************************/

find(node)
        int     node;
{
        REG     i;                      /* symbols index              */

for (i = sym_ptr ; i ; --i)
        if (compare(symbol[i].sym_node,node) == TRUE)
                return i;
return ERROR;
}

/**********************************************************************/
/*                                                                    */
/*      place                                                         */
/*              Place a new item in the symbol table.  Return the new */
/*              symbol table index.                                   */
/*                                                                    */
/**********************************************************************/

int place(node,type)
        int     node,
                type;
{
if (sym_ptr+1 >= SYMBOLS) (              /* limit symbol count          */
        if (!serror) {
                puts("Symbol table overflow.");
                serror = TRUE;
                ++errors;
                }
        }
else {
        symbol[++sym_ptr].sym_node = node;
        symbol[sym_ptr].sym_level = level;
        symbol[sym_ptr].sym_addr = offset++;
        symbol[sym_ptr].sym_type = type;
        }
return sym_ptr;
}
```

APPENDIX F

```
/*********************************************************************/
/*                                                                   */
/*    error                                                          */
/*           Mark the specified node with an error and output the    */
/*           supplied error message.  Return TRUE when all error     */
/*           markers have been used.                                 */
/*                                                                   */
/*********************************************************************/

int error(node,str)
        int     node;                   /* node in error             */
        char    *str;                   /* error message string      */
{
        REG     i;                      /* marker index              */

/* find the first open error marker or reuse one with the same node  */
for (i = 5;
     i <= 8 && src_info.f_mark[i] && src_info.f_mark[i] != node;
     ++i);

puts("ERROR MARKER ");                  /* message                   */
putchar(i+'0'); puts(":  ");
puts(str); puts(".");
if (src_info.f_mark[i])
        clr_mark(node);                 /* clear any previous mark   */
set_flag(src_info.f_mark[i] = node,MARK);       /* set error marker  */
if (i >= 9)
        {
        puts("Error limit exceeded.");
        return TRUE;
        }
++errors;
return FALSE;
}
```

```
/*******************************************************************/
/*                                                                 */
/*      node_type                                                  */
/*              Return node type of the node specified.            */
/*                                                                 */
/*******************************************************************/

int node_type(node)
        int     node;
{
        REG     i;                      /* types index             */
        int     loc;

loc = s_ptr(link(link(prod(node))));   /* string location is in table  */
for (i = 0; types[i]; ++i)
        if (types[i] == loc)
                return i;
return ERROR;                           /* not found               */
}

/*******************************************************************/
/*                                                                 */
/*      fill_types                                                 */
/*              Fill types array with string locations in syntax   */
/*              description corresponding to strings in argument.  */
/*                                                                 */
/*******************************************************************/

fill_types()
{
        register char   *s;
        int             i;              /* types index             */

for (i = 0; types[i]; ++i) {            /* stop at 0 entry         */
s = s_ptr(link(4));                     /* location of first string     */
/*puts(types[i]);           make this into a non comment to find errors*/
                          /* in the types array.                   */
fflush(stdout);
        while (strcmp(s,types[i])) {    /* while strings not equal      */
                if (*s == 0xFFFF) {     /* error if no more in SDF      */
                        puts("Type-fill error: ");
                        puts(types[i]);
                        exit();
                        }
                while (*s++) ;          /* find next SDF string         */
                }
        types[i] = s;                   /* type is loc of SDF string    */
        }
}
```

APPENDIX F

```
#include "adal.h"

/*****************************************************************/
/*                                                               */
/*      old_ident(ifier)                                         */
/*                                                               */
/*****************************************************************/

int old_ident(node)
  int     node;
{
  REG     i;

if ((i = find(node)) != ERROR)
  return i;
if (error(node,"undeclared identifier"))
  return ERROR;
return place(node,TYPE_VAR);
}

/*****************************************************************/
/*                                                               */
/*      new_ident(ifier)                                         */
/*          Undeclared identifier search.  Return ERROR if the   */
/*          identifier is found in the symbol table at the current */
/*          level, otherwise enter it as a new symbol and return */
/*          the symbol table index.                              */
/*                                                               */
/*****************************************************************/

int new_ident(node)
  int     node;
{
  REG     i;                            /* symbol table index    */

                                    /* must not be at same level  */
if ((i = find(node)) != ERROR && symbol[i].sym_level == level)
  if (error(node,"identifier already declared"))
    return ERROR;
return place(node,0);                   /* place new symbol      */
}

/*****************************************************************/
/*                                                               */
/*      integer                                                  */
/*                                                               */
/*****************************************************************/

int integer(node)
  int     node;
{
return number(node,10);                 /* base 10 number        */
}

/******************************************************************/
/*                                                                */
/*      n_exp                                                     */
/*                                                                */
/******************************************************************/

int n_exp(node)
  int     node;
{
return exp(son(node));                      /* <expression>        */
}

/******************************************************************/
/*                                                                */
/*      dec_num                                                   */
/*                                                                */
/******************************************************************/

int dec_num(node)
  int     node;
{
return integer(son(node));
}

/******************************************************************/
/*                                                                */
/*      mul_op                                                    */
/*                                                                */
/******************************************************************/

int mul_op(node)
  int     node;
{
if (!son(node))
  if (error(node,noson))
    return ERROR;
  else
    return _MUL;
switch (node_type(son(node)))
  {
  case MUL_:                                          /* "*"      */
    return _MUL;
  case DIV_:                                          /* "/"      */
    return _DIV;
  case REM_:                                          /* "REM"    */
    return _REM;
  case MOD_:                                          /* "MOD"    */
    return _MOD;
  }
}
```

```
/******************************************************************/
/*                                                                */
/*      primary                                                   */
/*                                                                */
/******************************************************************/

int primary(node)
  int     node;
{
  REG     next;
  int     i;

if (!(next = son(node)))
  if (error(node,noson))
    return ERROR;
  else {
    gen(_LIT,i,0,node);
return TYPE_INT;
}
switch (node_type(next)) {
  case DEC_NUM_:                              /* <decimal_number>     */
    if ((i = dec_num(next)) == ERROR)
      return ERROR;
    gen(_LIT,i,0,next);                             /* load literal   */
    return TYPE_INT;
  case NAME_:                                          /* <name>      */
    if ((i = name(next)) == ERROR)
      return ERROR;
    if (!(symbol[i].sym_flags && TYPE_VAR))         /* VAR or CONST   */
      if (error(next,"must be constant or variable name"))
        return ERROR;
      else {
        gen(_LIT,i,0,next);
        return TYPE_INT;
          }
    gen(_LOD,level-symbol[i].sym_level,symbol[i].sym_addr,next);
    return symbol[i].sym_type;                /* load variable        */
  case N_EXP_:                                       /* <nested_exp>  */
    return n_exp(next);
  case CHAR_LIT_ :
    if ((i = char_lit(next)) == ERROR)
      return  ERROR;
    gen(_LIT,i,0,next);
    return  TYPE_CHAR;
  case BOOLEAN_VALUE_ :
    if ((i = bool_val(next)) == ERROR)
      return  ERROR;
    gen(_LIT,i,0,next);
    return  TYPE_BOOL;
  case F_CALL_ :
    return func_call(next);
  }
}

/*******************************************************************/
/*                                                                 */
/*      character literal                                          */
/*                                                                 */
/*******************************************************************/

int char_lit(node)
  int  node;

{
 int  next;

 next = son(node);
 if (chk_flag(next,OPEN))
   if (error(next,"character needed"))
     return  ERROR;
   else
     return  0;
 else
   return  value(next);
}

/*******************************************************************/
/*                                                                 */
/*      boolean value                                              */
/*                                                                 */
/*******************************************************************/

int bool_val(node)
  int  node;

{
 int  next;

 if (!(next = son(node)))
   if (error(next,noson))
     return  ERROR;
   else
     return  TRUE;
 if (node_type(next) == TRUE_)
   return  TRUE;
 else
   return  FALSE;
}
```

APPENDIX F

```
#include "ada1.h"
/*****************************************************************/
/*                                                               */
/*      rel_op                                                   */
/*                                                               */
/*****************************************************************/

int rel_op(node)
  int     node;
{
if (!son(node))
  if (error(node,noson))
    return ERROR;
  else
    return _EQU;
switch (node_type(son(node))) {
  case EQU_:                          /* "="                    */
    return _EQU;
  case NEQ_:                          /* "/="                   */
    return _NEQ;
  case LES_:                          /* "< "                   */
    return _LES;
  case LEQ_:                          /* "<="                   */
    return _LEQ;
  case GRT_:                          /* "> "                   */
    return _GRT;
  case GEQ_:                          /* ">="                   */
    return _GEQ;
  }
}

/*****************************************************************/
/*                                                               */
/*      add_op                                                   */
/*                                                               */
/*****************************************************************/

int add_op(node)
  int     node;
{
if (!son(node))
  if (error(node,noson))
    return ERROR;
  else
    return _ADD;
switch (node_type(son(node))) {
  case PLUS_:                         /* "+"                    */
    return _ADD;
  case MINUS_:                        /* "-"                    */
    return _SUB;
  }
}
```

```
/*****************************************************************/
/*                                                               */
/*      fact(or)s                                                */
/*                                                               */
/*****************************************************************/

int facts(node)
  int     node;
{
  REG     next;
  int     op,                        /* multiply operator       */
          type;

if ((op = mul_op(next = son(node))) == ERROR)   /* <mul_op>         */
  return ERROR;
if ((type = factor(right(next))) == ERROR)           /* <factor>  */
  return ERROR;
if (type != TYPE_INT)
  {
   error(next,"integer type expected");
   return ERROR;
  }
gen(op,0,0,next);                      /* multiply operation   */
return type;
}

/*****************************************************************/
/*                                                               */
/*      rel(ation)al                                             */
/*                                                               */
/*****************************************************************/

int relal(node)
  int     node;
{
  REG     next;
  int     op,                        /* relational operators   */
          type;

if ((op = rel_op(next = son(node))) == ERROR)   /* <rel_op>         */
  return ERROR;
if ((type = s_exp(right(next))) == ERROR)        /* <simple_exp>  */
  return ERROR;
gen(op,0,0,next);                      /* relational operation */
return type;
}
```

APPENDIX F

```
/**********************************************************************/
/*                                                                    */
/*      terms                                                         */
/*                                                                    */
/**********************************************************************/

int terms(node)
  int     node;
{
  REG     next;
  int     op,                                  /* adding operator    */
         type;

if ((op = add_op(next = son(node))) == ERROR)     /* <add_op>        */
  return ERROR;
if ((type = term(right(next))) == ERROR)                /* <term>    */
  return ERROR;
if (type != TYPE_INT)
  {
  error(next,"integer type expected");
  return  ERROR;
  }
gen(op,0,0,next);                            /* adding operation     */
return type;
}

/**********************************************************************/
/*                                                                    */
/*      term                                                          */
/*                                                                    */
/**********************************************************************/

int term(node)
  int     node;
{
  REG     next,ltype,rtype;

if ((ltype = factor(next = son(node))) == ERROR)    /* <factor>      */
  return ERROR;
while (!chk_flag(next,RTMOST))                      /* {<factors>}   */
  if (!chk_flag(next = right(next),OPEN))
    if ((rtype = facts(next;) == ERROR)
      return ERROR;
    else if ((ltype != TYPE_INT)  ||
        (rtype != TYPE_INT))
       {
        error(node,"type mismatch.  operation not defined");
        return ERROR;
       }
return ltype;
}
```

```
/**********************************************************************/
/*                                                                    */
/*      factor                                                        */
/*                                                                    */
/**********************************************************************/

int factor(node)
  int     node;
{
 int next;

 if (!(next = son(node)))
   if (error(node,noson))
     return  ERROR;
   else
    {
     gen(_LIT,0,0,node);
     return  TYPE_INT;
    }

 switch (node_type(next))
  {
   case EXP_PRIMARY_ :
     return  exp_prim(next);
   case ABS_PRIMARY_ :
     return  abs_prim(next);
   case NOT_PRIMARY_ :
     return  not_prim(next);
  }
}
```

APPENDIX F

```
#include "adal.h"

/*****************************************************************/
/*                                                               */
/*      u_op                                                     */
/*                                                               */
/*****************************************************************/

int u_op(node)
  int     node;
{
if (!son(node))
  if (error(node,noson))
    return ERROR;
  else
    return _NOOP;
switch (node_type(son(node))) {
  case    PLUS_:                           /* "+"          */
    return _NOOP;
  case    MINUS_:                            /* "-"        */
    return _NEG;
  }
}

/*****************************************************************/
/*                                                               */
/*      or_rel(ation)                                            */
/*                                                               */
/*****************************************************************/

int or_rel(node)
  int     node;
{
 int type;
if ((type = relation(son(node))) == ERROR)       /* <relation>    */
  return ERROR;
if (type != TYPE_BOOL)
  {
   error(son(node),bool_exp);
   return  ERROR;
  }
gen(_OR,0,0,node);                         /* or operation     */
return type;
}
```

APPENDIX F

```
/*****************************************************************/
/*                                                               */
/*      w_clause                                                 */
/*                                                               */
/*****************************************************************/

int w_clause(node)
  int     node;
{
return exp(son(node));                        /* <expression>    */
}

/*****************************************************************/
/*                                                               */
/*      and_rel(ation)                                           */
/*                                                               */
/*****************************************************************/

int and_rel(node)
  int     node;
{
 int type;

if ((type = relation(son(node))) == ERROR)        /* <relation>   */
  return ERROR;
if (type != TYPE_BOOL)
  {
   error(son(node),bool_exp);
   return  ERROR;
  }
gen(_AND,0,0,node);                           /* and operation   */
return type;
}

/*****************************************************************/
/*                                                               */
/*      and_then_rel(ation)                                      */
/*                                                               */
/*****************************************************************/

int and_then_rel(node)
  int   node;
{
  int   ltype;

 if ((ltype = relation(son(node))) == ERROR)      /* <relation>   */
   return ERROR;
 if (ltype != TYPE_BOOL)
   if (error(son(node),bool_exp))
     return  ERROR;
 return ltype;
}
```

APPENDIX F

```
/*******************************************************************/
/*                                                                 */
/*       or_else_rel(ation)                                        */
/*                                                                 */
/*******************************************************************/

int or_else_rel(node)
  int   node;
(
  int   ltype;

 if ((ltype = relation(son(node))) == ERROR)       /* <relation>    */
   return ERROR;
 if (ltype != TYPE_BOOL)
   if (error(son(node),bool_exp))
     return  ERROR;

 return ltype;
)

/*******************************************************************/
/*                                                                 */
/*       xor_rel(ation)                                            */
/*                                                                 */
/*******************************************************************/

int xor_rel(node)
  int   node;
(
  int   ltype;

 if ((ltype = relation(son(node))) == ERROR)       /* <relation>    */
   return ERROR;
 if (ltype != TYPE_BOOL)
   if (error(son(node),bool_exp))
     return  ERROR;
 gen(_XOR,0,0,node);                             /* xor operation    */
 return ltype;
)

/*******************************************************************/
/*                                                                 */
/*       rel(ation)_part                                           */
/*                                                                 */
/*******************************************************************/

int rel_part(node)
  int     node;
(
return relal(son(node));                        /* <relational>     */
)
```

APPENDIX F

```
/*****************************************************************/
/*                                                               */
/*      s_exp                                                    */
/*                                                               */
/*****************************************************************/

int s_exp(node)
  int     node;
{
  REG     next;
  int     op,                              /* unary op-code     */
          type;

op = _NOOP;
if (node_type(next = son(node)) == U_OP_) {
  if (!chk_flag(next,OPEN))
    if ((op = u_op(next)) == ERROR)       /* [<unary_operator>]  */
      return ERROR;
  next = right(next);
  }
if ((type = term(next)) == ERROR)            /* <term>         */
  return ERROR;
if ((type != TYPE_INT) &&
    (op != _NOOP))
  {
   error(next,int_exp);
   return  ERROR;
  }
gen(op,0,0,son(node));                   /* unary operation      */
while (!chk_flag(next,RTMOST))
 {
  if (type != TYPE_INT)
    if (error(next,int_exp))
      return  ERROR;
  if (!chk_flag(next = right(next),OPEN))
    if ((type = terms(next)) == ERROR)         /* {<terms>}      */
      return ERROR;
  }
 return type;
 }
```

```
/*****************************************************************/
/*                                                               */
/*      i_clause                                                 */
/*                                                               */
/*****************************************************************/

int i_clause (node)
  int     node;
{
return w_clause(son(node));            /* <while_clause>        */
}

/*****************************************************************/
/*                                                               */
/*      else_part                                                */
/*                                                               */
/*****************************************************************/

int else_part(node)
  int     node;
{
return seq_of_stats(son(node));        /* <seq_of_stats>        */
}
```

APPENDIX F

```
#include "adal.h"

/*********************************************************************/
/*                                                                   */
/*      elsif_part                                                   */
/*                                                                   */
/*********************************************************************/

int elsif_part(node)
  int     node;
{
  REG     next;
  int     label1,                           /* label address storage  */
          type;

if ((type = exp(next = son(node))) == ERROR)    /* <expression>     */
  return ERROR;
if (type != TYPE_BOOL)
  if (error(next,bool_exp))
    return  ERROR;
label1 = gen(_JPC,0,0,0);                   /* jump to next else      */
if (seq_of_stats(right(next)) == ERROR)    /* <seq_of_stats>         */
  return ERROR;
return label1;
}
```

```
/***********************************************************************/
/*                                                                     */
/*      and_comp                                                       */
/*                                                                     */
/***********************************************************************/

int and_comp(node)
  int     node;
{
  REG     next,ltype;

 if ((ltype = relation(next = son(node))) == ERROR)   /* <relation> */
   return ERROR;
 if (ltype != TYPE_BOOL)
   if (error(son(node),bool_exp))
     return  ERROR;
 while (!chk_flag(next,RTMOST))                  /* {<and_rel>}   */
   if (!chk_flag(next = right(next),OPEN))
     if ((ltype = and_rel(next)) == ERROR)
       return ERROR;
 return ltype;
}

/***********************************************************************/
/*                                                                     */
/*      or_comp                                                        */
/*                                                                     */
/***********************************************************************/

int or_comp(node)
  int     node;
{
  REG     next,ltype;

 if ((ltype = relation(next = son(node))) == ERROR)   /* <relation> */
   return ERROR;
 if (ltype != TYPE_BOOL)
   if (error(son(node),bool_exp))
     return  ERROR;
 while (!chk_flag(next,RTMOST))                  /* {<or_rel>}  */
   if (!chk_flag(next = right(next),OPEN))
     if ((ltype = or_rel(next)) == ERROR)
       return ERROR;
 return ltype;
}
```

APPENDIX F

```
/************************************************************************/
/*                                                                      */
/*        and_then_comp                                                 */
/*                                                                      */
/************************************************************************/

int and_then_comp(node)
  int     node;
{
  REG     next,ltype,chain,label;

 if ((ltype = relation(next = son(node))) == ERROR)   /* <relation> */
   return ERROR;
 if (ltype != TYPE_BOOL)
   if (error(son(node),bool_exp))
     return  ERROR;
 chain = label = gen(_LAB,0,0,next);
 while (!chk_flag(next,RTMOST))              /* <<and_then_rel>>  */
   if (!chk_flag(next = right(next),OPEN))
     {
      label = code[label].c_op1 = gen(_AND_THEN,0,0,next);
      if ((ltype = and_then_rel(next)) == ERROR)
       return ERROR;
     }
 do
  {
   label = code[chain].c_op1;
   fix(chain);
  }
 while (chain = label);

 return ltype;
}
```

APPENDIX F

```
/*********************************************************************/
/*                                                                   */
/*       or_else_comp                                                */
/*                                                                   */
/*********************************************************************/

int or_else_comp(node)
  int     node;
{
  REG     next,ltype,chain,label;

 if ((ltype = relation(next = son(node))) == ERROR)   /* <relation> */
   return ERROR;
 if (ltype != TYPE_BOOL)
   if (error(son(node),bool_exp))
     return  ERROR;
 chain = label = gen(_LAB,0,0,next);
 while (!chk_flag(next,RTMOST))                /* <<or_else_rel>>   */
   if (!(chk_flag(next = right(next)),OPEN))
    {
     label = code[label].c_op1 = gen(_OR_ELSE,0,0,next);
     if ((ltype = or_else_rel(next)) == ERROR)
       return ERROR;
    }
 do
  {
   label = code[chain].c_op1;
   fix(chain);
  }
 while (chain = label);

 return ltype;
}
```

APPENDIX F

```
/***********************************************************************/
/*                                                                     */
/*      xor_comp                                                       */
/*                                                                     */
/***********************************************************************/

int xor_comp(node)
  int     node;
{
  REG     next,ltype;

 if ((ltype = relation(next = son(node))) == ERROR)   /* <relation> */
   return ERROR;
 if (ltype != TYPE_BOOL)
   if (error(son(node),bool_exp))
     return  ERROR;
 while (!chk_flag(next,RTMOST))                 /* {<xor_rel>}    */
   if (!chk_flag(next = right(next),OPEN))
     if ((ltype = xor_rel(next)) == ERROR)
       return ERROR;
 return ltype;
}


/*********************************************************************/
/*                                                                   */
/*      relation                                                     */
/*                                                                   */
/*********************************************************************/

int relation(node)
  int     node;
{
  REG     next,ltype,rtype;

if ((ltype = s_exp(next = son(node))) == ERROR)   /* <s_expression> */
  return ERROR;
if (!chk_flag(next,RTMOST) && !chk_flag(next = right(next),OPEN))
  {
 if ((rtype = rel_part(next)) == ERROR)   /* [<relation_part>]      */
    return  ERROR;
 if (ltype != rtype)
   {
    if (error(next,"types must match"))
      return  ERROR;
   }
   else
   return  TYPE_BOOL;
  }
return ltype;
}
```

APPENDIX F

```
/**********************************************************************/
/*                                                                    */
/*      name                                                          */
/*                                                                    */
/**********************************************************************/

int name(node)
  int     node;
{
return old_ident(son(node));              /* (old) <identifier>     */
}

/**********************************************************************/
/*                                                                    */
/*      loop_stat                                                     */
/*                                                                    */
/**********************************************************************/

int loop_stat(node)
  int     node;
{
  REG     next;
  int     label1,label2;                  /* label address storage  */

label1 = gen(_LAB,0,0,0);                        /* start of loop    */
label2 = -1;
if (node_type(next = son(node)) == I_CLAUSE_) {
  if (!chk_flag(next,OPEN)) {
    if (i_clause(next) == ERROR)           /* [<iteration_clause>] */
      return ERROR;
    label2 = gen(_JPC,0,0,0);              /* jump to exit loop    */
    }
  next = right(next);
  }
if (seq_of_stats(next) == ERROR)            /* <seq_of_stats>       */
  return ERROR;
gen(_JMP,label1,0,0);                     /* jump to loop start     */
if (label2 != -1)
  fix(label2);                            /* fix jump to exit loop  */
return SUCCESS;
}
```

APPENDIX F

```
/**********************************************************************/
/*                                                                    */
/*      if_stat                                                       */
/*                                                                    */
/**********************************************************************/

int if_stat(node)
  int     node;
{
  REG     next;
  int     label1,label2,chain,          /* label address storage    */
          type;

if ((type = exp(next = son(node))) == ERROR)     /* <expression>    */
  return ERROR;
if (type != TYPE_BOOL)
  if (error(next,bool_exp))
    return  ERROR;
label1 = gen(_JPC,0,0,0);                       /* jump to else     */
if (seq_of_stats(next = right(next)) == ERROR)  /* <seq_of_stats>   */
  return ERROR;
chain = label2 = gen(_JMP,0,0,0);         /* jump past if_stat      */
while (!chk_flag(next,RTMOST) && node_type(right(next)) == ELSIF_P_)
  if (!chk_flag(next = right(next),OPEN)) {   /* {<elsif_part>}     */
    fix(label1);                          /* fix jump to else       */
    if ((label1 = elsif_part(next)) == ERROR)
      return ERROR;
          /* chain locations of jump-past-if_stat instructions      */
    label2 = code[label2].c_op1 = gen(_JMP,0,0,0);
    }
fix(label1);
if (!chk_flag(next,RTMOST))
  if (!chk_flag(next = right(next),OPEN))
    if (else_part(next) == ERROR)                /* [<else_part>]   */
      return ERROR;
do {
  label2 = code[chain].c_op1;
  fix(chain);                       /* fix jumps-past-if_stat       */
  } while (chain = label2);
return SUCCESS;
}
```

APPENDIX F

```
#include "adal.h"

/*********************************************************************/
/*                                                                   */
/*      proc_call                                                    */
/*                                                                   */
/*********************************************************************/

int proc_call(node)
  int     node;
{
  REG     next;
  int     sym;                              /* symbol table index   */
if ((sym = name(next = son(node))) == ERROR)            /* <name>  */
  return ERROR;
if (symbol[sym].sym_flags && TYPE_PROC) /* must be procedure name */
  if (error(next,"must be procedure name"))
    return ERROR;
  else
    return SUCCESS;
gen(_CAL,level-symbol[sym].sym_level,symbol[sym].sym_addr,next);
return SUCCESS;
}

/*********************************************************************/
/*                                                                   */
/*      ident(ifier)s                                                */
/*                                                                   */
/*********************************************************************/

int idents(node)
  int     node;
{
return new_ident(son(node));              /* (new) <identifier>  */
}
```

```
/*****************************************************************/
/*                                                               */
/*      a(ssignment)_stat                                        */
/*                                                               */
/*****************************************************************/

int a_stat(node)
  int     node;
{
  REG     next;
  int     sym,                          /* symbol table index     */
          ltype,rtype;

if ((sym = name(next = son(node))) == ERROR)      /* <name>      */
  return ERROR;
if (!(symbol[sym].sym_flags && TYPE_VAR))         /* must be VAR */
  if (error(next,"must be variable name"))
    return ERROR;
if (!(symbol[sym].sym_flags && TYPE_CONST))
  if (symbol[sym].sym_flags && TYPE_INIT)
    if (error(next,"constant initialization allowed only once"))
      return  ERROR;
ltype = symbol[sym].sym_type;
if ((rtype = exp(right(next))) == ERROR)         /* <expression> */
  return ERROR;
if (ltype != rtype)
  if (error(node,"types must match"))
    return  ERROR;

gen(_STO,level-symbol[sym].sym_level,symbol[sym].sym_addr,son(node));
return SUCCESS;
}
```

APPENDIX F

```
/*****************************************************************/
/*                                                               */
/*      exp(ression)                                             */
/*                                                               */
/*****************************************************************/

int exp(node)
  int     node;
{
  REG     next;

if (!(next = son(node)))
  if (error(node,noson))
    return ERROR;
  else
    return SUCCESS;
switch (node_type(next)) {
  case RELATION_:
          return relation(next);            /* <relation>      */
  case AND_COMP_:
    return and_comp(next);                  /* <and_comp>      */
  case OR_COMP_:
    return or_comp(next);                   /* <or_comp>       */
  case AND_THEN_COMP_:
    return  and_then_comp(next);          /* <and_then_comp> */
  case OR_ELSE_COMP_:
    return  or_else_comp(next);            /* <or_else_comp> */
  case XOR_COMP_:
    return  xor_comp(next);                 /* <xor_comp>      */
  }
}
```

APPENDIX F

```
/*****************************************************************/
/*                                                               */
/*     sub(type)_ind(ication)                                    */
/*                                                               */
/*****************************************************************/

int sub_ind(node)
  int     node;
{
 int     next,sym;

 if (!(next = son(node)))
  if (error(node,noson))
    return ERROR;
  else
    return TYPE_INT;

 switch (node_type(next))
  {
   case  INTEGER_:
     return  TYPE_INT;
   case  BOOLEAN_:
     return  TYPE_BOOL;
   case  CHAR_:
     return  TYPE_CHAR;
   case  IDENT_:
     sym = old_ident(next);
     if (!(symbol[sym].sym_flags && TYPE_TYPE))
       if (error(next,"type name expected"))
         return  ERROR;
       else
         return  TYPE_INT;
     else
       return  symbol[sym].sym_type;
  }
}
```

APPENDIX F

```
/*****************************************************************/
/*                                                               */
/*      s(imple)_stat                                            */
/*                                                               */
/*****************************************************************/

int s_stat(node)
  int     node;
{
  REG     next;

if (!(next = son(node)))
  if (error(node,noson))
    return ERROR;
  else
    return SUCCESS;
switch (node_type(next)) {
  case A_STMT_:
    return a_stat(next);                  /* <assignment_stat>   */
  case P_CALL_:
    return proc_call(next);                     /* <proc_call>   */
  case IF_STMT_:
    return if_stat(next);                       /* <if_stat>     */
  case LP_STMT_:
    return loop_stat(next);                     /* <loop_stat>   */
  case RET_STMT_ :
    return  ret_stat(next);
  case COMMENT_ :
  case NULL_STMT_ :
    break;

  }
}

/*****************************************************************/
/*                                                               */
/*      initial                                                  */
/*                                                               */
/*****************************************************************/

int initial(node)
  int     node;
{
return exp(son(node));                          /* <expression>  */
}
```

APPENDIX F

```
/**********************************************************************/
/*                                                                    */
/*      obj(ect)_decl                                                 */
/*                                                                    */
/**********************************************************************/

int obj_decl(node)
  int     node;
{
  REG     next;
  int     stype,                  /* type of variables declared  */
          type,                   /* symbol table flags          */
          strt;                   /* first symbol table entry    */

strt = sym_ptr + 1;
if (id_list(next = son(node)) == ERROR)          /* <id_list>     */
  return ERROR;
type |= TYPE_VAR;
if (node_type(next = right(next)) == CONST_) {   /* ["constant"]  */
  if (!chk_flag(next,OPEN))
    type |= TYPE_CONST;
  next = right(next);
  }
if ((stype = obj_type(next)) == ERROR)           /* <object_type> */
  return ERROR;
if (!chk_flag(next,RTMOST) && !chk_flag(next = right(next),OPEN)) {
  if (initial(next) == ERROR)                    /* [<initial>]   */
    return ERROR;
  else
    type |= TYPE_INIT;
  }
else
 {
  gen(_LIT,0,0,next);                 /* no initialization, use 0 */
 }
symbol[strt].sym_type = stype;            /* set each symbol type  */
symbol[strt].sym_flags |= type;           /* set flags for symbol  */
while (++strt <= sym_ptr) {
  symbol[strt].sym_type = stype;
  symbol[strt].sym_flags |= type;
  gen(_CPY,0,0,symbol[strt].sym_node);    /* copy initial for each */
  }
return SUCCESS;
}
```

APPENDIX F

```
/*******************************************************************/
/*                                                                 */
/*       seq_of_stats                                              */
/*                                                                 */
/*******************************************************************/

int seq_of_stats(node)
  int     node;
{
  REG     next;

if (stat(next = son(node)) == ERROR)          /* <stat>          */
  return ERROR;
while (!chk_flag(next,RTMOST))                /* {<stat>}        */
  if (!chk_flag(next = right(next),OPEN))
    if (stat(next) == ERROR)
      return ERROR;
return SUCCESS;
}

/*******************************************************************/
/*                                                                 */
/*       prog(ram)_comp(onent)                                     */
/*                                                                 */
/*******************************************************************/

int prog_comp(node)
  int     node;
{
 int  next;

if (!(next = son(node)))
  if (error(node,noson))
    return ERROR;
  else
    return SUCCESS;
switch (node_type(next))
 {
  case P_BODY_ :
      return  proc_body(next);
  case F_BODY_ :
      return  func_body(next);


  }
}
```

```
/****************************************************************/
/*                                                              */
/*      decl                                                    */
/*                                                              */
/****************************************************************/

int decl(node)
  int     node;

{
return obj_decl(son(node));
}

/****************************************************************/
/*                                                              */
/*       proc_spec                                              */
/*                                                              */
/****************************************************************/

int proc_spec(node)
  int     node;
{
  REG     sym;                              /* symbol table index  */

if ((sym = new_ident(son(node))) == ERROR)   /* new <identifier> */
  return ERROR;
symbol[sym].sym_type |= TYPE_PROC;           /* set symbol type */
return sym;
}

/****************************************************************/
/*                                                              */
/*      comp(ilation)_unit                                      */
/*                                                              */
/****************************************************************/

comp_unit(node)
  int     node;
{
 int  next;

if (!(next = son(node)))
  if (error(node,noson))
    return ERROR;
  else
    return SUCCESS;
switch (node_type(next))
 {
  case P_BODY :
      return  proc_body(next);
  case F_BODY :
      return  func_body(next);
 }
}


/****************************************************************/
/*                                                              */
/*      proc_body                                               */
/*                                                              */
/****************************************************************/

int proc_body(node)
  int     node;
{
  REG     next;                         /* thru sibling list    */
  int     label,                        /* label resolution     */
          cnt,                  /* number of variables declared */
          sym,                  /* symbol entry # of proc name  */
          i;

 if (node_type(next = son(node)) == PROG_HDR_)
   next = right(next);
 if ((sym = proc_spec(next)) == ERROR)      /* <proc_spec>          */
   return ERROR;
 symbol[sym].sym_addr = gen(_LAB,0,0,0);  /* proc entry address  */
 ++level;                                 /* bump lexical level  */
 offset = 0;                              /* zero level offset   */
 while (node_type(next = right(next)) == DECL_)       /* {<decl>} */
   if (!chk_flag(next,OPEN))
     if (decl(next) == ERROR)
       return ERROR;
 cnt = offset;
 offset = 0;
 label = gen(_JMP,0,0,0);               /* jump around procedures */
 while (node_type(next) == P_COMP_) {   /* {<program_component>}  */
   if (!chk_flag(next,OPEN))
     if (prog_comp(next) == ERROR)
       return ERROR;
   next = right(next);
   }
 fix(label);                          /* fix jump around procedures */
 body_type = TYPE_PROC;
 ret_lst = 0;
 if (seq_of_stats(next) == ERROR)          /* <seq_of_stats>       */
   return ERROR;
 if (!chk_flag(next,RTMOST) && !chk_flag(next = right(next),OPEN)) {
   if ((i = old_ident(next)) == ERROR)       /* [<identifier>]     */
     return ERROR;
   if (i != sym)
     if (error(next,"procedure id's do not match"))
       return ERROR;                   /* must == first identifier */
   }
```

```
if (ret_lst)
  do
   {
    i = code[ret_lst].c_op];
    fix(ret_lst);
   } while(ret_lst = i);

gen(_DCS,cnt,0,node);            /* remove variable stack space */
--level;                              /* restore lexical level  */
while (symbol[sym_ptr].sym_level > level)   /* peel symbol table */
  symbol[sym_ptr--].sym_flags = 0;  /* clear flags field.        */
                                    /* leave procedure name.     */
gen(_RET,0,0,node);                   /* return from procedure  */
return SUCCESS;
}
```

APPENDIX F

```
#include  "adal.h"

/***********************************************************************/
/*                                                                     */
/*      goal                                                           */
/*          <compilation> is ADAO language grammar goal symbol.       */
/*                                                                     */
/***********************************************************************/

int goal(node)
 int     node;
{
 noson = "incomplete program fragment";
 bool_exp = "boolean type expected";
 int_exp = "integer type expected";

 types[MUL_]     =      "*";
 types[PLUS_]    =      "+";
 types[MINUS_]   =      "-";
 types[DIV_]     =      "/";
 types[NEQ_]     =      "/=";
 types[LES_]     =      "< ";
 types[LEQ_]     =      "<=";
 types[EQU_]     =      "=";
 types[GRT_]     =      "> ";
 types[GEQ_]     =      ">=";
 types[A_STMT_]  =      "assignment_stat";
 types[AND_COMP_]=      "and_comp";
 types[CONST_]   =      "constant";
 types[DEC_NUM_] =      "decimal_number";
 types[DECL_]    =      "decl";
 types[ELSIF_P_] =      "elsif_part";
 types[IF_STMT_] =      "if_stat";
 types[I_CLAUSE_]=      "iteration_clause";
 types[LP_STMT_] =      "loop_stat";
 types[NAME_]    =      "name";
 types[N_EXP_]   =      "nested_exp";
 types[OR_COMP_] =      "or_comp";
 types[P_CALL_]  =      "proc_call";
 types[P_COMP_]  =      "program_component";
 types[RELATION_]=      "relation";
 types[U_OP_]    =      "unary_operator";
 types[REM_]     =      "rem";
 types[MOD_]     =      "mod";
 types[AND_THEN_COMP_] =     "and_then_comp";
 types[OR_ELSE_COMP_] =      "or_else_comp";
 types[XOR_COMP_]     =      "xor_comp";
 types[INTEGER_] =      "integer";
 types[BOOLEAN_] =      "boolean";
 types[CHAR_LIT_] =     "char_lit";
 types[IDENT_]   =      "identifier";
 types[TRUE_]    =      "true";
types[FALSE_]   =       "false";
types[F_CALL_] =       "func_call";
types[F_BODY_] =        "func_body";
types[P_BODY_] =        "proc_body";
types[RET_STMT_] =      "return_stat";
types[BOOLEAN_VALUE_] =  "boolean_value";
types[EXP_PRIMARY_] =   "exp_primary";
types[ABS_PRIMARY_] =   "abs_primary";
types[NOT_PRIMARY_] =   "not_primary";
types[EXPON_PART_] =    "expon_part";
types[CHAR_]   =        "char";
types[PROC_RET_]  =     "procedure_return";
types[FUNC_RET_]  =     "function_return";
types[COMMENT_]   =     "comment";
types[NULL_STMT_] =     "null_stat";
types[PROG_HDR_]  =     "program_header";
types[LAST_PROD_] =       0;
fill_types();
return comp_unit(son(node));          /* <compilation_unit>     */
}
```

APPENDIX F

```
#include "adal.h"

/*****************************************************************/
/*                                                               */
/*      func_body                                                */
/*                                                               */
/*****************************************************************/

int func_body(node)
       int     node;
{
       REG     next;               /* thru sibling list           */
       int     label,              /* label resolution            */
               cnt,                /* number of variables declared */
               sym,                /* symbol entry # of func name  */
               i;
 if (node_type(next = son(node)) == PROG_HDR_)
   next = right(next);
 if ((sym = func_spec(next)) == ERROR)   /* <func_spec> */
   return ERROR;
 symbol[sym].sym_addr = gen(_LAB,0,0,0);      /* func entry address */
 ++level;                                     /* bump lexical level */
 offset = 0;                                 /* zero level offset   */
 while (node_type(next = right(next)) == DECL_)       /* <<decl>>  */
   if (!chk_flag(next,OPEN))
     if (decl(next) == ERROR)
       return ERROR;
 cnt = offset;
 offset = 0;
 label = gen(_JMP,0,0,0);                 /* jump around functions  */
 while (node_type(next) == P_COMP_)       /* <<program_component>> */
  {
   if (!chk_flag(next,OPEN))
     if (prog_comp(next) == ERROR)
       return ERROR;
    next = right(next);
 }
 fix(label);                         /* fix jump around functions   */
 body_type = TYPE_FUNC;              /* indicate working on function*/
 ret_type = symbol[sym].sym_type;    /* indicate return type        */
 ret_lst = 0;
 if (seq_of_stats(next) == ERROR)     /* <seq_of_stats>              */
   return ERROR;
 if (!chk_flag(next,RTMOST) && !chk_flag(next = right(next),OPEN))
  {
   if ((i = old_ident(next = son(next))) == ERROR)
      return ERROR;
   if (i != sym)
     if (error(next,"function designators do not match"))
       return ERROR;               /* must == first identifier    */
  }
if (ret_lst)                      /* check if return list exists  */
  do
   {
    i = code[ret_lst].c_op;
    fix(ret_lst);
   } while (ret_lst = i);
else
  if (error(node,"functions must contain a return statement"))
    return ERROR;
gen(_STO,0,-4,node);              /* save return value            */
gen(_DCS,cnt,0,node);             /* remove variable stack space  */
--level;                             /* restore lexical level     */
while (symbol[sym_ptr].sym_level > level) /* reset symbol table   */
  --sym_ptr;                          /*   leaving function name  */
gen(_RET,0,0,node);                /* return from function        */
return SUCCESS;
}

/*****************************************************************/
/*                                                               */
/*      function_specification                                   */
/*                                                               */
/*****************************************************************/

int func_spec(node)
  int  node;

{
 REG  next,sym;

 if ((sym = desig(next = son(node))) == ERROR)
   return  ERROR;

 symbol[sym].sym_flags |= TYPE_FUNC;
 if ((symbol[sym].sym_type = sub_ind(right(next))) == ERROR)
   return  ERROR;
 return  sym;
}
```

```
/*******************************************************************/
/*                                                                 */
/*      function call                                              */
/*                                                                 */
/*******************************************************************/

int func_call(node)
  int  node;

{
 int  next,sym;

 if ((sym = old_ident(next = son(son(node)))) == ERROR)
   return  ERROR;
 if (!(symbol[sym].sym_flags && TYPE_FUNC))
   { error(next,"function name expected");
     return ERROR;
   }
 gen(_LIT,0,0,node);                    /* create space for return  */
                                        /* value                    */
 gen(_CAL,level-symbol[sym].sym_level,symbol[sym].sym_addr,next);
 return symbol[sym].sym_type;
}


/*******************************************************************/
/*                                                                 */
/*      designator                                                 */
/*                                                                 */
/*******************************************************************/

int desig(node)
  int  node;

{
 return  (new_ident(son(node)));
}
```

APPENDIX F

```
/*********************************************************************/
/*                                                                   */
/*      return_statement                                             */
/*                                                                   */
/*********************************************************************/

int ret_stat(node)
  int  node;

{
 REG next;

 if (!(next = son(node)))
  {
   error(node,noson);
   return ERROR;
  }

 switch  (node_type(next))
  {
   case  PROC_RET_ :
       return  proc_ret(next);
   case  FUNC_RET_ :
       return  func_ret(next);
  }
}

/*********************************************************************/
/*                                                                   */
/*      procedure_return - accept a procedure return.                */
/*                                                                   */
/*********************************************************************/

int proc_ret(node)
  int  node;

{
 if (body_type == TYPE_FUNC)
   if (error(node,"functions must return a value"))
     return  ERROR;
   else
     gen(_LIT,0,0,node);

 ret_lst = gen(_JMP,ret_lst,0,node);
 return  SUCCESS;
}
```

APPENDIX F

```
/*********************************************************************/
/*                                                                   */
/*      function_return - accept a function return.                  */
/*              it must return the correct type.  return type        */
/*              is stored globally in ret_type.                      */
/*                                                                   */
/*********************************************************************/

int  func_ret(node)
  int  node;

{
 int type,next;

 if (body_type == TYPE_PROC)
   if (error(node,"procedures cannot return a value"))
     return  ERROR;
   else
    {
     ret_lst = gen(_JMP,ret_lst,0,node);
     return  SUCCESS;
    }

 if ((type = exp(next = son(node))) == ERROR)
   return  ERROR;
 if (type != ret_type)
   if (error(next,"return expression of wrong type"))
     return  ERROR;
 ret_lst = gen(_JMP,ret_lst,0,node);
 return  SUCCESS;
}
```

APPENDIX F

```
#include "ada1.h"

/*****************************************************************/
/*                                                               */
/*      exponent primary - accept a primary with an optional     */
/*              exponent.  if the exponent exists it must be     */
/*              an integer and positive.                         */
/*                                                               */
/*****************************************************************/

int  exp_prim(node)
  int  node;

{
 int  type1,type2,next;

 type1 = primary(next = son(node));
 if (node_type(next = right(next)) != EXPON_PART_)
   return type1;
 if (type1 != TYPE_INT)
   if( error(next,int_exp))
     return  ERROR;
 if (expon_part(next) != TYPE_INT)
   if (error(next,int_exp))
     return  ERROR;
 gen(_EXP,0,0,node);
 return type1;
}

/*****************************************************************/
/*                                                               */
/*      exponent part - accept the exponent part of a factor     */
/*              generate code to do exponentiation.  Exponent    */
/*              must be of type integer.                         */
/*                                                               */
/*****************************************************************/

int  expon_part(node)
  int  node;

{
 return  primary(son(node));
}
```

```
/*****************************************************************/
/*                                                               */
/*     abs primary - accept an absolute value primary.           */
/*             generate code to find the absolute value          */
/*             of its arguement.                                 */
/*                                                               */
/*****************************************************************/

int  abs_prim(node)
  int  node;

{
 int  next,type;

 if ((type = primary(next = son(node))) != TYPE_INT)
   if (error(next,int_exp))
     return  ERROR;
 gen(_ABS,0,0,node);
 return  type;
}

/*****************************************************************/
/*                                                               */
/*     not primary - accept a negated boolean primary.           */
/*             its arguement must be a boolean expression.       */
/*                                                               */
/*****************************************************************/

int  not_prim(node)
  int  node;

{
 int  type,next;

 if ((type = primary(next = son(node))) != TYPE_BOOL)
   if (error(next,bool_exp))
     return  ERROR;
 gen(_NOT,0,0,node);
 return  type;
}
```

APPENDIX F

```
/*************************************************************************/
/*            00000   00    00   00   00   00000    00000              */
/*            00       00  00    000  00   00  00   00                 */
/*            00000     0000     00 0 00   00  00   0000               */
/*               00      00      00  000   00  00   00                 */
/*            00000      00      00   00   00000    00000              */
/*                                                                       */
/*       SYNtax-Directed Editor (c) Copyright December 1982             */
/*         CAPT. Scott Edward Ferguson, USAF, AFIT GCS-82D              */
/*                     Modified October 1983                             */
/*         CAPT. Michael L. McCracken, USAF, AFIT GCS-83D               */
/*                                                                       */
/*                            INTERP.C                                   */
/*    SYNDE system "dynamic display" interpreter.  The interpreter      */
/*    itself is language independent, using a stack pseudo-machine,    */
/*    but must be linked with language dependent compiler routines.     */
/*************************************************************************/

#include "synde.h"            /* system global information structures */

#include "types.h"            /* production type definitions         */

#include "code.h"             /* pseudo-code structures              */

#include <curses.h>


extern                                  /* source file information      */
struct  file_info src_info;             /*       in AST.C               */

extern                                  /* terminal display information */
struct  term_info tdf_data;             /*       in DISPLAY.C           */

extern
struct  code_word *code;                /* pseudo-code memory           */

extern                                  /* in EXECUTE.C                 */
int     inst_ptr,                       /* instruction pointer          */
        stk_ptr,                        /* stack pointer                */
        *stack;                         /* stack space                  */

char    *codes[_NOOP],                  /* instruction names            */
        *types[LAST_PROD_],
        *noson,*int_exp,*bool_exp,
        str[WIDE];                      /* display line to build        */

int     ch;                             /* command input character      */
```

APPENDIX F

```
/**************************************************************************/
/*      main                                                              */
/*              Entry point and driver for interpreter.                   */
/**************************************************************************/

main(argc,argv)
        int     argc;                   /* input argument count         */
        char    *argv[];                /* input argument ptrs          */
{
puts("INTERPRETER 11/11/83 ");
if (argc < 2) {                         /* prepare source file name     */
        puts("Unspecified source file. ");
        exit();
        }

/* initialize                                                           */
if (a_init(argv[1]) == ERROR || s_init(src_info.f_lang) == ERROR)
        exit();

if (compile()) {                /* check generated code for errors      */
        puts("Errors in source program. Continue with interpreter?");
        if (toupper(getchar()) == 'Y')
                interpret();    /* interpret pseudo-code any way        */
        }
else
        interpret();            /* no errors, interpret pseudo-code     */

if (a_wrap() == ERROR)
        exit();
s_wrap();

if (src_info.f_edit) {
        execl("synde","synde",argv[1],2);
        puts("Cannot access SYNDE. ");
        }
}
```

APPENDIX F

```
/***********************************************************************/
/*      interpret                                                      */
/*              Interpret the generated code with dynamic display of  */
/*              program AST tree.  Allow user interaction to execute, */
/*              single-step and terminate execution.                  */
/***********************************************************************/

interpret()
{
        int     focus;                  /* dislay image focus          */
        BOOL    cont;                   /* continue execution          */

codes[_JMP]     = "JMP";                /* instruction names           */
codes[_CPY]     = "CPY";
codes[_STO]     = "STO";
codes[_CAL]     = "CAL";
codes[_JPC]     = "JPC";
codes[_EQU]     = "EQU";
codes[_NEQ]     = "NEQ";
codes[_LES]     = "LES";
codes[_LEQ]     = "LEQ";
codes[_GRT]     = "GRT";
codes[_GEQ]     = "GEQ";
codes[_NEG]     = "NEG";
codes[_ADD]     = "ADD";
codes[_SUB]     = "SUB";
codes[_MUL]     = "MUL";
codes[_DIV]     = "DIV";
codes[_LIT]     = "LIT";
codes[_LOD]     = "LOD";
codes[_RET]     = "RET";
codes[_AND]     = "AND";
codes[_OR]      = "OR";
codes[_NOT]     = "NOT";
codes[_DCS]     = "DCS";
codes[_MOD]     = "MOD";
codes[_REM]     = "REM";
codes[_AND_THEN]= "AND_THEN";
codes[_OR_ELSE] = "OR_ELSE";
codes[_XOR]     = "XOR";
codes[_ABS]     = "ABS";
codes[_EXP]     = "EXP";
codes[_NOOP]    = "NOOP";
```

```
 system ("stty cbreak");
 system ("stty -echo");

 if (d_init() == ERROR || i_init() == ERROR || e_init() == ERROR)
         exit();
 focus = NIL;
 ch = 0;
 cont = TRUE;
 while (cont) {
         if (focus = code[inst_ptr].c_node)
                 window(0,focus,TRUE);
         do_refresh();
         show_stack();
         show_inst();
         if (ch != 'C')
                 {
                 message("Single-step, Continue, Restart or Exit?");
                 ch = toupper(keyin());
                 message("");
                 }
         switch (ch) {
                 case 'C': message("");
                 case ' ':
                 case 'S':
                         if (execute() == ERROR)
                           {
                            message("End of valid program.");
                            ch = 0;
                           }
                         else
                                 break;

                 case 'R':
                         restart(); break;

                 case 'E':
                         cont = FALSE;
                 }
         }
d_wrap();
i_wrap();
e_wrap();

system("stty echo");
system("stty -cbreak");
}
```

APPENDIX F

```
/*************************************************************************/
/*      show_stack                                                      */
/*              Display a portion of the top of stack on the screen.    */
/*************************************************************************/

show_stack()
{
        REG     i;

strcpy(str,"Stack:  ");
for (i = stk_ptr - 1; i >= 0 && strlen(str) < tdf_data.chars; --i) {
        catnum(str,stack[i]);
        strcat(str,"  ");
        }
dsp_line(str,tdf_data.lines - 3);
}

/*************************************************************************/
/*      show_inst                                                       */
/*              Display the next instruction to be executed on the      */
/*              screen.                                                 */
/*************************************************************************/

show_inst()
{
        REG     i;

strcpy(str,"Next instruction @");
catnum(str,inst_ptr);
strcat(str,":  ");
if ((i = code[inst_ptr].c_opcode) > 0) {
        strcat(str,codes[i]);                   /* instruction mnemonic */
        strcat(str,"   ");
        catnum(str,code[inst_ptr].c_op1);       /* operand 1            */
        strcat(str,",");
        catnum(str,code[inst_ptr].c_op2);       /* operand 2            */
        }
else
        strcat(str,"INVALID");
dsp_line(str,tdf_data.lines - 2);
}
```

APPENDIX F

```
/*************************************************************************/
/*      catnum                                                          */
/*              Concatenate the ASCII representation of a signed number */
/*              to the end of the given string.                         */
/*************************************************************************/

catnum(string,val)
        char    *string;
        int     val;
{
        char    s[7];                   /* temporary string             */
        REG     i;                      /* string index                 */

if (val < 0) {
        strcat(string,"-");             /* negative value               */
        val = -val;
        }
s[i = 6] = 0;                           /* generate characters in       */
do {                                    /*      reverse order           */
        s[--i] = val % 10 + '0';
        } while (val = val / 10);
strcat(string,&s[i]);                   /* concatenate number string    */
}
```

APPENDIX F

```
/*********************************************************************/
/*                                                                   */
/*          00000   00    00   00   00   00000    00000          */
/*          00       00  00    000  00   00  00   00             */
/*          00000     0000     00 0 00   00  00   0000           */
/*             00      00      00  000   00  00   00             */
/*          00000      00      00   00   00000    00000          */
/*                                                                   */
/*       SYNtax-Directed Editor (c) Copyright December 1982         */
/*        CAPT. Scott Edward Ferguson, USAF, AFIT GCS-82D           */
/*                    Modified October 1983                          */
/*        CAPT. Michael L. McCracken, USAF, AFIT GCS-83D            */
/*                                                                   */
/*                          EXECUTE.C                                */
/*     SYNDE system pseudo-machine emulator.  Stack organization:   */
/*                                                                   */
/*                          ...        <- stack pointer              */
/*    top of stack     [   work space   ]                            */
/*                          ...                                      */
/*                     [ local variables ]                           */
/*                     [   static link  ] <- base pointer            */
/*                     [  dynamic link  ]                            */
/*   bottom of stack   [ return address  ]                           */
/*                          ...                                      */
/*                                                                   */
/*********************************************************************/

#include "synde.h"          /* system global information structures */

#include "code.h"           /* pseudo-machine structures            */

extern
struct  code_word *code;           /* instruction space             */

#define STACK   100                /* stack size                    */

int     inst_ptr,                  /* program instruction pointer  */
        base_ptr,                  /* base pointer register         */
        stk_ptr,                   /* stack pointer register        */
        *stack;                    /*  machine stack                */
```

APPENDIX F

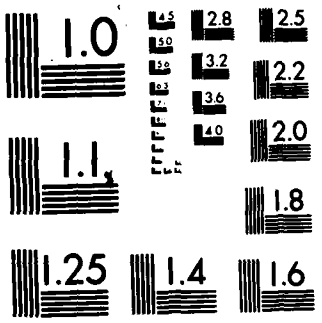
```
/*********************************************************************/
/*                                                                   */
/*      e_init                                                       */
/*              Initialize execution emulator.                       */
/*                                                                   */
/*********************************************************************/

int e_init()
{
if (!(stack = malloc(STACK*2)))         /* allocate stack space       */
        return ERROR;
restart();                              /* set processor at beginning */
}

/*********************************************************************/
/*                                                                   */
/*      restart                                                      */
/*              Set the execution emulator to resume at program start. */
/*                                                                   */
/*********************************************************************/

restart()
{
stk_ptr = inst_ptr = 0;                 /* clear stack, inst pointer  */
push(0);                                /* return loc for function    */
push(-1);                               /* "program end" return addr  */
push(0);                                /* initial dynamic link       */
base_ptr = stk_ptr;                     /* base pointer to static link */
push(0);                                /* initial static link        */
return SUCCESS;
}

/*********************************************************************/
/*                                                                   */
/*      execute                                                      */
/*              Execute the next machine instruction.  Return ERROR  */
/*              upon reaching end of program or invalid instruction. */
/*                                                                   */
/*********************************************************************/

int execute()
{
        register
        struct  code_word *inst_reg;    /* pointer to current inst    */
        int     i,j,k,l;

inst_reg = &code[inst_ptr++];           /* fetch next instruction     */

switch (inst_reg->c_opcode) {           /* decode instruction         */

        case _JMP:
                inst_ptr = inst_reg->c_op1; break;
```

AD-A138 027 ADA1: AN ADA SUBSET COMPILER FOR THE AFIT SYNTAX DIRECTED PROGRAMMING ENV..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. M L MCCRACKEN

3/4

UNCLASSIFIED 05 DEC 83 AFIT/GCS/MA/83D-4 F/G 9/2 NL

END
DATE
FILMED
3-84
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

APPENDIX F

```
case _CPY:
        push(push(pop())); break;

case _STO:
        stack[base(inst_reg->c_op1) + 1 + inst_reg->c_op2] = pop();
        break;

case _CAL:
        push(inst_ptr);                 /* save return address  */
        push(base_ptr);                 /*      dynamic link    */
        push(base(inst_reg->c_op1));    /*      static link     */
        base_ptr = stk_ptr - 1;
        inst_ptr = inst_reg->c_op2;     /* vector to routine    */
        break;

case _JPC:
        if (!pop())
                inst_ptr = inst_reg->c_op1;
        break;

case _EQU:
        push(pop() == pop()); break;

case _NEQ:
        push(pop() != pop()); break;

case _LES:
        i = pop();
        j = pop();
        push(j < i);
        break;

case _LEQ:
        i = pop();
        j = pop();
        push(j <= i);
        break;

case _GRT:
        i = pop();
        j = pop();
        push(j > i);
        break;

case _GEQ:
        i = pop();
        j = pop();
        push(j >= i);
        break;
```

```
case _NEG:
        push(-pop()); break;

case _ADD:
        push(pop() + pop()); break;

case _SUB:
        i = pop();
        j = pop();
        push(j-i);
        break;

case _MUL:
        push(pop() * pop()); break;

case _DIV:
        i = pop();
        push(pop() / i); break;

case _LIT:
        push(inst_reg->c_op1); break;

case _LOD:
        push(stack[base(inst_reg->c_op1) + 1 + inst_reg->c_op2]);
        break;

case _RET:
        pop();                          /* discard static link  */
        base_ptr = pop();               /* restore base_ptr     */
        if ((inst_ptr = pop()) == -1)   /* restore inst_ptr     */
                return ERROR;           /* exit if "end"        */
        break;

case _AND:
        push(pop() && pop()); break;

case _OR:
        push(pop() || pop()); break;

case _NOT:
        push(!pop()); break;

case _DCS:
        stk_ptr -= inst_reg->c_op1; break;

case _REM:
        i = pop();
        j = pop();
        push(j - ((j/i) * i));
        break;
```

```
case _MOD:
        i = pop();
        j = pop();
        if ((k = j/i) >= 0)
          push(j - (k * i));
        else
          push(i + j - (k * i));
        break;

case _AND_THEN:
        if (!pop())
          inst_ptr = inst_reg->c_op1;
        break;

case _OR_ELSE:
        if (pop())
          inst_ptr = inst_reg->c_op1;
        break;

case _XOR:
        i = pop();
        j = pop();
        push(!((i && j) || (!i && !j)));
        break;

case _ABS:
        if ((i = pop()) < 0)
          push(-i);
        else
          push(i);
        break;

case _EXP:
        i = pop();
        if (i < 0)
         {
          puts("ERROR - negative exponent");
          return ERROR;
         }
        j = pop();
        l = j;
        for(k = 1; k < i; k++)
          l = l * j;
        push(l);
        break;

case _NOOP:
        break;

default:
        return ERROR;
}
```

```
return SUCCESS;
}

/**********************************************************************/
/*                                                                    */
/*      push                                                          */
/*              Push a value onto the stack.  The pushed value is also */
/*              returned.  Stack overflow exits with an error message. */
/*                                                                    */
/**********************************************************************/

int push(val)
        int     val;
{
if (stk_ptr >= STACK) {
        puts("Stack overflow.");
        exit();
        }
return stack[stk_ptr++] = val;
}

/**********************************************************************/
/*                                                                    */
/*      pop                                                           */
/*              Return the value popped from the top of stack.        */
/*                                                                    */
/**********************************************************************/

int     pop()
{
if (stk_ptr)
  return stack[--stk_ptr];

puts("Stack underflow.");
exit();
}
```

APPENDIX F

```
/************************************************************************/
/*                                                                      */
/*      base                                                            */
/*              Return the base pointer for the given static level      */
/*              difference.                                             */
/*                                                                      */
/************************************************************************/

int base(level)
        int     level;
{
        REG     new_base;

new_base = base_ptr;                    /* start with current base_ptr  */
while (level) {
        new_base = stack[new_base];     /* chain to base pointer at the */
        --level;                        /*      desired level           */
        }
return new_base;
}

/************************************************************************/
/*                      \                                               */
/*      e_wrap                                                          */
/*              Terminate use of execution emulator.                    */
/*                                                                      */
/************************************************************************/

e_wrap()
{
free(stack);                            /* restore stack space          */
}
```

APPENDIX F

```
#include "synde.h"

/****************************************************************/
/*                                                              */
/*      streq - test strings for equality                       */
/*                                                              */
/****************************************************************/

streq(str1,str2)          /*string equality test */
   char str1[], str2[];
{
  return (!strcmp(str1,str2));
}

/****************************************************************/
/*                                                              */
/*      prestr - test if str1 is beginning of str2              */
/*                                                              */
/****************************************************************/

prestr(str1,str2)       /* test if str1 is beginning of str2 */
   char str1[], str2[];  /* return 1 if true, 0 otherwise */
{
  int i;

  i=0;
  while(str1[i] == str2[i])
    if (str1[i++] == 0)
        return(1);
  if (str1[i] == 0)
    return(1);
  else return(0);
}

/****************************************************************/
/*                                                              */
/*      isnumeric - test if a character is a number             */
/*                                                              */
/****************************************************************/

isnumeric(num, max)     /* numeric set test */
    char num;
    int max;
{
  return isdigit(num) && ( (num - '0') < max);
}
```

APPENDIX F

```
/*******************************************************************/
/*                                                                 */
/*      toupper - convert lowercase input to uppercase             */
/*                                                                 */
/*******************************************************************/

toupper(ch)             /* convert lower case character to upper case */
   int ch;              /*  return all others unmodified              */
{
  return (islower(ch) ? (ch)-'a'+'A' : ch);
}

/*******************************************************************/
/*                                                                 */
/*      tolower - convert lowercase inputs to uppercase            */
/*                                                                 */
/*******************************************************************/

tolower(ch)             /* convert upper case character to lower case */
   int ch;              /*  return all others unmodified              */
{
  return (isupper(ch) ? (ch)-'A'+'a' : ch);
}

/*******************************************************************/
/*                                                                 */
/*      ioerr - display I/O error message                          */
/*                                                                 */
/*******************************************************************/

ioerr(fp)           /* bogus ioerr function */
  int *fp;
{
  puts("I/O error occured");
}
```

APPENDIX F

```
/*****************************************************************/
/*                                                               */
/*      clear_finfo - clear file_info block                     */
/*                                                               */
/*****************************************************************/

clear_finfo(fptr)               /* clear file_info block */
struct file_info *fptr;
{  int i;

 fptr->f_buf = 0;
 strcpy(fptr->f_name,"");
 strcpy(fptr->f_lang,"");
 strcpy(fptr->f_creat,"");
 strcpy(fptr->f_last,"");
 strcpy(fptr->f_conf,"");
 fptr->f_edit = 0;
 fptr->f_update = 0;
 fptr->f_avail = 0;
 fptr->f_root = 0;
 fptr->f_clip = 0;
 for (i=0; i<10; i++)
   fptr->f_mark[i] = 0;
}

/*****************************************************************/
/*                                                               */
/*      clear_ast - clear ast_node                              */
/*                                                               */
/*****************************************************************/

clear_ast(ast_ptr)                    /* clear ast_node       */
struct ast_node *ast_ptr;
{
 ast_ptr->a_flags = 0;
 ast_ptr->a_value = 0;
 ast_ptr->a_prod = 0;
 ast_ptr->a_right = 0;
 ast_ptr->a_son = 0;
}
```

APPENDIX F

```
/**********************************************************************/
/*                                                                    */
/*      clear_tinfo - clear terminal information structure           */
/*                                                                    */
/**********************************************************************/

clear_tinfo(tdf_ptr)                    /* clear term_info      */
struct term_info *tdf_ptr;
{  int i;

 tdf_ptr->lines = 0;
 tdf_ptr->chars = 0;
 tdf_ptr->wsize = 0;
 tdf_ptr->xxx[1] = 0;
 tdf_ptr->xxx[2] = 0;
 tdf_ptr->xxx[3] = 0;
 tdf_ptr->xxx[4] = 0;
 tdf_ptr->xxx[5] = 0;
 for (i=0; i<29; i++)
   strcpy(tdf_ptr->cmds[i],"");
 strcpy(tdf_ptr->init,"");
 strcpy(tdf_ptr->tab,"");
 strcpy(tdf_ptr->elide,"");
 strcpy(tdf_ptr->div,"");
 strcpy(tdf_ptr->clr,"");
 strcpy(tdf_ptr->pos,"");
 strcpy(tdf_ptr->eol,"");
 strcpy(tdf_ptr->dc,"");
 strcpy(tdf_ptr->rev,"");
 strcpy(tdf_ptr->norm,"");
 strcpy(tdf_ptr->ion,"");
 strcpy(tdf_ptr->ioff,"");
 strcpy(tdf_ptr->il,"");
 strcpy(tdf_ptr->dl,"");
 strcpy(tdf_ptr->fini,"");
}
```

APPENDIX F

```
/****************************************************************   ***********/
/*                                                                            */
/*              ADA1 CODE LISTER (c) Copyright Novemeber 1983                 */
/*              CAPT. Michael L. McCracken, USAF,AFIT GCS-83D                 */
/*                                                                            */
/*                              CODE_LISTER                                   */
/*                                                                            */
/*      Read code file generated by the Ada1 compiler and produce a          */
/*      formatted file of the code for user inspection.                       */
/*                                                                            */
/******************************************************************************/

#include "code.h"                     /* pseudo-code structures   */

#define  ERROR  -1
#define  READ    0
#define  REG     int

struct  code_word  code;              /* pseudo-code memory       */

int     inst_ptr,                     /* instruction pointer      */
        code_file;                    /* listing file pointer     */

char    code_name[20],                /* output file name         */
        *codes[_NOOP],                /* instruction names        */
        str[40];                      /* display line to build    */
```

```
/**********************************************************************/
/*                                                                    */
/*      main                                                          */
/*              Entry point and driver for code lister.              */
/*                                                                    */
/**********************************************************************/

main(argc,argv)
        int     argc;                   /* input arguement count      */
        char    *argv[];                /* input arguement ptrs       */

{
 puts("CODE LISTER 11/11/83");

 if (argc < 2)                         /* prepare source file name */
  {
   puts("Unspecified source file.");
   exit();
  }

 c_init();

 if (argc > 2)
   strcpy(code_name,argv[2]);
 else
  {
   puts(" Filename for code listing? (default is name.codlst)");
   gets(code_name);
   puts("");
   if (!strlen(code_name))
    {
     strcpy(code_name,argv[1]);
     strcat(code_name,".codlst");
    }
  }
 printf("Code list output file = %s.",code_name);

 if ((code_file = creat(code_name,0644)) == ERROR)
  {
   puts("Cannot create code listing file.");
   exit();
  }
 else
   c_list(argv[1]);

 close(code_file);
}
```

APPENDIX F

```
/**********************************************************************/
/*                                                                    */
/*      c_init - initialize codes array with instruction strings.     */
/*                                                                    */
/**********************************************************************/

c_init()

{
 codes[_JMP]     = "JMP";                /* instruction names        */
 codes[_CPY]     = "CPY";
 codes[_STO]     = "STO";
 codes[_CAL]     = "CAL";
 codes[_JPC]     = "JPC";
 codes[_EQU]     = "EQU";
 codes[_NEQ]     = "NEQ";
 codes[_LES]     = "LES";
 codes[_LEQ]     = "LEQ";
 codes[_GRT]     = "GRT";
 codes[_GEQ]     = "GEQ";
 codes[_NEG]     = "NEG";
 codes[_ADD]     = "ADD";
 codes[_SUB]     = "SUB";
 codes[_MUL]     = "MUL";
 codes[_DIV]     = "DIV";
 codes[_LIT]     = "LIT";
 codes[_LOD]     = "LOD";
 codes[_RET]     = "RET";
 codes[_AND]     = "AND";
 codes[_OR]      = "OR";
 codes[_NOT]     = "NOT";
 codes[_DCS]     = "DCS";
 codes[_MOD]     = "MOD";
 codes[_REM]     = "REM";
 codes[_AND_THEN]= "AND_THEN";
 codes[_OR_ELSE] = "OR_ELSE";
 codes[_XOR]     = "XOR";
 codes[_ABS]     = "ABS";
 codes[_EXP]     = "EXP";
 codes[_NOOP]    = "NOOP";
}
```

APPENDIX F

```
/*****************************************************************/
/*                                                               */
/*      c_list - produce code listing and output to a file       */
/*                                                               */
/*****************************************************************/

c_list(c_name)
   char  c_name[];

{
 int  c_file,num_read,instr_num,i,cont;
 char str[40];

 instr_num = 0;
 strcat(c_name,".cod");
 if ((c_file = open(c_name,READ)) == ERROR)
  {
   puts("I/O error opening code file ");
   puts(c_name);
   puts(".");
  }
 else
  {                                          /* read until end of file   */
   while (num_read = read(c_file,&code,(sizeof(struct code_word)))
          && cont)
    {
     if (num_read == ERROR)
       puts("I/O error reading code file.");
     else
      {
       strcpy(str,"");               /* initialize as empty      */
       catnum(str,instr_num++);        /* instruction number       */
       strcat(str," :  ");
       if ((i = code.c_opcode) > 0)
        {
         strcat(str,codes[i]);         /* instruction string       */
         strcat(str,"    ");
         catnum(str,code.c_op1);       /* operand 1                */
         strcat(str,",");
         catnum(str,code.c_op2);       /* operand 2                */
        }
       else
         if (i == 0)                   /* end of file before       */
                                       /* actual file end          */
          {
           strcpy(str,"");
           cont = 0;
          }
         else
           strcat(str,"INVALID");        /* invalid instruction      */
       strcat(str,"");
       i = strlen(str);
      if (write(code_file,str,i) != i)
        puts("I/O error writing code listing file.");
      }
    }
  }
 close(c_file);
}

/****************************************************************/
/*                                                              */
/*     catnum                                                   */
/*             Concatenate the ASCII representation of a signed */
/*             number to the end of the given string.           */
/*                                                              */
/****************************************************************/

catnum(string,val)
        char    *string;
        int     val;
{
        char    s[7];                   /* temporary string     */
        REG     i;                      /* string index         */

if (val < 0) {
        strcat(string,"-");             /* negative value       */
        val = -val;
        }
s[i = 6] = 0;                           /* generate characters in */
do {                                    /*      reverse order   */
        s[--i] = val % 10 + '0';
        } while (val = val / 10);
strcat(string,&s[i]);                   /* concatenate number string*/
}
```

APPENDIX G

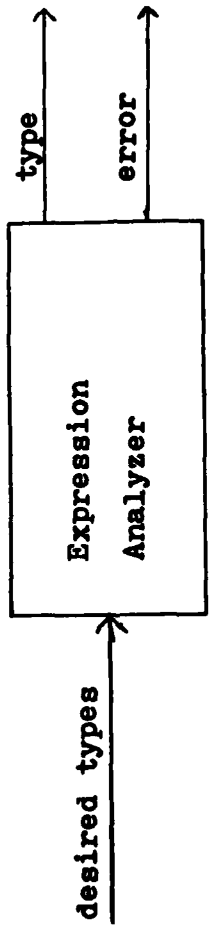


Figure G-1 Semantic Analyzer

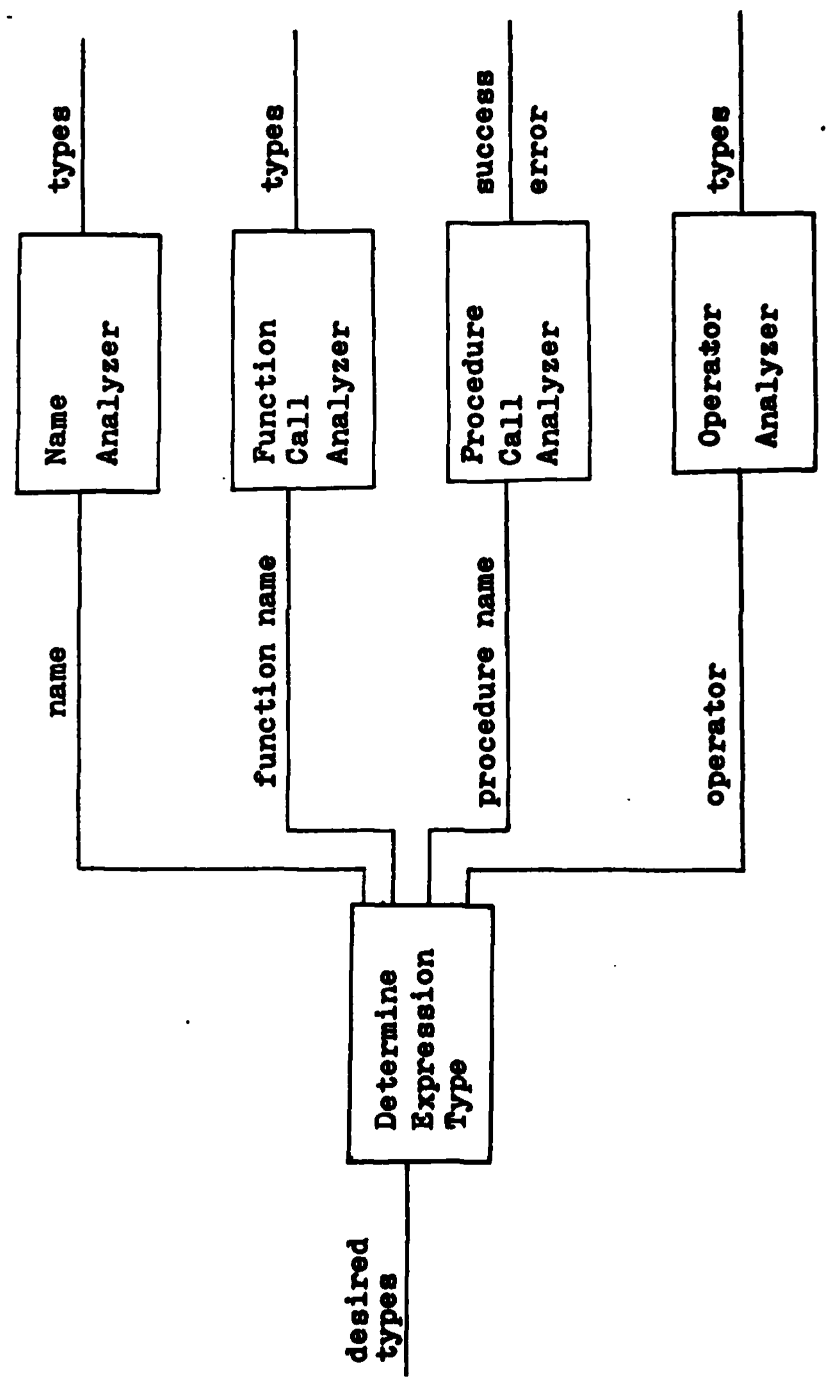


Figure G-2 Expression Analyzer

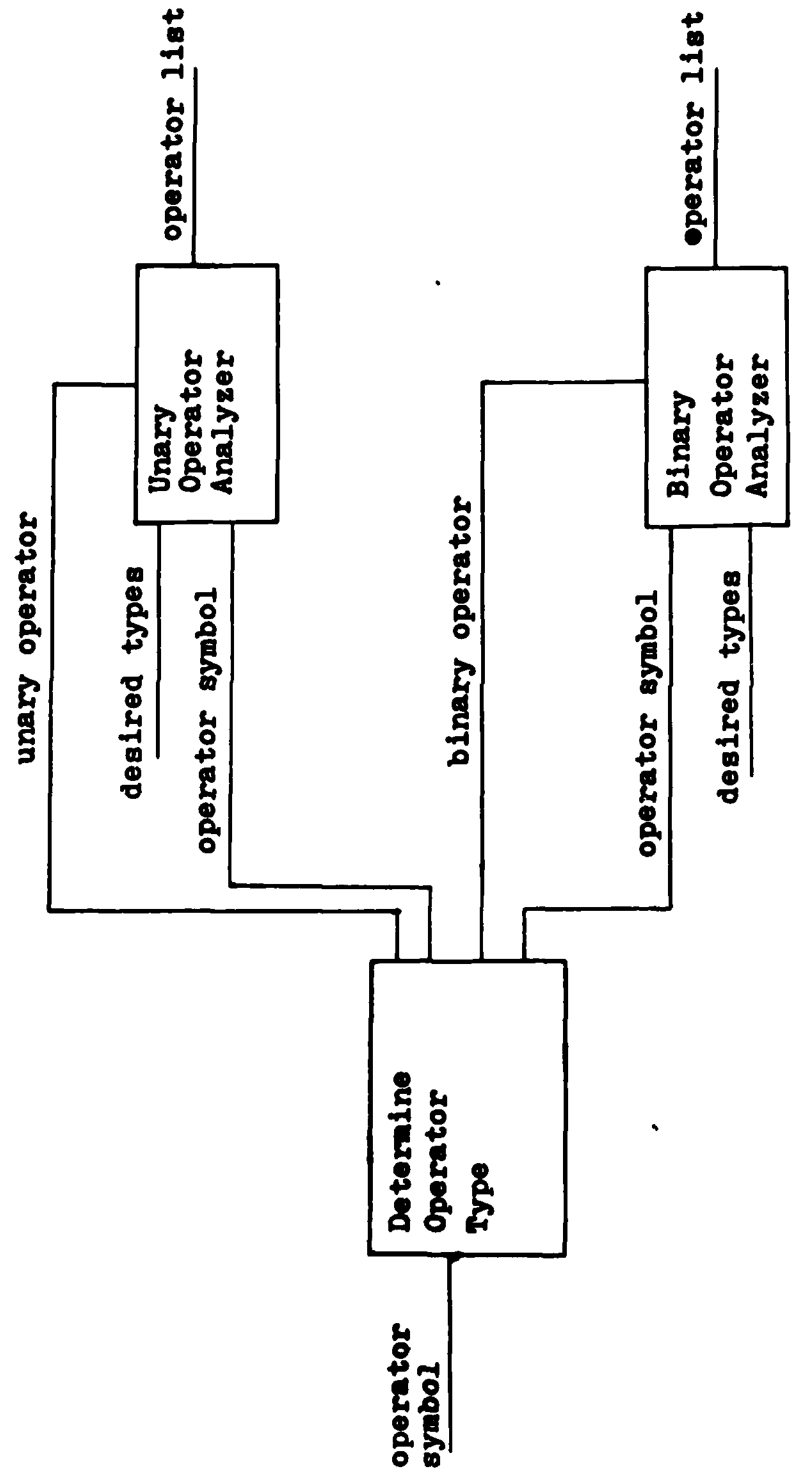


Figure G-3 Operator Analyzer

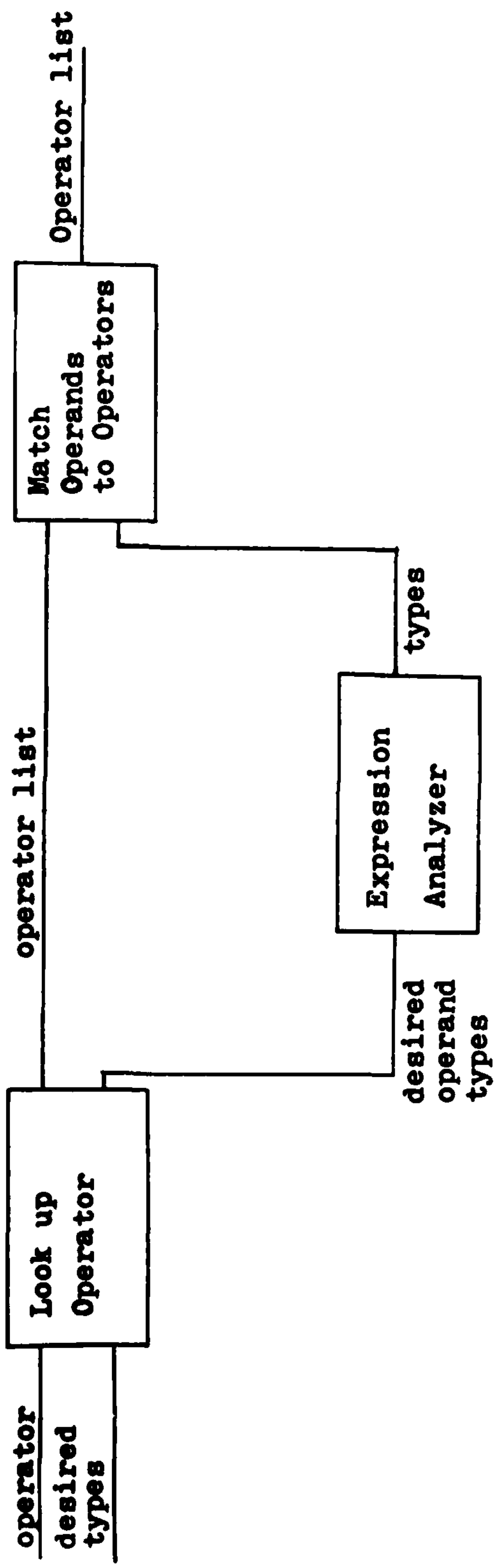


Figure G-4 Unary Operator Analyzer

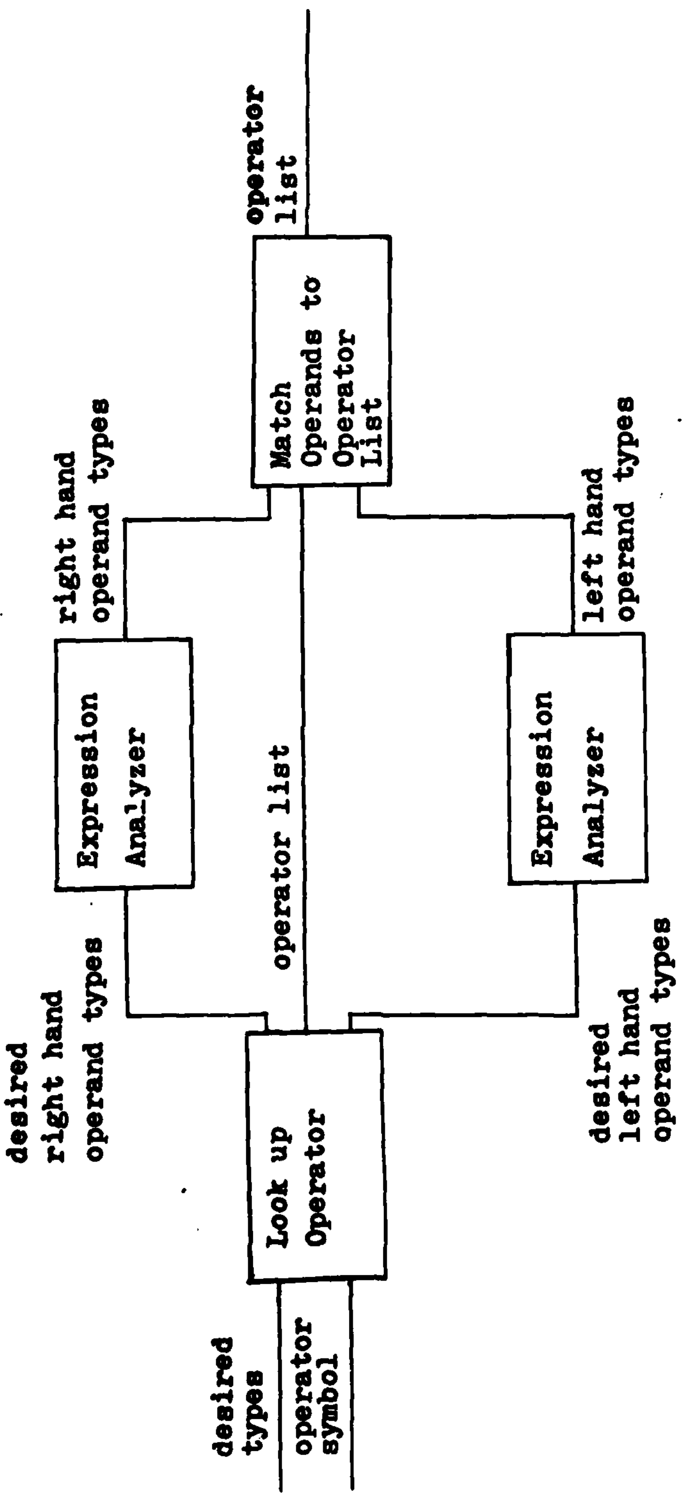


Figure G-5 Binary Operator Analyzer

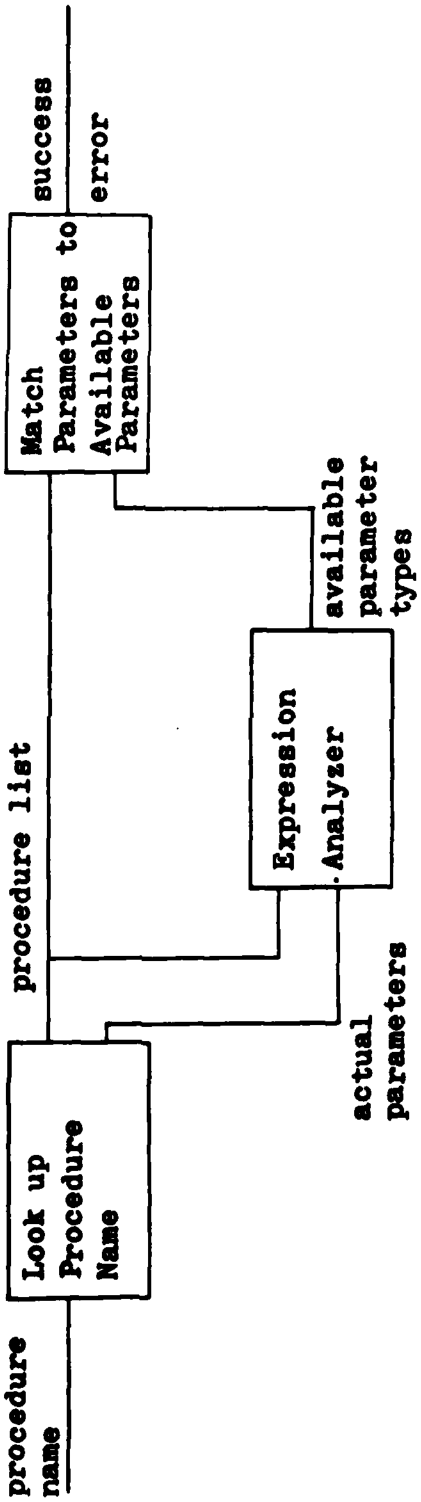


Figure G-6 Procedure Call Analyzer

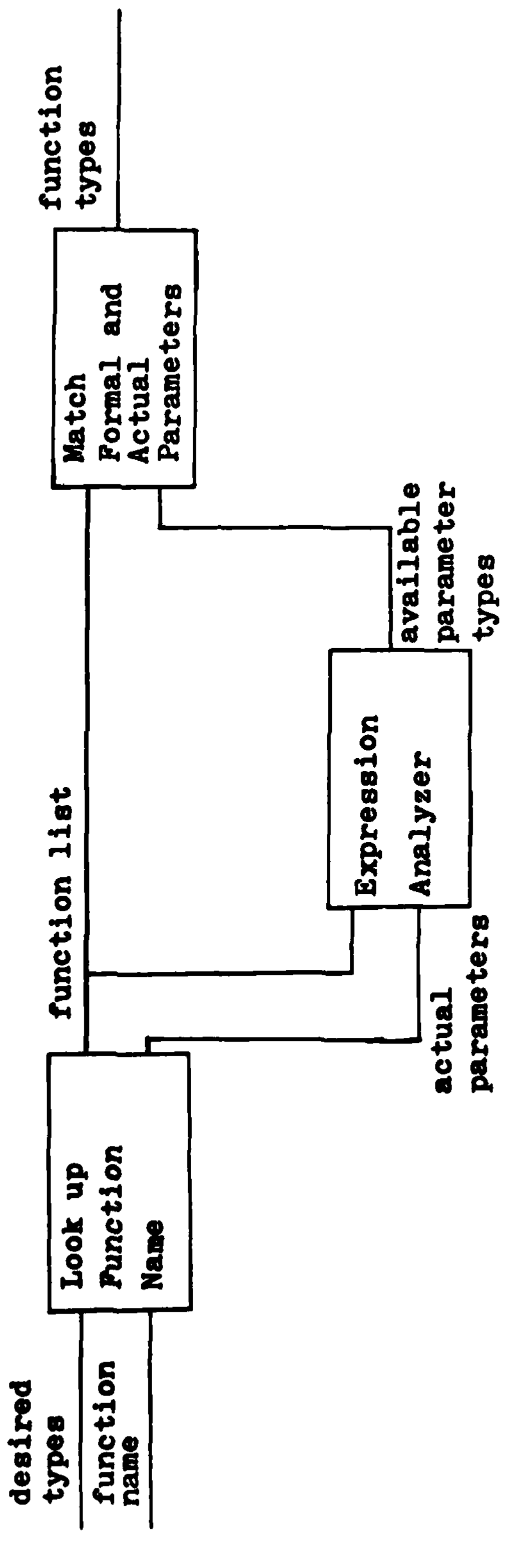


Figure G-7 Function Call Analyzer

VITA

VITA

Michael L. McCracken was born on 25 July 1952 in Glen Ridge, New Jersey to Robert A. McCracken and Lilian B. (Sarter) McCracken. He attended Sehome High School in Bellingham, Washington and graduated cum laude in 1970. He attended Western Washington State College in Bellingham, Washington where he earned a Bachelors of Arts degree in mathematics in June 1977. In July 1978 he entered Officer's Training School at Medina Annex, Lackland Air Force Base and was commissioned in October. His first active duty Air Force assignment was with the 2nd Communications Squadron which later became Detachment 1 4602 CPUSS at Lowry AFB, Colorado. He then entered the Air Force Institute of Technology in June 1982 as a graduate student in computer science.

Captain McCracken was married to Kerri Lee Lobberegt on 15 May 1976 in Port Gamble, Washington. They have a daughter, April Lynn, born 25 February 1980 and a son, Patrick Logan, born 12 December 1981.

Permanent address: 5191 N.E. Ponderosa Drive
Hansville, Washington 98340

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A138 027

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/MA/83D-4	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG
7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS *(City, State and ZIP Code)* Air Force Institute of Technology Wright Patterson AFB, Ohio 45433	7b. ADDRESS *(City, State and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL *(If applicable)*
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS *(City, State and ZIP Code)*	10. SOURCE OF FUNDING NOS.
11. TITLE *(Include Security Classification)* See Box 19	

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.

12. PERSONAL AUTHOR(S)
Michael L. McCracken, B.A., Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Yr., Mo., Day)*	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1983 December 5	214

16. SUPPLEMENTARY NOTATION

Approved for public release: IAW AFR 190-17.
[illegible]
LYNN E. WOLAVER
Dean for Research and Professional Development
Wright-Patterson AFB OH 45433

17. COSATI CODES

FIELD	GROUP	SUB. GR.
09	02	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Title: ADA1 - AN ADA SUBSET COMPILER FOR THE AFIT SYNTAX DIRECTED PROGRAMMING ENVIRONMENT

Thesis Advisor: Capt Patricia K. Lawlis

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER *(Include Area Code)*	22c. OFFICE SYMBOL
Capt Patricia K. Lawlis	(513) 255-3636	AFIT/ENC

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

Abstract

This document describes the effort involved in moving the Ada0 compiler and interpreter developed by Capt. Scott E. Ferguson as part of the AFIT syntax directed editor environment from a microcomputer to the VAX 11/780.

As part of this effort the compiler and interpreter were expanded to accept a larger suset of Ada. The compiler and interpreter work with an abstract syntax representation of a computer program produced by the syntax directed editor. This abstract representation, which is guaranteed to be syntactically correct, makes the compiler much easier to write and understand. The compiler in a top-down compiler but no backtracking is needed since the program is known to be syntactically correct. The interpreter is able to use the abstract representation to give the user an interactive display of the program during execution.

Designs to allow overloading of names and operators, and passing parameters to subprograms are also presented.

UNCLASSIFIED

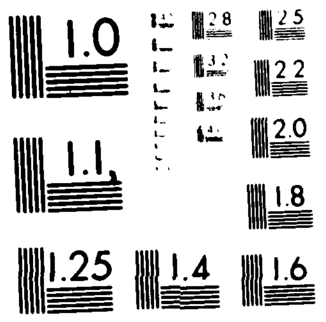
SECURITY CLASSIFICATION OF THIS PAGE

DATE
ILME

AD-A138 027 ADA1: AN ADA SUBSET COMPILER FOR THE AFIT SYNTAX DIRECTED PROGRAMMING ENV..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. M L MCCRACKEN 4/4

UNCLASSIFIED 05 DEC 83 AFIT/GCS/MA/83D-4 F/G 9/2 NL

SUPPLEMENTARY INFORMATION



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SUPPLEMENTARY

INFORMATION

ERRATA

AD-A138 027

Page 162 is missing and is not available.

11 May 84