IMPLEMENTATION OF MINI-TASK: A LANGUAGE BASED ON ADA'S TASKING MODEL

By

Monty D. Bates

Bachelor of Science in Arts and Sciences

Oklahoma State University

Stillwater, Oklahoma

1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1987

IMPLEMENTATION OF MINI-TASK: A LANGUAGE

BASED ON ADA'S TASKING MODEL

Thesis Approved:

D. E. Hedrick
Thesis Adviser

[illegible]

J P Chandler

Norman N. Durham
Dean of the Graduate College

1290872

PREFACE

Mini-task is an efficient compiler for multi-tasking programs. Mini-task takes the advantageous features of Ada's tasking model and produces a small and efficient target code.

I would like to thank Dr. G. E. Hedrick for providing the initial topic for this thesis and for serving as my major advisor. I would like to express my sincerest appreciation to Dr. J. P. Chandler for his guidance in my education and for serving on my committee. I owe a special thanks to Dr. K. M. George for his continual assistance, his valuable insight, and for serving on my committee.

I would also like to thank Brother Isidore for his assistance and insight in completing this paper.

Finally, I would like to express my appreciation for the support and patience my parents offered during my studies.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Mini-task is a language based on Ada's tasking model. The development of the Ada language followed a unique pattern in the history of programming languages. In the early 1970's the United States Department of Defense decided to develop a standard programming language for computers contained in larger systems such as an aircraft, ships, and communication systems. Pratt [Pratt, 1984] gives a more extended history of the development of Ada.

The Pascal design was the starting point for the design of Ada, but the resulting language has many differences from Pascal. Ada is a much larger and more complex language than Pascal, and it includes several major sets of features that are not contained in the Pascal Language. Some of these features include tasks and concurrent execution, real-time control of tasks, exception handlers, abstract data types, as well as separate compilation. Due to the advantageous attributes listed above as well as others, the study in this thesis is based on the Ada's tasking model.

In the programming language Ada, tasks are program segments which can be executed in parallel with one another. Each task can be considered to execute on a separate processor. Different tasks process independently, except at points where they synchronize. The Ada ANSI Reference Manual states "Parallel tasks (parallel logical processors) may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor" (ARM 9.0.5).

Although Ada has many excellent features, it is such a large language that its compile time is considered to be inefficient, and the size of the target code produced is long. These drawbacks cause Ada to be impracticable when tasking is the only needed feature. Since Mini-task is a smaller language, it results in more efficient processing while still retaining the necessary tasking features.

Statement of Problem

This thesis addresses the language Mini-task. Mini-task is a language based on Ada's tasking model. A front-end compiler which produces intermediate code based on Descriptive Intermediate Attributed Notation for Ada (DIANA) for Mini-task is implemented. Each phase of the compiler is independent of any target machine. The implementation includes lexical analysis, symbol table management, semantic analysis, and the generation of the external intermediate code which is based on DIANA. Also included in the implementation are the error handling routines which correspond with each phase.

Although Mini-task possesses the Ada tasking features, Mini-task does not contain separate compilation, packages, subprograms, and other Ada entities. Mini-task is designed so that the code for tasking programs will compile faster and more efficiently.

The intermediate language that Mini-task produces is based on DIANA, a general purpose intermediate language. DIANA is flexible, and it has the potential to become a widely accepted intermediate language. A strong advantage of DIANA becoming a widely accepted intermediate language would be only one standardized back-end would need to be developed for each machine type. DIANA has the versatility to be used as the intermediate language for many different compilers.

Due to the increasing interest in a common intermediate language for Ada, DIANA was designed in January 1981 by teams from Karlsruhe University, Carnegie-Mellon University, Intermetrics and Softech. DIANA reflects the abstract syntax structure of an Ada program together with the additional information gained by lexical and semantic analysis.

Introduction to the language implemented

A Mini-task program always begins with the reserved word PROCEDURE. The user defined program name follows and then the reserved word IS. All declarations for the main program follow, separated by commas, and ending with a colon. The type, which is always an INTEGER in Mini-task, follows. All statements are terminated by semicolons. Output statements consist of the reserved word PUT followed by a single variable in parenthesis. The Ada notation is followed for assignment statements. Tasks are always declared beginning with the reserved word TASK followed by the user defined task name. Entries are declared in the task declaration, as shown in chapter V. The task body begins with the reserved words TASK BODY followed by a structure similar to the main program. A simple example follows.

```
PROCEDURE main IS
   num, x, y : INTEGER;

   TASK tl;

   TASK BODY tl is
      result, a, b : INTEGER;
   BEGIN
      a := 3;
      b := 3;
      result := a + b;
      PUT(result);
   END tl;


BEGIN
   x := 2;
   y := 2;
   num := x + y;
   PUT(num);
END main;
```

Figure 1. Simple Mini-task example.

Chapter III contains a more detailed discussion of Mini-task.

CHAPTER II

LITERATURE REVIEW

Literature which discusses some important factors encountered when implementing languages similar to Mini-task is currently available. One such factor is deciding on which intermediate language is suited best for use.

An intermediate code to represent the interface of a portable compiler front-end to the various code generators must fulfill the following requirements:

- It must provide a füll and detailed description of the semantics of the source program.
- It must provide a machine-independent representation of a given program.
- It must provide a flexible notation with respect to the different implementation choices which may be taken from a specific implementation.
- It must allow for the application of extensive and sophisticated optimization techniques.
- It must allow for easy and systematic generation of efficient code [Lorho, 1984, Pratt, 1984, Waite, 1984].

DIANA fulfills all of the above requirements [Goos, 1981, Rogers, 1984, Rosenblum, 1985].

Intermediate representations of Ada programs as they appear at the end of the analysis phase are used not only as input for a back-end, but also for various other uses within a programming environment. The uses include semantic analyzers, optimizers, and syntax-directed editors. DIANA is based on the formal definition of Ada. For each DIANA tree the meaning of the tree must be defined. In DIANA a single definition exists for each Ada entity. The authors of DIANA have defined an externally visible ASCII form of DIANA representation for Ada programs because it is essential to have a representation that can be communicated between computing systems. From a representation of DIANA the original source program can be recreated. This requirement was introduced to support all kinds of program manipulation [Rosenblum, 1985, Goos, 1982].

A DIANA tree exists in two forms: before and after semantic analysis. The tree before semantic analysis represents the abstract syntax. It only contains information such as the source position of each entity, the external representation of values, the string representation of identifiers, and similar entities as attributes. From this information the source program can be recreated, since all of the information to recreate the program is stored. The tree after semantic analysis represents an attributed parse tree, in which the semantic attributes are completed.

Another factor encountered in implementing Mini-task was examining similar compilers already in existence. One is the Ada Compiler Karlsruhe.

The Ada Compiler Karlsruhe

The Ada Compiler Karlsruhe consists of a front-end generating the intermediate language DIANA, a middle-part mapping DIANA to the low level intermediate language AIM, and a back-end producing machine code either for the SIEMENS 7000 series or the MC68000. At the time of the publication, the tasking model had not been implemented.

Several tools have been constructed around the compiler. A formatter which is based on DIANA permits printing the Ada program in pretty format. Another such tool is a back-trace tool for program debugging which provides the position in the source program corresponding with the object program. A third tool is a library-user-system which provides the state of the library and the effect of recompilations.

The front-end of the compiler consists of the following: A LALR(1) parser for syntax analysis, an attribute grammar for semantic analysis to produce the intermediate language DIANA, and an optimization phase. The DIANA tree reflecting the Ada source program structure is shaped into a form which is suitable for code generation with the help of DIANA attributes and attributes computed by the middle-end. The optimization phase of the front-end of the compiler reduces its definition table space by importing only the entities whose identifier occurs within the source program.

The techniques used in the Ada Compiler Karlsruhe are supported automatically and machine independent. Supposedly, this will meet the goals of a reliable and maintainable compiler [Persch, 1983].

Another factor is how the compiler should be structured. Following is a diagram displaying the phases of the compiler.

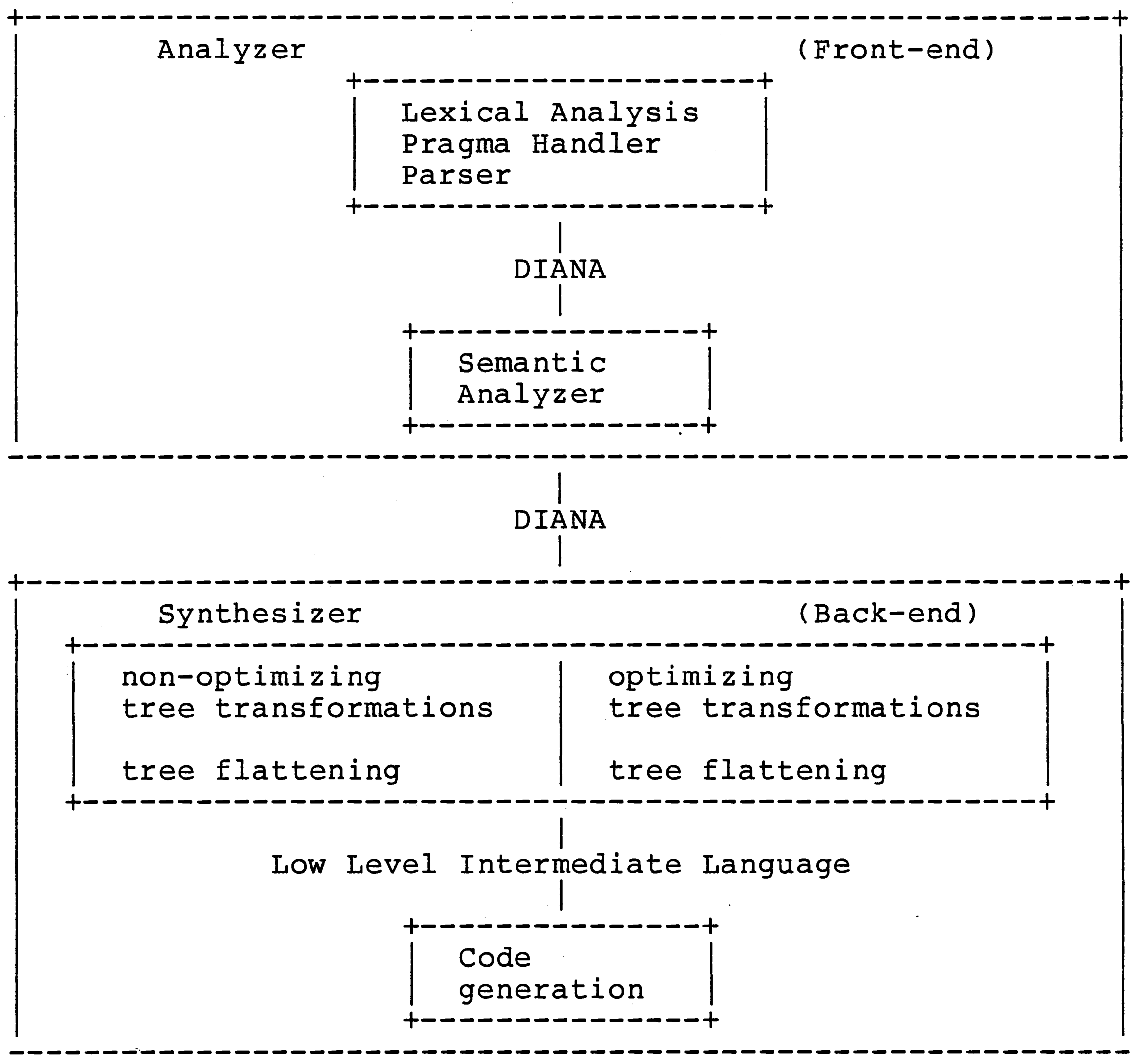

Figure 2. Structure of an Ada compiler [Baker, 1985, Waite, 1984].

The analyzer, or compiler front-end contains the parser, the lexical analyzer, pragma handler, and the semantic analyzer. The parser produces the intermediate code DIANA and the semantic analyzer traverses through the DIANA tree completing semantic attributes.

The synthesizer, or back-end of the compiler, contains the optimizing routines, tree transformation routines, tree flattening routines, which produce the low level intermediate language, and finally producing the target code [Aho, 1986, Halstead, 1974, Lorho, 1984, Waite, 1984].

CHAPTER III

DISCUSSION OF LANGUAGE DESCRIPTION, SYNTAX, AND SEMANTICS

A Discussion of Mini-task

Mini-task is a language based on Ada's tasking model. Tasks in Mini-task are program blocks that may be executed in parallel or concurrently with other tasks in the same program. Each task is considered to be executed by its own logical processor and proceeds independently of the other tasks, except at points where they synchronize.

The semantics of the tasks in Mini-task are similar to the Ada tasks. Every task in Mini-task is declared in the declarative part of some enclosing program unit, which is referred to as its parent. The parent task may be a main program, or it may be another task. The execution of the parent task results in the concurrent execution of the declared tasks. If more than one task is specified in the declarative part of the parent, all the tasks will be executed concurrently both with one another and with the parent task. Each task is executed in its own sequential order, independent of the order of others, except when explicit statements cause synchronization. Each task may finish executing its statements or it may be aborted, but the parent task does not complete its execution until all of the tasks declared in it have completed their execution.

Synchronization between tasks is achieved by a rendezvous between a task issuing an ENTRY call and a task accepting the call by an ACCEPT statement. An ENTRY of a task is called by other tasks, and it may have parameters. ENTRY calls and ACCEPT statements are the principle means of inter-process communication.

Mini-task does not specify the order in which tasks declared are activated. A Mini-task program with more than one task usually consists of a parent containing the tasks that perform the required actions. The body of the parent is usually responsible for the control of the tasks.

Each task consists of a task specification and a task body. A task can be declared as a type if its declaration starts with the reserved words TASK TYPE. Without the reserved word TYPE, a single task is declared. The body of the task is defined by the corresponding TASK BODY.

The task name at the beginning of a task specification must correspond with the task body's name. If the simple task name appears at the the end of the task specification or match body, it must be identical to the name at the start of its task specification and its task body.

An ENTRY call consists of the name of the task in which the ENTRY ACCEPT is located and the name of the ENTRY ACCEPT statement accepting the call, separated by a period.

Parameters follow in parenthesis similar to a procedure call in Ada. An example of an entry call calling taskl entry tl with parameter "a" follows.

```
taskl.tl(a);
```

An ENTRY ACCEPT statement, as described above, is similar to the syntax of a procedure call in Ada. The ENTRY specification may have parameters with the binding modes: IN, OUT, IN OUT, and may be called from other tasks by ENTRY calls. The modes will be discussed later in this chapter. An ENTRY call cannot be executed until it has been synchronized with an ACCEPT statement in the body of the task. The syntax for an ENTRY call is simply the ENTRY name followed by any actual parameters in parentheses.

The actions to be performed when an ENTRY is called are specified in the corresponding ACCEPT statement. The syntax of an ACCEPT statement is:

```
ACCEPT entry_name(formal_parameters) do
   sequence_of_statements;
end entry_name;
```

The formal parameters in the ENTRY declaration and the ACCEPT statement must be identical. An ACCEPT statement for an ENTRY may appear only in the task body of the task which had the declared ENTRY. Tasks can execute ACCEPT statements only for their own entries. The sequence of statements between the "do" and the "end" in the ACCEPT statement is referred to as the critical section. The execution of the calling task is suspended during the execution of the critical section.

The modes discussed above, IN, OUT, IN OUT, imply the direction of the variables in the critical section. If the mode is IN, the value of the variable is transferred into the critical section. If the mode is OUT, the value is transferred out of the critical section. IN OUT implies the value is transferred in at the beginning of the critical section and transferred out at the end of the critical section.

More than one task can issue an ENTRY call for the same ACCEPT statement, which means it is possible for ENTRY calls to occur faster than they can be processed by the corresponding ACCEPT statements. If this occurs the ENTRY calls are stored in a queue that is associated with the ENTRY name and processed on a first-come-first-served basis. Each execution of a corresponding ACCEPT statement removes one ENTRY call from the queue and allows the process with that ENTRY call to continue executing. A task may have more than one ACCEPT statement for an ENTRY declaration, but there will be only a single queue of waiting ENTRY calls for each ENTRY name. To prevent confusion a task may be in only one queue at a time. Each entry possesses its own queue to contain all unserviced entry calls.

The next entities that need to be discussed are the SELECT statements: the selective waits, the conditional entry calls, and the timed entry calls.

The select statement allows tasks to choose their next action among several entry calls. The SELECT statement begins with the reserved word SELECT and terminates with the reserved words END SELECT. The number of guards in a SELECT statement is at the discretion of the programmer. When working with SELECT statements, some basic rules must be followed.

1) A select alternative may be one of three entities: A) an accept statement B) a delay statement c) a terminate statement
2) If the select statement contains a delay statement, the select statement cannot have an ELSE alternative or a TERMINATE.
3) If the select statement does contain a TERMINATE alternative, it cannot have an alternative beginning with DELAY or an ELSE clause.
4) No more than one TERMINATE alternative may be available.

The guards are evaluated by the select statement. Either a guard that is evaluated as true, or an absent guard, is considered to be open. If a guard is evaluated as false, then it is considered to be closed. A rendezvous occurs when an open alternative begins with an ACCEPT statement, and a corresponding entry call has been received. In some cases several entry calls may occur before a select statement is executed, which causes the entry calls to be inserted into the queue. In this case, it is possible to have more than one guard open, which results in one open guard selected at random. It is also acceptable to have more than one unconditional statement; which participates in the same random selection. A delay statement following an open alternative produces a delay; however, if another guard becomes true before the delay elapses, then that guard will be evaluated. An else statement can follow if there is not an ELSE and all of the guards are closed, the task waits for a guard to open.

A conditional entry call issues an entry call that is then revoked if a rendezvous is not immediate.

The DELAY statement suspends the execution of a task for a specified amount of time. The syntax for the DELAY statement is:

```
DELAY simple_expression;
```

where simple_expression is the number of seconds to be delayed. The execution of the DELAY statement evaluates the simple_expression and suspends the task for at least the number of seconds specified by the expression. If the result is a negative number, the DELAY will be the same as that of a delay with a zero specification. The maximum and minimum values are implementation dependent, but the upper limit should be at least the number of seconds in a day.

The DELAY statement can be used with the SELECT statement to create a timed ENTRY call. A timed ENTRY call issues a call if, and only if, a rendezvous is possible within the specified delay time. The DELAY in the following example is given in an 'or' clause. If the rendezvous occurs within the time period, the ENTRY call and any statements associated with it are executed. If the rendezvous does not occur within the time period, the sequence of statements following the DELAY statement is executed. Example:

```
SELECT
   taskl.request;
OR
   DELAY 10.0;
   put(" REQUEST can't be satisfied");
END SELECT;
```

This SELECT statement limits the request call to 10 seconds. If the task does not accept the ENTRY within 10 seconds, the message will be printed and execution will continue with the statements following the END SELECT.

If the above example would have contained "ACCEPT request" instead of "request", then the ENTRY taskl.request is called within the prescribed delay time. If the delay expires, the message is printed and the request is no longer acceptable, and execution continues just as before.

```
SELECT
   ACCEPT request;
OR
   DELAY 10.0;
   put(" REQUEST can't be satisfied");
END SELECT;
```

If the example is changed to a SELECT .. ELSE as follows:

```
SELECT
   ACCEPT request;
ELSE
   DELAY 10.0;
   put(" REQUEST can't be satisfied");
END SELECT;
```

The execution is quite different than the above two examples. If the ENTRY call of request cannot be accepted at once, then the "else" part is executed. A 10 second delay occurs, and the message is printed. The processing continues after the "END SELECT", as before [Gilpin, 1986, Helmbold, 1985, Wiener, 1983].

CHAPTER IV

INTRODUCTION TO DIANA

DIANA is an intermediate form of Ada programs that is especially suitable for an interface between the front and back ends of Ada compilers. It is also well suited for pretty printers and other tools in the Ada support environment. DIANA is based on the formal definition of Ada. It encodes the results of lexical, syntactic, and static semantic analysis, but it does not include the results of dynamic semantic analysis, of optimization, or of code generation.

DIANA is best viewed as an abstract data type that can be modeled as an attributed parse tree. The actual data or file structures used to present the abstract data type are hidden.

DIANA is referred to as a "tree", "abstract syntax tree", or "attributed parse tree" and nodes are referred to in the trees. Although the word tree is used, the authors of DIANA make no reference that the data structure used to implement DIANA is a tree using pointers. Instead, the notion of attributed trees is the abstract model for the definition of DIANA.

There are two types of DIANA users: DIANA producers and DIANA consumers. The study in this thesis is concerned with the DIANA producer which is a compiler front-end based on Ada's tasking model.

A DIANA tree can be represented in an ASCII form to assist debugging and to allow communication between computing systems, but it is not the typical communication between tools.

The implementation may decide how accurately comment positions are recorded and how to associate comments with particular nodes. DIANA has no requirement about either the internal or the external representation of comments, and an implementation does not have to support the screen position of the tokens (lx_scrpos) or the comments in the Ada code (lx_comments) to be considered a DIANA producer or DIANA consumer [Goos, 1982, Rosenblum, 1985].

There are four kinds of attributes defined in DIANA:

as_	Structural attributes define the abstract syntax tree of an ADA program.
lx_	Lexical attributes provide information about the source form of the program. Examples: spelling of identifiers, position in the source file.
sm_	Semantic attributes encode the results of semantic analysis. Example: type and overload resolution.
cd_	Code attribute provides information from representation specifications that must be observed by the Back End of the compiler.

Figure 3. Types of DIANA attributes.

Notation of DIANA

In the DIANA Reference Manual, DIANA is presented in a notation called IDL, Interface Description Language. The advantages of presenting DIANA in IDL are its simplicity and its similarities with Backus-Naur Form (BNF). Upper case character strings represent IDL class names; lower case strings beginning with lx_, as_, sm_, or cd_ are IDL attributes. Other lower case strings represent node names, and strings beginning with an upper case letter represent reserved words in IDL. Examples:

DECL OP DEF_OCCURRENCE	examples of IDL class names
lx_srcpos sm_address	examples of IDL attributes
constant var_const_id	examples of IDL node names
Structure Root Type	examples of IDL reserved words

The language can be given in a form similar to BNF. The set of abstract trees used to model the DIANA abstract data type can be viewed as a language, where the terminal sentences are attributed parse trees instead of character strings. Consider an example IDL class name:

```
EXP ::= leaf | tree ;
```

The definition may be read: 'The notion of an EXP is defined to be either a leaf or a tree'. Class names, the equivalent to nonterminals in BNF, never appear in the sentences of the language; their only use is to define the language. Node names, the equivalent to terminals in BNF, appear in the sentences or trees of the language. IDL requires a semicolon to terminate a definition. As with BNF, more than one production with the same left-hand side may occur; after the first definition, the following definitions introduce other alternatives.

The definition of the node specifies the attributes that are present in the node, as well as the names and types of the attributes. An example of a node definition with three attributes with their names op, left, right, and their types OPERATOR, EXP, and EXP respectively, follow.

```
tree => op     : OPERATOR,
        left   : EXP,
        right  : EXP ;
```

Unlike BNF, the order of the attribute specifications is insignificant. The right-hand side of the production must be a sequence of zero or more attributes specifications separated by commas and terminated by a semicolon. Multiple definitions of a node are permitted, but the additional attribute specifications are concatenated on to the previous specifications. For example:

```
tree => op     : OPERATOR;
tree => right  : EXP ;
tree => left   : EXP;

and

tree => op     : OPERATOR,
        left   : EXP,
        right  : EXP ;
```

both possess the same attributes.

Some nodes have no attributes. Example:

```
foo => ;
```

As in Ada, a comment is introduced by a double hyphen '--' and is terminated by the end of the line. Also the IDL is case sensitive, which means identifiers that are spelled identically except for the case are considered to be different. Identifiers in IDL consist of a letter followed by an optional sequence of letters, digits, and isolated underscore characters.

External Representation of DIANA

A standard external form of DIANA is defined to help in debugging and to allow communication between computing systems. The square brackets surround the attributes of a node, and the angle brackets surround items of a sequence. Examples of ASCII representation of DIANA nodes:

```
plus
leaf [ name "A" ]
tree [ left leaf [ name "A" ] ]
```

The node 'plus' has no attributes, and the node 'leaf' has one attribute 'name' which is 'A'. Also the node 'tree' has one attribute 'left leaf', which contains another node with one attribute 'name'.

Each node is represented by its corresponding name. The representations of its attributes are separated by semicolons, and are surrounded by opening and closing brackets. If there are no attributes, the brackets may be omitted.

Each attribute in the nodes is represented by the name of the attribute, followed by the representation of the attribute's value.

In the nodes, comments begin with the double hyphen and terminate with the end of the line. Spaces are insignificant except to separate tokens, while case distinctions are significant.

To have a shared attribute value, one occurrence of the value must be labeled and all other occurrences must refer to that label. Any attribute may be labeled. Each label is followed by a colon and its node name. Each label reference consists of the label identifier followed by a circumflex instead of the usual representation of the attribute value. A label identifier consists of a letter followed by an optional sequence of letters, digits, and isolated underscore characters. There are many ways in which A+A can be represented. Two of these ways are shown below.

```
tree [ left leaf [ name "A"] ;
       op plus ;
       right leaf [ name "A] ]

tree [ left A01^ ;
       op plus ;
       right A01^ ]
A01: leaf [ name "A"]
```

A complete external representation begins with the root node of the structure followed by a sequence of zero or more nodes. The root indication can be either a label referencing itself or another node. Since the representation of the subnodes can be contained within the parent node, it is possible for the entire external representation to be given by the root. It is also legal to represent the DIANA tree in a flat form, where node-valued attributes are always referring to labels of non-nested nodes.

Following is an example of a partial Mini-task program and its DIANA external representation.

```
procedure progl is
   a,b,c : integer;

   task tl;

   task body tl is
      f,g,h : integer;
```

Figure 4. Sample partial Mini-task program.

The corresponding DIANA representation. The formal definition of DIANA in chapter V may be needed to completely understand the tree.

```
A0: compilation               [ as_list < Al^ > ]

Al: comp_unit                 [ as_unit_body A2^ ;
                                as_pragma_s ]
```

Figure 5. Sample partial DIANA program.

```
A2: subprogram_body              [ as_designator A3^ ;
                                   as_header A4^ ;
                                   as_block_stub A6^ ]

A3: proc_id                      [ lx_symrep "prog1" ;
                                   sm_spec ;
                                   sm_body ;
                                   sm_location ;
                                   sm_stub ;
                                   sm_first ]

A4: procedure                    [ as_param_s A5^ ]

A5: param_s                      [ as_list < > ]

A6: block                        [ as_item_s A7^ A19^ ;
                                   as_stm_s ;
                                   as_alternative_s ]

A7: item_s                       [ as_list < A8^ A15^ > ]

A8: var                          [ as_id_s A9^ ;
                                   as_type_spec A13^ ;
                                   as_object_def void ]

A9: id_s                         [ as_list < A10^ A11^ A12^ >]

A10: var_id                       [ lx_symrep "a" ;
                                    sm_obj_type ;
                                    sm_address ;
                                    sm_obj_def ]

A11: var_id                       [ lx_symrep "b" ;
                                    sm_obj_type ;
                                    sm_address ;
                                    sm_obj_def ]

A12: var_id                       [ lx_symrep "c" ;
                                    sm_obj_type ;
                                    sm_address ;
                                    sm_obj_def ]

A13: constrained                  [ as_name A14^ ;
                                    as_constraint void ;
                                    cd_impl_size ;
                                    sm_type_struct ;
                                    sm_base_type ;
                                    sm_constraint ]

A14: used_name_id                 [ lx_symrep "integer" ;
                                    sm_defn ]
```

Figure 5. (Continued)

```
A15: task_decl                [ as_id A18^ ;
                                as_task_def A16^ ]

A16: task_spec                [ as_decl_s A17^ ;
                                sm_body ;
                                sm_address ;
                                sm_storage_size ]

A17: decl_s                   [ as_list < > ]

A18: var_id                   [ lx_symrep "tl" ]

A19: item_s                   [ as_list < A20^ > ]

A20: var                      [ as_id_s A21^ ;
                                as_type_spec A25^ ;
                                as_object_def void ]

A21: id_s                     [ as_list < A22^ A23^ A24^ >]

A22: var_id                   [ lx_symrep "f" ;
                                sm_obj_type ;
                                sm_address ;
                                sm_obj_def ]

A23: var_id                   [ lx_symrep "g" ;
                                sm_obj_type ;
                                sm_address ;
                                sm_obj_def ]

A24: var_id                   [ lx_symrep "h" ;
                                sm_obj_type ;
                                sm_address ;
                                sm_obj_def ]

A25: constrained              [ as_name A26^ ;
                                as_constraint void ;
                                cd_impl_size ;
                                sm_type_struct ;
                                sm_base_type ;
                                sm_constraint ]

A26: used_name_id             [ lx_symrep "integer" ;
                                sm_defn ]
```

Figure 5. (Continued)

CHAPTER V

FORMAL LANGUAGE DESCRIPTION, SYNTAX, AND SEMANTICS

The context-free syntax of the language implemented is described using a simple variant of Backus-Naur-Form. Square brackets imply optional items; braces imply zero or more occurrences.

Below each set of production rules for Ada, the definition of the DIANA abstract type is given.

```
--      task-declaration ::= task-specification ;
--
--      task-specification ::= TASK [TYPE] identifier [is
--                                  { entry_declaration }
--                             END [task_simple_name]]

         TASK_DEF ::= task_spec;

         task_decl => as_id              : ID,
                      as_task_def        : TASK_DEF;

         TYPE_SPEC ::= task_spec;

         task_spec => as_decl_s          : DECL_S,
         task_spec => sm_body            : BLOCK_STUB_VOID,
                      sm_address         : EXP_VOID,
                      sm_storage_size    : EXP_VOID;

         BLOCK_STUP_VOID ::= block | stub | void;
```

Figure 6. Formal description of Mini-task and DIANA

```
--   task-body ::= TASK BODY task_simple_name IS
--                    [declarative_part]
--                 BEGIN
--                    sequence_of_statements
--                 END [task_simple_name] ;

        task_body => as_id            : ID,
                     as_block_stub    : BLOCK_STUB;

        DEF_ID ::= task_body_id;

                        lx_symrep     : symbol_rep;
        task_body_id => sm_type_spec  : TYPE_SPEC,
                        sm_body       : BLOCK_STUB_VOID,
                        sm_first      : DEF_OCCURRENCE,
                        sm_stub       : DEF_OCCURRENCE;

--   entry_declaration  ::= ENTRY
--                            identifier [(discrete_range)]
--                            [formal_part] ;

        HEADER ::= entry;

        DSCRT_RANGE_VOID ::= DSCRT_RANGE | void;

        entry => as_dscrt_range_void : DSCRT_RANGE_VOID,
                 as_param_s          : PARAM_S;

        DEF_ID ::= entry_id;

                    lx_symrep        : symbol_rep;
        entry_id => sm_spec          : HEADER,
                    sm_address       : EXP_VOID;


--   entry_call_statement ::= entry_name
--                               [actual_parameter_part] ;

        entry_call => as_name                : NAME,
                      as_param_assoc_s       : PARAM_ASSOC_S;
        entry_call =>  sm_normalize_param_s  : EXP_S;


--   accept_statement ::= ACCEPT entry_simple_name
--                           [{entry_index}]
--                           [formal_part] [DO
--                           sequence_of_statements
--                        END [entry_simple_name]];
--
--   entry_index ::= expression
```

Figure 6. (Continued)

```
        accept => as_name                  : NAME,
                  as_param_assoc_s         : PARAM_ASSOC_S;
        accept => as_stm_s                 : STM_S;


--    delay_statement ::= DELAY simple_expression ;

         delay => as_exp          : EXP;


--    select_statement ::= selective_wait
--                       | conditional_entry_call
--                       | timed_entry_call
--
--    selective_wait ::= SELECT
--                          select_alternative
--                       { OR
--                          select_alternative }
--                       [ELSE
--                          sequence_of_statements]
--                       END SELECT ;

        select => as_select_clause_s       : SELECT_CLAUSE_S,
                  as_stm_s                 : STM_S;

        SELECT_CLAUSE_S ::= select_clause_s;

        select_clause_s => as_list    : seq of SELECT_CLAUSE;


--    select_alternative ::= [WHEN condition =>]
--                                selective_wait_alternative
--
--    selective_wait_alternative ::= accept_alternative
--                                 | delay_alternative
--                                 | terminate_alternative
--
--    accept_alternative ::= accept_statement
--                                [sequence_of_statements]
--
--    delay_alternative ::= delay_statement
--                                [sequence_of_statements]
--
--    terminate_alternative ::= TERMINATE ;
--

        SELECT_CLAUSE ::= select_clause;
        SELECT_CLAUSE ::= pragma;

        select_clause => as_exp_void      : EXP_VOID,
                         as_stm_s         : STM_S;
```

Figure 6. (Continued)

```
--   conditional_entry_call ::= SELECT
--                                 entry_call_statement
--                                 [sequence_of_statements]
--                              ELSE
--                                 sequence_of_statements
--                              END SELECT ;

        cond_entry => as_stm_sl        : STM_S,
                      as_stm_s2        : STM_S;


--   timed_entry_call ::= SELECT
--                           entry_call_statement
--                           [sequence_of_statements]
--                        OR
--                           delay_alternative
--                        END SELECT;

        timed_entry => as_stm_sl        : STM_S,
                       as_stm_s2        : STM_S;


--   abort_statement ::= ABORT task_name {, task_name } ;

        NAME_S ::= name_s;

        name_s => as_list          : seq of NAME;

        abort => as_name_s         : NAME_S;
```

Figure 6. (Continued)

CHAPTER VI

IMPLEMENTATION TECHNIQUES

Introduction to Compilers

A compiler is a program that reads a program written in the source language and translates it into an equivalent program in the target language. There are many types of source and target languages. Source languages range from the traditional programming languages FORTRAN and COBOL to specialized languages for solving certain problems. A target language may be another programming language, or a certain machine language.

There are two parts to a compiler: the analysis and the synthesis. The analysis of the compiler consists of the following three phases. Lexical analysis groups characters read from the input into meaningful tokens. Hierarchical analysis, also called syntax analysis, groups tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Semantic analysis checks to ensure that the components of a program fit together meaningfully. The synthesis phase translates the intermediate representation into the desired target program [Aho, 1986, Lorho, 1984].

An essential task of a compiler is to record the identifiers used in the source program and maintain information about various attributes of each identifier. A symbol table is a data structure, used by all phases of the compiler, which contains a record for each identifier. Each record contains fields for the attributes of the identifier which contain pertinent information such as its type, its scope, and if it is a procedure name, information about its arguments and the type returned, if any.

Every phase of the compiler can possibly encounter errors. When a phase does encounter an error, it must deal with the error, and continue processing to allow additional errors to be detected.

The front end of the compiler consists of the analysis, as described above, symbol table creation, the generation of the intermediate code, and, of course, the necessary error handling routines that go along with each phase. The back-end of the compiler consists of the code optimization phase, the code generation phase, and also the necessary symbol table management and error handling routines. The study in this thesis deals with the front end of the Mini-task compiler. A possible back-end is discussed briefly in chapter VII.

Implementation Decisions

It is necessary to address specific implementation techniques of the compiler developed in this thesis.

As previously mentioned, the external form of DIANA is used to assist in debugging and to allow communication between computing systems, but it is not the typical communication between tools. However, this study primarily is concerned with the compiler front-end only. Therefore, a compiler back-end needs to be developed to consume the external representation produced by this study's compiler.

The flat form of external representation was chosen from the forms described above. The flat form is more direct in the areas of reading and debugging; consequently, the possible occurrence of errors is reduced.

Implementation Techniques

The implementation of Mini-task is broken into six stages: the lexical analysis, the symbol table routines, the generation of the intermediate code based on DIANA, the syntactic analysis, the semantic analysis, and the error handling routines. These stages will be discussed in various segments of this chapter.

The main program of the compiler is in the file "main.c" and consists of several of the phases drivers. The first step of the main program is to read the command line to determine which source file to compile. This is done through the C language's way of passing the command-line arguments to the main. When main is called to begin execution, it is called with two arguments. The first, (argc) is the number of command-line arguments with which the program was invoked; the second (argv) is a pointer to an array of character strings that contain the arguments [Kernighan, 1978]. "If" statements are used to test the number of arguments. If there are no arguments, Mini-task terminates. If there is more than one argument, an error message is printed, and the program terminates. Since Mini-task requires that all Mini-task programs end with filenames ending with ".mt", the last three characters are tested. If the characters are not ".mt", an error message is printed, and the program terminates. If the constant "DEBUG" is assigned a positive value in the header file "mt.h", the debug file is opened for a compiler trace. If the constant "DEBUG" is assigned the value "0", no debugging statements are printed. The second phase of the main program is the initialization of the variables. The third phase calls the parser (yyparse) which in turn calls the lexical analyzer (yylex) and other routines discussed in this chapter. The fourth and final phase of the main program calls the routines "chkblks" and "sm_diana" to complete the final semantic analysis phases.

As mentioned above, the third phase of the main program results in calling the lexical analyzer. The lexical analyzer is responsible for reading the source file and producing tokens for the parser. A token consisting of all alphabetic characters could be an identifier name or a reserved word; therefore, the routine "rwtable" is called to determine if the token is in the list of reserved words.

"rwtable" consists of a binary search which searches for the reserved words.

<u>Reserved Words</u>
abort
accept
begin
body
delay
do
else
end
entry
in
integer
is
null
or
out
procedure
put
select
task
terminate
type
when

Figure 7. Mini-task reserved words.

If the token is identified as a reserved word, the value of the reserved word is returned to the lexical analyzer, which is returned to the parser. If the token is not a reserved word, it is obviously an identifier, which causes the value of the identifier token to be returned.

All strings regardless of whether they are uppercase or lowercase are sent to the routine "lcase" which copies the original string (yytext) to a new string (temp), while converting all uppercase to lowercase. This conversion allows for different programmer styles of case distinction. The programmer never sees the case conversion, because the original string is printed in the listing file. The lower case string is only used for the symbol table routines and the reserved word lookup routines.

The listing file is produced at the time of lexical analysis. In order to have error messages printed below the source line, a buffer is used to store the error messages. The source line is also stored in a buffer in order to keep track of the characters for error displaying. A dollar sign is printed below the first character of the token that causes the error. Warnings are displayed in a similar manner, except a circumflex is displayed below the tokens causing warnings. The only so called warnings in Mini-task are undecodable characters. If the lexical analyzer cannot identify a character, a warning is produced, and the character is ignored by the parser.

The file "mt.h" is a header file that is included in most of the routines in the compiler. "Mt.h" contains the data structures used for the DIANA templates, the symbol table routines, and various stacks throughout the compiler. During the running of a compiler, a task control block is kept of the current task. The data structure works as a stack, where the parser pushes the name of the task on the stack when it encounters the declaration of the task or the task body.

A stack is used to keep track of the current task. When the parser encounters a task declaration or a task body, it pushes the task name on the stack. When the parser encounters the corresponding end to one of these entities, the parser pops the blockname off of the top of the stack. Also, the testing to see if the declared task name matches the ending name is accomplished at this point. The stack uses the following data structure.

```
struct stacker {
   char blockname[MAXIDLEN]
   } ;
```

During the semantic analysis phase, a routine (checkends) is called to see if the the declared tasks match the tasks bodies. If they do not match, "yyerror" is called.

"Yyerror" is the error handling procedure. Errors can be detected in all phases of the compiler. When "yyerror" is called, the number of errors, "nerrs", for the line is set to true, and the total number of errors for the program, "tnerrs", is incremented. A dollar sign is inserted into the error message buffer, "temperrl", and the error message is concatenated to the buffer also.

The parser is the heart of the compiler. Yacc, Yet Another Compiler Compiler, is used to build the parser.

"Insert" is a routine that inserts DIANA nodes into the DIANA tree. A template is allocated when a node is needed by the routine "getnode". "Getnode" requires one parameter: the node name. Each DIANA node is inserted into a DIANA node template.

```
struct template {
   char name[MAXIDLEN];
   struct {
      char name[MAXIDLEN];
      struct anode *list;
          } att[MAXATT];
       } ;
```

MAXIDLEN is a constant containing the maximum identifier length. MAXATT is a constant containing the maximum number of attributes each DIANA node has. If this program were to be expanded, MAXATT would probably increase, because some DIANA nodes require more attributes. Each DIANA node has attributes which may have any number of children. Since the number may be small or large, a linked list is used to keep the list of children. The following data structure is used to keep a list of the children nodes.

```
struct anode {
          struct anode *alist;
          int dnode;
       } ;
```

The routine "alocnode" allocates the nodes. Each node contains a pointer for the next node, which has the value of NULL when it is a leaf node, and and integer pointing to the leaf. The symbol table and supporting routines are very simple because the symbol table is used only for semantic analysis, and there is only one type: INTEGER; The symbol table does not contain the storage for the values of the variables as in most languages. DIANA is the only structure that is passed to the back-end of the compiler, consequently, the DIANA nodes are used for the actual storage of the variables. The data structure for the symbol table follows.

```
struct stemplate {
   char blockname[MAXIDLEN];
   struct {
      char name[MAXIDLEN];
   } symbol[MAXSYM];
   } ;
```

MAXIDLEN is a constant that contains the maximum identifier length. The constant MAXSYM contains the maximum number of elements for the symbol table [Johnson, 1975, Lesk, Schreiner, 1985].

Following is an example of how nodes are inserted into the DIANA tree.

```
   if (strcmp(type,"compilation") == 0){
                                          /* Compilation */
      dpointer = getnode("compilation");
      addatt(dpointer,"as_list <");

                                          /* Comp_unit */
      dpointer = getnode("comp_unit");
      dlink(dcurrent,"as_list <",dpointer);
      addatt(dpointer,"as_unit_body");
      addatt(dpointer,"as_pragma_s");

      dpointer = getnode("subprogram_body");
      dlink(comp_unit,"as_unit_body",dpointer);
      addatt(dpointer,"as_designator");
      addatt(dpointer,"as_header");
      addatt(dpointer,"as_block_stub");
   }
```

Figure 8. Example of "insert" program.

Before the first line of the source program is parsed, the parser calls the routine "insert" and sends the node to be inserted which is "compilation". An "if" construct is used to find the correct entry". When the correct entry is found, a node is allocated by the routine "getnode". "Dcurrent" and "dpointer" point to the new node. Since the node "compilation" has only one node, "addatt", a routine to insert attributes into the node, is only called once. "Addatt" sends the pointer to the node in the tree and the attribute to be inserted into that node. The nodes "comp_unit" and "subprogram_body" are inserted into the tree at the same time. "Dlink" is a routine used to link the nodes together. For a more detailed description of the compiler, refer to the Appendix.

CHAPTER VII

DISCUSSION OF POSSIBLE BACK-END TECHNIQUES

The implementation of Mini-task in this thesis consists of the front-end only. However, the back-end deserves a short discussion. To begin, the compiler writer would need to gain a thorough understanding and a working knowledge of DIANA and the machine that is used for implementation. Also, the compiler writer would need to be familiar with the following techniques. Two important factors in implementing a back-end of Mini-task are memory management techniques and task switching.

Memory Management

Many modern high-level languages are implemented using a stack-based memory management system. The available run-time storage is organized as a single stack with code and static data at the bottom and free storage at the top. A stack pointer is positioned at the bottom of the free area. As memory is allocated the stack pointer moves up the stack, and as memory is released it moves down. Storage must therefore be freed in the reverse order to that of allocation. Most operations take place on the top of the stack [Burns, 1985, Lorho, 1984, Waite, 1984].

When a subprogram is called, a new activation record is placed on the top of the stack which includes local variables and a return address. Chains of subprogram calls therefore cause no difficulty, nor does recursive calling or reentrant usage. As Ada has recursive subprogram calls and data structures whose size is calculated at run-time, it encourages the use of a stack-based memory management structure. What loss of efficiency there might be by not utilizing fully the available hardware registers is compensated by portability considerations [Aho, 1986, Lorho, 1984, Pratt, 1981, Waite, 1984, Burns, 1985].

With a multi-tasking program each task can, to some extent, be seen as a separate program, so each task will have its own stack. However, creation of a task is dependent on some state of the parent (master). A child task may also require access to shared variables held on its parent's stack. From these considerations a structure known as a cactus stack is used. A cactus stack consists of a variable number of stacks, one for each task in the program [Burns, 1985, Baker, 1985].

Task Switching

When there is only one processor, a context switch must let the different tasks take turns on the CPU. In order to switch tasks the old task's activation record must be stored on the run time stack and the new task's record mounted. Ideally, this should be done with very few machine instructions, or maybe even just one. If this cannot be accomplished, the code should be optimized so that fewer context switches are needed [Burns, 1985].

CHAPTER VIII

SUMMARY, CONCLUSIONS, AND SUGGESTED FURTHER RESEARCH

Summary and Conclusions

In this thesis Mini-task and its front-end implementation are discussed. Mini-task is a language based on Ada's tasking model. The front-end of the compiler produces an intermediate code based on DIANA. The intermediate language DIANA has proven to be a well suited intermediate language for Mini-task. DIANA is flexible, and it has the potential to become a widely accepted intermediate language. DIANA also possesses the versatility to be used as the intermediate language for many different compilers.

The front-end of Mini-task is currently implemented on a Perkin Elmer 3230 running UNIX System V. The implementation is written with approximately 1200 lines of "C" code and 250 lines of "lex" and "yacc" code. The routines are divided into fifteen files. UNIX's "makefile" utility is used to compile the programs.

Mini-task is a well suited language for programmers who want to write multi-tasking programs. Due to its small size, Mini-task has the potential to produce an efficient target program. The intermediate code produced, as shown in this thesis, is considered compact, as compared with many Ada program's intermediate codes.

Suggestions for Future Research

During the investigation of Mini-task and its front-end implementation, it became clear that developing a standardized intermediate language would be beneficial to computer scientists. It would enable computer scientists to have one intermediate language, and each machine would only require a standardized back-end compiler. When a new language is designed, only the front-end of the compiler for the new language would need to be developed to produce the standardized intermediate code.

There are some directions suitable for extending the current work: an implementation of a DIANA consumer and a DIANA employer. The DIANA consumer would accept as input the output of the compiler discussed in this thesis, and other compilers producing DIANA. Implementation of a DIANA employer is another interesting research topic. A DIANA employer is a program that uses DIANA as an intermediate language, without producing the external representation. The program would consist of a front-end producing DIANA and a back-end consuming DIANA.

There is also a potential to expand the current implementation of Mini-task. Ideally, Mini-task would contain features such as more types, preferably programmer defined types, separate compilation, and exception handlers.

SELECTED BIBLIOGRAPHY

Aho, A. B., Sethi, R, Ullman, J. D. (1986). Compilers Principles, Techniques, and Tools. Reading, MA: Addison-Wesley Publishing Company.

Baker, T. P., Riccardi, G.A. (1985). Ada Tasking: from Semantics to efficient implementation. IEEE Software. 2,2, Pages 9-22.

Burns, A. (1985). Concurrent Programming in Ada. Cambridge: Cambridge University Press.

Gilpin, G. (1986). Ada: A guided Tour and Tutorial. New York, NY: Prentice Hall Press.

Goos, G., Harmanis, J. (1982). An Attribute Grammar for the Semantic Analysis of Ada. New York, NY: Springer-Verlag.

Goos, G., Winterstein, G. (1981). Trends in Information Processing Systems. 3rd Conference of the European Cooperation in Informatics. Berlin, Germany: Sprinter-Verlag.

Halstead, M. H. (1974). A Laboratory Manual for Compiler and Operating System Implementation. New York, NY: American Elsevier Publishing Company, Inc.

Helmbold, D., Luckham, D. (1985). Debugging Ada Tasking Programs. IEEE Software. 2,2, Pages 47-57.

Johnson, S. C. (1975). Yacc: Yet Another Compiler-Compiler. Bell Laboratories, Murray Hill, NJ.

Kernighan, B. W., Ritchie, D. M. (1978). The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Ledgard, H., Marcotty, M. (1981). The Programming Language Landscape. Chicago, IL: Science Research Associates, Inc.

Lesk, M. E., Schmidt E. LEX - A lexical Analyzer Generator. Bell Laboratories, Murray Hill, NJ.

Lorho, B. (1984). Methods and Tools for Compiler Construction. Cambridge: Cambridge University Press.

Persch, G. (1983). Ada Compiler Karlsruhe Overview. Ada-Europe/Adatec Joint Conference on Ada. Pages 2.1-2.4.

Pratt, T. W. (1984). Programming Languages Design and Implementation. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Rogers, M. W. (1984). Ada: Language, Compilers and Bibliography. Cambridge: Cambridge University Press.

Rosenblum, D. S. (1985). A Methodology for the design of Ada transformation tools in a DIANA environment. IEEE Software, 2, 2, Pages 34-36.

Schreiner, A. T., Friedman, Jr. G. H. (1985). Introduction to Compiler Construction with UNIX. Englewood Cliffs, NJ: Prentice-Hall, Inc.

United States Department of Defense, (1983). Reference Manual for the Ada Programming Language. New York, NY: Sprintger-Verlag.

Waite, W. M., Goos, G. (1984). Compiler Construction. New York, NY: Springer-Verlag.

Wiener, R., Sincovec, R. (1983). Programming in Ada. New York, NY: John Wiley & Sons.

APPENDIXES

MINI-TASK COMPILER

```
#include <stdio.h>
addonword(i,sl,s2)
   int i;
   char *sl,*s2;{

   int j;

   for(j=0; (strcmp(dnode[i].att[j].name,sl) != 0); )
      if(++j >= MAXATT){
         fprintf(stderr,"addonword: can't find the att0);
         exit(4);
      }
   strcat(dnode[i].att[j].name," ");
   strcat(dnode[i].att[j].name,s2);
}
addonstring(i,sl,s2)
   int i;
   char *sl,*s2;{

   int j;

   for(j=0; (strcmp(dnode[i].att[j].name,sl) != 0); )
      if(++j >= MAXATT){
         fprintf(stderr,"astring: can't find the att0);
         exit(4);
      }
   strcat(dnode[i].att[j].name,"
   strcat(dnode[i].att[j].name,s2);
   strcat(dnode[i].att[j].name,"
}
addatt(i,s)
   int i;
   char *s;{

   struct anode *alocnode();
   int j;

   for(j=0; (strcmp(dnode[i].att[j].name,"") != 0); )
      if(++j >= MAXATT){
         fprintf(stderr,"Not enough atts; fix MAXATT0);
         exit(4);
      }
   strcpy(dnode[i].att[j].name,s);
}
dlink(i,s,j)
```

```
   int i;
   char *s;
   int j;{

   struct anode *alocnode(), *temp, *prev;
   int k;

   for(k=0; (strcmp(dnode[i].att[k].name,s) != 0); )
      if((++k >= MAXATT) ||
         (strcmp(dnode[i].att[k].name,"") == 0)){
         fprintf(stderr,"can't find att: %s0,s);
         exit(4);
      }
   prev = temp = dnode[i].att[k].list;
   while (temp != (struct anode *) NULL){
      if (temp->dnode == -1){
         temp->dnode = j;
         return;
      }
      prev = temp;
      temp = temp->alist;
   }
   if (prev == (struct anode *) NULL)
      dnode[i].att[k].list = temp = alocnode();
   else
      prev->alist = temp = alocnode();
   temp->dnode = j;
}
addent(s)
   char *s;{

   int i;

   for(i=0 ; i < MAXSYM; i++){
      if(strcmp(stable[curb].entry[i].name,s) == 0){
         yyerror("Entry already declared ");
         return(i);
      }
      else if(strcmp(stable[curb].entry[i].name,"") == 0){
         strcpy(stable[curb].entry[i].name,s);
         return(i);
      }
   }
   fprintf(stderr,"addent: MAXSYM too small0);
   exit(4);
}
chkent(tname,ename)
   char *tname;
   char *ename;{

   int block,i;

   for(block=0 ; block < MAXSYM; block++)
```

```
      if(strcmp(stable[block].blockname,tname) == 0)
         break;

   if(block >= MAXSYM){
      yyerror("Undefined task ");
      return;
   }

   for(i=0 ; i < MAXSYM; i++)
      if(strcmp(stable[block].entry[i].name,ename) == 0)
         return;

   yyerror("Undefined entry ");
}
dinit(ptr)
   int ptr; {

   int i;

   strcpy(dnode[ptr].name,"");
   for(i=0; i<MAXATT; i++){
      strcpy(dnode[ptr].att[i].name,"");
      dnode[ptr].att[i].list = (struct anode *) NULL;
   }
}
ainit(ptr)
   struct anode *ptr; {

   ptr->dnode = -1;
   ptr->alist = (struct anode *) NULL;
}
getnode(s)
   char *s;{

   if(++nnodes > MAXDNODES){
      fprintf(stderr,"not enough nodes allocated0);
      exit(4);
   }
   dinit(nnodes-1);
   strcpy(dnode[nnodes-1].name,s);
   return(nnodes-1);
}
struct anode *alocnode(){

   struct anode *ptr;

   ptr = (struct anode *) malloc(sizeof(struct anode));
   ainit(ptr);
   return(ptr);
}
int nnodes;     /* Curent number of DIANA nodes */

struct template dnode[MAXDNODES];
struct anode *aprevious, *acurrent, *apointer;

insert(type) char *type; {

   struct anode *alocnode();

   static int dnumber,
       mode,
       dcurrent,
       dpointer,
       block,
       var,
       id_s,
       var_id,
       subprogram_decl,
       entry,
       TASK_id_s,
       task_decl,
       param_s,
       task_spec,
       in,
       in_out,
       out,
       decl_s,
       constrained,
       subprogram_body,
       procedure,
       comp_unit;

   int itop;

   if (strcmp(type,"compilation") == 0){
      dcurrent = dpointer = getnode("compilation");
      addatt(dpointer,"as_list <");

      comp_unit = dpointer = getnode("comp_unit");
      dlink(dcurrent,"as_list <",dpointer);
      addatt(dpointer,"as_unit_body");
      addatt(dpointer,"as_pragma_s");

      subprogram_body = dpointer =
                     getnode("subprogram_body");
      dlink(comp_unit,"as_unit_body",dpointer);
      addatt(dpointer,"as_designator");
      addatt(dpointer,"as_header");
      addatt(dpointer,"as_block_stub");
   }
   else if (strcmp(type,"proc_id") == 0){
      dcurrent = dpointer = getnode("proc_id");
      addatt(dpointer,"lx_symrep");
      addonstring(dpointer,"lx_symrep",yytext);
      addatt(dpointer,"sm_spec");
      addatt(dpointer,"sm_body");
   addatt(dpointer,"sm_location");
   addatt(dpointer,"sm_stub");
   addatt(dpointer,"sm_first");
   dlink(subprogram_body,"as_designator",dpointer);
}
else if (strcmp(type,"procedure") == 0){
   procedure = dpointer = getnode("procedure");
   addatt(dpointer,"as_param_s");
   dlink(subprogram_body,"as_header",dpointer);

   dpointer = getnode("param_s");
   addatt(dpointer,"as_list <");
   dlink(procedure,"as_param_s",dpointer);

   block = dpointer = getnode("block");
   addatt(dpointer,"as_item_s");
   addatt(dpointer,"as_stm_s");
   addatt(dpointer,"as_alternative_s");
   dlink(subprogram_body,"as_block_stub",dpointer);
}
else if (strcmp(type,"item_s") == 0){
   stable[curb].item_s = dpointer = getnode("item_s");
   addatt(dpointer,"as_list <");
   dlink(block,"as_item_s",dpointer);
}
else if (strcmp(type,"var") == 0){
   var = dpointer = getnode("var");
   addatt(dpointer,"as_id_s");
   addatt(dpointer,"as_type_spec");
   addatt(dpointer,"as_object_def");
   addonword(dpointer,"as_object_def","void");
   dlink(stable[curb].item_s,"as_list <",dpointer);
}
else if (strcmp(type,"id_s") == 0){
   id_s = dpointer = getnode("id_s");
   addatt(dpointer,"as_list <");
   dlink(var,"as_id_s",dpointer);
}
else if (strcmp(type,"var_id") == 0){
   var_id = dpointer = getnode("var_id");
   addatt(dpointer,"lx_symrep");
   addonstring(dpointer,"lx_symrep",yytext);
   addatt(dpointer,"sm_obj_type");
   addatt(dpointer,"sm_address");
   addatt(dpointer,"sm_obj_def");
   dlink(id_s,"as_list <",dpointer);
}
else if (strcmp(type,"constrained") == 0){
   constrained = dpointer = getnode("constrained");
   addatt(dpointer,"as_name");
   addatt(dpointer,"as_constraint");
   addatt(dpointer,"cd_impl_size");
   addatt(dpointer,"sm_type_struct");
```

```
      addatt(dpointer,"sm_base_type");
      addatt(dpointer,"sm_constraint");
      addonword(dpointer,"as_constraint","void");
      dlink(var,"as_type_spec",dpointer);

      dpointer = getnode("used_name_id");
      addatt(dpointer,"lx_symrep");
      addonstring(dpointer,"lx_symrep",yytext);
      addatt(dpointer,"sm_defn");
      dlink(constrained,"as_name",dpointer);
   }
   else if (strcmp(type,"task_decl") == 0){
      task_decl = dpointer = getnode("task_decl");
      addatt(dpointer,"as_id");
      addatt(dpointer,"as_task_def");
      dlink(stable[curb].item_s,"as_list <",dpointer);

      task_spec = dpointer = getnode("task_spec");
      addatt(dpointer,"as_decl_s");
      addatt(dpointer,"sm_body");
      addatt(dpointer,"sm_address");
      addatt(dpointer,"sm_storage_size");
      dlink(task_decl,"as_task_def",dpointer);

      decl_s = dpointer = getnode("decl_s");
      addatt(dpointer,"as_list <");
      dlink(task_spec,"as_decl_s",dpointer);
   }
   else if (strcmp(type,"TASK_var_id") == 0){
      dpointer = getnode("var_id");
      addatt(dpointer,"lx_symrep");
      addonstring(dpointer,"lx_symrep",yytext);
      dlink(task_decl,"as_id",dpointer);
   }
   else if (strcmp(type,"subprogram_decl") == 0){
      subprogram_decl = dpointer =
         getnode("subprogram_decl");
      addatt(dpointer,"as_designator");
      addatt(dpointer,"as_header");
      addatt(dpointer,"as_subprogram_def");
      dlink(decl_s,"as_list <",dpointer);

      entry = dpointer = getnode("entry");
      addatt(dpointer,"as_dscrt_range_void");
      addonword(dpointer,"as_dscrt_range_void","void");
      addatt(dpointer,"as_param_s");
      dlink(subprogram_decl,"as_header",dpointer);

      param_s = dpointer = getnode("param_s");
      addatt(dpointer,"as_list <");
      dlink(entry,"as_param_s",dpointer);
   }
   else if (strcmp(type,"entry_id") == 0){
      dpointer = getnode("entry_id");
      addatt(dpointer,"lx_symrep");
      addonstring(dpointer,"lx_symrep",yytext);
      addatt(dpointer,"sm_spec");
      addatt(dpointer,"sm_address");
      dlink(subprogram_decl,"as_designator",dpointer);
   }
   else if (strcmp(type,"in") == 0){
      mode = in = dpointer = getnode("in");
      addatt(dpointer,"as_id_s");
      addatt(dpointer,"as_name");
      addatt(dpointer,"as_exp_void");
      addonword(dpointer,"as_exp_void","void");
      dlink(param_s,"as_list <",dpointer);

      TASK_id_s = dpointer = getnode("id_s");
      addatt(dpointer,"as_list <");
      dlink(in,"as_id_s",dpointer);

      for(itop=0; itop < top; itop++){
         dpointer = getnode("in_id");
         addatt(dpointer,"lx_symrep");
         addonstring(dpointer,"lx_symrep",stack[itop].id);
         addatt(dpointer,"sm_obj_type");
         addatt(dpointer,"sm_first");
         dlink(TASK_id_s,"as_list <",dpointer);
      }
      top = 0;
   }
   else if (strcmp(type,"in_out") == 0){
      mode = in_out = dpointer = getnode("in_out");
      addatt(dpointer,"as_id_s");
      addatt(dpointer,"as_name");
      addatt(dpointer,"as_exp_void");
      addonword(dpointer,"as_exp_void","void");
      dlink(param_s,"as_list <",dpointer);

      TASK_id_s = dpointer = getnode("id_s");
      addatt(dpointer,"as_list <");
      dlink(in_out,"as_id_s",dpointer);

      for(itop=0; itop < top; itop++){
         dpointer = getnode("in_out_id");
         addatt(dpointer,"lx_symrep");
         addonstring(dpointer,"lx_symrep",stack[itop].id);
         addatt(dpointer,"sm_obj_type");
         addatt(dpointer,"sm_first");
         dlink(TASK_id_s,"as_list <",dpointer);
      }
      top = 0;
   }
   else if (strcmp(type,"out") == 0){
      mode = out = dpointer = getnode("out");
      addatt(dpointer,"as_id_s");
      addatt(dpointer,"as_name");
      addatt(dpointer,"as_exp_void");
      addonword(dpointer,"as_exp_void","void");
      dlink(param_s,"as_list <",dpointer);

      TASK_id_s = dpointer = getnode("id_s");
      addatt(dpointer,"as_list <");
      dlink(out,"as_id_s",dpointer);

      for(itop=0; itop < top; itop++){
         dpointer = getnode("out_id");
         addatt(dpointer,"lx_symrep");
         addonstring(dpointer,"lx_symrep",stack[itop].id);
         addatt(dpointer,"sm_obj_type");
         addatt(dpointer,"sm_first");
         dlink(TASK_id_s,"as_list <",dpointer);
      }
      top = 0;
   }
   else if (strcmp(type,"used_name_id") == 0){
      dpointer = getnode("used_name_id");
      addatt(dpointer,"lx_symrep");
      addonstring(dpointer,"lx_symrep",yytext);
      addatt(dpointer,"sm_defn");
      dlink(mode,"as_name",dpointer);
   }
   else if (strcmp(type,"delay") == 0){
      dpointer = getnode("delay");
      addatt(dpointer,"as_exp");
      dlink(mode,"xxxxxxx",dpointer);
   }
   else if (strcmp(type,"template") == 0){
      dpointer = getnode("template");
      addatt(dpointer,"lx_symrep");
      dlink(mode,"as_name",dpointer);
   }
   else if (strcmp(type,"template") == 0){
      dpointer = getnode("template");
      addatt(dpointer,"lx_symrep");
      dlink(mode,"as_name",dpointer);
   }
   else if (strcmp(type,"template") == 0){
      dpointer = getnode("template");
      addatt(dpointer,"lx_symrep");
      dlink(mode,"as_name",dpointer);
   }
   else if (strcmp(type,"template") == 0){
      dpointer = getnode("template");
      addatt(dpointer,"lx_symrep");
      dlink(mode,"as_name",dpointer);
   }
   else if (strcmp(type,"template") == 0){
      dpointer = getnode("template");
      addatt(dpointer,"lx_symrep");
      dlink(mode,"as_name",dpointer);
   }
   else{
      fprintf(stderr,"insert: can't find %s0,type);
      exit(4);
   }
}
islist(s)
char s[];{

   int i;

   for(i=0; s[i] != ' '; i++)
      if(s[i] == '<')
         return(1);
   return(0);
}
itoa(n,s)        /* convert n to characters in s */
char s[];
int n;{

   int i, sign;

   if ((sign = n) < 0)
      n = -n;
   i = 0;
   do {
      s[i++] = n % 10 + '0';
   } while ((n /= 10) > 0);
   if (sign < 0)
      s[i++] = '-';
   s[i] = ' ';
   reverse(s);
}
reverse(s)
char s[]; {

   int c, i, j;

   for (i = 0, j = strlen(s)-1; i<j; i++, j--){
      c = s[i];
      s[i] = s[j];
      s[j] = c;
   }
}
struct stemplate stable[MAXSYM];
struct stacker sstack[MAXSYM];
int stop;
int curb;

main(argc,argv)
```

```
         int argc;
         char *argv[]; {

         int i,j,k;
         FILE *fopen();
         char tempfn[20], dianafn[20];

         if(argc < 2)
            exit();
         else if(argc > 2){
            fprintf(stderr,"mt: too many arguements0);
            exit();
         }
         strcpy(tempfn,argv[1]);
         if((tempfn[strlen(tempfn)-3] != '.') ||
            (tempfn[strlen(tempfn)-2] != 'm') ||
            (tempfn[strlen(tempfn)-1] != 't')){
            fprintf(stderr,"input file must end with .mt0);
            exit();
         }
         if((yyin = fopen(tempfn,"r")) == (FILE *) NULL){
            printf("mt: can't open %s0,tempfn);
            exit();
         }
         tempfn[strlen(tempfn)-2] = ' ';
         strcpy(dianafn,tempfn);
         strcat(dianafn,"diana");
         strcpy(templine,"");
         strcpy(temperr1,"");
         strcpy(temperr2,"");
         strcpy(tempwarns,"");
         nerrs = 0;
         nwarns = 0;
         tnerrs = 0;
         tnwarns = 0;
         linenum = 0; /* Current line number  */
         nnodes = 0;  /* Number of DAINA Nodes in tree */
         for(j=0; j < MAXSYM; j++){
            strcpy(stable[j].blockname,"");
            for(k=0; k < MAXSYM; k++)
               strcpy(stable[j].symbol[k].name,"");
         }
         for(j=0; j < MAXSYM; j++)
            strcpy(sstack[j].blockname,"");
         stop = -1;
         curb = -1;
         while ((i = yyparse()) != 0);
         fprintf(stdout,"  Number of errors: %d0,tnerrs);
         fprintf(stdout,"Number of warnings: %d0,tnwarns);
         if(tnerrs == 0)
            treedump(dianafn,dnode,nnodes);
         fclose(yyin);
      }
```

```
yyerror(s)
char *s; {

   int i;

   nerrs++; tnerrs++;
   for(i=0; temperrl[i] != ' '; i++) ;
   temperrl[i-strlen(yytext)] = '$';
   strcat(temperr2,s);
}
spush(s,decmode)
   char *s;
   short decmode;{

   if(stop >= MAXSYM){
      fprintf(stderr,"too many levels; adust MAXSYM0);
      exit(4);
   }
   strcpy(sstack[++stop].blockname,s);
   setcurb();
   if(decmode == 0) return;
   if(decmode == 1 && stable[curb].decval > 0)
      yyerror("Task already declared ");
   else if(decmode == 2 && stable[curb].decval > 1)
      yyerror("Task body already defined ");
   else if(decmode == 2 && stable[curb].decval == 0)
      yyerror("Task undeclared");
   else if(decmode != 1 && decmode != 2)
      fprintf(stderr,"decmode is: %d",decmode);
   else{
      if(decmode == 1 && stop > 0)
         stable[curb].parent = findpar();
      else if(stable[curb].parent != findpar()
            && stop > 0)
         yyerror("Mismatch of parent");
      stable[curb].decval = stable[curb].decval + decmode;
   }
}
spop(){

   if(stop < 1){
      fprintf(stderr,"spop: too many pops0);
      exit(4);
   }
   stop--;
   setcurb();
}
setcurb(){

   int i;

   for(i=0 ; i < MAXSYM; i++)
      if(strcmp(stable[i].blockname,
             sstack[stop].blockname) == 0){
            curb = i;
            return(curb);
         }
      for(i=0 ; i < MAXSYM; i++)
         if(strcmp(stable[i].blockname,"") == 0){
            strcpy(stable[i].blockname,
              sstack[stop].blockname);
            curb = i;
            return(curb);
         }
      printf("ERROR: setcurb can't find the block0);
      exit(4);
   }
   chkblks(){

      int i,flag;
      char s[132];

      flag = 0;
      s[0] = ' ';
      for(i=1 ; i < MAXSYM; i++)
         if ((stable[i].parent == curb)
            && (stable[i].decval == 1)){
            flag++;
            strcat(s,stable[i].blockname);
         }
      if(flag){
         tnerrs++;
         printf("0** ERROR: Missing task bodies: %s0,s);
      }
   }
   findpar(){

      int i;

      for(i=0 ; i < MAXSYM; i++)
         if(strcmp(sstack[stop-1].blockname,
            stable[i].blockname) == 0)
            return(i);

      fprintf(stderr,"findpar can't find the parent0);
      exit(4);
   }
   st_isit(id)
      char id[]; {

      int i,j,k,temp;

      for(j=stop; j >= 0; j--){
         for(k=0 ; k < MAXSYM; k++)
            if(strcmp(stable[k].blockname,
              sstack[j].blockname) == 0)
            temp = k;
      for(i=0 ; i < MAXSYM; i++)
         if(strcmp(id,stable[temp].symbol[i].name) == 0)
            return(1);
   }

   yyerror("Variable undeclared");
}
st_insert(id)
   char id[]; {

   int i;

   for(i=0 ; i < MAXSYM; i++)
      if(strcmp(id,stable[curb].symbol[i].name) == 0){
         yyerror("Variable previously declared");
         return(-1);
      }

   for(i=0 ; i < MAXSYM; i++)
      if(strcmp(stable[curb].symbol[i].name,"") == 0){
         strcpy(stable[curb].symbol[i].name,id);
         return(1);
      }
   yyerror("Out of room in symbol table: adjust MAXSYM");
}
sendtest(s)
   char s[]; {

   if(strcmp(s,sstack[stop].blockname) != 0)
         yyerror("Identifier doesn't match block");
}
#define MAXTOP 20

struct distack stack[MAXTOP];

int top = 0;

push(s)
   char *s;{

   if(top >= MAXTOP){
      fprintf(stderr,"too many variables0);
      exit(4);
   }
   strcpy(stack[top++].id,s);
}
FILE *fopen(), *diana;

treedump(dianafn,dnode,nnodes)
   char dianafn[];
   struct template dnode[];
   int nnodes; {
```

```
   int i,j,isflag;
   char tlistl[130], tlist2[130];
   struct anode *templ;

   if((diana = fopen(dianafn,"w")) == (FILE *) NULL){
      fprintf(stderr,"error opening %s0,dianafn);
      return;
   }
   for(i=0; i<nnodes; i++){
      fprintf(diana,"A%d:",i);
      fprintf(diana," %-25s [ ",dnode[i].name);
      for(j=0; (j < MAXATT) &&
        (strcmp(dnode[i].att[j].name,"") != 0);j++){
         templ = dnode[i].att[j].list;
         isflag = islist(dnode[i].att[j].name);
         strcpy(tlistl,"");
         while ((templ != (struct anode *) NULL) &&
           (templ->dnode != -1)){
            strcat(tlistl,"A");
            itoa(templ->dnode,tlist2);
            strcat(tlistl,tlist2);
            strcat(tlistl,"^ ");
            templ = templ->alist;
         }
         if (isflag)
            strcat(tlistl,"> ");
         if (j > 0)
            fprintf(diana,";                    ");
         if ((i > 99) && (j > 0))
            fprintf(diana,"  ");
         else if ((i > 9) && (j > 0))
            fprintf(diana," ");
         fprintf(diana,"%s ",dnode[i].att[j].name);
         if (strcmp(tlistl,"") != 0)
            fprintf(diana,"%s",tlistl);
      }
      fprintf(diana,"]0);
   }
   fclose(diana);
}
#define MAXDNODES 100
#define MAXATT     10
#define MAXSYM     10
#define MAXIDLEN   25

struct template {
   char name[MAXIDLEN]; /* DIANA node name */
   struct {
      char name[MAXIDLEN];
      struct anode *list;
          } att[MAXATT];
         } ;
```

```
struct anode {
          struct anode *alist;
          short dnode;
       } ;

struct stacker {
   char blockname[MAXIDLEN]
   } ;

struct stemplate {
   char blockname[MAXIDLEN];
   short decval;  /* declaration value   */
   short parent;  /* points to parent    */
   short item_s;  /* items in block      */
   short stm_s;   /* statements in block */
   struct {
      char name[MAXIDLEN];
      struct {
         char name[MAXIDLEN];
         short val;
      } entryvar[MAXSYM]; /* array of entry variables */
   } entry[MAXSYM];
   struct {
      char name[MAXIDLEN];
   } symbol[MAXSYM]; /* array of regular variables */
   } ;

struct distack {
   char id[MAXIDLEN]
   } ;

%{
char templine[133], temperr1[133],
        temperr2[133], tempwarns[133];
char recentid[133];
int linenum, nerrs, nwarns;
int tnerrs, tnwarns;
%}

rword   [a-zA-Z]+
id      [a-zA-Z]([a-zA-Z0-9]*_?[a-zA-Z0-9]+)*
taskbody [tT][aA][sS][kK][ ][bB][oO][dD][yY]
%%

{rword} { addit(); return(rw_lookup()); }

{taskbody} {addit(); return(399); }

{id}    { addit(); strcpy(recentid,yytext); return(350); }

";"     { addit(); return(351); }

")"     { addit(); return(352); }
```

```
"("      { addit(); return(353); }

"=>"     { addit(); return(354); }

","      { addit(); return(355); }

":"      { addit(); return(356); }

":="     { addit(); return(357); }

"*"      { addit(); return(358); }

"/"      { addit(); return(359); }

"+"      { addit(); return(360); }

"-"      { addit(); return(361); }

[0-9]+   { addit(); return(362); }

"."      { addit(); return(363); }

"0       { printit(); }

[ ]+     { addit(); }


"--".*   { addit(); }

%%
static struct rwtable {
   char *rw_name;
   int tok_val;
   } rwtable[] = {
   "abort",     301,
   "accept",    302,
   "begin",     303,
   "body",      304,
   "delay",     305,
   "do",        306,
   "else",      307,
   "end",       308,
   "entry",     309,
   "in",        310,
   "integer",   311,
   "is",        312,
   "null",      313,
   "or",        314,
   "out",       315,
   "procedure", 316,
   "put",       317,
   "select",    318,
   "task",      319,
   "terminate", 320,
   "type",      321,
   "when",      322,
   };

static int rw_lookup(){

   int c;
   char temp[132];
   struct rwtable *low = rwtable,
      *high = (rwtable-1 +
        sizeof(rwtable)/sizeof(rwtable[0])),
      *mid;

   lcase(temp,yytext);
   while (low <= high){
      mid = low + (high-low)/2;
      if ((c = strcmp(mid->rw_name, temp)) == 0)
         return(mid->tok_val);
      else if (c < 0)
         low = mid + 1;
      else
         high = mid - 1;
   }
   strcpy(recentid,yytext);
   return(350);
}
lcase(s,t)                    /* copy tolower(t) to s */
   char s[], t[]; {

   int i;

   i = 0;
   while((s[i] = tolower(t[i])) != ' ')
      i++;
}
warning(){

      strcat(temperrl,"^");
      if (! nwarns)
         strcat(tempwarns,
          "Undecodable Character(s) Ignored");
      strcat(templine,yytext);
      nwarns++; tnwarns++;
}
addit(){

   int i;

   strcat(templine,yytext);
   for(i=0; i< (strlen(yytext)); i++)
      strcat(temperrl," ");
}
```

```
printit(){

   printf("%3d %s0,++linenum,templine);
   strcpy(templine,"");
   if(nerrs || nwarns)
      printf("    %s0,temperrl);
   if(nerrs){
      nerrs = 0;
      printf("    *** $ ERROR: %s0,temperr2); }
   if(nwarns){
      nwarns = 0;
      printf("    ** ^ WARNING: %s0,tempwarns); }
   strcpy(temperrl,"");
   strcpy(temperr2,"");
   strcpy(tempwarns,"");
}
%{
char recent2[133];
%}

%token ABORT            301
%token ACCEPT           302
%token BEGIN            303
%token BODY             304
%token DELAY            305
%token DO               306
%token ELSE             307
%token END              308
%token ENTRY            309
%token IN               310
%token INTEGER          311
%token IS               312
%token ADANULL          313
%token OR               314
%token OUT              315
%token PROCEDURE        316
%token PUT              317
%token SELECT           318
%token TASK             319
%token TERMINATE        320
%token TYPE             321
%token WHEN             322

%token TASKBODY         399

%token IDENTIFIER       350
%token SCOLON           351
%token RPAREN           352
%token LPAREN           353
%token POINTER          354
%token COMMA            355
%token COLON            356
%token COLEQU           357
```

```
%token MULTIPLY         358
%token DIVIDE           359
%token ADD              360
%token SUBTRACT         361
%token CONSTANT         362
%token PERIOD           363

%%

compilation : {insert("compilation");} compilation_unit ;

compilation_unit : library_unit ;

library_unit : subprogram_body ;

subprogram_body : subprogram_specification
                  IS {insert("procedure");}
                     declarative_part_or_not
                  BEGIN {chkblks();}
                     sequence_of_statements
                  END end_id_or_not SCOLON
                | error
                ;

subprogram_specification : PROCEDURE IDENTIFIER
   {spush(recentid,0); insert("proc_id");} ;

declarative_part_or_not : {insert("item_s");}
                          declarative_part
                        | ;

declarative_part : basic_declarative_item
                 | declarative_part basic_declarative_item
                 | later_declarative_item
                 | declarative_part later_declarative_item
                 ;

basic_declarative_item : {insert("var"); insert("id_s");}
                         identifier_list COLON
                         INTEGER
                         {insert("constrained");} SCOLON
                       ;

identifier_list : identifier_list COMMA IDENTIFIER
                 {st_insert(recentid); insert("var_id");}
                | IDENTIFIER
                 {st_insert(recentid); insert("var_id");}
                ;

later_declarative_item : task_specification
                       | task_body
                       ;
```

```
task_specification : {insert("task_decl");} TASK
                     type_or_not
                     IDENTIFIER {spush(recentid,1);
                     insert("TASK_var_id");}
                     ts_or_not SCOLON {spop();} ;

task_body : TASKBODY IDENTIFIER {spush(recentid,2);} IS
               declarative_part_or_not
            BEGIN {chkblks();}
               sequence_of_statements
            END end_id_or_not SCOLON {spop();} ;

type_or_not : TYPE
              | ;

ts_or_not : IS
               entry_declaration_seq
            END end_id_or_not
          | ;

end_id_or_not : IDENTIFIER {sendtest(recentid);}
              | ;

id_or_not : IDENTIFIER
          | ;

entry_declaration_seq : entry_declaration_seq
                        entry_declaration
                      | ;

entry_declaration : {insert("subprogram_decl");}
                    ENTRY IDENTIFIER
                    {addent(recentid);
                    insert("entry_id");}
                    formal_part_or_not SCOLON
                  ;

formal_part_or_not : formal_part
                   | ;

formal_part : LPAREN parameter_specification RPAREN ;

parameter_specification : entry_id_list COLON mode
                        INTEGER {insert("used_name_id");}
                        | parameter_specification SCOLON
                          entry_id_list COLON mode
                        INTEGER {insert("used_name_id");}
                        ;

mode : IN {insert("in");}
     | IN OUT {insert("in_out");}
     | OUT {insert("out");}
     | ;
```

```
entry_id_list : entry_id_list COMMA
     IDENTIFIER {st_insert(recentid); push(recentid);}
     | IDENTIFIER {st_insert(recentid); push(recentid);}
     ;

sequence_of_statements_or_not : sequence_of_statements
                              | ;

sequence_of_statements : sequence_of_statements statement
                       | statement
                       ;

statement : null_statement
          | assignment_statement
          | delay_statement
          | entry_call_statement
          | abort_statement
          | accept_statement
          | select_statement
          | put_statement
          ;

null_statement : ADANULL SCOLON
               ;

put_statement : PUT LPAREN IDENTIFIER
     {st_isit(recentid);} RPAREN SCOLON

assignment_statement : IDENTIFIER {st_isit(recentid);}
                       COLEQU expression SCOLON
                     ;

expression : expression ADD operand
           | expression SUBTRACT operand
           | factor
           ;

factor : operand
       | factor MULTIPLY operand
       | factor DIVIDE operand
       ;

operand : IDENTIFIER {st_isit(recentid);}
        | CONSTANT
        | LPAREN expression RPAREN
        ;

accept_statement : ACCEPT IDENTIFIER
                   ac_formal_part_or_not
                   ac_stuff_or_not
                   SCOLON
                 ;
```

```
ac_formal_part_or_not : ac_formal_part
                      | ;

ac_formal_part : LPAREN ac_parameter_specification
                    RPAREN ;

ac_parameter_specification : ac_entry_id_list COLON mode
                        INTEGER {insert("used_name_id");}
                      | ac_parameter_specification SCOLON
                        ac_entry_id_list COLON mode
                        INTEGER {insert("used_name_id");}
                      ;

ac_entry_id_list : ac_entry_id_list COMMA
         IDENTIFIER {st_isit(recentid); push(recentid);}
       | IDENTIFIER {st_isit(recentid); push(recentid);}
       ;

ac_stuff_or_not : DO sequence_of_statements END id_or_not
                | ;

abort_statement : ABORT task_abort_seq SCOLON
                ;

task_abort_seq : task_abort_seq COMMA IDENTIFIER
             | IDENTIFIER
             ;

delay_statement : DELAY expression SCOLON
                ;

entry_call_statement :
             entry_name actual_parameter_part_or_not
             SCOLON
             ;

entry_name : IDENTIFIER {strcpy(recent2,recentid);}
             PERIOD IDENTIFIER {chkent(recent2,recentid);}
           ;

actual_parameter_part_or_not : actual_parameter_part
                             | ;

actual_parameter_part : LPAREN expression RPAREN
                      ;

select_statement : selective_wait
                 | conditional_entry_call
                 | timed_entry_call
                 ;

conditional_entry_call : SELECT entry_call_statement
                            sequence_of_statements_or_not
```

```
                          ELSE
                             sequence_of_statements
                          END SELECT SCOLON

timed_entry_call : SELECT entry_call_statement
                      sequence_of_statements_or_not
                   OR
                      delay_statement
                   END SELECT SCOLON

selective_wait : SELECT
                    select_alternative
                 selmore_or_not
                 elsemore_or_not
                 END SELECT SCOLON
               ;

selmore_or_not : selmore_or_not OR select_alternative
               | ;

elsemore_or_not : ELSE sequence_of_statements
                ;

select_alternative : selective_wait_alternative
                   ;

selective_wait_alternative : accept_alternative
                           | delay_alternative
                           | terminate_alternative
                           ;

accept_alternative : accept_statement
        sequence_of_statements_or_not
                   ;

delay_alternative : delay_statement
      sequence_of_statements_or_not
                   ;

terminate_alternative : TERMINATE SCOLON
                      ;

%%
```

VITA 2

Monty Dale Bates

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION OF MINI-TASK: A LANGUAGE BASED ON ADA'S TASKING MODEL

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Ponca City, Oklahoma, June 20, 1963, the son of Kenneth and Carlene Bates.

Education: Graduated from Ponca City High School, Ponca City, Oklahoma, in May 1981. Attended Northern Oklahoma College from August, 1981 to December 1982. Received a Bachelor of Science degree in Computer Science from Oklahoma State University, July 1985. Completed the requirements for a Master of Science degree in Computer Science at Oklahoma State University, December 1987.

Professional Experience: Programmer, Geography Department/NASA, Oklahoma State University, September, 1985 to August 1986; Programmer, Horizon Insurance Agency, May 1986 to May 1987; Technical Writer, Department of Correspondence and Independent Study, Oklahoma State University, May 1984 to May 1986; Programmer, Seminole Tribe of Florida, February 1986 to October, 1986; Graduate Teaching Assistant, Department of Computer Science, Oklahoma State University, February, 1984 to May, 1987.