A
(Ada*) Compiler
User's Manual
Release 3.00

and

SuperSoft, Inc.

*Ada is a trademark of the Department of Defense (Ada Joint Program Office)

Disclaimer

SuperSoft, Inc. and Maranatha Software Systems make no representations or warranties with respect to the contents herein. While every precaution has been taken in the preparation of this manual, no responsibility is assumed for errors or omissions. Further, no liability is assumed for damages resulting from the use of this product. SuperSoft, Inc. and Maranatha Software Systems reserve the right to revise this publication and to make changes from time to time without obligation to notify any person of such revision or changes.

SOFTWARE NON-DISCLOSURE AGREEMENT

As with all SuperSoft software, acceptance of this or other products, both in machine and human readable form, implies agreement with the principles and concepts below.

1. All software is sold on an individual CPU basis. Usage on secondary machines without permission constitutes a violation of this agreement.
2. Five (5) backup copies may be made by the user. These are for the protection of user's investment only.
3. The ideas, concepts and machine/human interface of software are all considered the property of SuperSoft Inc. and its authors.
4. The user agrees to non-disclosure of the following:
   - Underlying concepts
   - Documentation
   - Code and code fragments (in both source and object)
   - User interface
   - Any and all aspects of software which SuperSoft developed.
5. All software is non-transferable and may not be re-sold without permission.
6. User modification of software completely removes SuperSoft Inc. from any liability regarding the operation or reliability of the software.

STATEMENT OF WARRANTY

SuperSoft disclaims all warranties with regard to the software contained on diskette, tape, or in printed form, including all warranties of merchantability and fitness; and any stated express warranties are in lieu of all obligations or liability on the part of SuperSoft for damages, including, but not limited to special, indirect or consequential damages arising out of or in connection with the use or performance of the software licensed.

Title and ownership shall at all times remain with SuperSoft and its authors.

Guide to the A Software Package

- What you get

In your Maranatha A package you should have recieved this Guide, a User's Manual, a Language Reference Manual, the government document MIL-STD-1815A, and 2 or more disks containing the software outlined in section 1.2 of the User's Manual.

- How to use this Package

You should first buy a good book on Ada; I recommend "Software Engineering with Ada" by Grady Booch (see bibliography, Appendix B of the User's Manual). While writing an Ada program, you will probably be referring to your book, the A Language Reference Manual, and (to a lesser extent) the MIL-STD-1815A. While compiling and linking programs, you only need to refer to the User's Manual.

- Optimization Hints

The following hints are provided for those who wish their programs to run as fast as possible, or take up as little space as possible. Note that, in some cases, these hints go against many good programming practices, and so they should be used with caution. They are listed here so users may take advantage of the particular way A has been implemented.

- Use the pragma RECURSION(off). This may have a disastrous effect when used with a recursive subprogram such as a factorial program, so be sure that none of the subprograms are recursive. It does provide excellent results in that variables can be accessed more efficiently.

- Use the pragma OPTIMIZE(time). This, in effect, forces the optimizer to to optimize the intermediate code twice. Sometimes this will have no effect, other times it will have only a minimal effect. The only trade-off is compile-time versus run-time, so for those programs you do not expect to have to re-compile often, use of this pragma can't hurt.

• Use address specifications for arrays. Because of the way arrays are implemented, the RECURSION pragma cannot place arrays in the local data area. Use an address specification to declare the start location of an array that is used often. Be sure it does not interfere with the program or system (CP/M) code!

• Use declared constants instead of attributes. For example, use declared constants instead of array'first..array'last to loop through an array. The attribute takes up run-time to evaluate, whereas the constant does not.

• Re-write TEXT_IO. The present TEXT_IO package contains more code than you will probably use. For example, a short program to print "hello" will link in the entire TEXT_IO package, including all file i/o operations (over 16K!). Write a new I/O package that contains only those procedures you will need for your program. Alternatively, the i/o operations from version 2 of the compiler are still embedded in the system (for compatibility). For such short programs, which only do minimal i/o (i.e., integer or string input/output), it is entirely possible to leave out the "with TEXT_IO" at the beginning of the program, since these inherent routines are contained in ADALIB. This is bad programming practice, of course, since it goes against the philosophy of Ada packages. These routines will be removed sometime in the future without notice, but may be used temporarily.

• Avoid numerous procedure calls. Because of the large amount of overhead in calling a subprogram, avoid them when possible. A must keep track of a large amount of data when a subprogram is called such as return addresses, exception data, dynamic and static links, return values, heap pointers, etc. Again, this is bad programming practice; so only use this technique when you are desperate for speed or space.

• The Prime Number Benchmark

The prime number benchmark program appears as a sample program, and is taken from the Byte prime number benchmark article(s). I've worked very hard to provide as much optimization as possible, and the result is a benchmark that ranks among the top languages available for CP/M systems. On a Z80 system running at 4 MHz, the benchmark clocks in at 17.5 seconds! Compare this to your favorite language; A does fairly well.

• The Maranatha Bulletin Board System

A computerized bulletin-board system has been set up dedicated to Maranatha A in Seattle, Washington. It is currently on-line 24 hours/day, 7 days a week (300 baud only). It is an RCPM/RBBS system with 2 600K floppies filled with Ada information. I hope to create a public domain database of Ada programs compatible with Maranatha A. Send in your favorite program and help me to make this service a success. Dial (206)939-6179.

– David Norris
– Author, Maranatha A

COPY THE MASTER DISK

As with all master disks the first thing you should do is make copies of the original Factory Master Disks you received from SuperSoft then use the copy not the original disk

You are allowed under the user agreement to make a maximum of 5 working copies for **your use** only. Put the factory disk away after making copies and only get it out again if something disasterous happens to your working disk

Make the copy on a new and freshly formatted disk If you have programs that check the integrity of the surface media, such as the Disk Doctor set of utilities offered by SuperSoft, use those before copying

Then format the new disk and copy the system tracks from your own master onto the new disk

CARE OF FLOPPY DISKS

Floppy disks are durable, long-lasting and are among the most reliable mass storage devices known They are something of a cross between a phonograph record and a cassette tape, with the speed of the former and the sensitivity and ease of use of the tape But they require careful handling Do **NOT**·

- —touch the magnetic surface
- —bend the disks, even though they are pliable
- —expose them to excessive heat or cold
- —allow them to come near magnetic substances or fields such as those generated by television sets, transmitters, or medium sized electric motors
- —leave them out of their protective sleeves for any longer than necessary
- —store them in a dirty or dusty environment, as motes of dust can permanently damage the disk surface

If you want to write on a disk cover, use a soft-tip pen, it's a much better idea to write on the self-stick label that comes with new disks before putting the label on the disk

"BACKING-UP" DATA DISKS

If this program uses continuing data disks which are regularly updated, as is the practice in most business and financial programs, it is a prudent and useful process to make back-up copies of those disks.

A number of completely uncontrollable events can cause irreparable damage to a disk, such as a line surge while the disk drives are active, a power outage during certain internal memory manipulations, a bad sector on disk media, accidental disconnect of the machine, (or some helpful soul turning it off because it didn't appear to be doing anything) and similar occurences that have nothing to do with the integrity of your machine or the quality of the software you are using.

In all of the above cases, the data probably will be lost forever and the time and expense of reconstructing it may be almost prohibitive.

Making regular back-up copies of your updated files takes no longer than photocopying an important document. Get in the habit of making back-up copies of your working disks regularly, at the end of each computer session, or even every half-hour or so during a long session where much data is being entered.

It is also a good idea to produce hard-copy (paper) printouts of key data files as another form of back-up.

Preface

Thank you in advance for purchasing this our third major release of the Maranatha A compiler. Additions to this version include:

- Packages
- Subunits
- Linking Loader
- Separate Run-time Library
- Relocatable Run-time modules
- Exceptions
- Shorter Run-time programs
- Faster, more optimized programs
- Upgraded to ANSI Standard Ada syntax (MIL-STD-1815A)
- Membership Tests
- Short-circuit conditionals
- Reformatter

Table of Contents **Page**

1.0 Introduction

The User's Manual describes the Maranatha Software Systems implementation of the Ada language for microcomputers in the CP/M operating system environment; i.e., how to compile, load and run an Ada program. While this manual does describe what standard Ada constructs are not yet implemented in this release, it is not intended to serve as a tutorial to the Ada programming language. Users are referred to the bibliography section for tutorial and historic information on the subject. The Language Reference Manual describes specifically what has been implemented and gives some examples.

1.1 Forward

In 1975, the Department of Defense, plagued with ballooning software development costs began a program to create a new language for all embedded military software systems. Many other languages were evaluated and none were found that could fill all of the DOD's rigid requirements; included in the languages tested were Pascal, Algol, PL/I, Jovial, Fortran and Cobol. The new language was named Ada in honor of Augusta Ada Byron, the first computer programmer.

Ada has many problems despite its backing by the U.S. Government and years of language development behind it. Ada is one of the largest languages (in terms of syntax) of any language yet developed; its sheer size makes it a language difficult to implement, especially on a microcomputer. Some say Ada is too loose and inefficient for the purpose for which it was created. The input/output facilities are extremely awkward for formatted I/O. Even so, its strong support from the DoD, its accomplishment of a recognized standard and the fact that it is a well-rounded language for use in systems programming and structured language instruction will make Ada the language of the 1980's.

Maranatha Software Systems is dedicated to making our implementation one of the most complete and usable Ada language systems for CP/M computers. This, the third major release, contains most of the constructs which make A a usable language system and stand out from other microcomputer languages such as Pascal and Basic.

1.2 Contents Of The Distribution Disk

On the distribution disk you will find the following compiler tools:

A.COM	- A lexical analyzer (scanner)
A2.COM	- A syntactical/semantical analyzer (parser)
A3.COM	- A optimizer
A4.COM	- A 8080/8[illegible] code generator
L.COM	- A Linking Loader
PUTREL.COM	- Relocatable file printer
REFORMAT.COM	- the Ada source text reformatter

The following compiler programs are also required:

ADALIB.REL	- The A Standard Library
ERRORS.TXT	- text file containing A error messages
INIT.REL	- The initial start-up module

The following libraries and demonstration programs have been included in source form, and need to be compiled before use:

ASCII.ADA	- limited package with ASCII constants
CALC.ADA	- RPN calculator program
FACT.ADA	- function to compute n!
IOEXCEPT.ADA	- The package IO_EXCEPTIONS
MATHBODY.ADA	- Ada math library package body
MATHLIB.ADA	- Ada library of math functions (used by CALC.ADA)
PRIMES.ADA	- Ada benchmark program
QUEENS.ADA	- solves "Eight Queens" problem
SHELL.ADA	- badly formatted program (demo for Reformatter)
SORT.ADA	- demonstration of three sorting techniques in Ada
STRINGS.ADA	- demonstrates Ada string manipulation capability
TERMIO.ADA	- Terminal i/o driver package specification
TERMBODY.ADA	- Terminal i/o driver package body (Televideo 950)
TEXTBODY.ADA	- The package body TEXT_IO
TEXTIO.ADA	- The package specification TEXT_IO
TOUPPER.ADA	- program that prints any text file in upper case
TOWERS.ADA	- solves "Towers of Hanoi" problem

2.0 Compiler Operation

2.1 Compile-Time System Requirements

The A compiler requires an 8080, 8085 or Z80 CPU running the CP/M operating system. The four main compiler files (A.COM, A2.COM, A3.COM, and A4.COM) should all be resident on the current disk, although source files may exist on other disks. The A error file "ERRORS.TXT" should be with the main compiler files but is optional. If it is not on the same disk as the main compiler, only the error number will be printed, along with the offending line number (check Appendix A for the error numbers and corresponding explanations). The compiler itself requires 50K of transient program area. A 64K system is recommended, although small programs can be compiled in as little as a 56K system.

For systems with extremely limited disk space, the four compiler passes can be executed separately. Each pass will output an error when it is finished if the next pass is not on the disk, but will leave the intermediate file (.TOK, .INT, or .OPT) intact. Run the next pass with the intermediate file as an argument, i.e.:

```
A>A2 TEST.TOK
```

2.2 Run-Time System Requirements

The code generator indicates the number of bytes of object code it has generated. The total program contains code generated by the main subprogram, packages and subunits, plus various routines as needed from the A library ADALIB. Note that the amount of "white space" (comments, blank lines, etc.) in the source code does not affect the amount of actual code produced, since Ada comments are filtered during parsing and produce no run-time code. Variable storage is allocated within the stack which is automatically set at the top of the transient program area (TPA) and is not included in the above figure.

2.3 Compiler Invocation

To execute the A compiler type:

A <filename>[.ADA]

Examples:

```
B>A CALC

A>A B:TOWERS.ADA

A>B:A TEST        ; This is invalid, the compiler will abort after the first pass!
```

Note that the file extension ".ADA" is automatically appended unless another file extension has been provided by the user. The only compiler toggles presently available are through pragmas and no compiler switches are available in the command line.

2.3.1 Lexical Analyzer (A.COM)

The first pass lexically analyzes the source text and divides it into lexical chunks called tokens (see Figure 1) such as the Ada keywords, identifiers, special symbols, etc. The output is placed into a ".TOK" file. The lexical analyzer is responsible for processing all pragmas. This pass has its own set of error messages for lexical errors; they are described in Appendix A.

2.3.2 Parser (A2.COM)

The stream of tokens in the ".TOK" file is then processed by the parser. The parser is the workhorse of the compiler, checking for syntax and semantic errors. It produces a ".INT" intermediate code file. The errors which may be produced by the parser are listed in Appendix A. The parser may have to read symbol tables generated by other program units; these files have CP/M extensions of ".SYM."

2.3.3 Optimizer (A3.COM)

The intermediate code, which resembles the *Forth* language, is processed by the optimizer. The optimizer searches through the code generated by the parser and attempts to reduce redundant operations, eliminate unneeded code, and increase the speed and efficiency of the resulting program. Normally, once the compiler has reached this pass, they code is considered error-free, and no error messages should occur. However, both the optimizer and the code generator contain code that checks the internal operation of the compiler to insure there are no internal errors. If a message such as "Internal compiler error" is printed, contact us. Note that the optimizer will not be executed if the *optimize* pragma has been turned off.

2.3.4 Code Generator (A4.COM)

The file produced by the optimizer (".OPT") is processed by the code generator to produce a relocatable object ".REL" file. The code generator converts the optimized intermediate code into machine code, attempting to make the best use of the 8080/8085/Z80 register set. The code generator may print a table overflow error if the program is too large, in which case it will have to be broken down into smaller chunks. The first pass of the code generator resolves all internal references and completes internal tables. The second pass actually outputs the object code and requires more time than any of the other compiler phases.

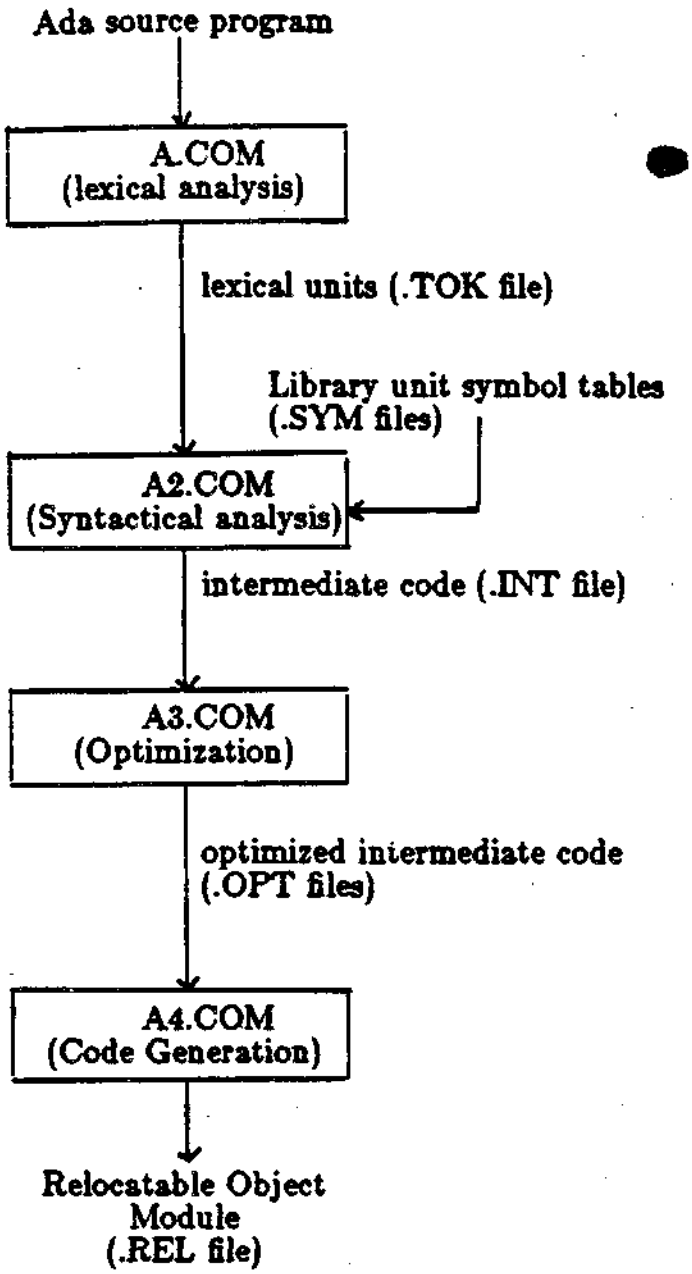


Figure 1.
A Compiler Operation

3.0 Linking Loader Operation

The Maranatha A compiler generates .REL files which are not executable. To create an executable file, you need to invoke the Linking Loader and link the resulting ".REL" file with a number of other ".REL" files *in a particular order* to produce an executable ".COM" object file.

3.1 Invoking the Loader

To invoke the loader, type "L" followed by a carriage return. The linking loader will print a sign-on message, then the prompt "*". Commands to the loader consist of strings of filenames and switches separated by commas; i.e.:

*filename/switch,filename,filename/switch,/switch.....etc.

When a filename is used, that file is loaded from the disk into memory (a default extension of .REL is assumed). The file can optionally be searched (as in ADALIB) by using the /S switch.

3.2 Loader Switches

There are several switches available to specify actions which affect the loading process:

/E - Exit the Linker and return to CP/M. If the /N switch has been used, the .COM file will be written to the disk before exit.

/M - List global data map of defined and undefined globals.

/N - If <filename>/N is entered, the program will be saved on the disk with the specified name (with a default extension of .COM if no extension has been provided) when the Linker is exited.

/R - Reset the Linker to its initial state.

/S - If <filename>/S is entered, the file specified by <filename> will be searched, loading only those modules which are currently undefined.

/U - List undefined globals, origin and end of the program.

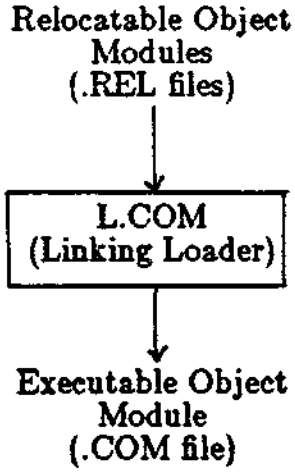


Figure 2.
Linking Loader Operation

3.3 Order of Elaboration

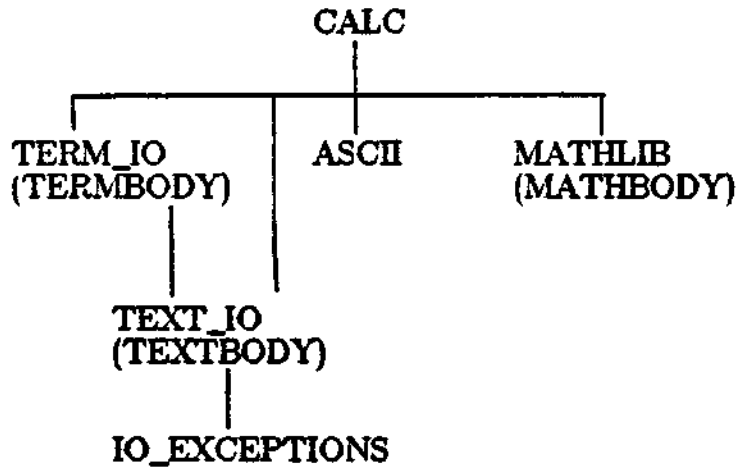
The order in which A relocatable modules are loaded is all-important. Packages must be elaborated before they are used or undesirable side-effects will occur. For the simplest program, which is not dependent on any packages, the following link sequence can be used:

```
*INIT,<program>,ADALIB/S

*<program>/N/E
```

INIT must always be the first module loaded. It is the kernel Ada program which sets up the stack and initializes the environment for all Ada programs. ADALIB must always be the last link item loaded (or searched). It contains all of the run-time math routines, relational subroutines, stack and heap manipulation routines, etc. Note that all ADALIB modules begin with a dollar sign; i.e., $IML (integer multiply).

In a more complicated example, let's compile and link the demo program CALC. Since CALC requires four packages (TEXT_IO, TERM_IO, ASCII, and MATH_LIB), they must be compiled first. The following tree shows the dependency between the units; the names in parentheses indicate the CP/M file name corresponding to the package body of a particular package:



This order requires us to compile IO_EXCEPTIONS first, then TEXT_IO, and then TERM_IO, ASCII, and MATH_LIB in any order (the package bodies can be compiled after all of the specifications have been compiled). CALC is compiled last. To link the entire program together, the following order should be used:

```
*INIT
*IOEXCEPT
*TEXTIO,TEXTBODY
*TERMIO,TERMBODY,MATHLIB,MATHBODY
*CALC
*ADALIB/S
*CALC/N/E
```

Why this order? INIT must *always* be loaded first. TEXT_IO cannot be loaded next because it is dependent on the package IO_EXCEPTIONS. Since IO_EXCEPTIONS does not depend on another package, it can be loaded. Packages that do not depend on other packages can be loaded in any order. Since the package TEXT_IO has two parts, the specification and the body, the specification TEXTIO must be loaded first, because the body depends on it. Next the package body TEXTBODY can be loaded. Next, the package specifications TERM_IO, ASCII, and MATH_LIB can be loaded in any order, followed by their respective package bodies (ASCII has no package body). Since all of the required packages have been loaded, the main subprogram CALC can be loaded. As of this release, a subprogram that is a library unit is automatically considered the main program. Lastly, the A library (ADALIB) is searched for undefined externals and the .COM file created. Observe carefully the undefined symbols during loading to insure the correct loading sequence and that all required packages have been loaded. The only undefined globals that should appear are ADALIB modules (begin with a dollar sign), and labels belonging to as yet unlinked subunits. If an undefined global appears that belongs to a package, the loading order is incorrect and the entire linking process must be done over. Such undefined globals always begin with the name of the package.

In this particular case, the package specification ASCII does not actually have to be loaded, because it contains extraneous code. Package specifications which contain only subprogram declarations or constant declarations (where the initialization expression is a single primary) do not need to be loaded.

Subunits may be loaded after everything else has been loaded, but before ADALIB has been searched. If a subunit has been declared, but never used, the Linker does not require that it be loaded.

4.0 Reformatter

The reformatter is a language design tool that is similar to other "pretty-printer" utilities. It takes as input any Ada program, and produces as output the same program re-justified, following the informal indentation rules presented in the MIL-STD-1815A:

1) Reserved words appear in lower case

2) Identifiers appear in upper case

3) Indentation follows examples presented in Ada LRM

The reformatter can be an invaluable tool to point out hard-to-find syntax errors and make plain embedded "nesting" mistakes which may cause the compiler to print an erroneous error message in the wrong place. The Ada program does not have to be syntactically correct, although an incorrect program may cause some unexpected results.

4.1 Invocation

To invoke the reformatter, type:

REFORMAT source_file_name[.ADA] [$options]

Examples:

A>reformat sqrt.ada

B>reformat c:towers $t2

The default file extension is ".ADA" and need not be included. The last example demonstrates the use of one of the reformatter options.

4.2 Options

To use a reformatter option, use the dollar ($) sign following the reformatter invocation, followed by any of the following:

Tn - Define n as tab length (default = 4)
For indentation purposes (i.e, between begin-end), this option defines the number of spaces to be used for tabs.

N - Print nest level
Instructs reformatter to print nest level at the beginning of each line of source text. A valid Ada program will have a nest level of "1" for the last line of text.

4.3 Examples of Reformatting

The following valid Ada program appears on the distribution disk as SHELL.ADA (not meant to be directly compiled). This program is an actual portion of Ada source code, and is "included" in the demonstration program SORT.ADA with the INCLUDE pragma:

```
A>type shell.ada

procedure shell_sort(a : in out sort_array) is done : boolean; jump : integer;
begin jump := a'length; while jump > a'first loop jump := jump/2; loop
done := true; for j in a'first..a'length-jump loop declare
i : integer := j+jump; begin if a(j)>a(i) then swap(a(j),a(i)); done := false;
end if; end; end loop; exit when done; end loop; end loop; end shell_sort;
```

Now, if you are prone to write programs in this way, you probably deserve what you get. At any rate, it would obviously be difficult to find the "mis-matched end" types of errors in a program that is so badly formatted. The Reformatter can make plain the begin-end relationships:

```
A>reformat shell.ada

        procedure SHELL_SORT(A : in out SORT_ARRAY) is
           DONE : BOOLEAN;
           JUMP : INTEGER;
        begin
           JUMP := A'LENGTH;
           while JUMP>A'FIRST loop
              JUMP := JUMP/2;
              loop
                 DONE := TRUE;
                 for J in A'FIRST..A'LENGTH-JUMP loop
                    declare
                       I : INTEGER := J+JUMP;
                    begin
                       if A(J)>A(I) then
                          SWAP(A(J), A(I));
                          DONE := FALSE;
                       end if;
                    end;
                 end loop;
                 exit when DONE;
              end loop;
           end loop;
        end SHELL_SORT;
```

The output of the Reformatter clearly shows the logical flow of the program, and the "end" constructs are easily matched up with their identically indented counterparts. Errors in control flow will become much more evident.

Appendix A
Error Messages

Lexical analyzer error messages:

Invalid digit -
A digit '0'..'9' was expected in a numeric literal.

Letter or digit expected following '_' -
Identifiers cannot end with an underscore.

Invalid character -
A character was read that does not conform to the basic graphic character set. A likely cause of this error is text produced by a word processing program that sets the 8th bit of a character.

Digit expected following underscore -
Numbers cannot end with an underscore.

Single quote expected following character literal -
A character literal must be followed by a quote.

Pragma identifier expected -
Self-explanatory.

Include file name expected -
A file name is required for the include pragma.

Left parenthesis expected -
Self-explanatory.

Unable to open include file -
Self-explanatory.

Identifier expected -
Self-explanatory.

Illegal pragma argument -
Self-explanatory.

Right parenthesis expected -
Self-explanatory.

Semicolon expected -
Self-explanatory.

The syntax/semantic errors are contained in the file "ERRORS.TXT." If this file is not present on the same disk as the compiler, only the offending line number and error number will be printed.

2. Intermediate file not found -

This error should not normally appear; it means the compiler was unable to find the output of the previous pass. If you invoked the parser directly with a tokenized file (".TOK"), insure the name was correct or that the file actually existed.

3. Discrete type_mark expected for discrete_range -

When the form "type_mark [range_constraint]" is used for a discrete_range, the type_mark must be discrete; that is, an enumeration or integer type.

4. Label already exists -

You have tried to re-define a label which already exists in the current sequence of statements.

5. ">>" Expected -

Self-explanatory.

6. Left parenthesis expected -

Self-explanatory.

7. Record must have at least one component -

Self-explanatory.

8. Right parenthesis expected -

Self-explanatory.

9. Semicolon expected -

Self-explanatory.

10. Subtype_indication expected -

Self-explanatory.

11. Package not visible -

For a use_clause, the given package not has not been used in a with_clause or does not appear in a visible declarative part.

12. Procedure or function expected -

A subprogram must begin with one of these keywords. You probably have either misspelled the keyword, or have submitted a non-Ada program for compilation.

13. "Is" expected -

Self-explanatory.

14. "End" expected -

Self-explanatory.

15. Designator or semicolon expected -

Following the final "end" of a subprogram, the subprogram designator or a terminating semicolon should be used.

16. No return statement found within subprogram body -

Every function must have at least one return statement.

17. Mis-match of begin and end designators -

The subprogram designator used after the terminating "end" does not match the initial designator. If you are sure it does, insure there are the correct number of "ends" within the program. The Reformatter may help.

18. Ellipsis (..) expected -

Self-explanatory.

19. Constraint error -

A range_constraint can only be used on an enumerated or integer type.

20. Not enough index constraints -

When constraining an array type, all of the indices must be constrained. Insure that an index_constraint exists for each index.

21. Index type mismatch -

The type of the index_constraint did not match the type of the corresponding index of the array type you are constraining.

22. Illegal index constraint for this type -

You cannot constrain an array type that has already been constrained.

23. Too many index constraints (5 maximum) -

Maranatha A allows for a maximum of 5 indices.

24. Comma or colon expected -

Self-explanatory.

25. Too many identifiers in identifier list -

You have tried to do too much at once by using too many identifiers in the same declaration. Break the declaration into two or more separate declarations.

26. Identifier expected -

Self-explanatory.

27. Undeclared type -

You have tried to use a type_mark which has not been declared.

28. Expression type mismatch -

The type of a processed expression did not match the required type.

29. Two different logical operators used within expression -

You may not mix logical operators. Use parentheses to resolve ambiguities.

30. Boolean relation expected -

Logical operators can only work on boolean operands.

31. Simple expression type mismatch in relation -

Simple expressions used as operands in a relation must have the same type.

32. Invalid relational operator, use "/=" instead -

You are probably a Pascal programmer. "<>" is the Pascal symbol for inequality, Ada uses "/=". The "<>" is called a "box" in Ada and has different uses.

33. Invalid relational operator for this type -

The relational operator is not defined for the operand type used.

34. Invalid unary operator for this type -

The unary operator is not defined for the operand type used.

35. Boolean term expected following unary NOT -

Following the unary NOT operator, the type of the term must be boolean.

36. Invalid adding operator for this type -

The adding operator is not defined for the operand type used.

37. Invalid operands for concatenation -

Self-explanatory.

38. Invalid multiplying operator for this type -

The multiplying operator is not defined for the operand type used.

39. Integer factors expected for MOD and REM -

MOD and REM are defined only for integer factors.

40. Invalid type for exponentiation -

The "**" operator can only be used on integer and floating point numbers.

41. Invalid primary -

A primary must be an integer, real, character, or string constant, begin with a left parenthesis, or start with a non-keyword identifier. Otherwise it is invalid.

42. Integer constant illegally used -

The integer constant does not match the existing expression type.

43. Floating point constant illegally used -

The floating point constant does not match the existing expression type.

44. String constant illegally used -

The string constant does not match the existing expression type.

45. Character constant illegally used -

The character constant does not match the existing expression type.

46. Name type mismatch -

A name processed in a primary must be a function, variable or constant.

47. Invalid identifier used in name processing -

The first identifier in a name must be a variable, constant, type_mark, or function name.

48. Left parenthesis or single quote expected following type name -

Following a type_mark used in a name, a left parenthesis (indicating type_conversion) or a single quote (indicating qualified expression) must be used.

49. Left parenthesis or identifier expected -

Self-explanatory.

50. Enumerated literal type mismatch -

The enumerated literal used did not match the existing expression type.

51. Undeclared identifier -

Self-explanatory.

52. Invalid type conversion -

Conversion to the type_mark given in the type conversion is not allowed or is not defined.

53. Function not found -

A function used in a primary was not declared. Check the designator spelling, the number and types of parameters.

54. Comma or right parenthesis expected -

Self-explanatory.

55. Illegal attribute -

The attribute identifier used is not implemented or is not defined. See Appendix A for a list of implemented attributes and how to use them.

56. Invalid attribute -

The attribute is not defined for the object it is used against. See Appendix A.

57. Invalid left parenthesis -

Self-explanatory.

58. Too many indices -

You have used too many indices in an indexed component. Check the declaration of the array.

59. Comma or left parenthesis expected -

Self-explanatory.

60. Comma expected, not enough indices -

You have not indexed the array sufficiently. Check the array declaration.

61. Invalid selected component -

You cannot select the component of the record given. Either the object is not a record or the selected component was not declared in the record type declaration.

62. Designator expected -

Self-explanatory.

63. Invalid binding modes for function parameters -

Only "IN" mode parameters are allowed for functions.

64. Subtype indication expected in function specification -

A return clause is expected to declare the type of a function.

65. Invalid subtype indication found in procedure specification -

Procedures do not return values and do not have return clauses.

66. Right parenthesis or semicolon expected -

Self-explanatory.

67. Bad order in declarative part -

In the declarative_part, declarative_items precede representation_specifications, which are then followed by program components. You have mixed the order of the declarative_part.

68. Invalid declarative item -

A declarative item begins with the reserved word "TYPE" or "SUBTYPE", or a non-keyword identifier which begins an object_declaration.

69. Semicolon or ":=" expected -

Self-explanatory.

70. Identifier expected -

Self-explanatory.

71. Invalid type definition -

The type_definition used has not yet been implemented or is not defined.

72. Identifier expected for enumeration literal -

Self-explanatory.

73. Index expected -

In an array_type_definition, an index cannot be followed by a discrete_range. The array indices must all be unconstrained.

74. Discrete range expected -

In an array_type_definition, when an index_constraint is used, an index cannot be substituted for a discrete_range. All of the indices must be constrained.

75. Illegal goto -

The goto statement illegally tried to goto this label. See section 5.9 for rules regarding goto statements.

76. "Of" expected -

Self-explanatory.

77. "Record" expected -

Self-explanatory.

78. Recursive record type definition not allowed -

A component of a record_type_definition cannot be the record type itself.

79. Statement expected -

In every sequence_of_statements, at least one statement must appear. If no action is to be performed, use the null statement.

80. Syntax error -

You have used an unrecognizable token to begin a statement.

81. Illegal function call as statement -

Only procedure calls may be used as statements; functions return values and are used in expressions.

82. Block or loop expected following block/loop identifier -

A block/loop identifier was used (identifier:) without a block or loop statement.

83. Invalid assignment to constant -

Assignments to constants are not allowed. Loop identifiers are considered constants, as are "IN" mode parameters.

84. Assignment operator expected -

Self-explanatory.

85. "Then" expected -

Self-explanatory.

86. "End if" expected -

Self-explanatory.

87. "End loop" expected -

Self-explanatory.

88. Mis-match of begin and end loop identifier -

The loop identifier given following the terminating end does not match the initial label.

89. Loop identifier or semicolon expected -

Self-explanatory.

90. Loop identifier not declared for this loop statement -

A loop identifier was given following the terminating end, but none was provided at the start of the loop.

91. Loop identifier expected -

The loop identifier given at the start of the loop statement must appear following the terminating end.

92. "Loop" expected -

Self-explanatory.

93. "In" expected -

Self-explanatory.

94. "Begin" expected -

Self-explanatory.

95. Block name expected -

The block identifier given at the start of the block must appear following the terminating end.

96. Mis-match of begin and end block identifier -

The block identifier given following the terminating end does not match the initial label.

97. Loop identifier, "when", or semicolon expected -

The token following the keyword "exit" is invalid.

98. "When" or semicolon expected -

The token following "exit loop_identifier" is invalid.

99. No enclosing loop for exit statement -

An exit statement was used without an enclosing loop statement.

100. Loop identifier not found -

The loop identifier given in the exit statement was not found.

101. Labels referenced but not declared -

In the sequence of statements of the preceding subprogram body, a goto statement referenced a label that was not declared (or not visible).

102. Return expression expected for function -

An expression must follow the return statement used within a function which has the type of the function.

103. Return expression not expected for procedure -

An expression was illegally used for a return statement used within a procedure, which does not return a value.

104. Discrete type expected for case expression -

The type of the expression in a case statement must be discrete; that is, enumerated or integer.

105. "=>" or "|" expected -

Self-explanatory.

106. "End case" expected -

Self-explanatory.

107. Invalid "when" found after choice "others" used -

You may not use any more choices in a case statement after the choice others has been used; it must be the last choice in the case statement construct.

108. Procedure not found -

Self-explanatory.

109. Semicolon or left parenthesis expected -

Self-explanatory.

110. GET undefined for this type -

Self-explanatory.

111. PUT undefined for this type -

Self-explanatory.

112. READ undefined for this type -

Self-explanatory.

113. Comma expected -

Self-explanatory.

114. WRITE undefined for this type -

Self-explanatory.

115. ABS undefined for this type -

Self-explanatory.

116. Unexpected end of file -

The compiler unexpectedly ran out of source text. Insure each procedure, function, and compound statement have a proper terminating end. The Reformatter may help to find such errors.

117. Fatal error - compilation aborted -

The compiler is unable to continue at this point. Correct the offending error(s) and re-compile.

118. Incomplete type definition for declaration -

An unconstrained array cannot be used in an object declaration; it must be constrained first.

119. Illegal actual parameter modes in procedure call -

The modes of the actual parameters did not match the binding modes of the formal parameters.

120. Symbol table overflow -

You have overflowed the symbol table. You need to get more memory, reduce the number of declarations in your program, or go buy a CRAY-1.

121. A3.COM (Optimizer) not found -

The optimizer was not found on the current disk. If it was accidently erased, you can continue compilation by using a backup copy and typing "ADA3 progname.INT."

122. Illegal prior reference to this label -

A goto statement used earlier in the program used this label illegally. See section 5.9 for rules regarding goto's.

123. "Use" expected -

Self-explanatory.

124. "AT" expected -

Self-explanatory.

125. Integer literal expected -

Self-explanatory.

127. "OTHERS" must be its only choice -

In case statements or in an exception handler, other choices may not be used in conjunction with OTHERS. It must be used alone.

128. Exception name expected -

Self-explanatory.

129. Package body is not a basic declarative item -

A package body may not appear in a package specification.

130. Invalid address specification -

Address specifications can only be used with variables and constants.

131. Subunit not found -

Self-explanatory.

132. Subprogram body not found for subprogram declaration(s) -

Subprogram body or bodies were not found for corresponding subprogram declarations in this declarative part (or package specification if in a package body).

133. Package specification required before package body -

Package specifications must be compiled before package bodies.

134. Undefined package body or bodies in declarative part -

One or more package specifications exist which require package bodies to fulfill subprogram declarations. The package bodies were not found.

135. Package bodies may not be separate. -

Self-explanatory.

136. Symbol table for ancestor unit not found -

The library unit of a subunit must be compiled before the subunit so its symbol table file may be accessed. Either the unit has not been compiled, or the symbol table file has been misplaced.

Appendix B
Selected Bibliography

The following is a list of selected textbooks and articles on Ada. While this list is for end-user information only and appearance in the list does not necessarily constitute endorsement, we strongly recommend Booch's book, "Software Engineering with Ada."

Barnes, J.G.P.
Programming in Ada
Addison-Wesley, 1982
(ISBN 0-201-13793-3, hardback)
(ISBN 0-201-13793-5, paperback)

Booch, Grady
Software Engineering with Ada
Benjamin/Cummings, 1983
(ISBN 0-8053-0600-5, paperback)

"Computer" magazine
June 1981
(Issue devoted to Ada)

Downes, Valerie A. and Goldsack, Stephen
Programming Embedded Systems with Ada
(both of the University of London)
400 pp., illustrated (April 1982)
Price: appr. $16.95

Freedman, Roy S.
Programming Concepts with the Ada Language
Petrocelli Books, Inc.
1101 State Road, Princeton, NJ 08540
Price: $12.00
(ISBN 089433-190-6)

Gehani, Narain H.
Ada: An Advanced Introduction
Prentice-Hall, 1983
Price: $18.95

Habermann, A. N., and Perry, DeWayne E.
Ada for Experienced Programmers
Addison-Wesley, 1983
(ISBN 0-201-11481-X, paperback)

Hibbard, P.; Hisgen, A.; Rosenberg, J.; Shaw, M. and Sherman, M.
Studies in Ada Style
Springer-Verlag, 1981
$11.20
(ISBN 0-387-90628-2)

Katzan, Harry Jr.
Invitation to Ada & Ada Reference Manual (July 1980)
Petrocelli Books, New York, 1982
(ISBN 089433-132-9)

Ledgar, Henry
Ada: An Introduction
Springer-Verlag
$12.95

Lewis, William E.
Problem Solving Principles for Ada Programmers: Applied Logic, Psychology, and Grit
Mail Request:
Dept. #CD 82
Hayden Book Company, Inc.
50 Essex St. Rochelle Park, NJ 07662
(Ada version, request #5211)
Price: $9.95
(Toll free: 1-800-631-0856)

Mayoh, Brian
Problem Solving with Ada
John Wiley & Sons, Ltd., 1982
(ISBN 0-471-10025-0)

Pyle, Ian C.
The Ada Programming Language
Prentice-Hall International, 1981
(ISBN 0-13-003921-7)

Stratford-Collins, Michael J.
Ada A Programmer's Conversion Course
John Wiley & Sons, New York, 1982
(ISBN 0-85312-250-4)

Wegner, Peter
Programming with Ada: An Introduction by Means of Graduated Examples
Prentice-Hall, January 1980
(ISBN 0-13-73-0697-0)

Wiener, Richard and Sincovec, Richard
Programming in Ada
John Wiley & Sons, New York, 1983
(ISBN 0-471-87089-7, hardback)

Young, S.J.
An Introduction to Ada
Ellis Horwood, Chichester, 1983
(ISBN 0-85312-535-X, paperback)

Appendix C
Unimplemented ANSI Standard Ada

Currently, Maranatha A fully implements 63% of the MIL-STD-1815A language, partially implementing another 1%, more than any other Ada compiler available for CP/M based microcomputers. The unimplemented syntax is broken down as follows:

10% - Generics

9% - Tasking

3% - Representation Specifications

3% - Variant Records and Discriminants

3% - Floating/fixed point implementation

2% - Access types

The remaining catagories each total 1% or less and include renaming declarations, aggregates, operator overloading, deferred constant declarations, private type declarations and slices.

These figures are based on the number of the Backus-Naur productions of the language that have been implemented. There are 342 such productions in standard Ada; Maranatha A fully implements 215 and partially implements 4. In comparison, Unix (tm) V7 "C" has 166 total productions, most standard Pascal implementations have about 100, and Modula-2 has 82. Appendix E of the Language Reference Manual mirrors Appendix E of the MIL-STD-1815A; compare the two for yourself.

Appendix D
Format of Relocatable (.REL) files

The relocatable modules produced by the Maranatha A compiler are similar but not identical to the Microsoft relocatable format. Differences between the two are:

1) The Maranatha Linker cannot accept all of the link items introduced in the full relocatable format, including:

item 1 - Select COMMON block
item 3 - Request library search
item 4 - Extension link items
item 5 - Define COMMON size
item 8 - External - offset
item 9 - External + offset
item 12 - Chain address

2) For link items requiring a "B-field", names with 1..7 characters are as the standard; however, since very long names may be required by the nature of Ada programs, long identifiers are used by declaring the "zzz" field to be zero (0), followed by the ASCII characters of the symbol and terminated by 16#FF#.

In spite of these differences, the format is very close. In fact, if your Ada program has a short program name and does not depend on any packages with long names, it is entirely possible to use the Microsoft or Digital Research linking loaders. (See note in Appendix E about using assembly language routines)

Appendix E
Writing Assembly Language Programs

Using assembly language programs with Maranatha A can best be demonstrated by example. Given the following program:

```
procedure t is
     y : integer;
     function inp(port : integer) return integer is separate;
begin
    loop
            get(y);
            put(inp(y));
     end loop;
end t;
```

If "inp" is to be an assembly language program which returns the value of the input port specified by the parameter "port", a suitable assembly language program would appear as follows.

Unfortunately, this method requires that the program name and function name have a combined length of only 4 or 5 characters. This is because standard CP/M assemblers only allow 6 or 7 characters for labels, and the A compiler adds two characters to create special labels for these routines. This will be remedied in a future release, probably by implementing the code statement.

```
;
; function inp(port : integer) return integer;
; begin
;    return in port;
; end inp;
;
; ROM-able code to fetch value from input port.
;
public t.inp,t.inp?,t.inp$

extrn  $rtp,$base
;
t.inp:                          ; start of routine
       lhld    $base
       lxi     d,-6
       dad     d
       mov     e,m              ;
       inx     h                ; fetch pointer to port number
       mov     d,m              ;
       xchg
        mov    h,m              ; fetch port number
;
; construct "mini-routine" on stack:  IN <port>; RET
;
       lxi     d,00C9H
       push    d                ; construct RET; NOP
       mvi     l,0DBH
       push    h                ; construct IN <port>;
       lxi     h,0
       dad     sp               ; compute calling address of mini-routine
       lxi     d,inp1
       push    d                ; create return address from mini-routine
       pchl                     ; "call" mini-routine
;
inp1:  lhld    $base
       lxi     d,4
       dad     d                ; hl -> return value
       mov     m,a
       inx     h                ; stuff return value
       mvi     m,0              ; into function return value location
;
; return from function
;
t.inp?:                         ; exception handler address (no exceptions)
       mvi     a,1
       call    $rtp             ; return from the function
;
t.inp$:                         ; <name>$ contains the 2's complement of the
                                ; amount of storage needed.
       dw      -2               ; storage space required (one parameter)
;
       end
```

We must first declare the symbols used externally in the program. $RTP is a routine in the A standard library (ADALIB.REL) which accomplishes a return from a procedure, recovering stack space, resetting links, etc. $BASE is a global variable that points to the static link in the current subprogram's activation record, and is used as an offset to all parameters and variables.

Using $BASE, we find the pointer to the input port by subtracting an offset corresponding to the parameter, using the formula:

offset = -4 - (2 * pointer #)

Scalar values have one pointer; composite values (arrays and records) have two pointers, one to data and the other to a special component which describes the data. To compute the pointer number, you must add up the number of pointers required by any previous parameters. Since our routine only has one scalar parameter, it will be the first pointer.

After we have the pointer to the parameter, we can fetch the port number into the H register (most significant byte ignored).

The next few lines use a programming trick to create ROM-able yet self-modifying code. The code to input from the port number given in the H register is stored onto the stack and called like a subroutine. An artificial return address is also stored on the stack. Upon return, the value from the input port is in the A register.

After fetching the contents of the input port, it is stored into the return value of the function. This address is computed by adding 4 to the value of $BASE. You must ALWAYS return from a subprogram by using the $RTP instruction. Using a "ret" will cause your program to crash.

Three external symbols are provided by the routine. All begin with the name of the procedure in selected form: <parent_name>.<subunit_name>. This form by itself forms the label for the program. The name followed by a question mark indicates the exception handler for the program; for assembly language programs this should be used immediately before the exit sequence. The name followed by a dollar sign indicates a label pointing to the two's complement of the amount of storage required for parameters and local variables. For assembly language programs, only the amount of storage space for parameters need be included here, since there will be no local variable lookup.

Note: when writing assembly language routines, be careful not to use constructs that will create link items unacceptable to the Linking Loader, such as "external plus offset" operands. The Linker will report an error message when it finds such a construct. See Appendix D for a list of link items that cannot be processed.

Example:

```
lxi     h,$base+4     ; invalid, external + offset
```

A
(Ada*) Compiler
Language Reference Manual
Release 3.00

*Ada is a trademark of the Department of Defense (Ada Joint Program Office)

Disclaimer

SuperSoft, Inc. and Maranatha Software Systems make no representations or warranties with respect to the contents herein. While every precaution has been taken in the preparation of this manual, no responsibility is assumed for errors or omissions. Further, no liability is assumed for damages resulting from the use of this product. SuperSoft, Inc. and Maranatha Software Systems reserve the right to revise this publication and to make changes from time to time without obligation to notify any person of such revision or changes.

Table of Contents **Page**

1.0 Introduction

This manual describes the Ada language as implemented in the Maranatha Software Systems A Compiler. The numbering system corresponds to the one used in the ANSI MIL-STD-1815A Ada Reference Manual, dated February 17, 1983. For those cases where no paragraph exists, the language feature is as yet not implemented. For instructions on the use of the compiler, linker, or other tools, refer to the User's Manual.

2.0 Lexical Elements

2.1 Character Set

All of the ASCII character set is acceptable to the compiler. These include graphic characters (those which are visible) and "format effectors", which include tabs, carriage return/line feeds, etc.

```
graphic_character ::= basic_graphic_character
  | lower_case_letter | other_special_character

basic_graphic_character ::=
    upper_case_letter | digit
  | special_character | space_character

basic_character ::=
    basic_graphic_character | format_effector
```

The basic graphic character set includes the following:

(a) upper case letters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(b) digits:

0 1 2 3 4 5 6 7 8 9

(c) special characters:

" # & ' ( ) * + , - . / : ; < = > _ |

(d) the space character

The remaining categories of graphic characters include:

(e) lower case letters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

(f) other special characters:

! $ % ? @ [ ] ^ ` { } ~

Remember that just because a character is in the character set does not necessarily mean it is legal in the context in which it is used. The names of each of the special characters (plus, tilde, etc.) are defined in the MIL-STD-1815A.

2.2 Lexical Elements, Separators, and Delimiters

The first pass of the compiler (ADA.COM) divides the source text into logical parts called lexical units or "tokens". Most editors available for CP/M insert a carriage return-line feed (CRLF) combination at the end of each line of text. The A compiler keeps track of the line number it is on by counting the number of carriage returns received. Users should be wary of word processing editors that may output special control characters which are not recognizable by the compiler and will produce an invalid character message. Definitions of the compound delimiters are in the MIL-STD.

2.3 Identifiers

Identifiers are implemented per the standard.

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

Remember that when naming objects, types or subprograms, all identifiers which differ only in the use of corresponding upper and lower case letters are considered the same, so "IdEnTiFiEr" and "iDeNtIfIeR" refer to the same thing. In Maranatha A the first 10 characters (including underlines) are significant, although an identifier can be any length. Users are cautioned about using identifiers with similar names such as ARRAY_OF_INTEGERS and ARRAY_OF_IDEAS, which are identical (ARRAY_OF_I).

2.4 Numeric Literals

There are two kinds of numeric literals, decimal literals and based literals.

numeric_literal ::= decimal_literal | based_literal

2.4.1 Decimal Literals

Underscore characters may be inserted between adjacent digits of a decimal number for clarity, but may not precede or follow the number.

decimal_literal ::= integer [.integer] [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | E - integer

Examples:

```
1_000_000     -- one million
3.1416        -- pi
10e2          -- one hundred
1.0e+6        -- one million
```

Examples of common mistakes in decimal literals:

```
12345_        -- illegal number, digit must follow underline
100.          -- illegal number, digit must follow decimal point
.25           -- illegal number, should be 0.25
10 + 3.4      -- illegal mismatch of integer and real types
```

2.4.2 Based Literals

Numbers may be represented with a base other than ten with a based number, which can have a base from 2 to 16:

based_literal ::= base # based_integer [.based_integer] # [exponent]

base ::= integer

based_integer ::= extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter

An exponent may be used to indicate the power of the base by which the preceding number is to be multiplied. Both the base and the exponent is in base ten.

Examples:

```
2#1111_1111# – 255
16#FF#       – 255
16#F.FF#E2   – 4095.0
2#101.11#    – 5.75
```

2.5 Character Literals

A character literal is formed by enclosing any of the printable ASCII characters within single quotes. The single quote may itself be used in this manner, represented as '''', and not with four single quotes as with some assemblers. Also keep in mind that there is a difference between a character literal and a string literal, and assignment from one to the other is illegal.

character_literal ::= 'graphic_character'

Examples:

```
'A'  '+'  – ascii characters
''''      – single quote
' '       – blank
'q'       – lower case character
```

2.6 String Literals

A string literal is zero or more printable ASCII characters prefixed and terminated by a string bracket character. The string bracket character can be either the double quote (") or the percent sign (%). Remember that the same string bracket must be used to terminate the sequence as the one used to prefix it. To enclose the string bracket character itself within a string it must be written twice.

string_literal ::= "{graphic_character}"

Examples:

```
"The percent sign (%) may appear in this string"

%String using percent signs as delimiters%

"He said, ""Hello, there!"""

"Example of an ILLEGAL string%     -- does not end with double quote
```

2.7 Comments

Comments start with two hyphens (dashes) and continue to the end of the line. Therefore, comments may not be embedded within a line of Ada source text. Comments are filtered out by the lexical analyzer and have no effect on the program whatsoever, except in one case: if the print pragma has been turned on, comments are passed through all compilation phases. They do not affect code generation, however.

Examples:

```
        -- this is a comment

for I in  -- badly placed comment -- 1..10 loop

    -- in the above example, the "1..10 loop" will be treated as
    -- part of the entire comment. Comments are ONLY ended by the
    -- end of a line. This statement will not compile correctly.

---------------------------- yes, this is a comment
---------------------------- the first two hyphens start
---------------------------- the comment
```

2.8 Pragmas

Pragmas are compile-time switches which allow the user to convey information to the compiler itself.

pragma ::= **pragma** identifier(argument_association);

argument_association ::= name

Most pragmas may appear anywhere within a program; some are restricted to the declarative part. The pragmas are outlined in Appendix B.

2.9 Reserved Words

The identifiers listed below are called reserved words. Identifiers declared by the user may not be reserved words. In this manual, reserved words appear in bold face as in the MIL-STD-1815A. In Maranatha A, all of the standard Ada reserved words are recognized, but not all of them (i.e., **renames**, **terminate**) are implemented. Thus, **renames**, although not implemented, nevertheless cannot be used as an identifier.

abort	**declare**	**generic**	**of**	**select**
abs	**delay**	**goto**	**or**	**separate**
accept	**delta**		**others**	**subtype**
access	**digits**	**if**	**out**	
all	**do**	**in**		**task**
and		**is**	**package**	**terminate**
array			**pragma**	**then**
at	**else**		**private**	**type**
	elsif	**limited**	**procedure**	
	end	**loop**		
begin	**entry**		**raise**	**use**
body	**exception**		**range**	
	exit	**mod**	**record**	**when**
			rem	**while**
		new	**renames**	**with**
case	**for**	**not**	**return**	
constant	**function**	**null**	**reverse**	**xor**

2.10 Allowable Replacements of Characters

The vertical bar character |, which appears on some terminals as a broken bar, may be replaced by the exclamation mark ! as a delimiter.

Example:

```
...
case day is
    when monday ! wednesday =>
...
```

The double quote character used as a string bracket may be replaced by a percent character (see String_Literals, section 2.6).

The pound sign may be replaced by a colon in based numbers (see Based_Literals, section 2.4.2.)

3.0 Declarations and Types

3.1 Declarations

Named entities in Ada can be numbers, enumerated literals, objects, record components, loop parameters, exceptions, types, subtypes, attributes, subprograms, packages, named blocks, named loops, or parameters of a subprogram.

A declaration associates an identifier with a declared entity. Only block and loop identifiers may be used without being declared; they are declared implicitly by their use. Also, labels are implicitly declared at the end of the declarative part of the subprogram in which they are used. Currently six forms of a basic_declaration have been implemented:

```
basic_declaration ::=
      object_declaration      | type_declaration
    | subtype_declaration     | subprogram_declaration
    | package_declaration     | exception_declaration
```

3.2 Objects and Named Numbers

An object is an entity that contains a value of a given type.

object_declaration ::=
    identifier_list : [**constant**] subtype_indication [:= expression];
  | identifier_list : [**constant**] constrained_array_definition [:= expression];

identifier_list ::= identifier {, identifier}

3.2.1 Object Declarations

All user defined variables and constants must be described in an object declaration before they are used. An object is considered a variable unless the reserved word **constant** appears following the colon. Usually a type must be predefined before it can be used in an object declaration; however, an array type definition is allowed which creates a new "unnamed" type for the variables in that identifier list. Assignment to or from that object, when it is not another variable from the same identifier list. must be done via a type conversion. Remember every type definition introduces a new type, so in the following example

```
A : array(1..10) of INTEGER;
B : array(1..10) of INTEGER;
```

The types of A and B appear identical, but Ada treats these two variables as having distinct and separate types. If the object is initialized by an optional expression, all identifiers in the identifier list are assigned that expression, so for

```
X,Y,Z : BOOLEAN := FALSE;
```

X, Y, and Z are all equal to false.

Examples:

```
THERMOMETER_READING : FLOAT := 32.0;
YOUR_AGE : INTEGER;
FINISHED : BOOLEAN := FALSE;
MATRIX : array(1..10,1..10) of INTEGER;
```

3.3 Types and Subtypes

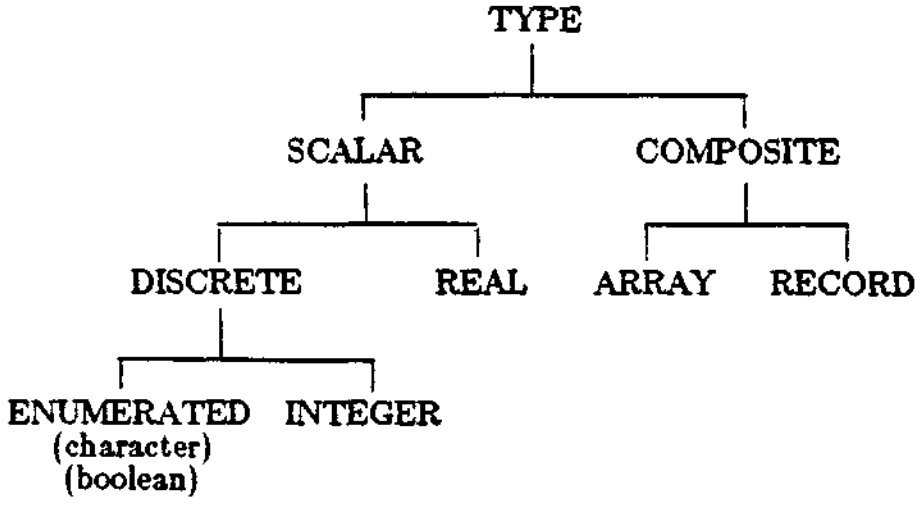
A data "type" determines a set of values which may be taken by data objects of the data type and a set of operations applicable to objects of the data type. Ada has some predefined data types including INTEGER, BOOLEAN, CHARACTER, and FLOAT. A programmer may introduce a new data type by using a type declaration.

3.3.1 Type Declarations

type_declaration ::= full_type_declaration

full_type_declaration ::= **type** identifier **is** type_definition;

A type declaration introduces a new type using a previously declared type or set of types in a unique fashion. A new type can be derived from a previous simple data type such as INTEGER. These types are called scalar types and include integer and real numbers, characters, truth values (BOOLEAN) and other user-defined enumeration types. Data types also include a number of components all of the same type; this composite type is an array. The most complex data type is the record which can have any number of components, each with its own separate data type. In Maranatha A, types can be broken down in the following manner:



A type definition is defined as follows:

```
type_definition ::=
    enumeration_type_definition | integer_type_definition
  | array_type_definition       | record_type_definition
  | derived_type_definition
```

3.3.2 Subtype Declaration

Often a programmer may wish to define a subset of values of a given type, and it may be more convenient to represent the new type as a subtype of the original type rather than introduce a totally distinct data type. This type is called a subtype and is defined as follows:

subtype_declaration ::= **subtype** identifier **is** subtype_indication;

subtype_indication ::= type_mark [constraint]

type_mark ::= *type*_name | *subtype*_name

constraint ::= range_constraint | index_constraint

It is important to point out that a subtype does NOT introduce a new type, but rather a constrained form of its base type. So for the following declarations

subtype OVEN_TEMPERATURE **is** INTEGER **range** 200..500;
TEMP : OVEN_TEMPERATURE;

The subtype OVEN_TEMPERATURE is not a distinct type and objects such as TEMP are of the type INTEGER, but constrained to the range 200 to 500, inclusive.

3.3.3 Classification of Operations

A basic operation is defined as:

- An assignment, membership test, or short-circuit control form
- A selected component or an indexed component
- A qualification (in qualified expressions) or a type conversion
- A numeric literal, string literal or an attribute

3.4 Derived Types

There are times when we would like to use the distinct typing rules of Ada to make a distinction between values of a similar type. We can use the classic example of apples and oranges:

```
type APPLES is new INTEGER;
type ORANGES is new INTEGER;

A : APPLES := 5;
B : ORANGES := 4;
```

In this example, the expression "A+B" would be illegal as we would be using the addition operator on two distinct types (It doesn't make sense to add apples and oranges). We can use a type conversion to forcibly convert from apples to oranges (see 4.6) and this will force the programmer to realize the distinction he has made. Another point to be made from this example is the use of literal integers in the assignments; in Ada, literals are treated as being of the types *universal_integer* and *universal_real* and can be assigned to a derived type such as this. The formal definition of a derived type is as follows:

derived_type_definition ::= **new** subtype_indication

3.5 Scalar Types

Scalar types include real numbers and discrete types. A range constraint may be used to specify the allowable range of a scalar type or subtype.

range_constraint ::= **range** range

range ::= simple_expression .. simple_expression

Examples:

```
type CENTS is range 0..99;
type SECONDS is new INTEGER range 0..59;
```

3.5.1 Enumeration Types

Enumerated types are used to more accurately describe a set of possibilities, such as a stoplight or a deck of cards. The position number of the first literal is zero.

Examples:

type CARD_SUIT **is** (CLUB, DIAMOND, HEART, SPADE);

type STOPLIGHT **is** (RED, AMBER, GREEN);

type CARD_COLOR **is** (RED, BLACK);

In the above definitions, the enumerated literal RED is overloaded. In expressions where the type of the literal cannot be determined from the context a qualified expression can be used to resolve this type ambiguity (see 4.7). The syntax for an enumeration type definition is given below.

enumeration_type_definition ::=
    (enumeration_literal_specification
      {, enumeration_literal_specification})

enumeration_literal_specification ::= enumeration_literal

enumeration_literal ::= identifier

3.5.2 Character Types

A character type is considered an enumeration type in Ada which contains character literals. The predefined type CHARACTER contains the 128 values of the ASCII character set, including control characters. Control characters can easily be used by using the VAL attribute discussed in 3.5.5; the unprintable characters have been thus defined in the package ASCII.

3.5.3 Boolean Types

The predefined type BOOLEAN is defined as

type BOOLEAN **is** (FALSE, TRUE);

Thus, BOOLEAN'POS(FALSE) = 0 and BOOLEAN'POS(TRUE) = 1. Unlike other languages, the logical operators **and**, **or** and **xor** will only work with boolean operands and not with integers.

3.5.4 Integer Types

An integer type definition introduces a set of consecutive integers using the bounds of a range constraint:

integer_type_definition ::= range_constraint

Integers in Maranatha A are implemented as a 15 bit value with a 1 bit sign. The definition of integer in the package STANDARD is

type INTEGER **is range** -32768 .. 32767;

NOTE: Arithmetic errors (underflow, overflow, divide by zero) will not produce a run time error (except in the case of divide by zero which prints an appropriate error message), since the predefined exceptions are not implemented.

3.5.5 Operations of Discrete Types

For every discrete type, the attributes POS, VAL and IMAGE have been implemented and are defined as follows:

T'POS(X) - The parameter X must be of type T; the result of the attribute is the position number of X in the series and is of type *universal_integer*.

T'VAL(N) - The parameter N must be a value of an integer type; the result is a value of the type T whose position number is N. No constraint error is raised if N is not within the range of T.

T'IMAGE(N) - The parameter N must be a value of an integer type; the result is a value of type STRING.

Examples:

STOPLIGHT'VAL(0) = RED

BOOLEAN'POS(TRUE) = 1

CARD_SUIT'VAL(2) = HEART

LINE_FEED : **constant** CHARACTER := CHARACTER'VAL(10);

PUT(INTEGER'IMAGE(10));

3.6 Array Types

Arrays are data values with a number of distinct components all of the same type. The components are distinguished by an index which must be a discrete type (integer or enumerated, including character and boolean). The elements of the array can be of any type, including record types or another array type. Note that the element type of an array must be a type_mark. Preliminary Ada allowed subtype indications to be used in the array definition; now they must be declared in a separate type or subtype declaration.

array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
    array(index_subtype_definition {,index_subtype_definition}) **of**
            *component*_subtype_indication

constrained_array_definition ::=
    array index_constraint **of** *component*_subtype_indication

index_subtype_definition ::= type_mark **range** <>

index_constraint ::= (discrete_range {,discrete_range})

discrete_range ::= *discrete*_subtype_indication | range

An array_type_definition will produce two kinds of array types, unconstrained and constrained. Definitions using an index will be unconstrained, while those having the index constraint will be constrained. Unconstrained array type definitions are useful for defining an array where the array bounds are flexible:

type SORT_ARRAY **is** **array**(INTEGER **range** <>) **of** CHARACTER;

These declarations are used in a normal object declaration where the type mark is that of the unconstrained array and an index constraint is provided. This type may also be used as the parameter type in a subprogram or return type of a function. The array bounds are considered to be an integral part of the array itself and are passed along with the array data. They may be accessed by the attributes FIRST and LAST (see section 3.6.2).

3.6.1 Index Constraints and Discrete Ranges

An example of using the above unconstrained array declaration follows:

```
TEST_ARRAY : MATRIX(1..10,1..10);
```

This is an object declaration with a subtype indication consisting of a type mark and an index constraint. Use of a constrained array declaration is identical except the index constraint is omitted.

3.6.2 Operations of Array Types

For an array object or for the type mark of a constrained array type, the following attributes have been implemented:

FIRST	The lower bound of the first index
LAST	The upper bound of the first index
FIRST(N)	The lower bound of the Nth index
LAST(N)	The upper bound of the Nth index
LENGTH	The number of values of the first index
LENGTH(N)	The number of values of the N'th index

No exception will be raised if the value of N does not correspond to an index of the array. Use of the attributes FIRST and LAST are highly recommended, especially when looping through the range of the array:

```
for I in A'FIRST .. A'LAST loop
```

This construct would be more readily implemented as

```
for I in A'RANGE loop
```

but since the RANGE attribute has not yet been implemented using FIRST and LAST can be used. These attributes are useful in procedures whose parameters are of an unconstrained type such as STRING.

3.6.3 The Type String

The predefined type STRING is an unconstrained one dimensional array of characters, indexed by a special integer subtype POSITIVE:

```
subtype POSITIVE is INTEGER range 1..INTEGER'LAST;
type STRING is array (POSITIVE range <>) of CHARACTER;
```

String literals are a special aggregate assignable to this type or any one dimensional array of characters.

3.7 Record Types

Record types introduce a new composite type with any number of distinct components, each with its own type.

```
record_type_definition ::=
    record
        component_list
    end record

component_list ::= component_declaration {component_declaration}
                 | null;

component_declaration ::=
    identifier_list : component_subtype_definition;

component_subtype_definition ::= subtype_indication
```

The components in a record may be any previously declared type including scalar types, arrays or records, with the exception that the definition may not be recursive (a component may not be the record type itself).

Examples:

```
type PERSON is record
    GENDER : SEX;
    AGE : INTEGER;
    MARRIED : BOOLEAN;
end record;
```

3.7.3 Variant Parts

Currently, record variant parts are not implemented. However, the syntactical construct "choice" is introduced here and implemented in Maranatha A.

```
choice ::= simple_expression | discrete_range | others
```

3.7.4 Operations of Record Types

The basic operations of a record type include assignment, membership tests, selection of record components, qualification and type conversion (for derived types).

3.9 Declarative Parts

Declarative parts describe a set of variables, constants, types, subprograms, and packages available to the main body of a subprogram or block.

```
declarative_part ::=
    {basic_declarative_item} {later_declarative_item}

basic_declarative_item ::=
    basic_declaration | representation_clause | use_clause

later_declarative_item ::= body
    | subprogram_declaration | package_declaration | use_clause

body ::= proper_body | body_stub

proper_body ::= subprogram_body | package_body
```

4.0 Names and Expressions

4.1 Names

At this point it becomes important to distinguish between an identifier and a name. An identifier is simply a word used by the programmer given to an object such as the procedure identifier FACTORIAL, or the type identifier STOPLIGHT. In Maranatha A, a name is defined syntactically as:

```
name ::= simple_name
  | character_literal      | indexed_component
  | selected_component     | attribute

simple_name ::= identifier

prefix ::= name | function_call
```

This is a recursive definition and describes a powerful method to access array elements, record components, variables and attributes. This release of Maranatha A does not support selected components fully (i.e., a variable may not be prefaced by its subprogram name as in FACTORIAL.X). Selected components may be used to access record components, of course. Normally, names could denote subprograms, packages, exceptions, labels, block and loop identifiers and operators, as well as variables.

4.1.1 Indexed Components

An indexed component is used to denote an element of an array:

```
indexed_component ::= prefix(expression {,expression})
```

Since "name" is a recursive definition, an array of an array must be referenced with two separate indexed components, as shown:

```
type TEST is array(1..10) of INTEGER;

TEST_VARIABLE : array(1..10) of TEST;

Z := TEST_VARIABLE(2)(8);
```

Accessing the array test_variable must be done with one indexed component, which will reference the second element in test_variable. Since this element is an array of integers, we use a second indexed component to reference the 8th element, which is an integer.

4.1.3 Selected Components

Selected components are used to denote a component of a record.

selected_component ::= prefix.selector

selector ::= simple_name

This definition, like indexed component, is recursive. This can leave us with such combinations as

Z := RECORD_OBJECT.COMPONENT_ONE(5).AGE

In this case, a selected component is used to reference "component_one" of that record. It is an array, and we select the fifth element of that array with an indexed component. Finally, the array has an element type which is a record and we reference a component "age" of that record.

4.1.4 Attributes

Attributes denote predefined characteristics of certain entities. Appendix A gives a list of all of the implemented attributes and how and when they may be used.

attribute ::= prefix'attribute_designator

attribute_designator ::= simple_name [(*universal_static*_expression)]

This definition is similar to indexed component and selected component in that it is recursive.

4.2 Literals

A literal is either a numeric literal (which can be decimal or based), an enumeration literal, or a string literal. Numeric literals are the literals of the type *universal_integer* and *universal_real*, overloaded for all numeric types. A string literal combines a sequence of characters into a value of a one- dimensional array of a character type.

4.4 Expressions

The expression processing mechanism in Ada is complex and defined as follows:

```
expression ::=
    relation {and relation}          | relation {and then relation}
  | relation {or relation}           | relation {or else relation}
  | relation {xor relation}

relation ::=
    simple_expression {relational_operator simple_expression}
  | simple_expression [not] in range
  | simple_expression [not] in type_mark

simple_expression ::=
    [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=
    numeric_literal | string_literal | name | function_call
  | type_conversion | qualified_expression | (expression)
```

Expressions are used mainly to calculate a numeric value of a given type, but may return a boolean result due to a logical operator or other types resulting from type conversions or the use of attributes.

4.5 Operators and Expression Evaluation

All standard Ada operators have been implemented in Maranatha A:

```
logical_operator          ::= and | or | xor

relational_operator       ::= = | /= | < | <= | > | >=

binary_adding_operator    ::= + | - | &

unary_adding_operator     ::= + | -

multiplying_operator      ::= * | / | mod | rem

highest_precedence_operator ::= ** | abs | not
```

4.5.1 Logical Operators and Short-Circuit Control Forms

The logical operators **and, or** and **xor** are only applicable to operands of a boolean type and return a boolean result. They are not available to operands of an integer type as in BASIC. The short circuit control forms **and then** and **or else** are defined for two operands of a boolean type a give a boolean result. The left operand (which is a boolean relation) is evaluated first. If this operand is evaluated as FALSE with a control form of **and then** the rest of the expression is not evaluated and the value of the expression is FALSE. If this operand is evaluated as TRUE with a control form of **or else**, the rest of the expression is not evaluated and the value of the expression is TRUE. Remember that any function calls in any of the right boolean relations may or may not be evaluated depending on the value(s) of the leftmost boolean relation(s). This was done to reduce the problems of function side- effects.

4.5.2 Relational Operators and Membership Tests

The predefined relational operators return a boolean value and operate on operands of the same type. Equality and inequality ("=" and "/=", respectively) are defined for any type, while the greater-than and less- than operators are defined only for scalar types or array types with discrete elements. The Pascal inequality operator "<>" is called a "box" in Ada and is incorrect as a relational operator.

4.5.3 Binary Adding Operators

The operators "+" and "-" are predefined and operate on any numeric type, returning a result of the same type. Concatenation ("&") is defined for any one dimensional array type, but is not yet implemented for an array type and its corresponding element type.

4.5.4 Unary Adding Operators

The unary operators "+" and "-" apply to a single operand and return a result of the same type.

4.5.5 Multiplying Operators

All of the multiplying operators have been implemented and are predefined for all numeric types. **Mod** and **rem** correspond to integer modulus and remainder operators. Page 4-19 of the Ada Reference Manual gives a good description of how these operators work.

4.5.6 Highest Precedence Operators

The operator **abs** is predefined for any numeric type and returns the absolute value of the primary which follows it. Note that in the early development of Ada, **abs** was considered a function call. Since a primary can be a parenthesized expression, the form **abs**(expression) is still legal. The **not** operator is predefined for any boolean type and returns the logical negation of the boolean primary which follows it. The exponentiating operator "**" applies to an integer type, returning the same integer type, or a floating point type, returning the same floating point type.

4.6 Type Conversions

Type conversions are defined as follows:

type_conversion ::= type_mark(expression)

To convert an integer to floating point, use the form FLOAT(X). To convert a floating point number to an integer (this involves rounding), use the form INTEGER(X). Type conversions are also allowed between derived types and their base or parent types.

Example:

```
total_fruit := integer(number_apples) + integer(number_oranges);
```

4.7 Qualified Expressions

A qualified expression is used to explicitly state the type of an expression enclosed in parenthesis. This construct is most often used to distinguish between overloaded enumeration literals, such as RED in the predefined types stoplight and card_color in section 3.5.1:

```
MY_CARD_COLOR := CARD_COLOR'(RED);
HOLLYWOOD_VINE := STOPLIGHT'(RED);
```

A qualified expression is defined as

qualified_expression ::= type_mark'(expression)

5.0 Statements

5.1 Simple and Compound Statements - Sequences of Statements

The programmer causes actions to be performed by a sequence of statements in the main subprogram body. A statement may be either simple (indicating a single action) or compound (which may contain internal sequences of statements). A statement may be labeled with an identifier enclosed by double angle brackets.

sequence_of_statements ::= statement {statement}

statement ::=
    {label} simple_statement | {label} compound_statement

simple_statement ::= null_statement
    | assignment_statement | procedure_call_statement
    | exit_statement | return_statement
    | goto_statement | raise_statement

compound_statement ::=
      if_statement | case_statement
    | loop_statement | block_statement

label ::= <<*label*_simple_name>>

null_statement ::= **null**;

In any sequence of statements, at least one statement *must* appear. In those cases where the programmer does not want any action to occur, or wants to leave that section "blank" and fill it in later, the **null** statement can be used to indicate no action is to be performed. The raise statement will be covered separately under exceptions (see section 11.3).

5.2 Assignment Statement

assignment_statement ::= *variable*_name := expression;

The assignment statement is used to change the value of a data item. Assignments to constants and **in** mode parameters are not allowed.

5.2.1 Array Assignments

Assignments to an array variable are allowed for data items of the same array type, concatenated objects, and strings. Array slices are not yet implemented.

5.3 If Statements

An **if** statement is used to execute a selected sequence of statements given a boolean condition. The **if** construct in Ada is different than in Algol 60 and its derivatives (i.e., Pascal) in that the "dangling-else" problem has been eliminated. For example, take the following Pascal programs:

Program 1:

```
if a then
    if b then
        write('X')
    else
        write('Y');
```

Program 2:

```
if a then
        begin
                if b then write('X')
        end
else
        write('Y')
```

In Program 1, does the else: correspond to the first or the second if? Pascal resolves the ambiguity by considering the "else" to be attached to the second if statement, requiring A to be true to print X. If the programmer wanted A to be false to print Y, Program 2 would have to be used. Ada solves this "dangling-else" ambiguity by using an **end if** following the sequence of statements of an **if** construct. The above programs would be written in Ada as

Program 1:

```
if A then
    if B then
        PUT('X');
    else
        PUT('Y');
    end if;
end if;
```

Program 2:

```
if A then
        if B then
                PUT('X');
        end if;
else
        PUT('Y');
end if;
```

The **if** statement is represented syntactically as

```
if_statement ::=
    if condition then
        sequence_of_statements
  { elsif condition then
        sequence_of_statements }
  [ else
        sequence_of_statements ]
    end if;

condition ::= boolean_expression;
```

5.4 Case Statements

The Ada **case** statement is used to select a sequence of statements to be executed based on a discrete value computed from an expression:

```
case_statement ::=
    case expression is
        case_statement_alternative
      {case_statement_alternative}
    end case;

case_statement_alternative ::=
    when choice {|choice} =>
        sequence_of_statements
```

The **case** statement is similar to the Algol-60 and "C" "switch", and the Pascal or Algol-68 case statement. Its usefulness can be seen when comparing a short program written using the **if** statement:

```
if DAY = SUNDAY then
    GO_TO_CHURCH;
elsif DAY = MONDAY then
    LATE_FOR_WORK;
elsif DAY in TUESDAY..THURSDAY then
    NORMAL_WORK_DAY;
elsif DAY = FRIDAY then
    LEAVE_WORK_EARLY;
elsif DAY = SATURDAY then
    WATCH_FOOTBALL;
else
    ERROR;
end if;
```

The same segment written using the **case** statement would appear as

```
case DAY is
    when SUNDAY => GO_TO_CHURCH;
    when MONDAY => LATE_FOR_WORK;
    when TUESDAY..THURSDAY => NORMAL_WORK_DAY;
    when FRIDAY => LEAVE_WORK_EARLY;
    when SATURDAY => WATCH_FOOTBALL;
    when others => ERROR;
end case;
```

The code in the branch of a **when** clause is not limited to one statement, and the **when** clause is not limited by a single choice. Any number of choices can be "or'ed" together into one choice, and a range of values may be defined as in TUESDAY..THURSDAY. The values do not have to be enumerated as in Pascal.

5.5 Loop Statements

A **loop** statement specifies that the embedded sequence of statements is to be executed any number of times.

```
loop_statement ::=
    [loop_simple_name:]
      [iteration_scheme] loop
        sequence_of_statements
      end loop [loop_simple_name];

iteration_scheme ::=
    while condition
  | for loop_parameter_specification

loop_parameter_specification ::=
    identifier in [reverse] discrete_range
```

There are three forms of the **loop** statement: the basic loop, the **while** loop, and the **for** loop. The basic loop results in an infinite execution of the sequence of statements, as in the following example:

```
loop
    WAIT_FOR_PHONE_TO_RING;
    ANSWER_PHONE;
    EXECUTE_COMPUTERIZED_BULLETIN_BOARD;
    HANG_UP_PHONE;
end loop;
```

This loop will execute indefinitely. Once a loop is entered, there are a number of ways of exiting:

1) An **exit** statement; (see 5.7)
2) A **raise** statement; (see 11.3)
3) A **return** statement; (see 5.8)
4) Reset the computer.

When the basic loop is modified by an iteration scheme, producing a **while** loop, the body of the loop is executed as long as the condition of the when clause is true. The loop will not be executed if the condition is initially evaluated as false.

```
while TEMPERATURE > 75 loop
    AIR_CONDITIONER;
end loop;
```

The last form of the loop statement is the **for** loop. In this case, the iteration scheme specifies a set of discrete values which are successively assigned to a loop parameter:

```
COMPUTE_TOTAL_HOURS:
for I in MONDAY..FRIDAY loop
    WEEKLY_HOURS := WEEKLY_HOURS + HOURS_WORKED(I);
end loop COMPUTE_TOTAL_HOURS;
```

In the last example we have shown the use of a labeled loop statement.

5.6 Block Statements

A block statement can be considered a miniature subprogram embedded directly in a sequence of statements. It introduces its own sequence of statements and, optionally, a declarative part:

```
block_statement ::=
    [block_simple_name:]
    [declare
        declarative_part]
     begin
        sequence_of_statements
    [exception
        exception_handler
       {exception_handler}]
     end [block_simple_name];
```

Example:

```
SWAP:
declare
    TEMP : INTEGER;
begin
    TEMP := X;
    X := Y;
    Y := TEMP;
end SWAP;
```

This labeled block (similar to the example on page 5-9 of the Ada Reference Manual) swaps the integer values of X and Y. Since the variable TEMP is a temporary variable and is only needed to store the value of X during the swap operation, it is declared as a variable, local to this block, and is only valid for the sequence of statements of the block. Following the block, the variable TEMP no longer exists.

5.7 Exit Statements

The exit statement is one method of terminating an enclosing loop:

exit_statement ::= **exit** [*loop*_name] [**when** condition];

An optional loop identifier may be given, either to improve the readability of the program by formally stating the loop it is exiting or to exit from an outer loop. If no loop identifier is given, the innermost enclosing loop is assumed. An optional condition may be included and the exit statement executed on the boolean condition. This feature is useful to simulate the Pascal "repeat-until" construct:

Pascal "repeat":

```
repeat
    read(ch)
until ch <> ' '
```

Ada "exit":

```
loop
        GET(CH);
        exit when CH /= ' ';
end loop;
```

The above program segments will input source text until a non-blank character is found.

5.8 Return Statements

A return statement terminates execution of a subprogram. A return statement for a function must include an expression of the same type as the return type of the function, and a return statement for a procedure must not include a return expression. A return statement is not required in a procedure if the programmer wishes to exit upon terminating the procedures sequence of statements. A function, however, must have a return statement within its sequence of statements and should not return from the end of the function; doing so will return an unspecified value.

return_statement ::= **return** [expression];

5.9 Goto Statements

Goto statements are provided for in Ada but their usage has been limited. A goto statement may not transfer control into a compound statement, nor between the arms of an if or case statement. A goto statement also may not transfer control into or out of a subprogram. Its basic syntax is

goto_statement ::= **goto** *label*_name;

6.0 Subprograms

A subprogram is a program unit, invoked by a subprogram call, which describes an action. There are two types of subprograms: procedures and functions. A procedure call is a statement; a function call returns a value and is used in expressions. A subprogram is externally defined by its subprogram specification. The subprogram specification tells you what kind of a subprogram it is (procedure or function), the number, types and modes of the parameters it requires and its return type (in the case of functions).

6.1 Subprogram Declarations

A subprogram declaration allows the programmer to "forward-reference" a subprogram, as in Pascal.

subprogram_declaration ::= subprogram_specification;

subprogram_specification ::=
    procedure identifier [formal_part]
  | **function** designator [formal_part] **return** type_mark

designator ::= identifier

formal_part ::= (parameter_specification {;parameter_specification})

parameter_specification ::= identifier_list : mode type_mark

mode ::= [**in**] | **in out** | **out**

6.2 Formal Parameter Modes

The formal part of the subprogram specification defines the subprogram's parameters. A parameter has one of three modes; **in** (default), **out**, or **in out**. These can be thought of as "read-only", "write-only" and "read-write" objects. If no mode is given, the parameter(s) are assumed to be read-only (mode **in**). The Ada language does not define what mechanism is used for parameter passing. In Maranatha A, for composite types, the formal parameter provides access to the corresponding actual parameter throughout the execution of the subprogram. This method reduces execution time by passing only the pointers to the object and not the entire object itself; this method also saves run-time storage space. The warning on page 6-3 of the Ada Reference manual, "A program that relies on one particular mechanism is therefore erroneous", applies.

6.3 Subprogram Bodies

The subprogram body defines a subprogram. It includes the subprogram specification, which contains all information describing the subprogram, a declarative part, which contains types, objects and subprograms local to the body of the subprogram, and a sequence of statements to be executed when the subprogram is called.

```
subprogram_body ::=
    subprogram_specification is
      [ declarative_part ]
    begin
        sequence_of_statements
    [exception
        exception_handler
      {exception_handler}]
    end [designator];
```

6.3.1 Conformance Rules

No variations are allowed where language rules permit the specification of a subprogram in more than one place; i.e., a forward-referenced subprogram, or a packaged subprogram.

- Since default parameters are not implemented, a numeric literal cannot appear in a subprogram specification.
- Selected components (for subprogram names) are not implemented.
- Operator symbols are not implemented.

6.4 Subprogram Calls

A subprogram call is either a procedure or a function call and invokes the execution of that subprogram. At present, there are no actual parameter associations or default actual parameters.

```
procedure_call_statement ::=
    procedure_name [actual_parameter_part];

function_call ::=
    function_name [actual_parameter_part]

actual_parameter_part ::=
    (parameter_association {, parameter_association})

parameter_association ::= actual_parameter

actual_parameter ::=
    expression | variable_name | type_mark(variable_name)
```

6.4.1 Parameter Associations

Each actual parameter must have the same type as the corresponding formal parameter (otherwise the compiler will not be able to find the subprogram with parameters of the wrong type). An actual parameter with a formal parameter of mode **in** must be an expression (or "r-value") and evaluated before the subprogram call. An actual parameter associated with a formal parameter of mode **in out** or **out** must be the name of a variable. The variable is evaluated before the call and passed by reference ("l-value").

6.5 Function Subprograms

A function is a subprogram that returns a value of any predefined type and may only have parameters of mode **in**, to reduce function side effects. If the body of a function is left by reaching the end of the function, the return value is undefined and such a program is erroneous.

6.6 Parameter and Result Type Profile - Overloading of Subprograms

In Ada, there may be several subprograms with the same name which may perform similar actions but with different parameters. A good example of this is the sample package MATHLIB on the distribution disk which contains the overloaded function SQRT, one for integer computations, and another for floating point square root operations. An overloaded subprogram must differ from others with the same identifier in at least one respect. For any two subprograms, if the order, number and types of parameters are the same and (for functions) the result type is the same the two subprograms are equivalent and one subprogram will "hide" the other.

7.0 Packages

Programs in Maranatha A are composed of subprograms and packages. Packages allow the programmer to group logically related objects and operations together.

7.1 Package Structure

Packages, like subprograms, are divided into a specification and a body. Every package has a specification, but the package body is not required if there are no subprogram declarations within the package specification, or if there is no sequence of statements the package should execute.

```
package_declaration ::= package_specification;

package_specification ::=
    package identifier is
        {basic_declarative_item}
    end [package_simple_name]

package_body ::=
    package body package_simple_name is
        [declarative_part]
  [ begin
        sequence_of_statements
  [ exception
        exception_handler
      {exception_handler}]]
    end package_simple_name;
```

The simple name of a package body must match the package specification. Like subprograms, if a simple name is given at the end of a specification or body, it must repeat the package simple name.

7.2 Package Specifications and Declarations

The declarations given in a package specification are automatically visible in the package body; no use clause is required (even if they are separately compiled). Remember, a subprogram or package body is not a basic declarative item and so cannot appear in a package specification.

7.3 Package Bodies

For the subprogram declarations which appear in the specification, a corresponding subprogram body must appear in the package body. Note that objects and types declared within the package body are not available outside of the package. Of course, these items are considered "global" in that, unlike a subprograms local objects, they do not "go away" after the package has been elaborated; packaged subprograms may continue to reliably use these variables. A package body must be elaborated before it is used; This is dependent on the order in which it is loaded, a facility outside of the Ada environment. See the linking loader manual for details on the order of linking/loading. The exception PROGRAM_ERROR has not been implemented and incorrect linking will result in unknown side effects.

8.0 Visibility Rules

The rules defining the "scope" of declarations and what identifiers are usable at various parts of the Ada program are defined here.

8.1 Declarative Region

A declarative region is used for the purposes of determining where certain identifiers are visible. A declarative region is formed by:

- A subprogram or package declaration, together with its corresponding body.
- A record type declaration
- A block statement or a loop statement

8.2 Scope of Declarations

For any given declaration, Ada defines a portion of the program called the scope of that declaration. The scope of any entity is the region of text where its declaration has effect. Its immediate scope extends to the end of the declarative region, and for most declarations the scope of the declarations extends beyond the immediate scope:

- A component declaration
- A parameter specification

8.3 Visibility

The visibility of a declaration defines where its name can be seen. In all cases, an entity is only visible within its scope. For each identifier and at each place in the text, Ada defines the possible meanings of an occurrence of the identifier. In some cases, the visibility rules determine no possible meaning. Either the identifier was not declared, is not visible, or we are not within the scope of the identifier. In the following example, X is not visible because we are outside of its scope:

Example:

```
declare
    X : INTEGER;
begin
    null;
end;
X := 5;          -- illegal (X is not visible)
```

In other cases, the visibility rules may determine more than one meaning of an identifier. In these cases the identifier is legal only if one declaration is acceptable, given the overloading rules in the given context. In the following example, the identifier RED has more than one meaning (see declarations in 3.5.1), but is legal because the overloading rules require the expression to have a certain type.

Example:

```
HOLLYWOOD'VINE := RED;
```

8.4 Use Clauses

A use clause achieves direct visibility of declarations that appear in the visible parts of named packages. Since named components are not fully implemented, a use clause *must* be given for each package:

use_clause ::= **use** *package*_name {,*package*_name};

The appearance of a use clause assumes that 1) a with clause specified this package was needed in a context specification, or 2) the associated package is already visible in a preceding declarative part. All potentially visible declarations are made visible; that is, potentially visible declarations that have the same identifier are automatically made visible regardless of their type, etc. Users are forewarned that this will produce unexpected errors in rare cases.

8.6 The Package STANDARD

The predefined types and operations are declared in a predefined package named STANDARD, and is described in Annex C. The package STANDARD forms a declarative region which encloses every library unit and thus the main program.

8.7 The Context of Overload Resolution

Overloading (in Maranatha A) is defined for subprograms and enumerated literals, as well as the inherent operations. The appropriate rules in section 8.7 of the Ada LRM apply.

10.0 Program Structure and Compilation Issues

10.1 Compilation Units - Library Units

The text of a program must be submitted in one compilation.

compilation ::= compilation_unit

compilation_unit ::=
  context_clause library_unit | context_clause secondary_unit

library_unit ::= package_declaration | subprogram_body

secondary_unit ::= library_unit_body | subunit

library_unit_body ::= subprogram_body | package_body

10.1.1 Context Clauses - With Clauses

A context clause informs the compiler that the named packages are needed within the following compilation unit.

context_clause ::= {with_clause {use_clause}}

with_clause ::= **with** *unit*_simple_name {,*unit*_simple_name};

10.2 Subunits of Compilation Units

A subunit allows subprograms of another program unit to be compiled separately.

body_stub ::= subprogram_specification **is separate**;

subunit ::= **separate** (*parent_unit*_name) proper_body

In Maranatha A, only subprograms may be subunits. Further, only subprograms of library units may be subunits (a program may not be a subunit of another subunit).

Restrictions have been placed on the visibility inside a subunit. A subunit may not access any subprograms or exceptions declared in the subunit's parent, with the exception that a packaged subprogram which is a subunit may access items within the package specification (including subprograms or exceptions).

10.3 Order of Compilation

The CP/M environment does not normally allow for time and date stamping of files, and so the order of compilation of program units is not strictly checked by the Maranatha A compiler. The compiler does ensure that package specifications are compiled before package bodies as the package specification symbol table (.SYM file) must appear on the disk to compile the package body. Likewise, subunits may not be compiled until their parent units have also been compiled.

10.4 The Program Library

A normal submission to the compiler consists of only the compilation unit. A future release will create reference files to ensure the order of compilation is correct.

10.5 Elaboration of Library Units

Before the main subprogram of a program is executed, all packages must first be elaborated. It is the responsibility of the user to ensure the package specifications, package bodies and subunits are loaded in the proper order to ensure correct elaboration. For more information, consult the linking loader manual, which has examples of program loading.

10.6 Program Optimization

Optimization of the program text is automatically performed by Maranatha A. If any statements will never be executed, the optimizer will omit the corresponding machine code. If the print pragma has been turned on, the unreachable source text will not appear in the listing.

11.0 Exceptions

This chapter defines the facilities in Ada for the creation and handling of run time errors, called exceptions. The instance of a run-time error is called "raising" the exception. Presently, only user-defined exceptions exist. The "suppress" pragmas are considered to be all turned on to increase program execution speed.

11.1 Exception Declarations

An exception declaration declares a name for a user-defined exception. Note that this name can only appear in a raise statement or an exception handler.

exception_declaration ::= identifier_list : **exception**;

If an exception declaration should appear in a recursive subprogram, the exception name denotes the same exception for all invocations of that subprogram (it is not "redeclared").

11.2 Exception Handlers

Run-time errors are responded to in an exception handler, which may appear in a subprogram body, package body, or block:

exception_handler ::=
    when exception_choice {| exception_choice} =>
        sequence_of_statements

exception_choice ::= *exception*_name | **others**

The exception choice **others** must appear last and by itself. Note that **others** may be used for exceptions that are not visible at the place of the exception handler. For example, if an embedded subprogram declares a local exception which is raised and not handled by that subprogram, it is no longer visible outside of the subprogram but still can be handled by the choice **others**:

```
procedure test is
...
    procedure inner is
        error : exception;
    begin
        raise error;
    end;
...
begin    -- test
    inner;
exception
    when others =>
        -- exception "error" can be handled here...
end test;
```

11.3 Raise Statements

A raise statement raises an exception; an example was seen in the last section.

raise_statement ::= **raise** [*exception*_name];

A raise statement without an exception name is allowed only within the sequence of statements of an exception handler, and "re-raises" the exception that caused the execution of this exception handler.

11.4 Exception Handling

When an exception is raised, a set of rules define the exception handler that is executed, depending on where the exception was raised.

11.4.1 Exceptions Raised During the Execution of Statements

If an exception is raised in a sequence of statements that has an exception handler, control is transferred to that handler. If the sequence of statements does not have an exception handler, then the action performed depends on the location in which the exception was raised:

subprogram_body : exception is re-raised at the point of call of the subprogram. If the subprogram is the main subprogram itself, the environment prints an "unhandled exception" message.

block : exception is re-raised immediately after the block statement.

package_body : if package body is a declarative item, exception is re-raised after the declarative item. If the package body is a library unit, the environment prints an "unhandled exception" message.

An exception that is "re-raised" is said to be propagated. If an exception handler does not handle the existing exception, it is propagated.

11.4.2 Exceptions Raised During the Elaboration of Declarations

The elaboration is abandoned if an exception is raised while a declaration is being evaluated. The next action depends on the location of the declaration:

subprogram_body : the exception is re-raised at the point of call of the subprogram. If this is the main subprogram, the environment prints an "unhandled exception" message.

block : the same exception is re-raised immediately following the block.

package_body : the exception is re-raised immediately after the package body, if it is a declarative item. If in a library unit, the environment prints an "unhandled exception" message.

13.0 Representation Clauses and Implementation Dependent Features

13.1 Representation Clauses

Representation clauses enable the user to tell the compiler how an entity is to be implemented. They do not alter the net effect of the program, but may be used to make a more efficient representation of an entity or to interface with features outside the domain of the compiler; i.e., the CP/M environment.

representation_clause ::= address_clause

13.5 Address Clauses

An address clause is used to assign an explicit address to an object.

address_clause ::= **for** simple_name **use at** integer_literal;

The full implementation of this feature would allow the assignment of an explicit address to a subprogram or hardware interrupt service routine; these features have not yet been implemented. Also, a single integer must be used instead of the full *static*_simple_expression for the address.

14.0 Input-Output

The input-output facilities provided by Maranatha A, as far as they are implemented, conform to those in the standard Ada language reference manual. They are provided in source form for those who wish to write their own input-output drivers. They have been written as a separate part of the compiler in an effort to emphasize the portability of the code and to try to conform with the spirit of the designers of the Ada language.

14.1 External Files and File Objects

External files are identified by a string (the name), and are documented in Appendix F. Input-output in human readable form is defined in the non-generic package TEXT_IO.

Before operations can be performed on an object of type FILE_TYPE, it must be associated with an external file. The association is made by the procedures "open" and "create". While this association is in effect, the file is said to be open.

A file has a mode, which is a value of the enumeration type

```
type FILE_MODE is (IN_FILE, OUT_FILE);
```

Exceptions that can be raised by any of the input-output subprograms are defined in the package IO_EXCEPTIONS.

14.2 Sequential and Direct Files

Currently, only sequential access is allowed to external files through the package TEXT_IO.

14.2.1 File Management

Before any file processing can be carried out, a file object within the Ada source program must be associated with an external CP/M file. When this association has taken place, the file is said to be "open." This association can take place by the procedure CREATE or the procedure OPEN.

```
procedure CREATE(FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE;
                 NAME : in STRING);
```

The procedure CREATE establishes a new external CP/M file and associates it with the given internal file object. The given file is left open, and the mode is set to the given access mode.

The exception STATUS_ERROR is raised if the given file is already open. The exception NAME_ERROR is raised if the string given as NAME does not allow the identification of an external file.

```
procedure OPEN(FILE : in out FILE_TYPE;
               MODE : in FILE_MODE;
               NAME : in STRING);
```

The procedure OPEN associates the given internal file object with an existing external file with the given name. The given file is left open, and the mode is set to the given access mode.

The exception STATUS_ERROR is raised if the given file is already open. The exception NAME_ERROR is raised if the string given as NAME does not allow the identification of an external file; in particular, this exception is raised if no external file with the given name exists.

```
procedure CLOSE(FILE : in out FILE_TYPE);
```

After processing has been completed on the external file, the association is severed by the procedure CLOSE. The internal disk buffers associated with the internal file will automatically be "flushed" to the disk if the file mode is OUT_FILE. This procedure need not be used if the file mode is IN_FILE and the file object will not be used again (i.e., end of main program).

The exception STATUS_ERROR is raised if the given file is not open.

procedure DELETE(FILE : **in out** FILE_TYPE);

Deletes the external file. The file is closed, and the external file ceases to exist.

The exception STATUS_ERROR is raised if the file is not open.

function MODE(FILE : **in** FILE_TYPE) **return** FILE_MODE;

Returns the current mode of the given file.

The exception STATUS_ERROR is raised if the given file is not open.

function NAME(FILE : **in out** FILE_TYPE) **return** STRING;

The function NAME returns a string representing the name of the external file associated with the internal file. The returning string is of the format "D:FILENAME.TYP", where "D" is the drive number of the file, "FILENAME" is the name of the file (up to 8 characters with no embedded blanks) and "TYP" is the three character extension of the file name.

The exception STATUS_ERROR is raised if the given file is not open.

function IS_OPEN(FILE : **in** FILE_TYPE) **return** BOOLEAN;

The boolean function IS_OPEN accepts a file object as an argument and returns true if the internal file is associated with an external file.

14.3 Text Input_Output

Many of the procedure in the package TEXT_IO have been reproduced in the Maranatha A package TEXTIO. Since the source code for this package has been provided, they may be adapted for your own purposes.

14.3.4 Operations on Columns, Lines, and Pages

The subprograms in 14.3.4 provide for control of line and page structure. In the absence of the FILE parameter, they act on the console device. The exception STATUS_ERROR is raised by any of these subprograms if the file to be used is not open.

```
procedure NEW_LINE(FILE : in out FILE_TYPE;
                   SPACING : in POSITIVE_COUNT := 1);
procedure NEW_LINE(SPACING : in POSITIVE_COUNT := 1);
```

Operates on a file of mode OUT_FILE.

For a SPACING of one: Outputs a carriage-return line-feed. For a SPACING greater than one, the above action is performed SPACING times.

The exception MODE_ERROR is raised if the mode is not OUT_FILE.

```
procedure SKIP_LINE(FILE : in out FILE_TYPE;
                    SPACING : in POSITIVE_COUNT := 1);
procedure SKIP_LINE(SPACING : in POSITIVE_COUNT := 1);
```

Operates on a file of mode IN_FILE.

For a SPACING of one, reads and discards all characters until a line terminator had been read. For a SPACING greater than one, the above action is repeated SPACING times.

The exception MODE_ERROR is raised if the mode is not IN_FILE. The exception END_ERROR is raised if an attempt is made to read past the end of the file.

```
function END_OF_LINE(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_LINE return BOOLEAN;
```

Operates on a file of mode IN_FILE, and returns TRUE if a line terminator or file terminator is next.

The exception MODE_ERROR is raised if the mode is not IN_FILE.

```
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_FILE return BOOLEAN;
```

Operates on a file of mode IN_FILE, and returns TRUE if a file terminator is next.

The exception MODE_ERROR is raised if the mode is not IN_FILE.

14.3.5 Get and Put Procedures

The procedures GET and PUT for all types are defined in these sections. GET and PUT for characters and strings work on a character by character basis; GET and PUT for numeric types treat the items as lexical elements.

All of the GET and PUT procedures have forms with a FILE parameter. Where this parameter is omitted, the console device is assumed. GET procedures operate on IN_FILE files; PUT procedures operate on OUT_FILE files.

The exception STATUS_ERROR is raised by any of the procedures GET, GET_LINE, PUT, and PUT_LINE if the file to be used is not open. The exception MODE_ERROR is raised by GET and GET_LINE, and by PUT and PUT_LINE if the file mode is not IN_FILE or OUT_FILE, respectively.

The exception END_ERROR is raised if an attempt is made to read past the end of a file. The exception DATA_ERROR is raised if an input sequence is not a lexical element of the input type.

14.3.6 Input-Output of Characters and Strings

For an item of type CHARACTER the following procedures are provided:

```
procedure GET(FILE : in FILE_TYPE;
              ITEM : out CHARACTER);
procedure GET(ITEM : out CHARACTER);
```

After skipping any line terminators, reads the next character from the input file and returns the value in the **out** parameter ITEM.

The exception END_ERROR is raised if an attempt is made to skip a file terminator (except in console input).

```
procedure PUT(FILE : in FILE_TYPE;
              ITEM : in CHARACTER);
procedure PUT(ITEM : in CHARACTER);
```

Outputs the given character to the file.

For items of the type STRING the following procedures are provided:

```
procedure GET(FILE : in FILE_TYPE;
              ITEM : out STRING);
procedure GET(ITEM : out STRING);
```

Performs a number of character GET operations for successive characters of the string.

```
procedure PUT(FILE : in FILE_TYPE;
              ITEM : in STRING);
procedure PUT(ITEM : in STRING);
```

Performs the number of character PUT operations equal to the length of the string.

```
procedure GET_LINE(FILE : in FILE_TYPE;
                   ITEM : out STRING;
                   LAST : out NATURAL);
procedure GET_LINE(ITEM : out STRING;
                   LAST : out NATURAL);
```

Replaces successive characters of the string by successive characters from the file. Reading stops if the end of line is met or the end of the string is met. Characters not replaced are left undefined.

If characters are read, returns in LAST the index value of the last character replaced in the string.

The exception END_ERROR is raised if an attempt is made to read past the end of a file (except console input).

```
procedure PUT_LINE(FILE : in FILE_TYPE;
                   ITEM : in STRING);
procedure PUT_LINE(ITEM : in STRING);
```

Calls the procedures PUT(ITEM) and NEW_LINE(1).

14.3.7 Input-Output for Integer Types

The following procedures are defined in TEXT_IO for the type INTEGER. Values are output as decimal literals, without underlines or exponents, and preceded by a minus sign if negative.

```
procedure GET(FILE : in FILE_TYPE;
              ITEM : out INTEGER);
procedure GET(ITEM : out INTEGER);
```

Skips any leading blanks or line terminators, reads a minus sign (if present), then according to the syntax of an integer literal (except based numbers).

Returns, in the parameter ITEM, the value of type INTEGER that corresponds to the sequence input.

The exception DATA_ERROR is raised if the input is not syntactically correct.

```
procedure PUT(FILE : in FILE_TYPE;
              ITEM : in INTEGER);
procedure PUT(ITEM : in INTEGER);
```

Outputs the parameter ITEM as an integer literal, with no underlines, no exponent, and no leading zeroes (but a single zero for the value zero), and a preceding minus sign for a negative value.

14.3.8 Input-Output for Real Types

The following procedures are defined in the package TEXT_IO for the standard type FLOAT.

```
procedure GET(FILE : in FILE_TYPE;
              ITEM : out FLOAT);
procedure GET(ITEM : out FLOAT);
```

Skips any leading blanks and line terminators, reads a plus or minus sign (if present), then according to the syntax of a real literal (except based numbers).

Returns, in the parameter ITEM, the value of type FLOAT that corresponds to the input sequence.

The exception DATA_ERROR is raised if the sequence input does have the required syntax.

```
procedure PUT(FILE : in FILE_TYPE;
              ITEM : in FLOAT);
procedure PUT(ITEM : in FLOAT);
```

Outputs the value of the parameter ITEM as a decimal literal in the following format:

```
0.000000E+00
```

If the value is negative, a minus sign is included in the integer part.

14.3.10 Specification of the Package Text_IO

The following is a listing of the TEXT_IO specification as it appears on the distribution disk. Note: Because this file was read verbatim from the file textio.ada, reserved words in this section are not in bold face.

```
pragma print(on);
with IO_EXCEPTIONS; use IO_EXCEPTIONS;
package TEXT_IO is

   type FILE_MODE is (IN_FILE, OUT_FILE);

   type FILE_CONTROL_BLOCK is
         record
               DRIVE : CHARACTER;
               NAME : STRING(1..8);
               EXTENSION : STRING(1..3);
               EXTENT,S1,S2,RC : CHARACTER;
               DISK_MAP : STRING(1..16);
               CR : CHARACTER;
               RANDOM : STRING(1..3);
         end record;

   type OPEN_STATUS is (OPENED,CLOSED);

   type FILE_TYPE is
         record
               FCB : FILE_CONTROL_BLOCK;
               BUFFER : STRING(1..128);
               OPEN : OPEN_STATUS;
               DATA_POINTER : INTEGER;
               NEEDS_FLUSHED : BOOLEAN;
               ENDOFFILE : BOOLEAN;
               MODE : FILE_MODE;
         end record;

   type COUNT is range 0..INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;
   UNBOUNDED : constant COUNT := 0;

   subtype FIELD       is INTEGER range 0..10;
   subtype NUMBER_BASE is INTEGER range 2..16;
```

```
procedure CREATE(FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE;
                 NAME : in STRING);

procedure OPEN  (FILE : in out FILE_TYPE;
                 MODE : in FILE_MODE;
                 NAME : in STRING);

procedure CLOSE (FILE : in out FILE_TYPE);
procedure DELETE(FILE : in out FILE_TYPE);
procedure RESET (FILE : in out FILE_TYPE; MODE : in FILE_MODE);
procedure RESET (FILE : in out FILE_TYPE);

function MODE   (FILE : in FILE_TYPE) return FILE_MODE;
function NAME   (FILE : in FILE_TYPE) return STRING;

function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- Column, Line, and Page control

procedure NEW_LINE(FILE : in out FILE_TYPE;
                   SPACING : in POSITIVE_COUNT);
procedure NEW_LINE(FILE : in out FILE_TYPE);
procedure NEW_LINE(SPACING : in POSITIVE_COUNT);
procedure NEW_LINE;

procedure SKIP_LINE(FILE : in out FILE_TYPE;
                    SPACING : in POSITIVE_COUNT);
procedure SKIP_LINE(FILE : in out FILE_TYPE);
procedure SKIP_LINE(SPACING : in POSITIVE_COUNT);
procedure SKIP_LINE;

function END_OF_LINE(FILE : in FILE_TYPE) return BOOLEAN;

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

-- Character Input-Output

procedure GET(FILE : in out FILE_TYPE; ITEM : out CHARACTER);
procedure GET(ITEM : out CHARACTER);
procedure PUT(FILE : in out FILE_TYPE; ITEM : in CHARACTER);
procedure PUT(ITEM : in CHARACTER);
```

```
    -- String Input-Output

    procedure GET(FILE : in out FILE_TYPE; ITEM : out STRING);
    procedure GET(ITEM : out STRING);
    procedure PUT(FILE : in out FILE_TYPE; ITEM : in STRING);
    procedure PUT(ITEM : in STRING);

    procedure GET_LINE(FILE : in out FILE_TYPE;
                       ITEM : out STRING; last : out NATURAL);
    procedure GET_LINE(ITEM : out STRING; last : out NATURAL);
    procedure PUT_LINE(FILE : in out FILE_TYPE; ITEM : in STRING);
    procedure PUT_LINE(ITEM : in STRING);

    -- Input-Output of Integer Types

    procedure GET(FILE : in out FILE_TYPE; ITEM : out INTEGER);
    procedure GET(ITEM : out INTEGER);
    procedure PUT(FILE : in out FILE_TYPE; ITEM : in INTEGER);
    procedure PUT(ITEM : in INTEGER);

    -- Input-Output of Real Types

    procedure GET(FILE : in out FILE_TYPE; ITEM : out FLOAT);
    procedure GET(ITEM : out FLOAT);
    procedure PUT(FILE : in out FILE_TYPE; ITEM : in FLOAT);
    procedure PUT(ITEM : in FLOAT);

end text_io;
```

14.4 Exceptions in Input-Output

The exceptions which can be raised by input-operations are declared in the package IO_EXCEPTIONS (see 14.5).

The corresponding section of the Ada LRM describes the conditions under which input-output exceptions are raised. Only LAYOUT_ERROR is not raised by COL, LINE or PAGE (since these do not exist), and MODE_ERROR is not raised by SET_INPUT, SKIP_PAGE, etc. (because these also do not exist).

14.5 Specification of the package IO_Exceptions

The following package defines the exceptions needed by TEXT_IO, and is included on the distribution disk. Note: Because this is also a verbatim listing, reserved words do not appear in bold face.

```
pragma print(on);
package io_exceptions is

   status_error : exception;
   mode_error   : exception;
   name_error   : exception;
   use_error    : exception;
   device_error : exception;
   end_error    : exception;
   data_error   : exception;
   layout_error : exception;

end io_exceptions;
```

Annex A
Predefined Language Attributes

P'ADDRESS
: For a prefix P that denotes an object, this attribute returns a number corresponding to the first storage unit occupied by P. Overloaded on all predefined integer types. This attribute is useful to pass pointers to strings and file objects using the BDOS call (see Appendix F).

P'FIMAGE
: For a prefix P that denotes a floating point type, this attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the predefined type STRING. The result is the image of the value of X, without underlines, containing one leading zero, a decimal point, six digit mantissa, and exponent.

P'FIRST
: The minimum value of P.

P'FIRST
: If P is a constrained array or subtype, or an array object, P'FIRST is the lower bound of the first index.

P'FIRST(J)
: Similarly, the lower bound of the J'th index.

P'IMAGE
: For a prefix P that denotes an integer type, this attribute is a function with a single parameter X, which must be of the base type of P. The result type is the predefined type STRING. The result is the image of the value of X, without leading zeroes, exponent, or trailing spaces, but with a one character prefix that is either a minus sign or a space.

P'LAST
: The maximum value of P.

P'LAST
: If P is a constrained array or subtype, or an array object, P'LAST is the upper bound of the first index.

P'LAST(J)
: Similarly, the upper bound of the J'th index.

P'LENGTH
: If P is a constrained array type or subtype, or an array object, P'LENGTH is the number of elements in the first dimension of P.

P'LENGTH(J)
: Similarly, the number of elements in the J'th dimension.

P'POS
: If X is a value of type P, P'POS(X) is the integer position of X in the ordered sequence of values P'FIRST..P'LAST, the position of P'FIRST being itself for integer types and zero for enumeration types.

P'VAL
: If J is an integer, P'VAL(J) is the value of enumeration type P whose POS is J.

Annex B
Predefined Language Pragmas

Pragma	Meaning
OPTIMIZE	Takes TIME, SPACE or OFF as argument. This pragma can only appear in a declarative part and it applies to the block or body enclosing the declarative part. It specifies whether time or space is the primary optimization criterion. The compiler can be directed to perform no optimization by using OFF as the argument. SPACE is the default optimization; TIME instructs the compiler to do as much intermediate code optimization as possible. Use this when compiling benchmark programs.
PRINT	Takes ON or OFF as argument. This pragma may appear anywhere and directs the compiler to output the source text as it is being processed by the different compiler phases. If these listings are not desired, it is suggested that they not be used. Compile time increases considerably as the source text must be included with the code passed from phase to phase.
SYSTEM	Takes 8080, 8085, or Z80 as argument. This pragma can appear anywhere and establishes the name of the object machine. When code generators are created for different machines, this pragma will affect the compiler output; until then the CP/M 8080 code generator is always used.
INCLUDE	Takes a string as argument, which is the name of an Ada text file. This pragma may appear anywhere and causes the text file to be included where the pragma is used. If a disk letter is not specified in the name, the default drive is assumed. Also, the print pragma will have no effect on the statements of a text file that has been included; to view these statements another print pragma must be used within the included text file itself. The include pragma has no "nesting limit" (but beware of "cyclic" includes!)
RECURSION	Takes ON or OFF as argument. Normally, all Ada programs are recursive. Turning recursion off allows the compiler to store data in a local data area as opposed to a dynamic stack, thus dramatically increasing the speed of the resulting object code. Use this pragma when compiling non-recursive benchmark programs. The pragma can be turned on and off within a program, so selected embedded subprograms may use recursion.

Annex C
Predefined Language Environment

```
package STANDARD is

   type BOOLEAN is (FALSE, TRUE);

   -- The predefined relational operators for this type are as
   -- follows:

   function "="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   function "<"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   function ">"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

   -- The predefined logical operators and the predefined negation
   -- operator are defined as follows:

   function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   function "or"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

   function "not" (RIGHT : BOOLEAN) return BOOLEAN;

   -- The universal type universal_integer is predefined.

   type INTEGER is range -32768 .. 32767;

   function "="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
   function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
   function "<"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
   function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
   function ">"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
   function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;

   function  "+" (RIGHT : INTEGER) return INTEGER;
   function  "-" (RIGHT : INTEGER) return INTEGER;
   function "abs"(RIGHT : INTEGER) return INTEGER;

   function  "+" (LEFT, RIGHT : INTEGER) return INTEGER;
   function  "-" (LEFT, RIGHT : INTEGER) return INTEGER;
   function  "*" (LEFT, RIGHT : INTEGER) return INTEGER;
   function  "/" (LEFT, RIGHT : INTEGER) return INTEGER;
   function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
   function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

   function "**"  (LEFT, RIGHT : INTEGER) return INTEGER;
```

```
-- The universal type universal_float is predefined.

type FLOAT is digits 6 range -3E31 .. 1E38;

-- The predefined operators for this type are as follows:

function "="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
function "/=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
function "<"  (LEFT, RIGHT : FLOAT) return BOOLEAN;
function "<=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
function ">"  (LEFT, RIGHT : FLOAT) return BOOLEAN;
function ">=" (LEFT, RIGHT : FLOAT) return BOOLEAN;

function  "+" (RIGHT : FLOAT) return FLOAT;
function  "-" (RIGHT : FLOAT) return FLOAT;
function "abs"(RIGHT : FLOAT) return FLOAT;

function  "+"  (LEFT, RIGHT : FLOAT) return FLOAT;
function  "-"  (LEFT, RIGHT : FLOAT) return FLOAT;
function  "*"  (LEFT, RIGHT : FLOAT) return FLOAT;
function  "/"  (LEFT, RIGHT : FLOAT) return FLOAT;

function "**"  (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;

type CHARACTER is

    (nul, soh, stx, etx,   eot, enq, ack, bel,
     bs,  ht,  lf,  vt,    ff,  cr,  so,  si,
     dle, dc1, dc2, dc3,   dc4, nak, syn, etb,
     can, em,  sub, esc,   fs,  gs,  rs,  us,

     ' ', '!', '"', '#',   '$', '%', '&', ''',
     '(', ')', '*', '+',   ',', '-', '.', '/',
     '0', '1', '2', '3',   '4', '5', '6', '7',
     '8', '9', ':', ';',   '<', '=', '>', '?',

     '@', 'A', 'B', 'C',   'D', 'E', 'F', 'G',
     'H', 'I', 'J', 'K',   'L', 'M', 'N', 'O',
     'P', 'Q', 'R', 'S',   'T', 'U', 'V', 'W',
     'X', 'Y', 'Z', '[',   '\', ']', '^', '_',

     '`', 'a', 'b', 'c',   'd', 'e', 'f', 'g',
     'h', 'i', 'j', 'k',   'l', 'm', 'n', 'o',
     'p', 'q', 'r', 's',   't', 'u', 'v', 'w',
     'x', 'y', 'z', '{',   '|', '}', '~', del);

for CHARACTER use  -- 128 ASCII character set without holes
    (0, 1, 2, 3, 4, 5, ..., 125, 126, 127);
```

```
-- Predefined subtypes:

subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;

-- Predefined string type:

type STRING is array(POSITIVE range <>) of CHARACTER;

-- The predefined operators for this type are as follows:

function "="  (LEFT, RIGHT : STRING) return BOOLEAN;
function "/=" (LEFT, RIGHT : STRING) return BOOLEAN;
function "<"  (LEFT, RIGHT : STRING) return BOOLEAN;
function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
function ">"  (LEFT, RIGHT : STRING) return BOOLEAN;
function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;

function "&" (LEFT, RIGHT : STRING) return STRING;

-- other definitions:

procedure BDOS(C : INTEGER; DE : INTEGER);
procedure BDOS(C : INTEGER; FILE : FILE := UFCB);
function BDOS(C : INTEGER; DE : INTEGER) return INTEGER;
function BDOS(C : INTEGER; FILE : FILE := UFCB) return INTEGER;

function RND(SEED : FLOAT) return FLOAT;

end STANDARD;
```

Appendix D
Glossary

The following glossary is a list of Ada terms as they apply to Maranatha A. For a complete Ada glossary see Appendix D of the Ada Reference Manual.

Attribute - An attribute is a predefined characteristic of a named entity.

Body - A body is a program unit defining the execution of a subprogram.

Compilation Unit - a compilation unit is a program unit presented for compilation as an independent text. A compilation unit in Maranatha A can be a subprogram body, package body or package specification.

Component - A component denotes a part of a composite object. An indexed component is a name containing expressions denoting indices, and names a component in an array. A selected component is the identifier of the component, prefixed by the name of the entity of which it is a component.

Composite type - An object of a composite type comprises several components. An array type is a composite type, all of whose components are of the same type and subtype; the individual components are selected by their indices. A record type is a composite type whose components may be of different types; the individual components are selected by their identifiers.

Constraint - a constraint is a restriction on the set of possible values of a type. A range constraint specifies lower and upper bounds of the values of a scalar type. An index constraint specifies lower and upper bounds of an array index.

Declarative Part - A declarative part is a sequence of declarations and related information such as subprogram bodies that apply over the region of a program text.

Derived Type - A derived type is a type whose operations and values are taken from those of an existing type.

Discrete Type - A discrete type has an ordered set of distinct values. The discrete types are the enumeration and integer types. Discrete types may be used for indexing and iteration, and for choices in case statements.

Elaboration - Elaboration is the process by which a declaration achieves its effect. For example it can associate a name with a program entity or initialize a newly declared variable.

Entity - An entity is anything that can be named or denoted in a program. Objects, types, values, program units, are all entities.

Enumeration Type - An enumeration type is a discrete type whose values are given explicitly in the type declaration. In Maranatha A these values can only be identifiers.

Expression - An expression is a part of a program that computes a value.

Introduce - An identifier is introduced by its declaration at the point of its first occurence.

Lexical Unit - A lexical unit is one of the basic syntactical elements making up a program. A lexical unit is an identifier, a number, a character literal, a string, a delimiter, or a comment.

Literal - A literal denotes an explicit value of a given type, for example a number, an enumeration value, a character, or a string.

Object - An object is a variable or a constant. An object can denote any kind of data element, whether a scalar or a composite value.

Overloading - Overloading is the property of literals and identifiers that can have several meanings within the same scope. For example an overloaded enumeration literal is a literal appearing in two or more enumeration types; an overloaded subprogram is a subprogram whose designator can denote one of several subprograms, depending upon the names and types of its parameters and its return type.

Parameter - A parameter is one of the named entities associated with a subprogram. A formal parameter is an identifier used to denote the named entity in the unit body. An actual parameter is the particular entity associated with the corresponding formal parameter in a subprogram call. A parameter mode specifies whether the parameter is used for input, output, or input-output of data.

Pragma - A pragma is an instruction to the compiler, and may be language defined or implementation defined.

Qualified Expression - A qualified expression is an expression qualified by the name of a type or subtype. It can be used to state the type or subtype of an expression, for example for an overloaded literal.

Range - A range is a contiguous set of values of a scalar type. A range is specified by giving the lower and upper bounds for the values.

Scalar Types - A scalar type is a type whose values have no components. Scalar types comprise discrete types (that is, enumeration and integer types) and real types.

Scope - The scope of a declaration is the region of text over which the declaration has an effect.

Subprograms - A subprogram is an executable program unit, possibly with parameters for communication between the subprogram and its point of call. A subprogram body specifies the execution of a subprogram. A subprogram may be a procedure, which performs an action, or a function, which returns a result.

Subtype - A subtype of a type is obtained from the type by constraining the set of possible values of the type. The operations over a subtype are the same as those of the type from which the subtype was obtained.

Type - A type characterizes a set of values and a set of operations applicable to those values. A type definition is a language construct introducing a new type. A type declaration associates a name with a type introduced by a type definition.

Visibility - At a given point in a program text, the declaration of an entity with a certain identifier is said to be visible if the entity is an acceptable meaning for an occurrence at that point of the identifier.

Appendix E
Syntax Summary

The following describes the implemented syntax of the Maranatha A compiler using a simple variant of Backus-Naur Form (See page 1-5 of the Ada Reference Manual).

2.1

```
graphic_character ::= basic_graphic_character
  | lower_case_letter | other_special_character

basic_graphic_character ::=
    upper_case_character | digit
  | special_character    | space_character

basic_character ::=
    basic_graphic_character | format_effector
```

2.3

```
identifier ::=
  letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter
```

2.4

```
numeric_literal ::= decimal_literal | based_literal
```

2.4.1

```
decimal_literal ::= integer [.integer] [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | E - integer
```

2.4.2

```
based_literal ::= base # based_integer [.based_integer] # [exponent]

base ::= integer

based_integer ::= extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter
```

2.5

character_literal ::= 'graphic_character'

2.6

string_literal ::= "{graphic_character}"

2.8

```
pragma ::=
  pragma identifier (argument_association);

argument_association ::= name
```

3.1

```
basic_declaration ::=
    object_declaration  | type_declaration
  | subtype_declaration | subprogram_declaration
  | package_declaration | exception_declaration
```

3.2

```
object_declaration ::=
    identifier_list : [constant] subtype_indication [:= expression];
  | identifier_list : [constant] constrained_array_definition [:= expression];

identifier_list ::= identifier {, identifier}
```

3.3.1

```
type_declaration ::= full_type_declaration

full_type_declaration ::= type identifier is type_definition;

type_definition ::=
    enumeration_type_definition | integer_type_definition
  | array_type_definition       | record_type_definition
  | derived_type_definition
```

3.3.2

subtype_declaration ::= **subtype** identifier **is** subtype_indication;

subtype_indication ::= type_mark [constraint]

type_mark ::= *type*_name | *subtype*_name

constraint ::= range_constraint | index_constraint

3.4

derived_type_definition ::= **new** subtype_indication

3.5

range_constraint ::= **range** range

range ::= simple_expression .. simple_expression

3.5.1

enumeration_type_definition ::=
  (enumeration_literal_specification
    {, enumeration_literal_specification})

enumeration_literal_specification ::= enumeration_literal

enumeration_literal ::= identifier

3.5.4

integer_type_definition ::= range_constraint

3.6

array_type_definition ::=
  unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
  array(index_subtype_definition {, index_subtype_definition}) **of**
      *component*_subtype_indication

constrained_array_definition ::=
  array index_constraint **of** *component*_subtype_indication

index_subtype_definition ::= type_mark **range** <>

index_constraint ::= (discrete_range {, discrete_range})

discrete_range ::= *discrete*_subtype_indication | range

3.7

```
record_type_definition ::=
  record
    component_list
  end record

component_list ::= component_declaration {component_declaration}
                 | null;

component_declaration ::=
  identifier_list : component_subtype_definition;

component_subtype_definition ::= subtype_indication
```

3.7.3

```
choice ::= simple_expression | discrete_range | others
```

3.9

```
declarative_part ::=
     {basic_declarative_item} {later_declarative_item}

basic_declarative_item ::=
  basic_declaration | representation_clause | use_clause

later_declarative_item ::= body
  | subprogram_declaration | package_declaration | use_clause

body ::= proper_body | body_stub

proper_body ::= subprogram_body | package_body
```

4.1

```
name ::= simple_name
   | character_literal | indexed_component
   | selected_component | attribute

simple_name ::= identifier

prefix ::= name | function_call
```

4.1.1

```
indexed_component ::= prefix(expression {, expression})
```

4.1.3

```
selected_component ::= prefix.selector

selector ::= simple_name
```

4.1.4

attribute ::= prefix'attribute_designator

attribute_designator ::= simple_name [(*universal_static*_expression)]

4.4

```
expression ::=
     relation {and relation} | relation {and then relation}
   | relation {or relation}  | relation {or else relation}
   | relation {xor relation}

relation ::=
     simple_expression [relational_operator simple_expression]
   | simple_expression [not] in range
   | simple_expression [not] in type_mark

simple_expression ::=
     [unary_adding_operator] term {binary_adding_operator term}

term ::=
     factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=
     numeric_literal | string_literal | name | function_call
   | type_conversion | qualified_expression | (expression)
```

4.5

logical_operator ::= **and** | **or** | **xor**

relational_operator ::= = | /= | < | <= | > | >=

binary_adding_operator ::= + | - | &

unary_adding_operator ::= + | -

multiplying_operator ::= * | / | **mod** | **rem**

highest_precedence_operator ::= ** | **abs** | **not**

4.6

type_conversion ::= type_mark (expression)

4.7

qualified_expression ::= type_mark'(expression)

5.1

```
sequence_of_statements ::= statement {statement}

statement ::=
     {label} simple_statement | {label} compound_statement

simple_statement ::= null_statement
   | assignment_statement | procedure_call_statement
   | exit_statement       | return_statement
   | goto_statement       | raise_statement

compound_statement ::=
    if_statement   | case_statement
  | loop_statement | block_statement

label ::= <<label_simple_name>>

null_statement ::= null;
```

5.2

```
assignment_statement ::= variable_name := expression;
```

5.3

```
if_statement ::=
     if condition then
           sequence_of_statements
     { elsif condition then
           sequence_of_statements }
     [ else
           sequence_of_statements ]
     end if;

condition ::= boolean_expression
```

5.4

```
case_statement ::=
  case expression is
     case_statement_alternative
    {case_statement_alternative}
  end case;

case_statement_alternative ::=
  when choice {| choice} =>
    sequence_of_statements
```

5.5

```
loop_statement ::=
   [loop_simple_name:]
      [iteration_scheme] loop
         sequence_of_statements
      end loop [loop_simple_name];

iteration_scheme ::=
    while condition
  | for loop_parameter_specification

loop_parameter_specification ::=
   identifier in [reverse] discrete_range
```

5.6

```
block_statement ::=
     [block_simple_name:]
      [declare
            declarative_part]
      begin
            sequence_of_statements
     [exception
          exception_handler
        {exception_handler}]
      end [block_simple_name];
```

5.7

```
exit_statement ::= exit [loop_name] [when condition];
```

5.8

```
return_statement ::= return [expression];
```

5.9

```
goto_statement ::= goto label_name;
```

6.1

```
subprogram_declaration ::= subprogram_specification;

subprogram_specification ::=
        procedure identifier [formal_part]
      | function designator [formal_part] return type_mark

designator ::= identifier

formal_part ::= (parameter_specification {; parameter_specification})

parameter_specification ::= identifier_list : mode type_mark

mode ::= [in] | in out | out
```

6.3

```
subprogram_body ::=
      subprogram_specification is
        [ declarative_part ]
      begin
        sequence_of_statements
      [exception
           exception_handler
          {exception_handler}]
      end [designator];
```

6.4

```
procedure_call_statement ::=
      procedure_name [actual_parameter_part];

function_call ::=
      function_name [actual_parameter_part]

actual_parameter_part ::=
      (parameter_association {, parameter_association})

parameter_association ::= actual_parameter

actual_parameter ::=
   expression | variable_name | type_mark(variable_name)
```

7.1

```
package_declaration ::= package_specification;

package_specification ::=
  package identifier is
      {basic_declarative_item}
  end [package_simple_name]

package_body ::=
  package body package_simple_name is
      [declarative_part]
[ begin
      sequence_of_statements
[ exception
      exception_handler
     {exception_handler}]]
  end [package_simple_name];
```

8.4

```
use_clause ::= use package_name {,package_name};
```

10.1

```
compilation ::= compilation_unit

compilation_unit ::=
   context_clause library_unit | context_clause secondary_unit

library_unit ::= package_declaration |  subprogram_body

secondary_unit ::= library_unit_body | subunit

library_unit_body ::= subprogram_body | package_body
```

10.1.1

```
context_clause ::= {with_clause {, use_clause}}

with_clause ::= with unit_simple_name {, unit_simple_name};
```

10.2

```
body_stub ::= subprogram_specification is separate;

subunit ::= separate (parent_unit_name) proper_body
```

11.1

exception_declaration ::= identifier_list : **exception**;

11.2

exception_handler ::=
  when exception_choice {| exception_choice} =>
    sequence_of_statements

exception_choice ::= *exception*_name | **others**

11.3

raise_statement ::= **raise** [*exception*_name];

13.1

representation_clause ::= address_clause

13.5

address_clause ::= **for** simple_name **use at** integer_literal;

Appendix F
Implementation Dependent Characteristics

Three areas of interest particular to the Maranatha implementation of the Ada programming language are covered in this appendix. These are the predefined subprograms BDOS and RND, and a peculiarity with indexes.

BDOS

Direct interface to the CP/M operating system is allowed through the predefined subprogram BDOS. Formally, it is defined as

```
procedure BDOS(C : INTEGER; DE : INTEGER);
procedure BDOS(C : INTEGER; FILE : FILE := UFCB);
function BDOS(C : INTEGER; DE : INTEGER) return INTEGER;
function BDOS(C : INTEGER; DE : FILE := UFCB) return INTEGER;
```

BDOS has two parameters. The first is an integer which is the function number, stored in the C register. The second is a default parameter and is stored in the DE register pair. This could be a file control block or another integer passed to the operating system. The second parameter can be omitted if it is not needed as some CP/M calls do not require it. If it is required and not supplied, it defaults to the user file control block (UFCB) at 16#5C#, thus allowing Ada programs to use bdos calls to access files in the default CP/M file control block. The function BDOS returns an integer from the HL register pair for calls returning a value. BDOS can also be called as a procedure in which case any return value is ignored. Great care should be taken in the use of this function as it is particular to this implementation and while BDOS conforms to the Ada syntax it is not an inherent part of the language. It should be avoided if possible.

Because BDOS will only allow integers for the second operand, passing a string to the BDOS call can be done by using the ADDRESS attribute.

RND

For users requiring a random number generator, one is supplied as part of the package STANDARD and is implemented as

```
function RND(SEED : FLOAT) return FLOAT;
```

The function returns a random value between 0.0 and 1.0. Seed > 1.0 restarts the random number generator with the given seed. Seed = 0.0 returns the next value in the sequence.

ARRAY INDEXES

Given the following:

```
type MEMORY is array(INTEGER) of CHARACTER;
for MEMORY use at 0;
```

The array memory would be an array consisting of 65536 elements of the type character. Of course, this much memory is not available to create such an array in the CP/M environment. Maranatha A will actually create an array of components zero bytes long (or 65536, which in the 8080 "wraps around" to 0). This is due to the method in which array lengths are computed. Users may define an array consisting of essentially their memory configuration, allowing access to any memory location.