THESIS

ADAMEASURE
AN ADA® SOFTWARE METRIC

by

Jeffrey L. Nieder
and
Karl S. Fairbanks, Jr.

March 1987

Thesis Advisor: Daniel L. Davis

Approved for public release; distribution is unlimited
Ada is a registered trademark of the U.S. Government (AJPO)
DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.
ADA Measure, Ada, Software Metrics

Software metrics is in its infancy, is a very new field and there are many scientists and educators who would choose to keep it that way. Software metrics is held in low esteem by many software engineers. There is much debate and argument about what a good metric should do, how it should work, and what it should produce for the user. Yet with the cost of software rising, it is becoming more and more critical for future cost control and expense management that some automatic metric tool be implemented. Interest is growing, as evidenced by the Department of Defense expressing a strong interest in the development of an effective and reliable metric tool. It is no doubt that if this can be accomplished the automated software metric would be a valuable asset to the software engineering effort.
Approved for public release; distribution is unlimited.

AdaMeasure
An Ada Software Metric

by

Jeffrey L. Nieder
Lieutenant Commander, United States Navy
B.S.A.S., Miami University of Ohio, 1976

and

Karl S. Fairbanks, Jr.
Lieutenant, United States Navy
B.S., United States Naval Academy, 1981

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1987

Authors:  

Jeffrey L. Nieder

Karl S. Fairbanks, Jr.

Approved by:  

Daniel L. Davis, Thesis Advisor

Gordon H. Bradley, Second Reader

Vincent Y. Lum, Chairman, Department of Computer Science

Kneale T. Marshall, Dean of Information and Policy Sciences
ABSTRACT

Software engineers in general, and the Department of Defense in particular, are looking for good software metrics to aid in software development. Maurice Halstead developed the theory of Software Science which includes the relation between program complexity and program length. Halstead's length metric deals with the properties of an algorithm that can be measured, either directly or indirectly, statically or dynamically, and with the relationships among these properties. A system has been developed which implements Halstead's length metric. This system, which is written in Ada, takes Ada programs as input, and outputs the length metric complexity analysis. Finally, recommendations for future work in this area are made.
# TABLE OF CONTENTS

## I. INTRODUCTION AND BACKGROUND ........................................ 7
   A. CONCERNS .................................................................. 7
   B. AVAILABLE METRICS .................................................. 8
   C. HALSTEAD .................................................................. 10
   D. OUR METRIC ............................................................. 11

## II. DEVELOPING AN ADA GRAMMAR .............................................. 13
   A. INTRODUCTION .......................................................... 13
   B. GRAMMAR FOR THE ADA LANGUAGE ............................. 14

## III. LEXICAL ANALYZER ............................................................. 18
   A. INTRODUCTION .......................................................... 18
   B. TOKENS .................................................................... 18
      1. Identifiers ................................................................ 19
      2. String Literal .......................................................... 19
      3. Character Literals ................................................... 20
      4. Comments .................................................................. 21
      5. Separators .................................................................. 21
      6. Delimiters .................................................................. 22
      7. Numeric Literal .......................................................... 23
   C. TOKEN USE ............................................................... 23

## IV. PARSER ........................................................................... 26
   A. INTRODUCTION .......................................................... 26
   B. TOP-DOWN RECURSIVE DESCENT PARSING .................... 27

## V. ADAMEASURE ................................................................... 29
   A. INTRODUCTION .......................................................... 29
   B. DATA COLLECTION ....................................................... 29
      1. Halstead Data ....................................................... 29
2. Comment Data .............................................. 30
3. Nesting Data .............................................. 30

VI. CONCLUSIONS ............................................. 31
   A. METRICS .............................................. 31
   B. IMPLEMENTATION .................................. 31
   C. THE FUTURE ........................................ 32

APPENDIX A: MODIFIED ADA GRAMMAR ......................... 34
APPENDIX B: 'ADAMEASURE' USERS GUIDE ...................... 42
APPENDIX C: 'ADAMEASURE' PROGRAM LISTING - PART 1 ....... 46
APPENDIX D: 'ADAMEASURE' PROGRAM LISTING - PART 2 ....... 92
APPENDIX E: 'ADAMEASURE' PROGRAM LISTING - PART 3 ...... 129
LIST OF REFERENCES ............................................. 163
INITIAL DISTRIBUTION LIST .................................... 164
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Finite State Machine for Identifiers</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Finite State Machine for String Literals</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Finite State Machine for Character Literals</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Finite State Machine for Comments</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>Finite State Machine for Separators</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Finite State Machine for Delimiters</td>
<td>22</td>
</tr>
<tr>
<td>3.7</td>
<td>Finite State Machine for Numeric Literals</td>
<td>24</td>
</tr>
</tbody>
</table>
I. INTRODUCTION AND BACKGROUND

A. CONCERNS

With computer software programs getting larger and larger all the time, the search is on for accurate and dependable aids for the developer to increase the productivity and efficiency of software engineering efforts. New tools and new methodologies are being sought in the effort to alleviate the "software crisis". This crisis, stated specifically, is that software is being delivered late, over budget, specifications are not being met, modifications are difficult and expensive, unresponsive to user needs, and unreliable.

Recently, it was reported that software costs are growing at the rate of 15% per year while productivity is increasing at less than 3% [Ref. 1: p.15]. Barry Boehm, a leading computer expert, asserted more than 10 years ago that, in the military application area, the cost of software was expected to reach about 80% of the total computer system budget by the year 1985 [Ref. 2: p.1]. His assertion now seems valid [Ref. 3]. Furthermore it appears that up to 90% of the total software budget for all organizations using computers is being devoted to maintenance [Ref. 2: p.2]. The Office of Naval Research and other Defense Research agencies are aware of these increasing costs. They are also acutely aware of the lack of quantitative measurement techniques which are desperately needed for assessing the quality and reliability of software as well as for the prediction and measurement of software production [Ref. 1: p.14]. Further evidence of this universal concern comes from a General Accounting Office report of June 78 on managing weapons systems. It stated that there exists no Department of Defense performance criteria to measure software quality and to establish a basis for its acceptance or rejection [Ref. 1: p.15]. The Secretary of Defense's response was brief and candid,

We concur. We regret and underscore the importance of the need. The Department of Defense will quickly embrace such measures when they are available.

The current Office of Naval Research (ONR) initiative to focus on software measurement is a result of the need for such metrics and the high level of interest that
the Secretary of Defense brings to bear. The ONR initiatives, stated specifically, are:
developing indices of merit that can support quantitative comparisons and evaluations;
designing a philosophical framework for understanding and defining software measurement; and focusing the attention of the scientific community on computer software. [Ref. 1: p. 15].

B. AVAILABLE METRICS

Software metrics are often classified as either process metrics or product metrics, and are applied to either the development process or the software product being developed [Ref. 2: p. 19]. Process metrics include resource metrics, such as the experience of programmers, and the cost of development and maintenance. Examples of metrics for the levels of personnel experience are the number of years that a team has been using a programming language, the number of years that a programmer has been with the organization, the number of years that a programmer has been associated with a programming team, and the number of years of experience constructing similar software [Ref. 2: p. 19]. Other factors considered in process metric measurement are development techniques (such as top-down or bottom-up development techniques, and structured programming), supervisory techniques (such as type of team organization and number of communication paths), and resources (human, computer, time schedule, and so on) [Ref. 2: p. 20]. Product metrics, on the other hand, are a measure of the software product. Product metrics include the size of the product (such as number of lines of code or some count of tokens in the program), the logic structure complexity (such as flow of control, depth of nesting, or recursion), the data structure complexity (such as the number of variables used), the function (such as type of software: business, scientific, systems, and so on), and combinations of these [Ref. 2: p. 20].

The emphasis in this thesis will be on product metrics to the total exclusion of process metric issues. We are interested in analyzing the static program, or product, in our effort to provide an automated tool.

There are a variety of different quantitative software metrics in use today. In an important paper by Boehm [Ref. 4], an attempt is made to define software quality in terms of some high level characteristics such as reliability, portability, efficiency, human engineering, testability, understandability, and modifiability. If we can define these characteristics, noting that a precise subjective definition is difficult to achieve.
and measure these characteristics with some precision, we could strive to maximize each of these characteristics [Ref. 2: p.7]. There are some difficulties here. First, some of the characteristics are potentially contradictory. For example, improvements in portability and understandability usually result in decreased efficiency [Ref. 2: p.7]. Secondly, there are significant cost-benefit tradeoffs. For example, the cost of producing highly reliable code may be several times more costly, in terms of time and money, than for less reliable code [Ref. 2: p.7].

The measurement of software complexity is receiving increased attention in recent years. Complexity has been a loosely defined term but Bill Curtis defined complexity to be a characteristic of the software interface which influences the resources another system will expend or commit while interfacing with the software [Ref. 5]. Two separate and distinct focuses have emerged in studying software complexity: computational and psychological complexity [Ref. 1: p.208]. Computational complexity relies on the formal mathematical analysis of such problems as algorithm efficiency and use of machine resources. In contrast, the empirical study of psychological complexity has emerged from the understanding that software development and maintenance are largely human activities. Psychological complexity is concerned with the characteristics of software which affect programmer performance [Ref. 1: p.208]. This thesis will focus entirely on program complexity in an effort to provide a representative metric from selected quantitative measures.

There are a variety of complexity metrics available and we will briefly highlight a few of them. A number of metrics having a base in graph theory have been proposed to measure complexity from control flow [Ref. 1: p.210]. Thomas McCabe devised one of the better known complexity metrics in relation to the decision structure of a program [Ref. 6]. McCabe argues that his metric assesses the difficulty of testing a program, since it is a representation of the control paths that must be exercised during testing [Ref. 1: p.210]. Victor Basili and Robert Reiter [Ref. 7], have developed different counting methods for computing cyclomatic complexity by counting rules for case statements and compound predicates [Ref. 1: p.210]. Definitive data on the most effective counting rules has yet to be presented. The best known and most thoroughly studied of the composite measures of complexity is Halstead's theory of Software Science [Ref. 1: p.211]. In 1972, Maurice Halstead argued that algorithms have measurable characteristics analogous to physical laws. We will focus this thesis on Halstead’s theory as the representative metric we implement. We will first look at Halstead’s theory.
C. HALSTEAD

Halstead’s software science theory applies the scientific method to the properties and structure of computer programs. It attempts to provide precise, objective measures of the complexity of existing software, which is then used to predict the length of the programs [Ref. 9: p.3]. Numerous statistical studies have shown very high correlations between the theory’s predictions and actual program measures such as mean number of bugs [Ref. 8: p.85]. Halstead defined four basic measures:

1. \( n_1 \): The number of distinct operators appearing in a program.
2. \( n_2 \): The number of distinct operands appearing in a program.
3. \( N_1 \): The total number of occurrences of the operators in a program.
4. \( N_2 \): The total number of occurrences of the operands in a program.

Halstead defined the size of the \textit{vocabulary} to be the operators plus the operands as in Equation 1.1.

\[
n = n_1 + n_2
\]  \hspace{1cm} \text{(eqn 1.1)}

Halstead’s theory [Ref. 8: p.11], says that actual program length can be calculated by adding the total number of operand references with the total number of operator references as in Equation 1.2.

\[
N = N_1 + N_2
\]  \hspace{1cm} \text{(eqn 1.2)}

Halstead, using information theory, computes the theoretical length or predicted length as in Equation 1.3.

\[
N = n_1 \times (\log_2 (n_1)) + n_2 \times (\log_2 (n_2))
\]  \hspace{1cm} \text{(eqn 1.3)}

Halstead also speaks of program volume as in Equation 1.4.

\[
V = N \log_2 n
\]  \hspace{1cm} \text{(eqn 1.4)}

The intuition is simple. For each of the \( N \) elements of a program, \( \log_2 (n) \) bits must be specified to choose one of the operators or operands for that element, thus, volume \( (V) \) measures the number of bits required to specify a program [Ref. 8: p.19].
Halstead also hypothesized a conservation law between the level of abstraction and the volume. The level is defined as the ratio of potential to actual volume where the potential volume is the volume of the most compact (highest-level) representation of the algorithm [Ref. 8: p.25]. Effort, another variable that Halstead suggests, is a measure of the mental effort required to create a program. He describes effort as the ratio of volume to program level which implies that programming difficulty increases as the volume of the program increases, and decreases as program level increases [Ref. 8: p.47]. Halstead hypothesized that programming Time (T) should be directly proportional to the Effort (E) in a program, as in Equation 1.5.

\[ T = E \cdot S \quad \text{(eqn 1.5)} \]

The constant S represents the Speed of a programmer, i.e., the number of mental discriminations per second of which he/she is capable [Ref. 3: p.48].

We agree with Alan Perlis, that regardless of the empirical support for many of Halstead’s predictions, the theoretical basis for his metrics needs considerable attention [Ref. 1: p.214]. Halstead, more than other researchers, tried to integrate theory from both computer science and psychology. Unfortunately, some of the psychological assumptions underlying his work are difficult to justify for the phenomena to which he applied them [Ref. 1: p.214]. Perlis states, and again we agree, that computer scientists would do well to purge from their memories the magic number 7 + or - 2, and the Stroud number of 18 mental discriminations per second. These numbers describe cognitive processes related to the perception or retention of simple stimuli, rather than the complex information processing tasks involved in programming [Ref. 1: p.214]. Broadbent [Ref. 10], argues that for complicated tasks (such as understanding a program) the magic number is substantially less than seven. For the above reasons, this thesis will focus on the actual count of the operators and operands and will totally exclude any discussion about Halstead’s other hypothesis.

D. OUR METRIC

The metric we have implemented will take, as input, an Ada program and analyze this program with respect to Halstead’s length hypothesis. To properly carry out this task, the input program must be decomposed into its most basic lexical elements, and then parsed to ensure that the program is syntactically correct. As the structure of the
program is being validated the data needed for metric implementation is collected and stored for later analysis. We have designed a generic front-end for this metric tool which means other metrics can be added at a later date, thus giving the program the ability to be expanded and provide a wider range of data. We will cover each of these front-end sections in detail and describe how and why our metric operates.
II. DEVELOPING AN ADA GRAMMAR

A. INTRODUCTION

The grammar of a language specifies the syntax of the language and is used to help guide the translation of programs. A grammar naturally describes the hierarchial structure of many programming language constructs [Ref. 11: p.261]. A grammar has four components:

1. A set of tokens, known as terminal symbols.
2. A set of nonterminals.
3. A set of productions which consists of a nonterminal, an arrow, and a sequence of terminals and or nonterminals.
4. A designation of one of the nonterminals as the start symbol.

Grammars are classified by many characteristics, and different parsing techniques are more or less effective on a particular class of grammar. The most efficient methods of parsing, top-down and bottom-up, which we will cover in chapter four, work only on certain subclasses of grammars. Several of these subclasses, such as the LL and LR grammars, are expressive enough to describe most syntactic constructs in programming languages [Ref. 11: p.160]. Parsers implemented by hand often work with LL grammars and parsers for the larger class of LR grammars are usually constructed by automated tools. The first L in LL stands for scanning the input from left-to-right, the second L for producing a leftmost derivation. Conversely the L in LR is again for left-to-right scanning of the input, the R for constructing a rightmost derivation in reverse [Ref. 11: p.215]. A grammar that is LL(1) can be deterministically parsed with a top down left to right scan by using only one token lookahead. Therefore, an LL(1) grammar has a parsing table with no multiply-defined entries. If the present parsing table has multidefined entries an attempt can be made to transform the grammar by eliminating all left recursion and left factoring whenever possible [Ref. 11: p.192]. There are some grammars for which no amount of alteration will yield an LL(1) grammar. Eliminating left recursion and then left factoring is easy to do but may make the resulting grammar hard to read and difficult to use for translation purposes. These procedures are covered in the next section.

A grammar generates strings by beginning with the start symbol and repeatedly replacing a nonterminal by the right side of a production for that nonterminal. The
terminal strings that can be derived from the start symbol form the language defined by the grammar [Ref. 11: p.28].

B. GRAMMAR FOR THE ADA LANGUAGE

Ada is a very large language consequently the grammar for this language is also very large. We chose to use a top-down, recursive-descent parser, which will be covered in greater detail in chapter four, as our method of analyzing our input program. We used the Ada language as defined in the Ada Language Reference Manual (LRM) [Ref. 12]. Our first step was to translate the Ada language from the Backus-Naur Form given in the Ada Language Reference Manual. In translating this grammar, which is not LL(1), into an LL(1)-like grammar, it was necessary to massage the language description given in the manual. Massaging is the process of removing all left recursion and then left factoring. Left recursion is when the leftmost symbol on the right side of a production is the same as the nonterminal on the left side of the production. Left recursion must be eliminated for this top-down, recursive descent parser to alleviate the possibility of infinite looping. It must be remembered, this process does not guarantee that the transformed language will be LL(1). However, we must be sure to perform transformations that lead to a grammar for the same language. The remainder of this chapter is devoted to the discussion and explanation of how we massaged the grammar. The complete grammar used by our parser can be found in Appendix A. Our translation key has terminal symbols as lowercase letters, non-terminal symbols as uppercase letters, and bold-faced symbols to indicate the meta-symbols of our grammar.

Once the initial grammar is expressed in our meta-symbology, the next step is to remove all left recursion. Since the BNF form in the LRM showed no left recursion, it appeared that this step would not be required. However, there was one case of left recursion that was not apparent until several substitutions of the productions had been made. This case involved the production rules for NAME, INDEXED_COMPONENT, SLICE, SELECTED_COMPONENT, ATTRIBUTE, and PREFIX. The production rules, when taken directly from the LRM, appear as the follows:

\[
\begin{align*}
\text{NAME} & \rightarrow \text{identifier} \\
& \quad \rightarrow \text{character_literal} \\
& \quad \rightarrow \text{string_literal} \\
& \quad \rightarrow \text{INDEX_COMPONENT}
\end{align*}
\]
When starting with NAME and substituting in the productions, the left recursion becomes readily apparent. For example:

\[ \text{name} \rightarrow \text{slice} \rightarrow \text{prefix} \left( \text{discrete range} \right) \rightarrow \text{name} \left( \text{discrete range} \right) \]

We see that the following production exists:

\[ \text{name} \rightarrow \text{name} \left( \text{expression} \right) \]

Several other productions, left recursive on \text{name}, can be generated using the other rules listed above.

Now that left recursion does exist, we expanded out the productions listed above (using the same technique previously demonstrated) and combined them all as production rules for \text{name}. The production rules for \text{indexed component}, \text{slice}, \text{selected component}, and \text{attribute} were incorporated into \text{name} so they were removed from our grammar. The final set of production rules for \text{name} can be found in Appendix A.

The third step in massaging our grammar is left factoring. Our parser could not function with one token lookahead if left factoring were possible. Left factoring is a grammar transformation which uses the basic idea that if it is not clear which of two alternative productions to use to expand a nonterminal, it may be possible to rewrite the productions to defer the decision until we have enough of the input to make the correct decision. To demonstrate this procedure, we will show the left factoring used on the productions for \text{relation}. Taken directly from the LRM the production rules for \text{relation} are as follows:
Applying the rule of left factoring, a new nonterminal, SIMPLE_EXPRESSION_TAIL, has been added to the grammar. The production rules for RELATION and SIMPLE_EXPRESSION_TAIL now look like the following:

```
RELATION --> SIMPLE_EXPRESSION
--> SIMPLE_EXPRESSION RELATIONAL_OPERATOR SIMPLE_EXPRESSION
--> SIMPLE_EXPRESSION in RANGES
--> SIMPLE_EXPRESSION not in RANGES
--> SIMPLE_EXPRESSION in TYPE_MARK
--> SIMPLE_EXPRESSION not in TYPE_MARK

SIMPLE_EXPRESSION_TAIL --> RELATIONAL_OPERATOR SIMPLE_EXPRESSION
--> in RANGES
--> not in RANGES
--> in TYPE_MARK
--> not in TYPE_MARK
```

Finally, in attempting to make our grammar LL(1) it was necessary to combine several similar constructs together so that it could be parsed by one function of the parser. For example, the reserved word `package` appears in several instances including a package specification, a package body declaration, a separate package body declaration, a generic instantiation of a package, and the renaming of a package. In each of these examples the reserved word `package` is used, and even with the ability to look ahead one token, it is impossible to tell which form of the package construct is being utilized. We massaged our grammar so that if `package` is encountered the function PACKAGE_DECLARATION is called. The function PACKAGE_DECLARATION first checks for the reserved word `body`, indicating a package body declaration, or a separate package body declaration. PACKAGE_DECLARATION then checks for an identifier, indicating a package specification, a generic instantiation, or a renaming declaration. If `body` is present then the function PACKAGE_BODY is called. If an identifier is present then the function PACKAGE_UNIT is called. This technique of decision making based on reserved word or terminal symbol presence is extended into the functions.
PACKAGE_BODY and PACKAGE_UNIT to further decide which form of package is being utilized. In essence, we have expanded the production rules to allow each new production the ability to correctly determine, with one token lookahead, what the next production rule will be. This entire process is also used for the different versions of procedures, functions, and tasks which can appear in an Ada program.
III. LEXICAL ANALYZER

A. INTRODUCTION

Ada is an extremely large language, comparable in size to PLI. It was developed on behalf of the Department of Defense for use in embedded systems [Ref. 13: p.xvi]. Based on Pascal, Ada is the first practical language to bring together important features such as data abstraction, multitasking, exception handling, encapsulation and generics [Ref. 13: p.xvi]. Our design approach utilizes a division of labor and we separate our metric into phases which perform a single, specific function. The first two phases, lexical analysis and parsing, combine to form a generic front-end machine. This front-end machine constructs an intermediate representation of the source program. The information necessary to implement the metric is then collected and analyzed from the intermediate form. We will look, in depth, at the lexical analyzer and identify how it operates and why it is necessary.

B. TOKENS

Lexical Analysis, often called linear analysis or scanning, is when a stream of characters making up the source program is read from left-to-right and grouped into tokens, which are sequences of characters having a collective meaning [Ref. 11: p.4]. The character sequence forming a token, with the legal characters as described in [Ref. 12: p.2-1], is called the lexeme for the token. This lexeme is what is used to identify the actual operators and operands that serve as the input for our metric. All variables will have a lexeme, such as sqrt, rate, answer, and so on. There are seven token classes in the Ada language. They are identifiers, separators, numeric literals, delimiters, comments, character literals, and string literals. The lexical analyzer takes the source program one character at a time, and builds the token lexeme as it determines the token class. Each token is generated by a finite state automaton. A finite state automaton, often called a finite state machine, is a mathematical model for a device that is capable of recognizing strings of characters defined by a certain class of grammars, called regular grammars. Our scanner, or lexical analyzer, can be in any one of a finite number of internal configurations or states [Ref. 14: p.13]. The state of the system summarizes the information concerning past inputs that is needed to determine the behavior of the system on subsequent inputs. The lexical analyzer scans
the symbols of a computer program to locate the strings of characters corresponding to one of the seven token types mentioned earlier. In this process the lexical analyzer needs to remember only a finite amount of information, such as how long a prefix of a reserved word it has seen since startup [Ref. 14: p.14].

We will now address these tokens individually and discuss not only their purpose and content but also the finite state machines we programmed to handle their recognition.

1. Identifiers

![Finite State Machine for Identifiers](image)

Identifiers are used as names and also as reserved words [Ref. 12: p.2-4]. An identifier must start with a letter and it can then be any combination of letters, digits or the underscore character (_). There cannot be two underscore characters side by side in the identifier and there is no maximum length specified for any identifier. Identifiers differing only in the use of corresponding upper and lower case letters are considered as the same [Ref. 12: p.2-4]. The finite state machine we programmed to identify and store token identifiers is seen in Figure 3.1.

2. String Literal

A string literal is formed by a sequence of zero or more graphic characters enclosed between two quotation characters ("), used as string brackets [Ref. 12: p.2-6]. A string literal has a value that is a sequence of character values corresponding to the graphic characters of the string literal apart from the quotation character itself.
If a quotation character value is to be represented in the sequence of character values, then a pair of adjacent quotation characters must be written at the corresponding place within the string literal. The length of a string literal is the number of character values in the sequence represented, except for doubled quotation characters which are counted as a single character [Ref. 12: p.2-6]. A string literal must fit on one line since it is a lexical element but longer sequences of graphic characters can be obtained by catenation of string literals [Ref. 12: p.2-7]. Except for the instance of doubled quotation characters, the finite state machine we programmed to identify and store token string literals can be seen in Figure 3.2.

3. Character Literals

Figure 3.2 Finite State Machine for String Literals.

Figure 3.3 Finite State Machine for Character Literals.
A character literal is formed by enclosing one of the 95 graphic characters (including the space), which are described in [Ref. 12: p.2-1], between two apostrophe characters ('). A character literal has a value that belongs to a character type. The finite state machine we created to identify and store token character literals can be seen in Figure 3.3.

4. Comments

![Finite State Machine for Comments](image)

Figure 3.4 Finite State Machine for Comments.

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a program [Ref. 12: p.2-7]. The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the effect of a program. The sole purpose of comments is to provide clarity and explanation to the human reader. The horizontal tabulation can be used in comments, after the double hyphen, and is equivalent to one or more spaces [Ref. 12: p.2-7]. The finite state machine we programmed to identify and store token comments can be seen in Figure 3.4.

5. Separators

In certain cases an explicit separator is required to separate adjacent lexical elements (namely, without separation, interpretation as a single lexical element is possible) [Ref. 12: p.2-3]. A separator is any of a space character, a format effector (such as horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed), or the end of a line [Ref. 12: p.2-3]. A space character is a separator except within a comment, a string literal, or a space character literal. The horizontal
tabulation is not a separator within a comment. One or more separators are allowed between any two adjacent lexical elements (tokens), and at least one separator is required between an identifier or a numeric literal and an adjacent identifier or numeric literal. The finite state machine we programmed to identify and store token separators is seen in Figure 3.5.

6. Delimiters

A simple delimiter is either one of the following special characters (in the basic character set):

\& \cdot ( ) * + , . / ; < = > | :
A compound delimiter is one of the following, each composed of two adjacent special characters

\[ \rightarrow .. \quad \ast : = \quad \rightarrow = \quad \rightarrow > \quad \rightarrow < \quad \rightarrow < \quad \rightarrow > \quad \rightarrow < \quad \rightarrow < \quad \rightarrow < \]

Any other combination of adjacent special characters is not a legal compound delimiter. The finite state machine we programmed for identifying and storing token delimiters is seen in Figure 3.6.

7. Numeric Literal

The numeric literal is by far the most complex and varied type of token. It encompasses real numbers, integer numbers, and based numbers which are numeric literals expressed in an explicitly specified base between 2 and 16 [Ref. 12: 2-5]. A real number is a number with a decimal point, an integer is a number without a point and a based literal is, again, a number whose base is explicitly stated. An underline character (_) inserted between adjacent digits of a numeric literal does not affect the value of this numeric literal. The only letters allowed as extended digits are the letters A through F, which stand for the digits ten through fifteen in hexadecimal. A letter in a based number can be written either in lower case or in upper case, with the same meaning [Ref. 12: p.2-5]. Leading zeros are allowed. No space is allowed in a numeric literal, not even between constituents of the exponent, since a space is a separator. A zero exponent is allowed for an integer literal. The finite state machine we programmed to identify and store token numeric literals can be seen in Figure 3.7.

C. TOKEN USE

As was seen in Chapter II, a grammar is made up of terminals, non-terminals, a start symbol, and productions. The terminals are the basic symbols from which strings are formed. These strings are the combinations of the most basic symbols, tokens, which form meaningful expressions to a particular language. To be able to analyze these strings and determine whether or not a given string is a legal statement in a given language we must first identify each token as it is entered by the program. Identification of the tokens permits the computer to compact the incoming data thus allowing the saving of space. For example, if someone placed ten blanks in an input program where only one was needed, lexical analysis would see the separator and flush the other unused blanks, thus saving space. Certain tokens will be augmented by a lexical value. For example, when an identifier like rate is found, the lexical analyzer not only generates a token, say id, but also enters the lexeme rate into the symbol
Figure 3.7  Finite State Machine for Numeric Literals.
table, if it is not already there [Ref. 11: p.12]. The lexical value associated with this occurrence of *id* points to the symbol-table entry for *id*. The construction of these tokens is done by reading one character at a time and building the lexeme of the token by appending the appropriate characters together. This translation from the input program to a simple stream of tokens is the sole job of the lexical analyzer. In the next chapter we will look at the system parser and its functions.
IV. PARSER

A. INTRODUCTION

The parser, which is the second component of our front-end machine, is the mainstay of our metric. Parsing is also called hierarchical analysis or syntax analysis. It involves grouping the tokens, created by the lexical analyzer, of the source program into grammatical phrases that are used to synthesize output [Ref. 11: p.61]. A parser can be constructed for any context-free grammar. The important factor in parsing is speed. Given a programming language, we can generally construct a grammar that can be parsed quickly. Most programming language parsers make a single left-to-right scan over the input, looking ahead one token at a time [Ref. 11: p.41]. In discussing this parsing problem, it is helpful to think of a parse tree being constructed, even though our front-end machine does not actually construct a tree. A parse tree describes the syntactic structure of the input. It pictorially shows how the start symbol of a grammar derives a string in the language [Ref. 11: p.29]. Formally, given a context-free grammar, a parse tree is a tree with the following properties:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a token or empty string.
3. Each interior node is labeled by a nonterminal.

The leaves of a parse tree, read from left to right, form the yield of a tree, which is the string generated or derived from the nonterminal at the root of the parse tree [Ref. 11: p.29]. The term, context-free grammar, which is mentioned earlier, defines a finite set of variables (also called nonterminals or syntactic categories), each of which represents a language [Ref. 14: p.77]. A context-free grammar is denoted \( G = (V,T,P,S) \), where \( V \) and \( T \) are finite sets of variables and terminals respectively [Ref. 14: p.79]. \( P \) is a finite set of productions; each production (\( A \rightarrow W \)) is of the form \( A \) produces \( W \) where \( A \) is a variable and \( W \) is a string of symbols from any combination of \( (V \cup T) \). Finally, \( S \) is the start symbol. It must be noted though that although our front-end machine is capable of constructing a parse tree (otherwise the translation would not be guaranteed correct) the actual tree is not necessary for our metric purposes and is therefore not built.
Most parsing methods fall into one of two classes, top-down and bottom-up methods [Ref. 11: p.41]. These terms refer to the order in which nodes in the parse tree, if the tree actually existed, were constructed. In the top-down method, construction starts at the root and proceeds towards the leaves going deeper and deeper until eventually reaching the bottom. In the bottom-up method it is just the opposite. Construction starts at the bottom and proceeds towards the root. The popularity of top-down parsers is due to the fact that efficient parsers can be constructed more easily by hand using top-down methods [Ref. 11: p.41]. Bottom-up parsing, however, can handle a larger class of grammars and translation schemes. Software tools for generating parsers directly from grammars have tended to use bottom-up methods [Ref. 11: p.41]. Because of its efficiency and ease of use, we have chosen to use the top-down method of parsing for the front-end machine of our metric. Furthermore, we chose a particular type of top-down parsing called recursive-descent. This technique, a classical method often used in industry, is very powerful. We describe its operation in the following section.

B. TOP-DOWN RECURSIVE DESCENT PARSING

Recursive-descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input [Ref. 11: p.44]. A function is associated with each nonterminal of a grammar. We now consider a special form of recursive-descent parsing, called predictive parsing, in which the token symbol unambiguously determines the function selected for each nonterminal [Ref. 11: p.44]. The sequence of functions called in processing the input implicitly defines a parse tree for the input.

Our procedure GET_CURRENT_TOKEN_RECORD builds an array of fifty tokens and, starting at the initial position controls a pointer which identifies the current token being parsed and another pointer which identifies the next or lookahead token to be parsed. The function BYPASS is the central control and workhorse for our parser. It compares the current token with predefined terminals. If there is a token-to-terminal match, BYPASS consumes the token by adjusting the index pointers. All of the terminal symbols in the Ada language are defined and any nonterminal is, as was stated earlier, a function call that returns a boolean value of true or false.

Parsing begins with a call for the starting nonterminal, which is COMPILATION in the Ada grammar. Parsing progresses as each function calls other functions.
descending into the parse structure until a call to BYPASS is performed and the appropriate boolean value is returned. This process of ascending and descending the parse structure continues until all the tokens created from the input file have been consumed or an error occurs.

We made an attempt to design the parser to be as robust as possible. However, due to the complexity of the language we were often forced to rely on the fact that the input file had been correctly compiled before being fed into our parser. The fact that the input file was precompiled allowed us to drop the italicized element of the nonterminals in the grammar. Some examples of this modification are:

1. A parser for a compiler would normally have to remember the type associated with NAME each time it was encountered. In our case we simply dropped the type requirement and parsed all of them with the function NAME. For example, all of the following are reduced to just NAME: TYPE_NAME, VARIABLE_NAME, PROCEDURE_NAME, FUNCTION_NAME, ENTRY_NAME, and this list is far from complete.

2. Another example occurred when we dropped the italicized element of the nonterminal EXPRESSION. Since the type had been checked by a full compiler, for our parser the following three nonterminals became simply EXPRESSION: UNIVERSAL_STATIC_EXPRESSION, QUALIFIED_EXPRESSION, and BOOLEAN_EXPRESSION. The following two nonterminals became SIMPLEXPRESSION for the same reasons: STATIC_SIMPLE_EXPRESSION, and DELAY_SIMPLE_EXPRESSION.

3. Our third example is that by dropping the italicized element of DISCRETE_SUBTYPE_INDICATION and COMPONENT_SUBTYPE_INDICATION, they can both be correctly parsed by SUBTYPE_INDICATION.

The changes highlighted above, and others that are not shown, were done to reduce the size and complexity of the parser. Having to retain all the type information would have required a much more extensive parsing element. This reduction allows our parser to be more efficient in its operation. We did not intend for our front-end machine to be a full compiler. We only needed to parse an input file in enough detail to be able to collect the meaningful and relevant metric data.
V. ADAMEASURE

A. INTRODUCTION

As was seen in the first chapter of our thesis there is a variety of different metric theories. By request of the Missile Software Branch, Naval Weapons Center, China Lake, we implemented the Halstead Software Science Metric. In the effort to provide as much information about the input program as possible, we also provide information on comments and nesting. We felt it was important to avoid trying to generate a single number, say between one and ten, which would be an attempt to quantify the given metric information into a single numeric statement. Instead we generate a few important numbers, then we apply some reasoning to what we believe these numbers mean in relation to program complexity and overall software quality. We stress what the program says about the software is merely suggestive.

B. DATA COLLECTION

In explaining the how and why of our data gathering, we will deal with each of the three types of information analysis separately.

1. Halstead Data

As stated in chapter one, we only implemented the program length metric from Halstead's software science theory. We gathered the operator data through our workhorse function BYPASS which counted every token-to-operator terminal match. To acquire the operand data we generated a symbol table of all identifiers, be they variables, procedures, functions, tasks, blocks, or numeric constants. Once this was completed we now had the four Halstead parameters \( n_1 \), \( n_2 \), \( N_1 \), and \( N_2 \). Having calculated the theoretical length and actual length we divided both of these numbers by total lines input to allow comparisons of results from programs of different size.

If the actual length is greater than the theoretical length Halstead hypothesized that the difference is caused by one or more of the following six classes of impurities:

- Cancellation: The occurrence of an inverse cancels the effect of a previous operator; no other use of the variable changed by the operator is made before the cancellation.
- Ambiguous operands: The same operand is used to represent two or more variables in an algorithm.
• Synonymous operands: Two or more operand names represent the same variable.
• Common subexpressions: The same subexpression occurs more than once.
• Unnecessary replacements: A subexpression is assigned to a temporary variable which is used only once.
• Unfactored expressions: There are repetitions of operators and operands among unfactored terms in an expression.

If the theoretical length is greater than the actual length then the following conditions could exist:
• Operands: There may be some variables which were declared but never referenced in the program.
• Globals: A large number of the variables referenced were declared in the package instantiated by the WITH statement.

2. Comment Data

As the input file is parsed, a count is kept of the comment lines. Upon completion of parsing the number of comment lines is divided by the total lines input, then multiplied by one-hundred to yield the overall comment percentage. On the basis of this percentage we make recommendations that might prove helpful to the software engineer. Briefly, we consider a comment percentage between zero and fifteen percent as low and we state that unless the program utilizes Ada's extensive variable identification ability then there may be too few comments for adequate reader comprehension. We consider a comment percentage between fifteen and fifty as a reasonable number and state that this could give the reader a good understanding of the program. A comment percentage between fifty and eighty-five percent is considered fairly high and we state that the program has good understandability but runs a small risk of obscuring the code. Lastly, a comment percentage between eighty-five and one-hundred percent is an extremely high percentage and we say that the program has a higher possibility of obscuring the code in the high number of comments.

3. Nesting Data

Determining a program's complexity is not an easy thing but it is generally accepted that as the nesting level increases so goes the complexity level. We have implemented a nesting level summary which counts how frequently a given nesting level was reached, maintains a record for each level, and keeps track of the maximum level used and where it was first encountered.
VI. CONCLUSIONS

A. METRICS

The Department of Defense's interest in metrics provides a powerful motivation for the continued research into possible metric tools. The software crisis is severe enough to warrant tools that will improve the software engineering process. Computer Science is such an infant in the world of academia and metrics is such a very small part of this child that we are aware of the difficulty in finding breakthroughs. The computer scientists may have tried to "catch-up" with the other sciences much too rapidly, consequently they may have missed laying some of the necessary foundations.

Our initial effort is a good start, a baseline from which future efforts can build upon. AdaMeasure is a helpful tool for providing the software engineer with the information that is presently gathered by hand. We do not feel software metrics should simply attempt putting a number onto a program in the effort to quantify its quality. Text describing what the metric has seen and how it relates to the programmer and the program is what is really needed for the output. It is this bridging from the concrete world of the metric to the abstract, metaphysical environment of the actual software that presents the real challenge. As we collect and analyze the data, we bridge this gap by providing recommendations to the user on how we see the interrelationship of these two entities. These recommendations are based on currently accepted software practices.

B. IMPLEMENTATION

In our effort to make the Ada grammar more LL(1)-like we were forced to perform extensive massaging. This massaging allowed us to use the top-down, recursive-descent parsing technique. This classical, time tested method proved to be easily implemented and debugged. The extensive massaging made it necessary for us to frequently check and reaffirm that the language generated by our newly transformed grammar was exactly the same as the initial language. This point cannot be overstated. The languages must be identical. Because of the complexity involved in massaging and checking the grammar, we feel we have some insight into how languages are created, how they build upon themselves, and where languages are going in the future.
When we originally accepted the proposal for creating a metric, we were not
aware of how complex and time consuming the implementation of the front-end
machine would be. This complexity, coupled with the fact that Ada is a brand new
language and relatively unknown other than by name, made it a struggle until we had
significantly progressed along the learning curve. We found the Ada compiler, which is
currently a state-of-practice compiler, to be extremely slow in comparison to the
compilers we were familiar with such as Pascal or Fortran. The Naval Postgraduate
School has no Ada compiler so we were forced to work over Arpanet or Telnet for the
actual programming. This presented problems and delays on a regular basis. The
expiration of access card numbers, the malfunctioning of the bridge box, the networks
going down, the slow baud rate between stations, and machine downtime at NWC
China Lake, both scheduled and nonscheduled, made information transfer a real hurdle
in the overall effort.

C. THE FUTURE

Having designed a generic metric tool, we hope the program will be expanded
and improved. There is already a plan in progress to have Sible Henry and Dennis
Karacha's Complexity Flow Metric implemented. There is a real need for user interface
improvements. Because of the limited hardware options available to us, we have
programmed our interface to deal with the VT-100 terminal. To make the system more
robust and transportable, an interface scheme that would be functional from a variety
of different terminals would be a useful endeavor.

This metric was undertaken at the request of NWC China Lake and all of our
efforts have been guided by their input to us. Pragmas, which are used to convey
information to the compiler, are currently being used in run-time systems only at China
Lake and although they are in the Ada grammar we have not implemented them in our
metric. It would greatly increase the value of the program if the pragma portion of the
grammar were implemented. The Software Missile Branch of NWC China Lake has
provided all the Ada programs at their disposal to help us test our metric for proper
parsing. We have successfully tested our metric on all these files and we have
successfully tested our own code by feeding our metric into itself. Although our tests
have been successful we have not tested all of our code. There is a particular need for
programs that will test our metric in the area of tasking and all the code that is
associated with it. This testing effort should be carried out as soon as possible.
Metrics are important. We hope that our initial work will be expanded and put to real use as a aid to the software engineer. Although Ada is new and relatively unexplored as a language, we feel it will begin to build and become more popular. We hope our metric adds to this building process.
APPENDIX A
MODIFIED ADA GRAMMAR

Our translation key has terminal symbols as lowercase letters, nonterminal symbols as uppercase letters, and bold-faced symbols to indicate the meta-symbols of our grammar.

(9.10) (parser3) ABORT_STATEMENT --> NAME [ , NAME]

(9.5) (parser1) ACCEPT_STATEMENT --> identifier [ (EXPRESSION) ?] [ FORMAL_PART ?] [ do SEQUENCE_OF_STATEMENTS end [ identifier ?] ? ]

(4.3) (parser3) AGGREGATE --> (COMPONENT_ASSOCIATION [ , COMPONENT_ASSOCIATION]*)

(4.8) (parser3) ALLOCATOR --> SUBTYPE_INDICATION [ 'AGGREGATE ?]

(3.6) (parser3) ARRAY_TYPE_DEFINITION --> (INDEX_CONSTRAINT of SUBTYPE_INDICATION

(5.2) (parser2) ASSIGNMENT_OR_PROCEDURE_CALL --> NAME := EXPRESSION ;
--> NAME ;

(4.1.4) (parser3) ATTRIBUTE_DESIGNATOR --> identifier [ (EXPRESSION) ?]
--> range [ (EXPRESSION) ?]
--> digits [ (EXPRESSION) ?]
--> delta [ (EXPRESSION) ?]

(3.1) (parser1) BASIC_DECLARATION --> type TYPE_DECLARATION
--> subtype SUBTYPE_DECLARATION
--> procedure PROCEDURE_UNIT
--> function FUNCTION_UNIT
--> package PACKAGE_DECLARATION
--> generic GENERIC_DECLARATION
--> IDENTIFIER_DECLARATION
--> task TASK_DECLARATION

(3.9) (parser1) BASIC_DECLARATIVE_ITEM --> BASIC_DECLARATION
--> REPRESENTATION_CLAUSE
--> use WITH_OR_USE_CLAUSE

(10.1) (parser0) BASIC_UNIT --> LIBRARY_UNIT
--> SECONDARY_UNIT

(4.5) (parser4) BINARY_ADDING_OPERATOR --> +
--> -
--> &

(15.6) (parser1) BLOCK_STATEMENT --> [ identifier : ?] [ declare DECLARATIVE_PART ?] begin
CASE_STATEMENT -- > EXPRESSION is [ CASE_STATEMENT_ALTERNATIVE]* end case 

CASE_STATEMENT_ALTERNATIVE -- > when CHOICE [ | CHOICE]* => SEQUENCE_OF_STATEMENTS 

CHOICE --> EXPRESSION ![SIMPLE_EXPRESSION?] 
--> EXPRESSION [ CONSTRAINT ] 
--> others 

COMPILATION --> [ COMPILATION_UNIT]* 

COMPILATION_UNIT --> CONTEXT_CLAUSE BASIC_UNIT 

COMPONENT_ASSOCIATION --> [ CHOICE [ | CHOICE]* => ?] EXPRESSION 

COMPONENT_DECLARATION --> IDENTIFIER_LIST : SUBTYPE_INDICATION [ := EXPRESSION ?]; 

COMPONENT_LIST --> [ COMPONENT_DECLARATION]* [ VARIANT_PART ?] 
--> null ; 

COMPUND_STATEMENT --> if IF_STATEMENT 
--> case CASE_STATEMENT 
--> LOOP_STATEMENT 
--> BLOCK_STATEMENT 
--> accept ACCEPT_STATEMENT 
--> select SELECT_STATEMENT 

CONSTANT_TERM --> array CONSTRUANED_ARRAY_DEFINITION [ := EXPRESSION ?]; 
--> := EXPRESSION 
--> NAME IDENTIFIER_TAIL 

CONSTRAINT --> range RANGES 
--> digits FLOATING_OR_FIXED_POINT_CONSTRAINT 
--> delta FLOATING_OR_FIXED_POINT_CONSTRAINT 
--> (INDEXCONSTRAINT 

CONTEXT_CLAUSE --> [ with WITHORUSE_CLAUSE [ use WITHORUSE_CLAUSE]*] 

DECLARATIVE_PART --> [ BASIC_DECLARATIVE_ITEM]* [ LATER_DECLARATIVE_ITEM]* 

DELAY_STATEMENT --> SIMPLE_EXPRESSION ; 

DESIGNATOR --> IDENTIFIER_TAIL 

DISCRETE_RANGE --> RANGES [ CONSTRAINT ] 

DISCRIMINANT_PART --> (DISCRIMINANT_SPECIFICATION [ , DISCRIMINANT_SPECIFICATION]* ) 

DISCRIMINANT_SPECIFICATION --> IDENTIFIER_LIST : NAME [ := EXPRESSION ?]
(9.5) (parser2)
ENTRY_DECLARATION --> entry identifier [(DISCRETE_RANGE)?] [(FORMAL_PART)?]

(3.5.1) (parser4)
ENUMERATION_LITERAL --> identifier
--> character_literal

(3.5.1) (parser4)
ENUMERATION_TYPE_DEFINITION --> (ENUMERATION_LITERAL [ , ENUMERATION_LITERAL]*)

(11.1) (parser2)
EXCEPTION_CHOICE --> identifier
--> others

(11.2) (parser1)
EXCEPTION_HANDLER --> when EXCEPTION_CHOICE [ , EXCEPTION_CHOICE]**
--> SEQUENCE_OF_STATEMENTS

(8.5) (parser2)
EXCEPTION_TAIL --> ;
--> renames NAME ;

(5.7) (parser3)
EXIT_STATEMENT --> [ NAME ?] [ when EXPRESSION ?] ;

(4.4.1) (parser3)
EXPRESSION --> RELATION RELATION_TAIL

(4.4.1) (parser3)
FACTOR --> PRIMARY [ ** PRIMARY ]
--> abs PRIMARY
--> not PRIMARY

(13.5.7) (parser2)
FLOATING_OR_FIXED_POINT_CONSTRAINT --> SIMPLE_EXPRESSION [ range RANGES ]

(6.4) (parser4)
FORMAL_PARAMETER --> identifier =>

(6.1) (parser2)
FORMAL_PART --> (PARAMETER_SPECIFICATION [ , PARAMETER_SPECIFICATION]*)

(6.1) (parser1)
FUNCTION_BODY --> is [ FUNCTION_BODY_TAIL ?]
--> ;

(6.1) (parser1)
FUNCTION_BODY_TAIL --> separate ;
--> <> ;
--> SUBPROGRAM_BODY
--> NAME ;

(6.1) (parser1)
FUNCTION_UNIT --> DESIGNATOR FUNCTION_UNIT_TAIL

(6.1) (parser1)
FUNCTION_UNIT_TAIL --> is new NAME [ GENERIC_ACTUAL_PART ?] ;
--> [ FORMAL_PART ?] return NAME FUNCTION_BODY

(12.1) (parser2)
GENERIC_ACTUAL_PART --> (GENERIC_ASSOCIATION [ , GENERIC_ASSOCIATION]*)

(12.1) (parser2)
GENERIC_ASSOCIATION --> [ GENERIC_FORMAL_PARAMETER ?] EXPRESSION

(12.1) (parser1)
GENERIC_DECLARATION --> [ GENERIC_PARAMETER_DECLARATION ?] GENERIC_FORMAL_PART
(12.1) (parser2)
GENERIC_FORMAL_PARAMETER -- > identifier =>
-- > string_literal =>

(12.1) (parser1)
GENERIC_FORMAL_PART -- > procedure PROCEDURE_UNIT
-- > function FUNCTION_UNIT
-- > package PACKAGE_DECLARATION

(12.1) (parser1)
GENERIC_PARAMETER_DECLARATION -- > IDENTIFIER_LIST : [ MODE ?] NAME [ := EXPRESSION ?]
-- > type private [ DISCRIMINANT_PART ?] is
PRIVATE_TYPE_DECLARATION
-- > type private [ DISCRIMINANT_PART ?] is
GENERIC_TYPE_DEFINITION
-- > with procedure PROCEDURE_UNIT
-- > with function FUNCTION_UNIT

(12.1) (parser2)
GENERIC_TYPE_DEFINITION -- > ( <> )
-- > range <>
-- > digits <>
-- > delta <>
-- > array ARRAY_TYPE_DEFINITION
-- > access SUBTYPE_DEFINITION

(5.9) (parser3)
GOTO_STATEMENT -- > NAME ;

(3.2) (parser2)
IDENTIFIER_DECLARATION -- > IDENTIFIER_LIST : IDENTIFIER_DECLARATION_TAIL

(3.2) (parser2)
IDENTIFIER_DECLARATION_TAIL -- > exception EXCEPTION_TAIL
-- > constant CONSTANT_TERM
-- > array CONSTRAINED_ARRAY_DEFINITION
[ := EXPRESSION ?] ;
-- > NAME IDENTIFIER_TAIL

(3.2) (parser2)
IDENTIFIER_LIST -- > identifier \[ , identifier \]^* 

(3.2) (parser2)
IDENTIFIER_TAIL -- > [ CONSTRAINT ?][ := EXPRESSION ?] ;
-- > [ renames NAME ?] ;

(5.3) (parser1)
IF_STATEMENT -- > EXPRESSION then SEQUENCE_OF_STATEMENTS
[ elsif EXPRESSION then SEQUENCE_OF_STATEMENTS]^* [ else
SEQUENCE_OF_STATEMENTS ?] end if ;

(3.6) (parser3)
INDEXCONSTRAINT -- > DISCRETE_RANGE \[ , DISCRETE_RANGE]^* ) 

(3.5.4) (parser3)
INTEGER_TYPE_DEFINITION -- > range RANGES

(5.5) (parser3)
ITERATION_SCHEME -- > while EXPRESSION
-- > for LOOP_PARAMETER_SPECIFICATION

(5.1) (parser2)
LABEL -- > << identifier >>

(3.9) (parser1)
LATER_DECLARATIVE_ITEM -- > PROPER_BODY
-- > generic GENERIC_DECLARATION
-- > use WITH_OR_USE_CLAUSE
(6.1) (parser2)
PARAMETER_SPECIFICATION --> IDENTIFIER_LIST : MODE NAME [ := EXPRESSION ]

(4.4) (parser3)
PRIMARY --> numeric_literal
--> null
--> string_literal
--> new_ALLOCATOR
--> NAME
--> AGGREGATE

(17.4) (parser2)
PRIVATE_TYPE_DECLARATION --> [ limited ? ] private

'6.1) (parser1)
PROCEDURE_UNIT --> identifier [ FORMAL_PART ] is SUBPROGRAM_BODY
--> identifier [ FORMAL_PART ]
--> identifier [ FORMAL_PART ] renames NAME;

(13.9) (parser1)
PROCEDURE_BODY --> procedure PROCEDURE_UNIT
--> function FUNCTION_UNIT
--> package PACKAGE_DECLARATION
--> task TASK_DECLARATION

(13.5) (parser3)
RANGES --> SIMPLE_EXPRESSION [ . . SIMPLE_EXPRESSION ]

'12.4) (parser2)
PAIR_STATEMENT --> [ NAME ];

(13.4) (parser2)
RECORD_REPRESENTATION_CLAUSE --> [ at mod SIMPLE_EXPRESSION ]
[ NAME at SIMPLE_EXPRESSION range RANGES ]
end record;

(13.7) (parser2)
RECORD_TYPE_DEFINITION --> COMPONENT_LIST end record

(4.4) (parser3)
RELATION --> SIMPLE_EXPRESSION [ SIMPLE_EXPRESSION TAIL ]

(4.4) (parser3)
RELATION_TAIL --> [ and [ then ? ] RELATION ]
--> [ or [ else ? ] RELATION ]
--> [ xor RELATION ]

(4.5) (parser4)
RELATIONAL_OPERATOR --> =
--> /=
--> <
--> <=
--> >
--> >=

(13.1) (parser2)
REPRESENTATION_CLAUSE --> for NAME use record RECORD_REPRESENTATION_CLAUSE
--> for NAME use [ at ? ] SIMPLE_EXPRESSION

(5.8) (parser3)
RETURN_STATEMENT --> [ EXPRESSION ]

(10.1) (parser0)
SECONDARY_UNIT --> LIBRARY_UNIT_BODY
--> SUBUNIT
(9.7.1) (parser1)
SELECT_ALTERNATIVE --> [ when EXPRESSION => ? ] accept ACCEPT_STATEMENT
SEQUENCE_OF_STATEMENTS
--> [ when EXPRESSION => ? ] delay DELAY_STATEMENT
SEQUENCE_OF_STATEMENTS
--> [ when EXPRESSION => ? ] terminate

(9.7.1) (parser1)
SELECT_ENTRY_CALL --> else SEQUENCE_OF_STATEMENTS
--> or delay DELAY_STATEMENT [ SEQUENCE_OF_STATEMENTS ]

(9.7) (parser1)
SELECT_STATEMENT --> SELECT_STATEMENT_TAIL SELECT_ENTRY_CALL and select ;

(9.7.1) (parser1)
SELECT_STATEMENT_TAIL --> SELECT_ALTERNATIVE [ or SELECT_ALTERNATIVE ]
--> NAME [ SEQUENCE_OF_STATEMENTS ]

(4.1.3) (parser6)
SELECTION --> identifier
--> character_literall
--> string_literall
--> all

(5.1) (parser1)
SEQUENCE_OF_STATEMENTS --> [ STATEMENT ]

(4.4) (parser1)
SIMPLE_EXPRESSION --> [ ] TERM [ BINARY_ADDING_OPERATOR TERM ]
--> [ ] TERM [ BINARY_ADDING_OPERATOR TERM ]

(4.4) (parser1)
SIMPLE_EXPRESSION_TAIL --> RELATIONAL_OPERATOR SIMPLE_EXPRESSION
--> [ not ] in RANGES
--> [ not ] in NAME

(5.1) (parser2)
SIMPLE_STATEMENT --> null
--> ASSIGNMENT_STATEMENT
--> PROCEDURE_CALL_STATEMENT
--> exit EXIT_STATEMENT
--> return RETURN_STATEMENT
--> goto GOTO_STATEMENT
--> delay DELAY_STATEMENT
--> abort ABORT_STATEMENT
--> raise RAISE_STATEMENT
--> ENTRY_CALL_STATEMENT
--> CODE_STATEMENT

(5.1) (parser1)
STATEMENT --> [ LABEL ] SIMPLE_STATEMENT
--> [ LABEL ] COMPOUND_STATEMENT

(6.3) (parser1)
SUBPROGRAM_BODY --> new NAME [ GENERIC_ACTUAI. PART ]
--> separate ;
--> [ ]
--> [ ] DECLARATIVE_PART begin SEQUENCE_OF_STATEMENTS
--> [ exception [ EXCEPTION_HANDLER ] ]
--> [ DESIGNATOR ]
--> NAME

(3.3.2) (parser2)
SUBTYPE_DECLARATION --> identifier is SUBTYPE_INDICATION

(3.3.2) (parser3)
SUBTYPE_INDICATION --> NAME [ CONSTRAINT ]
SUBUNIT -> separate (NAME) PROPER_BODY

(9.1) (parser1)
TASK_BODY -> identifier is TASK_BODY_TAIL

(9.1) (parser1)
TASK_BODY_TAIL -> separate

(9.1) (parser1)
TASK_BODY_TAIL -> separate

(9.1) (parser1)
TASK_DECLARATION -> body TASK_BODY ;

(4.4) (parser3)
TERM -> FACTOR [ MULTIPLYING_OPERATOR FACTOR ]

(3.3.1) (parser2)
TYPE_DECLARATION -> identifier [ DISCRIMINANT_PART ]

(3.3.1) (parser2)
TYPE_DEFINITION -> ENUMERATION_TYPE_DEFINITION

(3.7.3) (parser2)
VARIANT -> when CHOICE [ 1 CHOICE ] => COMPONENT_LIST

(3.7.3) (parser2)
VARIANT_PART -> case identifier is [ VARIANT ] and case ;

(10.1.1) (parser2)
WITH_OR_USE_CLAUSE -> identifier [ , identifier ]
APPENDIX B
'ADAMEASURE' USERS GUIDE

1. Once you have logged on a VT100 terminal, type RUN DEMON to begin execution of 'AdaMeasure'. The initial screen gives the general information about the program.

```
**H** **E** **L** **C** **O** **M** **E** **T** **O** 'AdaMEASURE'
**A** **U** **T** **H** **O** **R** **E** **D** **B** **Y**: LCDR JEFFREY L. NIEDER, USN
**L** **T** KARL S. FAIRBANKS, USN

**N** **A** **V** **A** **L** **P** **O** **S** **T** **G** **R** **A** **D** **U** **A** **T** **E** **S** **C** **H** **O** **O** **L**
**D** **E** **P** **A** **R** **T** **M** **E** **N** **T** **O** **F** **C** **O** **M** **P** **U** **T** **E** **R** **S** **C** **I** **E** **N** **C** **E**

**M** **O** **N** **T** **E** **R** **E** **Y**, **C** **A** **L** **I** **F** **O** **R** **N** **A** **S**

31 **O** **C** **T** **O** **B** **R** **E** **Y** 1986

**V** **E** **R** **S** **I** **O** **N** 1.0

* This program provides an automated software metric tool which uses quantitative measures in an effort to supply the user with helpful information about the static structure of a given input program. This program is public domain information.

*** Enter any letter to continue ***
```

2. Enter any letter to continue. This is required because Ada filters out carriage returns so just hitting ENTER will not cause execution to continue. The next screen shown is the MAIN SELECTION MENU. From here the user enters the digits 1, 2 or 3 to either parse a file, view previously gathered data, or quit to the operating system, respectively.

```
**M** **A** **I** **N** **S** **E** **L** **E** **C** **O** **N** **T** **I** **O** **N** **M** **E** **N** **U**
**H** **E** **R** **E** **E** **R** **A** **R** **E** **T** **H** **A** **C** **T** **I** **O** **N** **C** **H** **O** **I** **C** **E** **S** **A** **V** **A** **I** **L** **A** **B** **L** **E** **T** **O** **Y** **O** **U**

Simply enter the number of your choice

1 - PARSE AN INPUT FILE
2 - VIEW PREVIOUSLY GATHERED DATA
3 - EXIT TO OPERATING SYSTEM

```

3. If the user selects number one then he will be prompted for the file name of the file he wishes to have parsed. While parsing of the file is in progress, the user will see a message on the screen indicating at what line number, in the input file, the parser has
reached. When parsing of the input file commences, 'Ada.Measure' creates four new files. The four files have the same name as the user's input file with the following extensions: fn.DATA, fn.HALS, fn.RAND, fn.MISC. The meaning of each of these new files will be explained later in this user's guide.

4. Upon conclusion of parsing the input file, or if the user selects number two from the MAIN SELECTION MENU, the program displays the METRIC SELECTION PROGRAM. From this menu, the user can select any of the listed metrics, exit to the MAIN SELECTION MENU, or exit to the operating system.

```
***************

METRIC SELECTION MENU

HERE ARE THE INFORMATION CHOICES AVAILABLE TO YOU

Simply enter the number of your choice

1 - 'HALSTEAD' METRIC INFORMATION
2 - COMMENT AND NESTING METRIC INFORMATION
3 - 'HENRY and KAFURA' METRIC INFORMATION
4 - EXIT TO MAIN MENU
5 - EXIT TO OPERATING SYSTEM

Choice =
```

5. If number one is selected, the Halstead Metric choice, the next menu displayed is the HALSTEAD SELECTION MENU. From this menu, the calculations and conclusions of the Halstead Metric can be selected. There also exists the options of exiting to the METRIC SELECTION MENU, or exiting to the operating system.

```
***************

HALSTEAD SELECTION MENU

HERE ARE THE HALSTEAD METRIC OPTIONS AVAILABLE TO YOU

Simply enter the number of your choice

1 - HALSTEAD OPERATORS
2 - HALSTEAD OPERANDS
3 - HALSTEAD METRIC CONCLUSIONS
4 - EXIT TO METRIC SELECTION MENU
5 - EXIT TO OPERATING SYSTEM

Choice =
```
6. From the HALSTEAD SELECTION MENU, if number one is selected, the
Halstead operator data is displayed. The operator data is stored in the fn.DATA file.
The Halstead operator data includes the total number of different operators used, the
total number of occurrences of those operators, and the number of occurrences of each
individual operator.

7. If number two from the HALSTEAD SELECTION MENU is chosen, the
HALSTEAD OPERAND SELECTION MENU is displayed. From this menu the
different classes of operands and their data can be selected for viewing. Also available
for selection are exiting to the HALSTEAD SELECTION MENU, and exiting to the
operating system.

8. From the HALSTEAD OPERAND SELECTION MENU, selection of any of the
operand classes will show each identifier and number of occurrences of for that
particular class. The data for all operand classes is stored in the fn.RAND file.

9. Back to the HALSTEAD SELECTION MENU. If the selection is number three,
the Halstead Metric conclusions are displayed. The conclusions include the input file's
calculated theoretical length, its actual length, the difference between the two lengths,
and some comments about that difference. The Halstead Metric conclusion data is
stored in the fn.HALS file.

10. Back to the METRIC SELECTION MENU. If the selection number is two, the
comment and nesting metric information is displayed on the screen. The comment
metric information includes the total number of lines in the input file, the total number
of comment lines, a percentage of lines of comments to lines of code, and some observations about that percentage. The nesting metric information contains the type and total occurrences of nesting constructs used in the input file, the deepest level of nesting parsed, and how many times each nesting level, up to the deepest, was encountered. The comment and nesting metric information is stored in the fn.MISC file.

II. The final choice in the METRIC SELECTION MENU is the Henry and Kafura Complexity Flow Metric. At present, this metric is not implemented. Ongoing development of 'AdaMeasure' includes plans to implement this metric as well as other metrics and Ada tools.
APPENDIX C

'ADAMEASURE' PROGRAM LISTING - PART 1

with HALSTEAD_METRIC, BYPASS_SUPPORT_FUNCTIONS, GLOBAL, GLOBAL_PARSER;
use HALSTEAD_METRIC, BYPASS_SUPPORT_FUNCTIONS, GLOBAL, GLOBAL_PARSER;

package BYPASS_FUNCTION is
  function BYPASS(TOKEN_ARRAY_ENTRY_CODE : integer) return boolean is
    CONSUME : boolean := FALSE;
    LEXEME : string(1..LINESIZE);
    SIZE : natural;
    begin
      GET_CURRENT_TOKEN_RECORD(CURRENT_TOKEN_RECORD, LEXEME_LENGTH);
      LEXEME := CURRENT_TOKEN_RECORD.LEXEME;
      SIZE := CURRENT_TOKEN_RECORD.LEXEME_SIZE - 1;
      case TOKEN_ARRAY_ENTRY_CODE is
        when TOKEN_IDENTIFIER =>
          if (CURRENT_TOKEN_RECORD.TOKEN_TYPE = IDENTIFIER) then
            CONSUME := TRUE;
            CONDUCT_reserve_word_TEST(CONSUME);
          end if;
        if (CONSUME) then
          CONVERT_UPPER_CASE(LEXEME, SIZE);
          OPERAND_METRIC(HEAD_NODE, CURRENT_TOKEN_RECORD, DECLARE_TYPE);
          DECLARE_TYPE := VARIABLE_DECLARE;
        end if;
        when TOKEN_NUMERIC_LITERAL =>
          if (CURRENT_TOKEN_RECORD.TOKEN_TYPE = NUMERIC_LIT) then
            CONSUME := TRUE;
            DECLARE_TYPE := CONSTANT_DECLARE;
            OPERAND_METRIC(HEAD_NODE, CURRENT_TOKEN_RECORD, DECLARE_TYPE);
            DECLARE_TYPE := VARIABLE_DECLARE;
          end if;
      end case;
  end function;
end BYPASS_FUNCTION;
package body BYPASS_FUNCTION is
  -- this function compares the lexeme of the current token with the
  -- token currently being sought by the parser. If the current token
  -- type is identifier, then a test is conducted to ensure it is not
  -- a reserved word.
  function BYPASS(TOKEN_ARRAY_ENTRY_CODE : integer) return boolean is
    CONSUME : boolean := FALSE;
    LEXEME : string(1..LINESIZE);
    SIZE : natural;
    begin
      GET_CURRENT_TOKEN_RECORD(CURRENT_TOKEN_RECORD, LEXEME_LENGTH);
      LEXEME := CURRENT_TOKEN_RECORD.LEXEME;
      SIZE := CURRENT_TOKEN_RECORD.LEXEME_SIZE - 1;
      case TOKEN_ARRAY_ENTRY_CODE is
        when TOKEN_IDENTIFIER =>
          if (CURRENT_TOKEN_RECORD.TOKEN_TYPE = IDENTIFIER) then
            CONSUME := TRUE;
            CONDUCT_reserve_word_TEST(CONSUME);
          end if;
        if (CONSUME) then
          CONVERT_UPPER_CASE(LEXEME, SIZE);
          OPERAND_METRIC(HEAD_NODE, CURRENT_TOKEN_RECORD, DECLARE_TYPE);
          DECLARE_TYPE := VARIABLE_DECLARE;
        end if;
        when TOKEN_NUMERIC_LITERAL =>
          if (CURRENT_TOKEN_RECORD.TOKEN_TYPE = NUMERIC_LIT) then
            CONSUME := TRUE;
            DECLARE_TYPE := CONSTANT_DECLARE;
            OPERAND_METRIC(HEAD_NODE, CURRENT_TOKEN_RECORD, DECLARE_TYPE);
            DECLARE_TYPE := VARIABLE_DECLARE;
          end if;
      end case;
  end function;
end BYPASS_FUNCTION;
when TOKEN_CHARACTER_LITERAL =>
  if (CURRENT_TOKEN_RECORD.TOKEN_TYPE = CHARACTER_LIT) then
    CONSUME := TRUE;
  end if;
when TOKEN_STRING_LITERAL =>
  if (CURRENT_TOKEN_RECORD.TOKEN_TYPE = STRING_LIT) then
    CONSUME := TRUE;
  end if;
when TOKEN_END =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "end") then
    CONSUME := TRUE;
  end if;
when TOKEN_BEGIN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "begin") then
    CONSUME := TRUE;
  end if;
when TOKEN_IF =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "if") then
    CONSUME := TRUE;
  end if;
when TOKEN_THEN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "then") then
    CONSUME := TRUE;
  end if;
when TOKEN_ELSE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "else") then
    CONSUME := TRUE;
  end if;
when TOKEN_WHILE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "while") then
    CONSUME := TRUE;
  end if;
when TOKEN_LOOP =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "loop") then
    CONSUME := TRUE;
  end if;
when TOKEN_CASE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "case") then
    CONSUME := TRUE;
  end if;
when TOKEN_WHEN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "when") then
    CONSUME := TRUE;
  end if;
when TOKEN_DECLARE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "declare") then
    CONSUME := TRUE;
  end if;
when TOKEN_FOR =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "for") then
    CONSUME := TRUE;
when TOKEN OTHERS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "others") then
    CONSUME := TRUE;
  end if;
when TOKEN RETURN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "return") then
    CONSUME := TRUE;
  end if;
when TOKEN EXIT =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "exit") then
    CONSUME := TRUE;
  end if;
when TOKEN PROCEDURE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "procedure") then
    CONSUME := TRUE;
  end if;
when TOKEN FUNCTION =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "function") then
    CONSUME := TRUE;
  end if;
when TOKEN WITH =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "with") then
    CONSUME := TRUE;
  end if;
when TOKEN USE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "use") then
    CONSUME := TRUE;
  end if;
when TOKEN PACKAGE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "package") then
    CONSUME := TRUE;
  end if;
when TOKEN BODY =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "body") then
    CONSUME := TRUE;
  end if;
when TOKEN RANGE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "range") then
    CONSUME := TRUE;
  end if;
when TOKEN IN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "in") then
    CONSUME := TRUE;
  end if;
when TOKEN OUT =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "out") then
    CONSUME := TRUE;
  end if;
when TOKEN SUBTYPE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "subtype") then
    CONSUME := TRUE;
  end if;
when TOKEN TYPE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "type") then
CONSUME := \ U E;

when TOKEN_IS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "is") then
    CONSUME := TRUE;
  end if;

when TOKEN_NULL =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "null") then
    CONSUME := TRUE;
  end if;

when TOKEN_ACCESS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "access") then
    CONSUME := TRUE;
  end if;

when TOKEN_ARRAY =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "array") then
    CONSUME := TRUE;
  end if;

when TOKEN_DIGITS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "digits") then
    CONSUME := TRUE;
  end if;

when TOKEN_DELTA =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "delta") then
    CONSUME := TRUE;
  end if;

when TOKEN_RECORD_STRUCTURE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "record") then
    CONSUME := TRUE;
  end if;

when TOKEN_CONSTANT =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "constant") then
    CONSUME := TRUE;
  end if;

when TOKEN_NEW =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "new") then
    CONSUME := TRUE;
  end if;

when TOKEN_EXCEPTION =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "exception") then
    CONSUME := TRUE;
  end if;

when TOKEN_RENAMES =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "renames") then
    CONSUME := TRUE;
  end if;

when TOKEN_PRIVATE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "private") then
    CONSUME := TRUE;
  end if;

when TOKEN_LIMITED =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "limited") then
    CONSUME := TRUE;
  end if;

when TOKEN_TASK =>

49
if (ADJUST_LEXEME(LEXEME, SIZE) = "task") then
    CONSUME := TRUE;
end if;

when TOKEN_ENTRY =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "entry") then
        CONSUME := TRUE;
    end if;

when TOKEN_ACCEPT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "accept") then
        CONSUME := TRUE;
    end if;

when TOKEN_DELAY =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "delay") then
        CONSUME := TRUE;
    end if;

when TOKEN_SELECT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "select") then
        CONSUME := TRUE;
    end if;

when TOKEN_TERMINATE =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "terminate") then
        CONSUME := TRUE;
    end if;

when TOKEN_ABORT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "abort") then
        CONSUME := TRUE;
    end if;

when TOKEN_SEPARATE =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "separate") then
        CONSUME := TRUE;
    end if;

when TOKEN_RAISE =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "raise") then
        CONSUME := TRUE;
    end if;

when TOKEN_GENERIC =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "generic") then
        CONSUME := TRUE;
    end if;

when TOKEN_AT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "at") then
        CONSUME := TRUE;
    end if;

when TOKEN_REVERSE =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "reverse") then
        CONSUME := TRUE;
    end if;

when TOKEN_DO =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "do") then
        CONSUME := TRUE;
    end if;

when TOKEN_GOTO =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "goto") then
        CONSUME := TRUE;
    end if;
when TOKEN_OF =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "of") then
    CONSUME := TRUE;
  end if;

when TOKEN_ALL =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "all") then
    CONSUME := TRUE;
  end if;

when TOKEN_PRAGMA =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "pragma") then
    CONSUME := TRUE;
  end if;

when TOKEN_AND =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "and") then
    CONSUME := TRUE;
  end if;

when TOKEN_OR =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "or") then
    CONSUME := TRUE;
  end if;

when TOKEN_NOT =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "not") then
    CONSUME := TRUE;
  end if;

when TOKEN_XOR =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "xor") then
    CONSUME := TRUE;
  end if;

when TOKEN_MOD =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "mod") then
    CONSUME := TRUE;
  end if;

when TOKEN_Rem =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "rem") then
    CONSUME := TRUE;
  end if;

when TOKEN_ABSOLUTE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "abs") then
    CONSUME := TRUE;
  end if;

when TOKEN_ASTERISK =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "*") then
    CONSUME := TRUE;
  end if;

when TOKEN_SLASH =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "/") then
    CONSUME := TRUE;
  end if;
when TOKEN_EXPONENT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "**") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_EXPONENT, CONSUME, RESERVEWORD_TEST);

when TOKEN_PLUS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "+") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_PLUS, CONSUME, RESERVEWORD_TEST);

when TOKEN_MINUS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "-" ) then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_MINUS, CONSUME, RESERVEWORD_TEST);

when TOKEN_AMPERSAND =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "&") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_AMPERSAND, CONSUME, RESERVEWORD_TEST);

when TOKEN_EQUALS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "=") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_EQUALS, CONSUME, RESERVEWORD_TEST);

when TOKEN_NOT_EQUALS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "!"") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_NOT_EQUALS, CONSUME, RESERVEWORD_TEST);

when TOKEN_LESS_THAN =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "<") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_LESS_THAN, CONSUME, RESERVEWORD_TEST);

when TOKEN_LESS_THAN_EQUALS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "<=") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_LESS_THAN_EQUALS, CONSUME, RESERVEWORD_TEST);

when TOKEN_GREATER_THAN =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = ">") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_GREATER_THAN, CONSUME, RESERVEWORD_TEST);

when TOKEN_GREATER_THAN_EQUALS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = ">=") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_GREATER_THAN_EQUALS, CONSUME, RESERVEWORD_TEST);

when TOKEN_ASSIGNMENT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "=:") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_ASSIGNMENT, CONSUME, RESERVEWORD_TEST);

when TOKEN_COMMA =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = ",") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_COMMA, CONSUME, RESERVEWORD_TEST);
when TOKEN_SEMICOLON =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = ";") then
    CONSUME := TRUE;
  end if;

when TOKEN_PERIOD =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = ".") then
    CONSUME := TRUE;
  end if;

when TOKEN_LEFT_PAREN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "(" ) then
    CONSUME := TRUE;
  end if;

when TOKEN_RIGHT_PAREN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = ")") then
    CONSUME := TRUE;
  end if;

when TOKEN_COLON =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = ":") then
    CONSUME := TRUE;
  end if;

when TOKEN_APOSTROPE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "'" ) then
    CONSUME := TRUE;
  end if;

when TOKEN_RANGE_DOTS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "..") then
    CONSUME := TRUE;
  end if;

when TOKEN_ARROW =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "=>") then
    CONSUME := TRUE;
  end if;

when TOKEN_BAR =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = " " ) then
    CONSUME := TRUE;
  end if;

when TOKEN_BRACKETS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "<>") then
    CONSUME := TRUE;
  end if;

when TOKEN_LEFT_BRACKET =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "<<") then
    CONSUME := TRUE;
  end if;

when TOKEN_RIGHT_BRACKET =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = ">>") then
    CONSUME := TRUE;
  end if;

when others => null;
end case;

ADJUST_TOKEN_BUFFER(CONSUME, RESERVE_WORD_TEST);

return (CONSUME);
end BYPASS;
-- this procedure tests all identifiers to verify they are not reserved
-- words. The most common reserved words are tested first and the process
-- halts when a match is made or the test fails.
procedure CONDUCT_RESERVE_WORD_TEST(CONSUME : in out boolean) is
begin
  RESERVE_WORD_TEST := TRUE;
  for RESERVE_WORD_INDEX in TOKEN_END..TOKEN_ABSOLUTE loop
    if (BYPASS(RESERVE_WORD_INDEX)) then
      CONSUME := FALSE;
    end if;
    exit when not CONSUME;
  end loop;
  RESERVE_WORD_TEST := FALSE;
  end CONDUCT_RESERVE_WORD_TEST;
end BYPASS_FUNCTION;

---

**TITLE:**
AN ADA SOFTWARE METRIC
---
**MODULE NAME:**
PACKAGE BYPASS_SUPPORT_FUNCTIONS
**DATE CREATED:**
03 OCT 86
**LAST MODIFIED:**
03 DEC 86
**AUTHORS:**
LCDR JEFFREY L. NIEDER
LT KARL S. FAIRBANKS, JR.
**DESCRIPTION:**
This package contains the procedures and function required to support the function BYPASS.
---

with SCANNER, GLOBAL, GLOBAL_PARSER, TEXT_IO;
use SCANNER, GLOBAL, GLOBAL_PARSER, TEXT_IO;

package BYPASS_SUPPORT_FUNCTIONS is
  package NEW_INTEGER_IO is new TEXT_IO.INTEGER_IO(integer);
  use NEW_INTEGER_IO;

  procedure GET_CURRENT_TOKEN_RECORD
    (CURRENT_TOKEN_RECORD : in out TOKEN_RECORD_TYPE;
     LEXEME_LENGTH : in out integer);

  procedure TRACE
    (TRACE_TOKEN : in string;
     CONSUME, RESERVE_WORD_TEST : in boolean);

  procedure ADJUST_TOKEN_BUFFER
    (CONSUME, RESERVE_WORD_TEST : in boolean);

  function ADJUST_LEXEME
    (INPUT_LEXEME : string;
     SIZE : natural) return string;

  procedure CONVERT_UPPER_CASE
    (INPUT_LEXEME : in out string;
     LENGTH : in out integer);

end BYPASS_SUPPORT_FUNCTIONS;
---

package body BYPASS_SUPPORT_FUNCTIONS is

  -- this procedure handles the loading of the token record buffer, flushes out comments (while keeping count of them) and separators, and prints out the current line being parsed to the screen.

  procedure GET_CURRENT_TOKEN_RECORD
    (CURRENT_TOKEN_RECORD : in out TOKEN_RECORD_TYPE;
     LEXEME_LENGTH : in out integer) is
  begin
    DISPLAY_DELAY := constant integer := 250;

    if (FIRST_TIME_LOAD) then
      while (PLACEHOLDER_INDEX /= TOKEN_ARRAY_SIZE + 1) loop
        if not (END_OF_FILE_TEST_FILE) then
          GET_NEXT_TOKEN(TOKEN_RECORD);
          if ((TOKEN_RECORD.TOKEN_TYPE /= SEPARATOR) and (TOKEN_RECORD.TOKEN_TYPE /= COMMENT)) then
            TOKEN_RECORD_BUFFER(PLACEHOLDER_INDEX) := TOKEN_RECORD;
          elsif ((TOKEN_RECORD.TOKEN_TYPE = COMMENT) then
            COMMENT_COUNT := COMMENT_COUNT + 1;
          end if;
        else
          TOKEN_RECORD_BUFFER(PLACEHOLDER_INDEX).TOKEN_TYPE := ILLEGAL;
          TOKEN_RECORD_BUFFER(PLACEHOLDER_INDEX).LEXEME_SIZE := 1;
          PLACEHOLDER_INDEX := PLACEHOLDER_INDEX + 1;
        end if;
      end loop;
    FIRST_TIME_LOAD := FALSE;
  end if;
  end loop;
  FULL := TRUE;
  PLACEHOLDER_INDEX := 1;

55
elsif (TOKEN_ARRAY_INDEX = 0) and (not (FULL))) then
TOKEN_RECORD_BUFFER(0) := TOKEN_RECORD_BUFFER(TOKEN_ARRAY_SIZE))
while (PLACEHOLDER_INDEX /= TOken_ARRAY_SIZE + 1) loop
if not (END OF_FILE(TEST_FILE)) then
GET_NEXT_TOKEN(TOKEN_RECORD);
if (TOKEN_RECORD.TOKEN_TYPE /= SEPARATOR) and
(TOKEN_RECORD.TOKEN_TYPE /= COMMENT) then
TOKEN_RECORD_BUFFER(PLACEHOLDER_INDEX) := TOKEN_RECORD;
PLACEHOLDER_INDEX := PLACEHOLDER_INDEX + 1;
elsif (TOKEN_RECORD.TOKEN_TYPE = COMMENT) then
COMMENT_COUNT := COMMENT_COUNT + 1;
endif
else
TOKEN_RECORD_BUFFER(PLACEHOLDER_INDEX).TOKEN_TYPE := ILLEGAL;
TOKEN_RECORD_BUFFER(PLACEHOLDER_INDEX).LEXEME_SIZE := 1;
PLACEHOLDER_INDEX := PLACEHOLDER_INDEX + 1;
endif
end loop;
PLACEHOLDER_INDEX := 1;
FULL := TRUE;
endif
if not (RESERVE_WORD_TEST) then
CURRENT_TOKEN_RECORD := TOKEN_RECORD_BUFFER(TOKEN_ARRAY_INDEX);
endif
LEXEME_LENGTH := CURRENT_TOKEN_RECORD.LEXEME_SIZE - 1;
if (CURRENT_TOKEN_RECORD.TOKEN_TYPE = IDENTIFIER) then
CONVERT_LOWER_CASE(CURRENT_TOKEN_RECORD.LEXEME, LEXEME_LENGTH);
endif
STATUS_COUNTER := STATUS_COUNTER + 1;
if (STATUS_COUNTER = DISPLAY_DELAY) then
new_line;
CLEARSCREEN;
CONVERT_UPPER_CASE(DATA_FILE_NAME, DATA_FILE_SIZE);
put("Parse of "); put(ADJUST_LEXEME(DATA_FILE_NAME, DATA_FILE_SIZE));
put(" in progress, at line number "); put(TOTAL_LINES_INPUT, 5);
STATUS_COUNTER := 0;
endif
end GET_CURRENT_TOKEN_RECORD;

procedure TRACe(TRACE_TOKEN : in string;
CONSUME, RESERVE_WORD_TEST : in boolean) is
begin
if (CONSUME) and then not (RESERVE_WORD_TEST) then
put(RESULT_FILE, "Parsed a");
put(RESULT_FILE, TRACE_TOKEN);
new_line(RESULT_FILE);
endif
end TRACe;

-- this procedure adjusts the pointer to the token record buffer.
procedure ADJUST_TOKEN_BUFFER(CONSUME, RESERVE_WORD_TEST : in boolean) is
begin
if ((CONSUME) and not (RESERVE_WORD_TEST)) then
TOKEN_ARRAY_INDEX := (TOKEN_ARRAY_INDEX + 1) mod 50;
if (TOKEN_ARRAY_INDEX = 0) then
FULL := FALSE;
endif
endif
end ADJUST_TOKEN_BUFFER;

56
-- this function takes as input the 132 character input_lexeme and generates
-- a variable length string based on the actual size of the input_lexeme.
function ADJUST_LEXEME(INPUT_LEXEME : string; SIZE : natural) return string is
  subtype LEXEME_BUFFER is string(1..SIZE);
  ADJUSTED_LEXEME : LEXEME_BUFFER;
begin
  for I in 1..SIZE loop
    ADJUSTED_LEXEME(I) := INPUT_LEXEME(I);
  end loop;
  return (ADJUSTED_LEXEME);
end ADJUST_LEXEME;

procedure CONVERT_LOWER_CASE(INPUT_LEXEME : in out string;
  LENGTH : in out integer) is
  CONVERSION_FACTOR : constant integer := 32;
  LETTER_VALUE : integer;
begin
  for I in 1..LENGTH loop
    if (INPUT_LEXEME(I) in UPPERCASE_LETTER) then
      LETTER_VALUE := character'pos(INPUT_LEXEME(I)) + CONVERSION_FACTOR;
      INPUT_LEXEME(I) := character'val(LETTER_VALUE);
    end if;
  end loop;
end CONVERT_LOWER_CASE;

procedure CONVERT_UPPER_CASE(INPUT_LEXEME : in out string;
  LENGTH : in out integer) is
  CONVERSION_FACTOR : constant integer := 32;
  LETTER_VALUE : integer;
begin
  for I in 1..LENGTH loop
    if (INPUT_LEXEME(I) in LOWERCASE_LETTER) then
      LETTER_VALUE := character'pos(INPUT_LEXEME(I)) - CONVERSION_FACTOR;
      INPUT_LEXEME(I) := character'val(LETTER_VALUE);
    end if;
  end loop;
end CONVERT_UPPER_CASE;

end BYPASS_SUPPORT_FUNCTIONS;
-- TITLE: AN ADA SOFTWARE METRIC
--
-- MODULE NAME: PROCEDURE DEMON
--
-- DATE CREATED: 13 JUN 86
--
-- LAST MODIFIED: 03 DEC 86
--
-- AUTHORS: LCDR JEFFREY L. NIEDER
--
-- LT KARL S. FAIRBANKS, JR.
--
-- DESCRIPTION: This procedure is the driver for AdaMeasure.
--               It also contains the exception handler for the entire
--               set of packages which comprise AdaMeasure.
--
with MENUDISPLAY, GLOBAL_PARSER, GLOBAL, TEXT_IO;
use MENUDISPLAY, GLOBAL_PARSER, GLOBAL, TEXT_IO;

procedure DEMON is
package NEW_INTEGER_IO is new TEXT_IO.INTEGER_IO(integer);
  use NEW_INTEGER_IO;
begin
  DECLARATION := TRUE;
  INITIAL_MENU;

  exception
    when PARSER_ERROR => put(TOTAL_LINES_INPUT); new_line;
      put("Parser error");
    when SCANNER_ERROR => put(NEXT_CHARACTER); put(character_pos(NEXT_CHARACTER)); new_line;
      put(" Error occurred, program halted");
    when STATUS_ERROR => put("Status error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when MODE_ERROR => put("Mode error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when NAME_ERROR => put("Name error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when USE_ERROR => put("Use error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when DEVICE_ERROR => put("Device error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when END_ERROR => put("End error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when DATA_ERROR => put("Data error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when LAYOUT_ERROR => put("Layout error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when CONSTRAINT_ERROR => put("Constraint error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when NUMERIC_ERROR => put("Numeric error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when STORAGE_ERROR => put("Storage error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when PROGRAM_ERROR => put("Program error occurred at line ");
      put(TOTAL_LINES_INPUT);
    when QUIT_TO_OS => CLEARSCREEN;
  when others => put("Error occurred");
end DEMON;
package DISPLAY_SUPPORT is
  procedure GET_FILENAME(TYPE_PRESENT : in out boolean);
  procedure GET_ANSWERERROR, FINISHED : in out boolean;
  function ADJUST_EDIT_BUFFER(INPUT_STRING : string;
    FILL_LENGTH : integer) return string;
  procedure RESET_PARAMETERS;
  and DISPLAY_SUPPORT;

package body DISPLAY_SUPPORT is
  -- this is a user interface support procedure that prompts the user for
  -- the input file name, whenever the user must select a specific file.
  procedure GET_FILENAME(TYPE_PRESENT : in out boolean) is
    begin
      TYPE_PRESENT := FALSE;
      for I in 1..LINESIZE loop
        INPUT_FILE_NAME(I) := ' ';
        TEST_FILENAME(I) := ' ';
        DATA_FILE_NAME(I) := ' ';
      end loop;

      put("+---------------------------------------------------------------------+");
      put("+---------------------------------------------------------------------+");
      put("+---------------------------------------------------------------------+");
      put("+---------------------------------------------------------------------+");
      put("+---------------------------------------------------------------------+");
      new_line;
      put("+---------------------------------------------------------------------+");
      new_line;
      put("+---------------------------------------------------------------------+");
      new_line;
      put("Input the name of the file you wish to analyze");
      new_line;
      put("+---------------------------------------------------------------------+");
      new_line;
      put("+---------------------------------------------------------------------+");
      new_line;
      put("+---------------------------------------------------------------------+");
      new_line;
      put("+---------------------------------------------------------------------+");
      put("Filename = ");
      new_line;
      get_line(INPUT_FILE_NAME, LENGTH_OF_LINE); new_line(2);
      for I in 1..LENGTH_OF_LINE loop
        if (INPUT_FILE_NAME(I) = '.') then
          TYPE_PRESENT := TRUE;
        end if;
      end loop;
      if (TYPE_PRESENT) then
        for I in 1..LENGTH_OF_LINE-4 loop
          DATA_FILE_NAME(I) := INPUT_FILE_NAME(I);
        end loop;
        TEST_FILENAME := INPUT_FILE_NAME;
      end if;
    end procedure GET_FILENAME;

end DISPLAY_SUPPORT;

DATA_FILE_SIZE := LENGTH_OF_LINE - 4;

else
  DATA_FILE_NAME := INPUT_FILE_NAME;
  for I in 1..LENGTH_OF_LINE loop
    TEST_FILE_NAME(I) := INPUT_FILE_NAME(I);
  end loop;
  DATA_FILE_SIZE := LENGTH_OF_LINE;
end if;
end GET_FILENAME;

-- this user interface support procedure ensures that the user answers:
-- the question correctly, and determines if the user is finished.
procedure GET_ANSWER(FINISHED : in out boolean) is
begin
  FINISHED := false;
  get(ANSWER);
  if (ANSWER = 'N' or ANSWER = 'n') then
    FINISHED := true;
    ERROR := false;  -- user correctly said no
  elsif (ANSWER = 'Y') or (ANSWER = 'y') then
    ERROR := false;  -- user correctly said yes
  else
    ERROR := true;  -- user answered the question incorrectly
  end if;
end GET_ANSWER;

-- this formatting function places the input string in the edit buffer
-- and fills the remaining buffer spaces with periods.
function ADJUST_EDIT_BUFFER(INPUT_STRING : string; FILL_LENGTH : integer) return string is
begin
  for I in 1..FILL_LENGTH loop
    EDIT_BUFFER(I) := INPUT_STRING(I);
  end loop;
  for I in (FILL_LENGTH+1)..<EDIT_BUFFER_SIZE loop
    EDIT_BUFFER(I) := '.';
  end loop;
  return (EDIT_BUFFER);
end ADJUST_EDIT_BUFFER;

-- this procedure resets all of the metric parameters.
procedure RESET_PARAMETERS is
begin
  for I in TOKEN_AND..TOKEN_ASSIGNMENT loop
    OPERATOR_ARRAY(I) := 0;
  end loop;

  for I in IF_CONSTRUCT..CASE_CONSTRUCT loop
    CONSTRUCT_COUNT(I) := 0;
  end loop;

  for I in FIRST_LEVEL_NEST..MAXIMUM_NESTING loop
    NESTED_COUNT(I) := 0;
  end loop;

  TOKEN_ARRAY_INDEX := 0;
  PLACEHOLDER_INDEX := 0;
  TOTAL_LINES_INPUT := 0;
  COMMENT_COUNT := 0;
  CURRENT_NESTING_LEVEL := 0;
end RESET_PARAMETERS;
MAXIMUM_NESTING_LEVEL := 0;
NESTING_LINE_NUMBER := 0;
FIRST_TIME_LOAD := TRUE;
FULL := FALSE;
NESTED_LEVEL_INCREASE := TRUE;
and RESET_PARAMETERS;
and DISPLAY_SUPPORT;
package GENERAL_DATA is
  package NEWINTEGER_IO is new TEXTIO.INTEGER.IO(integer);
  use NEWINTEGER.IO;

  package REAL_IO is new TEXT_IO.FLOATIO0(float)*;
  use REAL_IO;

  procedure VIEWGENERAL;
end GENERAL_DATA;

package body GENERAL_DATA is

  -- this procedure computes the percentage of comments to total lines of the
  -- input file, and makes recommendations based on that percentage. It also
  -- displays what nesting constructs were utilized, and the count of each
  -- nesting level attained up to the maximum nesting level reached.

  procedure VIEWGENERAL is
    RESULT : float;
    HOLD_CHARACTER : character;
    COUNT, NEST : integer;
    begin
      GET_FILENAME(TYPE_PRESENT);
      CLEARSCREEN;
      open(DATA..FILE2, in_file, DATA_FILE_NAME & ".misc");

      CONVERT_UPPER_CASE(DATA_FILE_NAME, DATA_FILE..SIZE);
      put("");
      put("Comment count data for file ** ");
      put(ADJUST_LEXEME(DATAFILE_NAME, DATAFILE..SIZE));
      put(" ");
      new_line;
      put("-------------------------------------------------------------------------");
      new_line(2);
      put(ADJUST_EDIT_BUFFER("Total number of lines parsed", 28));
      get(DATA..FILE2, TOTAL_LINES_INPUT, 5);
      put(TOTAL_LINES_INPUT, 5);
      new_line;

      put(ADJUST_EDIT_BUFFER("Total number of comment lines parsed", 36));
      get(DATA..FILE2, COMMENT_COUNT, 5);
      put(COMMENT_COUNT, 5);
      new_line;
      put_line(DATA..FILE2, DUMMY_FILE_NAME, LENGTH_OF_LINE);

      RESULT := (float(COMMENT_COUNT) / float(TOTAL_LINES_INPUT)) * 100.0;
    end VIEWGENERAL;
end GENERAL_DATA;
IW
put(ADJUST_EDIT_BUFFER("Percentage of comments in the file", 34));
put(RESULT, 5, 1, 01);
new_line(2);
put("=================================================================================");
new_line(2);
if (RESULT >= 0.0) and (RESULT < 20.0) then
  put("There is a low percentage of comments to the total");
  new_line;
  put("number of lines in the file. Unless utilization of");
  new_line;
  put("Ada's extensive variable identification has been");
  new_line;
  put("applied, there may be too few comments for adequate");
  new_line;
  put("reader comprehension.");
  new_line;
elsif RESULT >= 20.0) and (RESULT < 50.0) then
  put("There is a reasonable number of comments to the");
  new_line;
  put("total number of lines in the file. This could help");
  new_line;
  put("a reader get a good understanding of the program.");
  new_line;
elsif RESULT >= 50.0) and (RESULT < 85.0) then
  put("There is a fairly high percentage of comments to the");
  new_line;
  put("total number of lines in the file. This could help");
  new_line;
  put("the reader get a good understanding of the program.");
  new_line;
  put("but could run the risk of obscuring the code in the");
  new_line;
  put("comments.");
else
  put("There is an extremely high percentage of comments to");
  new_line;
  put("the total number of lines in the file. With this high");
  new_line;
  put("number of comments, there is possibility of obscuring");
  new_line;
  put("the code in the comments.");
end if;
new_line;
put("It must be clearly understood, that this assessment of comment lines");
new_line;
put("to lines of code is not a hard and fast rule, but a suggestion that");
new_line;
put("may enhance the understanding of the code.");
new_line(2);
put("================================================================================");
new_line(2);
put("Enter any letter to continue ---");
new_line;
get(HOLD_CHARACTER);
CLEARSCREEN;
p",";
p"Nesting level data for file ** "");
p(ADJUST_LEXEME(DATA_FILE_NAME, DATA_FILE_SIZE));
p" **");
new_line;
p"================================================================================");
new_line(2);
p"DECISION TYPE UTILIZED") new_line;
p"---------------------------------------------------------------------------------";
new_line;
for I in IF_CONSTRUCT..CASE_CONSTRUCT loop
  get(DATA_FILE2, COUNT, 5);
  if (COUNT /= 0) then
    case I is

when IF_CONSTRUCT =>
    put(ADJUST_EDIT_BUFFER("IF construct", 12));
    put(COUNT, 5); new_line;
when LOOP_CONSTRUCT =>
    put(ADJUST_EDIT_BUFFER("LOOP construct", 14));
    put(COUNT, 5); new_line;
when WHILE_CONSTRUCT =>
    put(ADJUST_EDIT_BUFFER("WHILE construct", 15));
    put(COUNT, 5); new_line;
when FOR_CONSTRUCT =>
    put(ADJUST_EDIT_BUFFER("FOR construct", 13));
    put(COUNT, 5); new_line;
when CASE_CONSTRUCT =>
    put(ADJUST_EDIT_BUFFER("CASE construct", 14));
    put(COUNT, 5); new_line;
when others => null;
end cases;
end loop;
new_line;
get_line(DATA_FILE2, DUMMY_FILE_NAME, LENGTH_OF_LINE);
get(DATA_FILE2, MAXIMUM_NESTING_LEVEL, 5);
get_line(DATA_FILE2, DUMMY_FILE_NAME, LENGTH_OF_LINE);
get(DATA_FILE2, NESTING_LINE_NUMBER, 5);
get_line(DATA_FILE2, DUMMY_FILE_NAME, LENGTH_OF_LINE);
put(ADJUST_EDIT_BUFFER("Maximum nesting level", 21));
put(MAXIMUM_NESTING_LEVEL, 5);
new_line;
put(ADJUST_EDIT_BUFFER("Initial occurrence line number", 29));
put(NESTING_LINE_NUMBER, 5);
new_line;
for I in FIRST_LEVEL_NEST..MAXIMUM_NESTING_LEVEL loop
    get(DATA_FILE2, NEST, 5);
    put("Total nesting "); put(I, 2); put(" deep occurred");
    put(NEST, 3); put(" times."); new_line;
end loop;
new_line(2);
put(" --- Enter any letter to continue ---");
new_line;
get(HOLD_CHARACTER);
get_line(DATA_FILE2, DUMMY_FILE_NAME, LENGTH_OF_LINE);

close(DATA_FILE2);
end VIEW_GENERAL;
end GENERAL_DATA;
package GET_NEXT_CHARACTER is
    procedure GETNEXTCHARACTER(NEXTCHARACTER, LOOKAHEAD_ONE_CHARACTER : out character);
    procedure FILL_BUFFER(INPUTLINE : out INPUT_CODE_LINE);
end GET_NEXT_CHARACTER;

package body GET_NEXT_CHARACTER is

    -- this procedure gets the next character to be manipulated in
    -- the creation of each token
    procedure GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER : out character) is

        -- if the last character is read from the input buffer then it is
        -- time to refill the buffer, and reset the index variables
        begin
            if NEXT_BUFFER_INDEX = REFILL_BUFFER_INDEX then
                FILL_BUFFER(INPUT_LINE);
                CURRENT_BUFFER_INDEX := 0;
                NEXT_BUFFER_INDEX := 1;
            end if;
            if NEXT_BUFFER_INDEX = INPUT_LINE_SIZE then
                LOOKAHEAD_ONE_CHARACTER := ASCII.CR;
            else
                LOOKAHEAD_ONE_CHARACTER := INPUT_LINE(NEXT_BUFFER_INDEX + 1);
            end if;
            NEXT_CHARACTER := INPUT_LINE(NEXT_BUFFER_INDEX);
            CURRENT_BUFFER_INDEX := NEXT_BUFFER_INDEX;
            NEXT_BUFFER_INDEX := NEXT_BUFFER_INDEX + 1;
        end GETNEXTCHARACTER;

procedure FILL_BUFFER(INPUT_LINE : out INPUT_CODE_LINE) is

    begin
        for i in 1..INPUT_LINE_SIZE loop -- reset the input buffer to all $'s
            INPUT_LINE(i) := '$';
        end loop;
        get_line(TEST_FILE, INPUT_LINE, INPUT_LINE_SIZE);
        TOTAL_LINES_INPUT := TOTAL_LINES_INPUT + 1;
        REFILL_BUFFER_INDEX := INPUT_LINE_SIZE + 1;
    end FILL_BUFFER;
end GET_NEXT_CHARACTER;
-- AN ADA SOFTWARE METRIC

-- MODULE NAME: PACKAGE GLOBAL
-- DATE CREATED: 13 JUN 86
-- LAST MODIFIED: 16 OCT 86
-- AUTHORS: LCDR JEFFREY L. NIEDER
--          LT KARL S. FAIRBANKS, JR.
-- DESCRIPTION: This package contains all the global type,
--               subtype, and variable declarations.

with TEXT_IO;
package NEW_INTEGER_IO is new TEXT_IO.INTEGER_IO(integer);

with TEXT_IO, NEW_INTEGER_IO;
use TEXT_IO, NEW_INTEGER_IO;

package GLOBAL is

LINESIZE : constant integer := 100;

-- Ada token classes --
type TOKEN is (IDENTIFIER, SEPARATOR, NUMERIC_LIT, DELIMITER, COMMENT,
           CHARACTER_LIT, STRING_LIT, ILLEGAL);

-- record to hold the token built up by the scanner, the value is the
-- token's position (POS) in the token list, the lexeme is the actual
-- string for that particular token

type TOKEN_RECORD_TYPE is
record
  TOKENTYPE : TOKEN;
  TOKENVALUE : integer;
  LEXEME : string(1..LINESIZE);
  LEXEME_SIZE : natural;
end record;

-- this array is the input buffer, it holds each line of code when
-- read from the input file
subtype INPUT_CODE_LINE is string(1..LINESIZE);

subtype UPPER_CASE_LETTER is character range 'A'..'Z';
subtype LOWER_CASE_LETTER is character range 'a'..'z';
subtype UPPER_CASE_HEX is UPPER_CASE_LETTER range 'A'..'F';
subtype LOWER_CASE_HEX is LOWER_CASE_LETTER range 'a'..'f';
subtype DIGITS_TYPE is character range '0'..'9';

-- the following subtype declarations make use of the POS attribute
-- which returns the integer value of the particular ASCII
-- character argument

-- set of formators characterized by their ASCII value --
-- formators are horizontal tab, line feed, vertical tab, form feed,
-- and carriage return

subtype FORMATORS is integer
range character'pos(ASCII.TB)..character'pos(ASCII.CR);

-- first set of delimiters characterized by their ASCII value --
-- delimiters are ampersand, accent mark, left paren, right paren,
-- asterisk, plus sign, comma, dash, period, slash

subtype DELIMITER1 is integer

range character'pos('&').character'pos('/'));

-- second set of delimiters characterized by their ASCII value --
-- delimiters are colon, semi-colon, less than, equal, greater than
range character'pos(':')..character'pos('>'));

-- compound delimiters whose first symbol is in second set of delimiters --
-- the entire set of compound delimiters are <=, >=, /=, **, <<, >>, =>,
-- :=, <>...
subtype COMPOUND_DELIMITER is DELIMITER2
  range character'pos('<')..character'pos('>'));

TEST_FILE, RESULT_FILE : file_type;
INPUT_FILE_NAME : string(1..LINESIZE);
NEXT_CHARACTER : character;
LOOKAHEAD_ONE_CHARACTER : character;
CURRENT_BUFFER_INDEX : integer;
NEXT_BUFFER_INDEX : integer := 0;
TOKEN_RECORD : TOKEN_RECORD_TYPE;
INPUT_LINE : INPUT_CODE_LINE;
TOTAL_LINES_INPUT : integer := 0;
REFILL_BUFFER_INDEX : natural := 0;
LEXEME_LENGTH, INPUT_LINE_SIZE : natural;
TOKEN_CLASS : TOKEN;

procedure ERROR_MESSAGE(TOKEN_CLASS : in out TOKEN);

and GLOBAL;

package body GLOBAL is

-- procedure called when an error is detected by any of the token class
-- routines, an integer value is passed identifying which routine called
-- this procedure, and the SCANNER_ERROR exception is raised.
procedure ERROR_MESSAGE(TOKEN_CLASS : in out TOKEN) is
  SCANNER_ERROR_VALUE : integer;
begin
  SCANNER_ERROR_VALUE := TOKEN'pos(TOKEMCLASS);
  case SCANNER_ERROR_VALUE is
  when 0 => put(RESULT_FILE, "Illegal identifier at line number ");
  when 1 => put(RESULT_FILE, "Illegal separator at line number ");
  when 2 => put(RESULT_FILE, "Illegal numeric literal at line number ");
  when 3 => put(RESULT_FILE, "Illegal delimiter at line number ");
  when 4 => put(RESULT_FILE, "Illegal comment at line number ");
  when 5 => put(RESULT_FILE, "Illegal character literal at line number ");
  when 6 => put(RESULT_FILE, "Illegal string literal at line number ");
  when 7 => put(RESULT_FILE, "Illegal first character at line number ");
  when others => null;
  end case;
  put(RESULT_FILE, TOTAL_LINES_INPUT);
  new_line(RESULT_FILE));

  raise SCANNER_ERROR;
  and ERROR_MESSAGE;
end GLOBAL;
package GLOBAL_PARSER is

TOKEN_ARRAY_SIZE : constant integer := 501;
EDIT_LINE_SIZE : constant integer := 301;

TOKEN_IDENTIFIER : constant integer := 14;
TOKEN_NUMERIC_LITERAL : constant integer := 21;
TOKEN_CHARACTER_LITERAL : constant integer := 31;
TOKEN_STRING_LITERAL : constant integer := 41;
TOKEN_END : constant integer := 61;
TOKEN_BEGIN : constant integer := 71;
TOKEN_IF : constant integer := 81;
TOKEN_THEN : constant integer := 91;
TOKEN_ELSEIF : constant integer := 101;
TOKEN_ELSE : constant integer := 111;
TOKEN_WHILE : constant integer := 121;
TOKEN_LOOP : constant integer := 131;
TOKEN_CASE : constant integer := 141;
TOKEN_MHEN : constant integer := 151;
TOKEN_DECLARE : constant integer := 161;
TOKEN_FOR : constant integer := 171;
TOKEN_OTHERS : constant integer := 181;
TOKEN_RETURN : constant integer := 191;
TOKEN_EXIT : constant integer := 201;
TOKEN_PROCEDURE : constant integer := 211;
TOKEN_FUNCTION : constant integer := 221;
TOKEN_MITH : constant integer := 231;
TOKEN_USE : constant integer := 241;
TOKEN_PACKAGE : constant integer := 251;
TOKEN_BODY : constant integer := 261;
TOKEN_RANGE : constant integer := 271;
TOKEN_IN : constant integer := 281;
TOKEN_OUT : constant integer := 291;
TOKEN_SUBTYPE : constant integer := 301;
TOKEN_TYPE : constant integer := 311;
TOKEN_TS : constant integer := 321;
TOKEN_NULL : constant integer := 331;
TOKEN_ACCESS : constant integer := 341;
TOKEN_ARRAY : constant integer := 351;
TOKEN_DIGITS : constant integer := 361;
TOKEN_DELTA : constant integer := 371;
TOKEN_RECORD_STRUCTURE : constant integer := 381;
TOKEN_CONSTANT : constant integer := 391;
TOKEN_NEM : constant integer := 401;
TOKEN_EXCEPTION : constant integer := 411;
TOKEN_RENAMES : constant integer := 421;
TOKEN_PRIVATE : constant integer := 431;

68
TOKEN_TASK : constant integer := 44;
TOKEN_ENTRY : constant integer := 45;
TOKEN_ACCEPT : constant integer := 46;
TOKEN_DELAY : constant integer := 47;
TOKEN_SELECT : constant integer := 48;
TOKEN_TERMINATE : constant integer := 49;
TOKEN_ABORT : constant integer := 50;
TOKEN_SEPARATE : constant integer := 51;
TOKEN_RAISE : constant integer := 52;
TOKEN_GENERIC : constant integer := 53;
TOKEN_AT : constant integer := 54;
TOKEN_REVERSE : constant integer := 55;
TOKEN_DO : constant integer := 56;
TOKEN_GOTO : constant integer := 57;
TOKEN_OF : constant integer := 58;
TOKEN_ALL : constant integer := 59;
TOKENPragma : constant integer := 60;
TOKEN_AND : constant integer := 61;
TOKEN_OR : constant integer := 62;
TOKEN_NOT : constant integer := 63;
TOKEN_XOR : constant integer := 64;
TOKEN_MOD : constant integer := 65;
TOKEN_Rem : constant integer := 66;
TOKEN_ABSOLUTE : constant integer := 67;
TOKEN_ASTERISK : constant integer := 68;
TOKEN_SLASH : constant integer := 69;
TOKEN_EXPONENT : constant integer := 70;
TOKEN_PLUS : constant integer := 71;
TOKEN_MINUS : constant integer := 72;
TOKEN_HYPERSAND : constant integer := 73;
TOKEN_EQUALS : constant integer := 74;
TOKEN_NOT_EQUALS : constant integer := 75;
TOKEN_LESS_THAN : constant integer := 76;
TOKEN_LESS_THAN_EQUALS : constant integer := 77;
TOKEN_GREATER_THAN : constant integer := 78;
TOKEN_GREATER_THAN_EQUALS : constant integer := 79;
TOKEN_ASSIGNMENT : constant integer := 80;
TOKEN_SEMICOLON : constant integer := 81;
TOKEN_PERIOD : constant integer := 82;
TOKEN_LEFT_PAREN : constant integer := 83;
TOKEN_RIGHT_PAREN : constant integer := 84;
TOKEN_SEPARATOR : constant integer := 85;
TOKEN_RANGE_DOTS : constant integer := 86;
TOKEN_ARROW : constant integer := 87;
TOKEN_BAR : constant integer := 88;
TOKEN_BRACKETS : constant integer := 89;
TOKEN_LEFT_BRACKET : constant integer := 90;
TOKEN_RIGHT_BRACKET : constant integer := 91;

type TOKEN_RECORD_BUFFER_ARRAY is
array (0..TOKEN_ARRAY_SIZE) of TOKEN_RECORD_TYPE;

TOKEN_RECORD_BUFFER : TOKEN_RECORD_BUFFER_ARRAY;
CURRENT_TOKEN_RECORD : TOKEN_RECORD_TYPE;
LOOK_AHEAD_TOKEN : TOKEN_RECORD_TYPE;
TOKEN_ARRAY_INDEX : integer := 0;
LENGTH_OF_LINE : integer := 0;
PLACEHOLDER_INDEX : integer := 0;
COMMENT_COUNT : integer := 0;
DATA_FILE_SIZE : integer := 0;
STATUS_COUNTER : integer := 0;
FULL : boolean := FALSE;
FINISHED : boolean := FALSE;
RESERVE_MORD_TEST : boolean := FALSE;
FIRST_TIME_LOAD : boolean := TRUE;
PROCEDURE_TEST : boolean := FALSE;
ERROR : boolean := FALSE;
TYPE_PRESENT : boolean := FALSE;
DECLARATION : boolean;

TEST_FILE_NAME : string(1..LINESIZE);
DATA_FILE_NAME : string(1..LINESIZE);
DUMMY_FILE_NAME : string(1..LINESIZE);
EDIT_BUFFER : string(1..EDIT_LINE_SIZE);

ANSWER : character;
DATA_FILE1, DATA_FILE2 : file_type;
DATA_FILE3, DATA_FILE4 : file_type;
PARSER_ERROR : exception;
QUIT_TO_OS : exception;

procedure CLEARSCREEN;
procedure SYNTAX_ERROR(ERROR_MESSAGE : string);

end GLOBAL_PARSER;

package body GLOBAL_PARSER is

procedure CLEARSCREEN is
begin
put(ASCII.ESC "3J");
end CLEARSCREEN;

procedure SYNTAX_ERROR(ERROR_MESSAGE : string) is
begin
put("Incomplete ");
put(" at line number");
put(RESULT_FILE, "Incomplete ");
put(RESULT_FILE, ERROR_MESSAGE);
put(RESULT_FILE, " at line number");
raise PARSER_ERROR;
end SYNTAX_ERROR;

end GLOBAL_PARSER;
package HALSTEAD_DISPLAY is
    package NEW_INTEGER_IO is new TEXT_IO.INTEGER_IO(integer);
    use NEW_INTEGER_IO;

    package REAL_IO is new TEXT_IO.FLOAT_IO(float);
    use REAL_IO;

    procedure VIEW_OPERATORS;
    procedure OPERAND_MENU;
    procedure VIEW_SCOPE_STRUCTURES;
    procedure VIEW_VARIABLES;
    procedure VIEW_BLOCKS;
    procedure METRIC_CONCLUSIONS;
    function LOG2(NUMBER : float) return float;
    function NATURALLOG(NUMBER : float) return float;
end HALSTEAD_DISPLAY;

package body HALSTEAD_DISPLAY is
    -- this procedure provides a menu of selections for viewing Halstead's
    -- length metric data. It also loads the symbol table array.
    procedure HALSTEAD is
        DONE : boolean := FALSE;
        DUMMY_LINE_LENGTH : integer;
        LEXEME_NAME : string(1..LINESIZE);
    begin
        GET_FILENAME(TYPE_PRESENT);
        while not (DONE) loop
            CLEARSCREEN;
            new_line;
            put("****************************************************************************");
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
            put("#"); new_line;
Simply enter the number of your choice

1 - HALSTEAD OPERATORS

2 - HALSTEAD OPERANDS

3 - HALSTEAD METRIC CONCLUSIONS

4 - EXIT TO METRIC SELECTION MENU

5 - EXIT TO OPERATING SYSTEM

Choice = ";

get(ANSWER);

get_line(DUMMY_FILE_NAME, DUMMY_LINE_LENGTH); -- flush system input

-- load the symbol table array --

open(DATA_FILE3, in_file, DATA_FILE_NAME & ".rand");

get(DATA_FILE3, LAST_ENTRY_INDEX, 5);

get(DATA_FILE3, TOTAL_OPERAND_COUNT, 5);

get_line(DATA_FILE3, DUMMY_FILE_NAME, LENGTH_OF_LINE);

for I in 0..LAST_ENTRY_INDEX-1 loop

get(DATA_FILE3, SYMBOL_TABLE(I).SCOPE, 5);

get(DATA_FILE3, SYMBOL_TABLE(I).REFERENCE, 5);

get(DATA_FILE3, SYMBOL_TABLE(I).DECLARATION_TYPE, 5);

get_line(DATA_FILE3, LEXEME_NAME, LENGTH_OF_LINE);

NEW_NODE := new OPERAND_TYPE;

NEW_NODE.OPERAND := LEXEME_NAME;

NEW_NODE.SIZE := LENGTH_OF_LINE;

NEW_NODE.NEXTOPERAND := null;

SYMBOL_TABLE(I).LEXEME_ADDRESS := NEW_NODE;

end loop;

close(DATA_FILE3);

new_line(2);

case ANSWER is

when '1' => VIEW_OPERATORS;

when '2' => OPERAND_MENU;

when '3' => METRIC_CONCLUSIONS;

when '4' => DONE := TRUE;

when '5' => raise QUIT_TO_OS;

when others => null;

end case;
end loop;

end HALSTEAD;

-------------------------------------------------------------------

-- this procedure displays the Halstead operator metric data.

procedure VIEW_OPERATORS is

DONE : boolean;

OCCURENCES, OPERATORS_USED : integer := 0;

HOLD_CHARACTER : character;

LEXEME_NAME : string(1..LINESIZE);
begin
CLEARSCREEN;
open(DATA_FILE1, in_file, DATA_FILE_NAME & ".data");
put("*********************************************************" );
put("*********************************************************" ); new_line(2);

CONVERT_UPPER_CASE(DATA_FILE_NAME, DATA_FILE_SIZE);
pbutt Operator data for file ** " );
pbutt(ADJUST_LEXEME(DATA_FILE_NAME, DATA_FILE_SIZE));
pbutt(" "); new_line(2);
pbutt("OPERATOR UTILIZED" ); new_line;
pbutt("---------------------------------------------------------" );

for I in TOKEN_AND..TOKEN_ASSIGNMENT loop
get(DATA_FILE1, OCCURRENCES);
if OCCURRENCES /= 0 then
  case I is
  when TOKEN_AND =>
    butt(ADJUST_EDIT_BUFFER("Boolean 'AND'", 13));
  when TOKEN_OR =>
    butt(ADJUST_EDIT_BUFFER("Boolean 'OR'", 12));
  when TOKEN_NOT =>
    butt(ADJUST_EDIT_BUFFER("Boolean 'NOT'", 13));
  when TOKEN_XOR =>
    butt(ADJUST_EDIT_BUFFER("Boolean 'XOR'", 13));
  when TOKEN_MOD =>
    butt(ADJUST_EDIT_BUFFER("Modulus 'MOD'", 13));
  when TOKEN_REM =>
    butt(ADJUST_EDIT_BUFFER("Remainder 'REM'", 15));
  when TOKEN_ABSOLUTE =>
    butt(ADJUST_EDIT_BUFFER("Absolute Value 'ABS'", 20));
  when TOKEN_ASTERISK =>
    butt(ADJUST_EDIT_BUFFER("Multiplication '*'", 18));
  when TOKEN_SLASH =>
    butt(ADJUST_EDIT_BUFFER("Division '/'", 12));
  when TOKEN_EXPONENT =>
    butt(ADJUST_EDIT_BUFFER("Exponentiation '**'", 19));
  when TOKEN_PLUS =>
    butt(ADJUST_EDIT_BUFFER("Addition '+'", 12));
  when TOKEN_MINUS =>
    butt(ADJUST_EDIT_BUFFER("Subtraction '-'", 15));
  when TOKEN_AMPERSAND =>
    butt(ADJUST_EDIT_BUFFER("Catenation '&'", 14));
  when TOKEN_EQUALS =>
    butt(ADJUST_EDIT_BUFFER("Equality '='", 12));
  when TOKEN_NOT_EQUALS =>
    butt(ADJUST_EDIT_BUFFER("Inequality '/='", 15));
  when TOKEN_LESS_THAN =>
    butt(ADJUST_EDIT_BUFFER("Less Than '<'", 13));
  when TOKEN_LESS_THAN_EQUALS =>
    butt(ADJUST_EDIT_BUFFER("Less Than Equals '<='", 21));
  when TOKEN_GREATER_THAN =>
    butt(ADJUST_EDIT_BUFFER("Greater Than '>'", 16));
  when TOKEN_GREATER_THAN_EQUALS =>
    butt(ADJUST_EDIT_BUFFER("Greater Than Equals '=>'", 24));
  when others
    => null;
end case;
put(OCCURRENCES,5);
new_line;
end if; -- if occurrences /= 0
end loop;
get_line(DUMMY_FILE_NAME, LENGTH_OF_LINE);

for I in IF_CONSTRUCT..CASE_CONSTRUCT loop
get(DATA_FILE1, OCCURRENCES, 5);
if (OCCURRENCES /= 0) then
  case I is

73
when IF_CONSTRUCT =>
  put(ADJUST_EDIT_BUFFER("IF construct", 12));
when LOOP_CONSTRUCT =>
  put(ADJUST_EDIT_BUFFER("LOOP construct", 14));
when WHILE_CONSTRUCT =>
  put(ADJUST_EDIT_BUFFER("WHILE construct", 15));
when FOR_CONSTRUCT =>
  put(ADJUST_EDIT_BUFFER("FOR construct", 13));
when CASE_CONSTRUCT =>
  put(ADJUST_EDIT_BUFFER("CASE construct", 14));
when others => null;
end case;
put( OCCURENCES, 5); new_line;
end if;
end loop;

new_line;
put(ADJUST_EDIT_BUFFER("Number of individual operators used", 35));
get(DATA_FILE1, OPERATORSUSED);
put( OPERATORS_USED, 5));
new_line;
put(ADJUST_EDIT_BUFFER("Total number of occurrences", 26));
get(DATA_FILE1, OCCURENCES);
put(OCCURENCES, 5);
close(DATA_FILE1);

put("--- Enter any letter to continue ---");
new_line;
get(HOLD_CHARACTER);
end /ITEM_OPERATORS;

---------------------------------------------------------------
-- this procedure provides a menu of selections for viewing the--
-- Halstead operand metric data.
procedure OPERAND_MENU is
  DONE : boolean := FALSE;
  DUMMY_LINE_LENGTH : integer;
begin
while not (DONE) loop
  CLEARSCREEN;
  new_line;
  put(" "************** ""); new_line;
  put(" "************** ""); new_line;
  put(" "************** ""); new_line;
  put(" "************** ""); new_line;
  put(" "HALSTEAD OPERAND SELECTION MENU ");
  put(" "************** ""); new_line;
  put(" "HERE IS THE OPERAND DATA AVAILABLE ");
  put(" "************** ""); new_line;
  put(" "Simply enter the number of your choice ");
  put(" "************** ""); new_line;
  put(" 1 - PROCEDURE, FUNCTION, PACKAGE INFORMATION ");
  put(" "************** ""); new_line;
  put(" 2 - VARIABLES AND CONSTANTS INFORMATION ");
  new_line;
end loop;
end OPERAND_MENU;

END.

put("3 - TASKS AND BLOCKS INFORMATION")
put("4 - EXIT TO HALSTEAD SELECTION MENU")
put("5 - EXIT TO OPERATING SYSTEM")

put("Choice = ");
get(ANSWER);
get_line(DUMMY_FILE_NAME, DUMMY_LINE_LENGTH); -- flush system input

choice ANSWER is
  when '1' => VIEW_SCOPE_STRUCTURES;
  when '2' => VIEW_VARIABLES;
  when '3' => VIEW_BLOCKS;
  when '4' => DONE := TRUE;
  when '5' => raise QUIT_TO_OS;
  when others => null;
end case;
end OPERAND_MENU;

-- this procedure displays the Halstead operand metric data for
-- packages, procedures, and functions.
procedure VIEW_SCOPE_STRUCTURES is
  SCREEN_COUNTER : integer := 0;
  NAME : string(1..LINESIZE);
  SIZE, COUNT : integer;
  HOLD_CHARACTER : character;
begin
  CLEARSCREEN;
  put("**********************************");
  put("**********************************"); new_line(2);
  CONVERT_UPPERCASE( DATAFILENAME, DATAFILESIZE);
  for DECLARE_TYPE in PACKAGEDECLARE..FUNCTIONDECLARE loop
    case DECLARE_TYPE is
      when PACKAGE_DECLARE =>
        put("PACKAGES for file ");
      when PROCEDURE_DECLARE =>
        put("PROCEDURES for file ");
      when FUNCTION_DECLARE =>
        put("FUNCTIONS for file ");
      when others => null;
    end case;
  end loop;
end OPERAND_MENU;

  NAME := SYMBOL_TABLE(I).LEXEME_ADDRESS.OPERAND;
  SIZE := SYMBOL_TABLE(I).LEXEME_ADDRESS.SIZE;
  COUNT := SYMBOL_TABLE(I).REFERENCE;
  if SYMBOL_TABLE(I).DECLARATION_TYPE = DECLARE_TYPE then
    put(ADJUST_EDIT_BUFFER(NAME, SIZE));
    put(COUNT, 5); new_line;
    SCREEN_COUNTER := SCREEN_COUNTER + 1;
  end if;
end loop;
if (SCREEN_COUNTER = 10) then
  new_line(3);
end loop;

put("--- Enter any letter to continue --")
new_line;
get(HOLD_CHARACTER);
CLEARSCREEN;
SCREEN_COUNTER := 0;
case DECLARE_TYPE is
when PACKAGE_DECLARE =>
    put(" PACKAGES for file - ");
when PROCEDURE_DECLARE =>
    put(" PROCEDURES for file - ");
when FUNCTION_DECLARE =>
    put(" FUNCTIONS for file - ");
when others => null;
end case;
put(ADJUST_LEXEME(DATA_FILE_NAME, DATA_FILE_SIZE)); new_line(2);
put("NAME REFERENCED "); new_line;
put("---------------------------------------------------------"); new_line;
end if;
-- if screen_counter = 10
end if;
-- if symbol_table(i).declaration_type
for i in 0..(last_entry_index-1) loop
    NAME := SYMBOL_TABLE(I).LEXEME_ADDRESS.OPERAND;
    SIZE := SYMBOL_TABLE(I).LEXEME_ADDRESS.SIZE;
    COUNT := SYMBOL_TABLE(I).REFERENCE;
end loop;
--- for declare_type in
end VIEW_SCOPE_STRUCTURES;

-----------------------------------------------------------------------------------------------------------------------------------

--- this procedure displays the Halstead operand metric data for
--- variables, numeric constants, and global variables.

procedure VIEW_VARIABLES is
    SCREEN_COUNTER : integer := 0;
    NAME : string(1..LINESIZE);
    SIZE : integer;
    CONTINUE : boolean;
begin
    CLEARSCREEN;
    put("*************************************************************"); new_line(2);
    CONVERT_UPPER_CASE(DATA_FILE_NAME, DATA_FILE_SIZE);
    for DECLARE_TYPE in VARIABLE_DECLARE..NO_DECLARE loop
        SKIP := FALSE;
        CONTINUE := FALSE;
        case DECLARE_TYPE is
            when VARIABLE_DECLARE =>
                put(" VARIABLES for file - ");
            when CONSTANT_DECLARE =>
                put(" CONSTANTS for file - ");
            when NO_DECLARE =>
                put(" GLOBAL VARIABLES for file - ");
            when others => null;
        end case;
        put(ADJUST_LEXEME(DATA_FILE_NAME, DATA_FILE_SIZE)); new_line(2);
        put("NAME REFERENCED "); new_line;
        put("*************************************************************"); new_line;
    end loop;
    while not (SKIP) loop
        for I in 0..(LAST_ENTRY_INDEX-1) loop
            NAME := SYMBOL_TABLE(I).LEXEME_ADDRESS.OPERAND;
            SIZE := SYMBOL_TABLE(I).LEXEME_ADDRESS.SIZE;
            COUNT := SYMBOL_TABLE(I).REFERENCE;
        end loop;
        SKIP := TRUE;
    end while;
end VIEW_VARIABLES;
if (SYMBOL_TABLE(I).DECLARATION_TYPE = DECLARE_TYPE) then
  put(ADJUST_EDIT_BUFFER(NAME, SIZE));
  put(COUNT, 5); new_line;
  SCREEN_COUNTER := SCREEN_COUNTER + 1;
  if (SCREEN_COUNTER = 10) then
    new_line(3);
    put(" --- Enter 'S' to stop or any other letter to");
    put(" continue ");
    new_line;
    get(HOLD_CHARACTER);
    if (HOLD_CHARACTER = 'S') or (HOLD_CHARACTER = 's') then
      SKIP := TRUE;
    end if;
    CLEARSCREEN;
  end if;
  case DECLARE_TYPE is
    when VARIABLE_DECLARE =>
      put(" VARIABLES for file - ");
    when CONSTANT_DECLARE =>
      put(" CONSTANTS for file - ");
    when NO_DECLARE =>
      put(" GLOBAL VARIABLES for file - ");
    when others => null;
  end case;
  put(ADJUST_LEXEME(DATA_FILE_NAME, DATA_FILE_SIZE)); new_line(2);
  put("NAME REFERENCED");
  new_line;
  new_line;
  SCREEN_COUNTER := 0;
end if;
end if;
-- if symbol_table[i].declaration_type
if (I = LAST_ENTRY_INDEX - 1) then
  CONTINUE := TRUE;
end if;
-- if i = last_entry_index-1
exit when SKIP;
end loop;
-- for i in 0..(last_entry_index-1)
new_line;
exit when ((SKIP) or (CONTINUE));
end loop;
-- while not skip loop
new_line;
put(" --- Enter any letter to continue ---");
new_line;
get(HOLD_CHARACTER);
CLEARSCREEN;
SCREEN_COUNTER := 0;
-- for declare_type in
CLEARSEREN;
put("""";
new_line(2);
put(ADJUST_EDIT_BUFFER("Total number of operands used", 29));
put(LAST_ENTRY_INDEX - 1, 5); new_line;
put(ADJUST_EDIT_BUFFER("Total number of occurrences, all operands", 40));
put(TOTAL_OPERANDCOUNT, 5); new_line(3);
put(" --- Enter any letter to continue ---");
new_line;
get(HOLD_CHARACTER);
end VIEW_VARIABLES;

-- this procedure displays the Halstead operand metric data for
-- tasks, and blocks.
procedure VIEW_BLOCKS is
  SCREEN_COUNTER : integer := 0;
  NAME : string(1..LINESIZE);
  SIZE, COUNT : integer;
  HOLD_CHARACTER : character;
  CLEARSCREEN;
  C...
begin
CLEARSCREEN;
put("********************************************************************")
put("********************************************************************") new_line(2);
CONVERT_UPPER_CASE(DATA_FILE_NAME, DATA_FILE_SIZE);
for DECLARE_TYPE in TASK_DECLARE..BLOCK_DECLARE loop
  case DECLARE_TYPE is
    when TASK_DECLARE =>
      put(" TASKS for file ")
    when BLOCK_DECLARE =>
      put(" BLOCKS for file ")
    when others => null
  end case;
  put(ADJUST_LEXEME(DATA_FILE_NAME, DATA_FILE_SIZE)) new_line(2)
put("NAME REFERENCED") new_line;
put("--------------------------------------------------------") new_line;
for I in O..ILASTENTRYINDEX-1 loop
  NAME SYMBOLTABLE(I).LEXEMEADDRESS.OPERAND;
  SIZE SYMBOLTABLE(I).LEXEMEADDRESS.SIZE;
  COUNT SYMBOLTABLE(I).REFERENCE;
  if (SYMBOL-TABLE(I).DECLARATIONTYPE = DECLARETYPE) then
    put(ADJUST_EDITBUFFER(NAME, SIZE));
    putf(COUNT, 5); new_line;
    SCREEN_COUNTER := SCREEN_COUNTER + 1;
    if (SCREEN_COUNTER = 10) then
      new_line(3);
      put( new_line;
      CLEARSCREEN;
      SCREEN_COUNTER := 0;
      end case;
      put(ADJUST_LEXEME(DATA_FILE_NAME, DATA_FILE_SIZE)) new_line(2)
      put("NAME REFERENCED") new_line;
      put("--------------------------------------------------------") new_line;
      new_line;
      end if;
      -- if screen_counter = 10
      end if;
      -- if symbol_table(i).declaration_type
end loop;
new_line(2);
put("--------------------------------------------------------")
new_line;
get(HOLD_CHARACTER);
CLEARSCREEN;
SCREEN_COUNTER := 0;
end case;
end VIEW_BLOCKS;
for I in 0..(LAST_ENTRY_INDEX-1) loop
  NAME SYMBOL_TABLE(I).LEXEME_ADDRESS.OPERAND;
  SIZE SYMBOL_TABLE(I).LEXEME_ADDRESS.SIZE;
  COUNT SYMBOL_TABLE(I).REFERENCE;
  if (SYMBOL_TABLE(I).DECLARATIONTYPE = DECLARETYPE) then
    put(ADJUST_EDITBUFFER(NAME, SIZE));
    putf(COUNT, 5); new_line;
    SCREEN_COUNTER := SCREEN_COUNTER + 1;
    if (SCREEN_COUNTER = 10) then
      new_line(3);
      put(" Enter any letter to continue --") new_line;
      CLEARSCREEN;
      SCREEN_COUNTER := 0;
    end if;
    -- if screen_counter = 10
    end if;
    -- if symbol_table(i).declaration_type
end loop;
-- for I in 0..(last_entry_index-1)
new_line(2);
put(" Enter any letter to continue --")
new_line;
get(HOLD_CHARACTER);
CLEARSCREEN;
SCREEN_COUNTER := 0;
end loop;
-- for declare_type in
---------------------
-- this procedure calculates and displays all of the variables used in
-- evaluation of the Halstead length metric. The conclusions, which are
-- determined from the calculated lengths, are based on Halstead's
-- observations.
procedure METRIC_CONCLUSIONS is
HOLD_CHARACTER character;
LITTLE_NI, LITTLE_N2, BIG_N1, BIG_N2 : integer;
LOG_RESULT, DIFFERENCE, DISPARITY float;
ADD_RESULT integer;
begin
78
CLEARSCREEN;
open DATA_FILE4, in_file, DATA_FILE_NAME & ".hals"
get DATA_FILE4, BIG_N1, 51;
get DATA_FILE4, LITTLE_N1, 51;
get DATA_FILE4, BIG_N2, 51;
get DATA_FILE4, LITTLE_N2, 51;
ADD_RESULT := BIG_N1 + BIG_N2;
LOG_RESULT := float(LITTLE_N1) * LOG2(float(LITTLE_N1)) +
(float(LITTLE_N2) * LOG2(float(LITTLE_N2)));
DIFFERENCE := LOG_RESULT - float(ADD_RESULT);

DISPARITY := DIFFERENCE / float(TOTAL_LINES_INPUT);

if ((DISPARITY > 0.5) and (DISPARITY <= 1.0)) then
put("A very large positive disparity. Reasons:"); new_lines
put(" 1 - POSSIBILITY OF OPERANDS DECLARED BUT NOT USED "); new_line;
put(" There my be some variables which were declared "); new_line;
put(" but never referenced in the program"); new_line;
put(" 2 - USE OF GLOBAL VARIABLES."); new_line;
put(" A large number of the variables referenced were "); new_line;
put(" declared in the package instantiations by the "); new_line;
put(" WITH statements."); new_line;
elsif ((DISPARITY > 0.0) and (DISPARITY <= 0.5)) then
put("A small positive disparity. Reasons:"); new_line;
put(" 1 - SOME OF THE OPERANDS DECLARED HERE NOT USED "); new_line;
put(" There my be some variables which were declared "); new_line;
put(" but never referenced in the program"); new_line;
put(" 2 - SOME USE OF GLOBAL VARIABLES. "); new_line;
put(" A large number of the variables referenced were "); new_line;
put(" declared in the package instantiations by the "); new_line;
put(" WITH statements."); new_line;
elsif ((DISPARITY > -0.5) and (DISPARITY <= 0.0)) then
new_line;
put(" --- Enter any character to continue ---"); new_line;
get HOLD_CHARACTER;
CLEARSCREEN;
put("A small negative disparity. By Halstead's standards, this is "); new_line;
put("a well polished program. "); new_line;
put("However there may exist any of the following: "); new_line;
put(" 1 - CANCELLING OF OPERATORS "); new_line;
put(" The occurrence of an inverse cancels the effect of a "); new_line;
put("previous operator."); new_line;
put(" 2 - AMBIGUOUS OPERANDS "); new_line;
put(" Same operand represents two or more variables."); new_line;
put(" 3 - SYNONYMOUS OPERANDS "); new_line;
put(" Two or more operands represent the same variable."); new_line;
put(" 4 - COMMON SUBEXPRESSION "); new_line;
put(" The same subexpression occurs more than once. "); new_line;
put(" 5 - UNNECESSARY REPLACEMENTS "); new_line;
put(" A subexpression is assigned to a temporary "); new_line;
put(" variable which is used only once. "); new_line;
put(" 6 - UNFACTORED EXPRESSIONS "); new_line;
put(" Repetitions of operators and operands among unfactored "); new_line;
put(" terms."); new_line;
79
else
    new_line;
    --- Enter any character to continue ---"); new_line;
get(HOLD_CHARACTER);
CLEARSCREEN;
put("A large negative disparity. Halstead gives six reasons: "); new_line;
put("1 - CANCELLING OF OPERATORS "); new_line;
put("The occurrence of an inverse cancels the effect of a "); new_line;
put("previous operator."); new_line;
put("2 - AMBIGUOUS OPERANDS "); new_line;
put("Same operand represents two or more variables."); new_line;
put("3 - SYNONYMOUS OPERANDS "); new_line;
put("Two or more operands represent the same variable."); new_line;
put("4 - COMMON SUBEXPRESSION "); new_line;
put("The same subexpression occurs more than once."); new_line;
put("5 - UNNECESSARY REPLACEMENTS "); new_line;
put("A subexpression is assigned to a temporary "); new_line;
put("variable which is used only once."); new_line;
put("6 - UNFACTORED EXPRESSIONS "); new_line;
put("Repetitions of operators and operands among unfactored ");
put("terms. "); new_line;
end if;
new_line;
put("--- Enter any character to continue ---"); new_line;
get(HOLD_CHARACTER);
close DATAFILE4;
end

-- this function computes the log to the base 2 of a number by using 
-- natural logarithms.
function LOG2(NUMBER : float) return float is
    X, Y : float;
begin
    X := NUMBER;
    if (X < 0.0) then
        raise DEVICE_ERROR;
    end if;
    M := 0;
    while (X >= 2.0) loop
        M := M + 1;
        X := X/2.0;
    end loop;
    Y := 3.0 * (X - 1.0)/(X + 1.0);
    XP := Y * (Y - 2.0);
    B0 := 0.0;
    B1 := 0.0;
    for I in reverse 5..0 loop
        B2 := B1;
        B1 := B0;
        B0 := XP * B1 + B2;
    end loop;

-- this function computes the log to the base e of a number.
function LOG(NUMBER : float) return float is
begin
    X := NUMBER;
    if (X < 0.0) then
        raise DEVICE_ERROR;
    end if;
    M := 0;
    while (X >= 2.0) loop
        M := M + 1;
        X := X/2.0;
    end loop;
    Y := 3.0 * (X - 1.0)/(X + 1.0);
    XP := Y * (Y - 2.0);
    B0 := 0.0;
    B1 := 0.0;
    for I in reverse 5..0 loop
        B2 := B1;
        B1 := B0;
        B0 := XP * B1 + B2;
    end loop;
end LOG;

-- this function computes the natural logarithm of a number.
function NATURAL_LOG(NUMBER : float) return float is
    A : constant array (0..5) of float :=
        (0.68629150E+00, 0.67341785E-02, 0.11894142E-03, 0.25009347E-05, 0.57260501E-07, 0.13791205E-08);
    XP, Y, M, B0, B1, B2 : float;
begin
    XP := NUMBER;
    if (XP < 0.0) then
        raise DEVICE_ERROR;
    end if;
    M := 0;
    while (XP >= 2.0) loop
        M := M + 1;
        XP := XP/2.0;
    end loop;
    Y := 3.0 * (XP - 1.0)/(XP + 1.0);
    XP := Y * (Y - 2.0);
    B0 := 0.0;
    B1 := 0.0;
    for I in reverse 5..0 loop
        B2 := B1;
        B1 := B0;
        B0 := XP * B1 + B2;
    end loop;
BO := XP * B1 - B2 + A(I);
end loop;

return (float(M) * 0.69314718 + Y * (BO - B1));
end NATURAL_LOG;

end HALSTEAD_DISPLAY;
.package INITIAL_DISPLAY is
  procedure INITIAL_SCREEN;
  procedure INTRODUCTION;
.end INITIAL_DISPLAY;

.package body INITIAL_DISPLAY is
  -- this procedure opens all data files for the input file, starts the
  -- the parsing process, writes the metric data to the appropriate files,
  -- closes the data files, and prompts the user for further input files
  -- to parse.
  procedure INITIAL_SCREEN is
    DONE : boolean;
    begin
      FINISHED := FALSE;
      while not FINISHED loop
        open(RESULT_FILE, outfile, "RESULTS.ADA");
        GET_FILENAME(TYPE_PRESENT);
        if (TYPE_PRESENT) then
          open(TEST_FILE, in_file, TEST_FILE_NAME);
          create(DATA_FILE1, out_file, DATA_FILE_NAME & ".data");
          create(DATA_FILE2, out_file, DATA_FILE_NAME & ".misc");
          create(DATA_FILE3, out_file, DATA_FILE_NAME & ".rand");
          create(DATA_FILE4, out_file, DATA_FILE_NAME & ".hals");
        else
          open(TEST_FILE, in_file, TEST_FILE_NAME & ".ada");
          create(DATA_FILE1, out_file, DATA_FILE_NAME & ".data");
          create(DATA_FILE2, out_file, DATA_FILE_NAME & ".misc");
          create(DATA_FILE3, out_file, DATA_FILE_NAME & ".rand");
          create(DATA_FILE4, out_file, DATA_FILE_NAME & ".hals");
        end if;
        INITIALIZE_OPERAND_LIST(DATA_FILE_NAME, HEAD_NODE);
        if (COMPILATION) then
          new_line;
          CLEARSCREEN;
          put("Parse of Ada program");
          put("complete");
          new_line;
        end if;
      end while
    end if;
  end if;

WRITE_OPERATOR_TABLE(DATA_FILE1, DATA_FILE2, DATA_FILE4);
WRITE_OPERAND_TABLE(DATA_FILE3, DATA_FILE4);
WRITE_NESTING_TABLE(DATA_FILE2);
close(DATA_FILE1);
close(DATA_FILE2);
close(DATA_FILE3);
close(DATA_FILE4);
close(RESULT_FILE);
close(TEST_FILE);

while not (DONE) loop
    new_line(2);
    put("''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
-- this procedure produces the initial screen displayed to the user.
procedure INTRODUCTION is
HOLD_CHARACTER : character;
begin
CLEARSCREEN;
put(""; new_line);
put("WELCOME TO 'AdaMEASURE' ");
put(""; new_line);
put(""; new_line);
put(""; new_line);
put(""; new_line);
put("AUTHORED BY: LCDR JEFFREY L. NIEDER"; new_line);
put("", USN; new_line;
put("", USN; new_line;
put(""; new_line;
put("NAVAL POSTGRADUATE SCHOOL"; new_line;
put("DEPARTMENT OF COMPUTER SCIENCE"; new_line;
put("MONTEREY, CALIFORNIA"; new_line;
put(""; new_line;
put("31 OCTOBER 1986"; new_line;
put(""; new_line;
put(""; new_line;
put(""; new_line;
put(""; new_line;
put("This program provides an automated software ");
put("metric tool which ");
put("uses quantitative measures in an effort to suply the user with ");
put("helpful information about the static structure ");
put("of a given input ");
put("program. This program is public domain information. ");
put(""; new_line;
put(""; new_line;
put(""; new_line;
put(""; new_line;
put(""; new_line;
put("--- Enter any letter to continue ---"; new_line);
get(HOLD_CHARACTER);
end INTRODUCTION;
end INITIAL_DISPLAY;
package LOW_LEVEL_SCANNER is
   procedure GETIDENTIFIER(TOKEN_RECORD: in out TOKEN_RECORD_TYPE);
   procedure FLUSH_SEPARATORS(TOKEN_RECORD: in out TOKEN_RECORD_TYPE);
   procedure GET_DELIMITER(TOKEN_RECORD: in out TOKEN_RECORD_TYPE);
   procedure FLUSH_COMMENT(TOKEN_RECORD: in out TOKEN_RECORD_TYPE);
   procedure GET_CHARACTER_LITERAL(TOKEN_RECORD: in out TOKEN_RECORD_TYPE);
   procedure GET_STRING_LITERAL(TOKEN_RECORD: in out TOKEN_RECORD_TYPE);
end LOW_LEVEL_SCANNER;

package body LOW_LEVEL_SCANNER is
   -- an identifier can be any number of letters or digits following the
   -- first letter, with a single underscore allowed between any pair
   -- of letters and/or digits
   procedure GETIDENTIFIER(TOKEN_RECORD : in out TOKEN_RECORD_TYPE) is
      DONE : boolean := FALSE;
      begin
         while (not DONE) loop
            -- store the character in the lexeme buffer
            TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
            LEXEME_LENGTH := LEXEME_LENGTH + 1;
            if (LOOKAHEADONE_CHARACTER in UPPER_CASE_LETTER) or
               (LOOKAHEADONE_CHARACTER in LOWER_CASE_LETTER) or
               (LOOKAHEADONE_CHARACTER in DIGITS_TYPE) then
               GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEADONE_CHARACTER);
            elsif (LOOKAHEADONE_CHARACTER = '_') and (NEXT_CHARACTER = '_') then
               ERROR_MESSAGE(TOKEN_RECORD.TOKEN-TYPE); -- two consecutive underscores
            elsif (LOOKAHEADONE_CHARACTER = '_') then
               GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEADONE_CHARACTER);
            else
               DONE := TRUE; -- identifier token accepted
            end if;
         end loop;
      end GETIDENTIFIER;
end package body LOW_LEVEL_SCANNER;

-- this procedure removes all the separators, which are delineated
-- in GLOBAL, from the input code
procedure FLUSH_SEPARATORS(TOKEN_RECORD : in out TOKEN_RECORD_TYPE) is
    DONE : boolean := FALSE;
begin
    while (not DONE) loop
        TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
        LEXEME_LENGTH := LEXEME_LENGTH + 1;
        if ((LOOKAHEAD_ONE_CHARACTER = ' ') or
            (character'pos(LOOKAHEAD_ONE_CHARACTER) in FORMATORS) and
            (LOOKAHEAD_ONE_CHARACTER /= ASCII.CR)) then
            GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD.ONE_CHARACTER);
            else
                DONE := TRUE; -- completed flushing of separators
            end if;
        end loop;
    end FLUSH_SEPARATORS;
end FLUSH_SEPARATORS;

-- this procedure identifies both the simple and compound delimiters
-- which are delineated in GLOBAL
procedure GET_DELIMITER(TOKEN_RECORD : in out TOKEN_RECORD_TYPE) is
begin
    -- store the character in the lexeme buffer
    -- and increment the lexeme pointer
    TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
    LEXEME_LENGTH := LEXEME_LENGTH + 1;
    if (character'pos(NEXT_CHARACTER) in COMPOUND_DELIMITER) or
        (NEXT_CHARACTER = '.') or (NEXT_CHARACTER = '*') or
        (NEXT_CHARACTER = ':') or (NEXT_CHARACTER = '/') then
        if ((character'pos(LOOKAHEAD.ONE_CHARACTER) in COMPOUND_DELIMITER) or
            (LOOKAHEAD.ONE_CHARACTER = '.') or (LOOKAHEAD.ONE_CHARACTER = '*') then
            GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD.ONE_CHARACTER);
            else
                TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
                LEXEME_LENGTH := LEXEME_LENGTH + 1;
            end if;
        end if;
    end if;
end GET_DELIMITER;

-- this procedure removes all the comments from the input code
-- all comments start with a -- and end with a carriage return
procedure FLUSH_COMMENT(TOKEN_RECORD : in out TOKEN_RECORD_TYPE) is
    DONE : boolean := FALSE;
begin
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD.ONE_CHARACTER);
    while (not DONE) loop
        if (LOOKAHEAD.ONE_CHARACTER = ASCII.CR) then
            DONE := TRUE; -- end of comment
        else
            GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD.ONE_CHARACTER);
        end if;
    end loop;
end FLUSH_COMMENT;

-- this procedure identifies an individual character
-- formators are not allowed to be character literals
procedure GET_CHARACTER_LIT(TOKEN_RECORD : in out TOKEN_RECORD_TYPE) is
    STATE : positive := 1;
    DONE : boolean := FALSE;
begin
    86
while (not DONE) loop
  case STATE is
  -- store the character in the lexeme buffer
  -- and increment the lexeme pointer
  when 1 =>
    TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
    LEXEME_LENGTH := LEXEME_LENGTH + 1;
    if (LOOKAHEAD_ONE_CHARACTER = ")" then
      STATE := 2;
      GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
    elsif (character'pos! LOOKAHEAD_ONE_CHARACTER)
      in FORMATORS) then
      ERROR_MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
    else
      STATE := 3;
      GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);,
      and if
    when 2 =>
      if (LOOKAHEAD_ONE_CHARACTER = ")" then
        -- store the character in the lexeme buffer
        -- and increment the lexeme pointer
        TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
        LEXEME_LENGTH := LEXEME_LENGTH + 1;
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
        TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
      else
        -- one single quote found, classify as accent mark
        -- change token type from character literal to delimiter
        TOKEN_RECORD.TOKEN_TYPE := DELIMITER;
        NEXT_BUFFER_INDEX := CURRENT_BUFFER_INDEX;
        and if
      when others => null;
  end case,
  end loop;
end GET_CHARACTER_LIT;

-- this procedure identifies a string which is a sequence of zero or more characters between double quotes

procedure GET_STRING_LIT(TOKEN_RECORD : in out TOKEN_RECORD_TYPE is
  STATE : positive := 1;
  DONE : boolean := FALSE;
begin
  while (not DONE) loop
    -- store the character in the lexeme buffer
    -- and increment the lexeme pointer
    TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
    LEXEME_LENGTH := LEXEME_LENGTH + 1;
    case STATE is
      when 1 =>
        if (LOOKAHEAD_ONE_CHARACTER = "") then
          STATE := 2; -- two consecutive quotes seen
          else
            STATE := 4; -- one or more characters in the string
            and if
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
      when 2 =>
        if (LOOKAHEAD_ONE_CHARACTER = "") then
          -- three consecutive quotes seen
          GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
          else
            DONE := TRUE; -- string of zero characters accepted
            and if
    end case;
  end loop;
end GET_STRING_LIT;
when 3 => if (LOOKAHEAD_ONE_CHARACTER = '"') then
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
    TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
    L:EXEME LENGTH := L:EXEME LENGTH + 1;
    DONE := TRUE; -- four consecutive quotes
    -- string of one printable quote accepted
else
    STATE := 4;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
end if;

when 4 => if (LOOKAHEAD_ONE_CHARACTER = '"') then
    STATE := 5;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
elsif (ncharacter'pos(NEXT_CHARACTER) in FORMATORS) then
    ERROR_MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
else
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
end if;

when 5 => if (LOOKAHEAD_ONE_CHARACTER = '"') then
    STATE := 6;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
else
    DONE := TRUE; -- string literal accepted
    and if;

when 6 => if (LOOKAHEAD_ONE_CHARACTER = '"') then
    STATE := 5;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
else
    STATE := 4;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
end if;

when others => null;
end case;
end loop;
end GET_STRING_LIT;
end LOWLEVEL_SCANNER;
package MENU_DISPLAY is
  procedure MENU;
  procedure INITIAL_MENU;
  procedure VIEW_HENRY;
end MENU_DISPLAY;

package body MENU_DISPLAY is

  -- this procedure displays the metric selection menu from which the user
  -- can make the appropriate selection.

procedure MENU is
  DONE : boolean := FALSE;
begin
  while not DONE loop
    CLEARSCREEN;
    new_lines;
    put("*************************************************", new_lines);  
    put("*******************************", new_lines);
    put("***", new_lines);
    put("1 - 'HALSTEAD' METRIC INFORMATION", new_lines);
    put("2 - COMMENT AND NESTING METRIC INFORMATION", new_lines);
    put("3 - 'HENRY and KAFURA' METRIC INFORMATION", new_lines);
  end loop;
end MENU;
procedure INITIAL-MENU is
   DONE : boolean := FALSE;
begin
   INTRODUCTION;
   while not DONE loop
      CLEARSCREEN;
      put("4 - EXIT TO MAIN MENU"); new_line;
      put("5 - EXIT TO OPERATING SYSTEM"); new_line;
      put("Choice = "); new_line;
      get(ANSWER);
      get_line(DUMMY_FILE_NAME, LENGTH_OF_LINE);
      case ANSWER is
         when '1' => HALSTEAD;
         when '2' => VIEW_GENERAL;
         when '3' => VIEW_HENRY;
         when '4' => DONE := TRUE;
         when '5' => raise QUIT_TO_OS;
         when others => null;
      end case;
   end loop;
end INITIAL-MENU;

-- this procedure displays the main selection menu which allows the user
-- to choose to parse a file, view previously gathered data, or quit to
-- the operating system.
procedure INITIAL-MENU is
   DONE : boolean := FALSE;
begin
   INTRODUCTION;
   while not DONE loop
      CLEARSCREEN;
      put("4 - EXIT TO MAIN MENU"); new_line;
      put("5 - EXIT TO OPERATING SYSTEM"); new_line;
      put("Choice = "); new_line;
      get(ANSWER);
      get_line(DUMMY_FILE_NAME, LENGTH_OF_LINE);
      case ANSWER is
         when '1' => HALSTEAD;
         when '2' => VIEW_GENERAL;
         when '3' => VIEW_HENRY;
         when '4' => DONE := TRUE;
         when '5' => raise QUIT_TO_OS;
         when others => null;
      end case;
   end loop;
end INITIAL-MENU;
get_line(DUMMY_FILE_NAME, LENGTH_OF_LINE)); -- flush system input buffer
new_line(2);

 case ANSWER is
  when '1' => RESET_PARAMETERS;
  INITIAL_SCREEN;
  MENU;
  when '2' => MENU;
  when '3' => raise QUIT_TO_OS;
  when others => null;
  end case;
 end loop;
end INITIAL_MENU;

procedure VIEW_HENRY is
  HOLD_CHARACTER : character;
begin
  CLEARSCREEN;
  new_line;
  put("================================================================");
  new_line(2);
  put("This software metric has not yet been implemented into this");
  new_line;
  put("program. It is hoped that this information will be added in");
  new_line;
  put("the very near future.");
  new_line(2);
  put("================================================================");
  new_line(2);
  put("--- Enter any letter to continue ---");
  new_line;
  get(HOLD_CHARACTER);
end VIEW_HENRY;

end MENU_DISPLAY;
APPENDIX D

'ADAMEASURE' PROGRAM LISTING - PART 2

------------------------------
-- TITLE: AN ADA SOFTWARE METRIC
--
-- MODULE NAME: PACKAGE_HALSTEAD_METRIC
--
-- DATE CREATED: 04 OCT 86
--
-- LAST MODIFIED: 01 DEC 86
--
-- AUTHORS: LCDR JEFFREY L. NIEDER
--
-- LT KARL S. FAIRBANKS, JR.
--
-- DESCRIPTION: This package contains all of the declarations --
-- and data structures required by our metric, as well as --
-- all necessary procedures and functions for gathering --
-- and storing the metric data.
--
------------------------------

with GLOBAL, GLOBAL_PARSER, BYPASS_SUPPORT_FUNCTIONS, TEXT_IO;
use GLOBAL, GLOBAL_PARSER, BYPASS_SUPPORT_FUNCTIONS, TEXT_IO;

package HALSTEAD_METRIC is
  package NEW_INTEGER_IO is new TEXT_IO.INTEGER_IO(integer);
  use NEW_INTEGER_IO;

  SCOPE_ON : constant boolean := TRUE;
  SCOPE_OFF : constant boolean := FALSE;

  PACKAGE_DECLARE : constant integer := 0;
  PROCEDURE_DECLARE : constant integer := 1;
  FUNCTION_DECLARE : constant integer := 2;
  TASK_DECLARE : constant integer := 3;
  BLOCK_DECLARE : constant integer := 4;
  VARIABLE_DECLARE : constant integer := 5;
  CONSTANT_DECLARE : constant integer := 6;
  NO_DECLARE : constant integer := 7;

  IF_CONSTRUCT : constant integer := 0;
  LOOP_CONSTRUCT : constant integer := 1;
  WHILE_CONSTRUCT : constant integer := 2;
  FOR_CONSTRUCT : constant integer := 3;
  CASE_CONSTRUCT : constant integer := 4;
  IF_END : constant integer := 5;
  LOOP_END : constant integer := 6;
  CASE_END : constant integer := 7;

  NUMBER_OF_OPERANDS : constant integer := 500;
  FIRST_LEVEL_NEST : constant integer := 1;
  MAXIMUM_NESTING : constant integer := 15;

  type OPERATOR_ARRAY_TYPE is
    array (TOKEN_AND..TOKEN_ASSIGNMENT) of integer;

  type OPERAND_TYPE;
  type LINK is access OPERAND_TYPE;

  type OPERAND_TYPE is
    record
      OPERAND : string(1..LINESIZE);
      SIZE : natural;
      NEXT_OPERAND : LINK;
    end record;
type OPERAND_MATRIX is
record
  SCOPE : integer;
  REFERENCE : integer := 0;
  DECLARATION_TYPE : integer;
  LEXEME_ADDRESS : LINK;
end record;

type HALSTEAD_OPERAND_ARRAY is
array(0..NUMBER_OF_OPERANDS) of OPERAND_MATRIX;

type NESTED_COUNT_TYPE is
array(FIRST_LEVEL_NEST..MAXIMUM_NESTING) of integer;

type CONSTRUCT_COUNT_TYPE is
array(IF_CONSTRUCT..CASE_CONSTRUCT) of integers;

OPERATOR_ARRAY : OPERATOR_ARRAY_TYPE := (TOKEN_AND..TOKEN_ASSIGNMENT => 0);
NESTED_COUNT : NESTED_COUNT_TYPE := (FIRST_LEVEL_NEST..MAXIMUM_NESTING => 0);
CONSTRUCT_COUNT : CONSTRUCT_COUNT_TYPE := (IF_CONSTRUCT..CASE_CONSTRUCT => 0);
SYMBOL_TABLE : HALSTEAD_OPERAND_ARRAY;
HEAD_NODE, NEH_NODE : LINK;
DECLARE_TYPE : integer := VARIABLE_DECLARE;
CURRENT_NESTING_LEVEL : integer := 0;
MAXIMUM_NESTING_LEVEL : integer := 0;
TOTAL_OPERAND_COUNT : integer := 0;
NESTING_LINE_NUMBER : integer := 0;
SYMBOL_TABLE_INDEX : integer := 0;
SCOPE_LEVEL : integer;
LAST_ENTRY_INDEX : integer;
NESTED_LEVEL_INCREASE : boolean := TRUE;
NO_ITERATION : boolean := TRUE;

procedure OPERATOR_METRIC(OPERATOR_INDEX in integer;
  CONSUME, RESERVE_WORD_TEST in boolean); procedure OPERATOR_METRIC(HEAD_NODE : in out LINK;
  LEXEME_RECORD : in out TOKEN_RECORD_TYPE;
  DECLARE_TYPE : in integer); procedure INITIALIZE_OPERAND_LIST(DATA_FILE_NAME : in out string;
  HEAD_NODE : in out LINK); procedure ADD_SYMBOLITEM_TEMP_NODE : in out LINK; DECLARE_TYPE : in integer); function DUPLICATE_LEXEMEITEM_TEMP_NODE : in out LINK) return boolean;
procedure REFERENCE_UPDATE(SYMBOL_TABLE_INDEX : in out integer);
procedure WRITE_OPERATOR_TABLE(OUTPUT1, OUTPUT2, OUTPUT3 : in out file_type);
procedure WRITE_OPERAND_TABLE(OUTPUT1, OUTPUT2 : in out file_type);
procedure WRITE_NESTING_TABLE(OUTPUT_FILE : in out file_type);
end HALSTEAD_METRIC;

package body HALSTEAD_METRIC is
  -- this procedure updates the operator array based on the parsing of a
  -- valid Halstead operator.
  procedure OPERATOR_METRIC(OPERATOR_INDEX in integer;
    CONSUME, RESERVE_WORD_TEST in boolean) is
  begin
    if (CONSUME) and then not (RESERVE_WORD_TEST) then
      OPERATOR_ARRAY(OPERATOR_INDEX) := OPERATOR_ARRAY(OPERATOR_INDEX) + 1;
    end if;
end HALSTEAD_METRIC;
PROCEDURE OPERATOR_METRIC

-- this procedure builds the symbol table for the input file, and
-- calls the appropriate procedure for entering or updating Halstead
-- operand information.

procedure OPERAND_METRIC(HEAD_NODE : in out LINK;
                        LEXEME_RECORD : in out TOKEN_RECORD_TYPE;
                        DECLARE_TYPE : in integer) is

  TEMP_NODE, TRAILER : LINK;
  INPUT_LEXEME : string(1..LINESIZE);
  FOUND : boolean;
  SIZE : natural;
begin
  TRAILER := HEAD_NODE;
  TEMP_NODE := HEAD_NODE.NEXT_OPERAND;
  INPUT_LEXEME := LEXEME_RECORD.LEXEME;
  SIZE := LEXEME_RECORD.LEXEME_SIZE - 1;
  FOUND := FALSE;
  while (TEMP_NODE /= null) loop
    if (ADJUST_LEXEME(INPUT_LEXEME, SIZE) =
        ADJUST_LEXEME(TEMP_NODE.OPERAND, TEMP_NODE.SIZE)) then
      FOUND := TRUE;
    else
      TRAILER := TEMP_NODE;
      TEMP_NODE := TEMP_NODE.NEXT_OPERAND;
    end if;
    exit when FOUND;
  end loop;
  if not (FOUND) then
    NEW_NODE := new OPERAND_TYPE;
    NEW_NODE.OPERAND := INPUT_LEXEME;
    NEW_NODE.SIZE := LEXEME_RECORD.LEXEME_SIZE - 1;
    NEW_NODE.NEXT_OPERAND := null;
    TRAILER.NEXT_OPERAND := NEW_NODE;
    TEMP_NODE := NEW_NODE;
  end if;
  if not (DUPLICATE_LEXEME(TEMP_NODE)) then
    ADD_SYMBOL(TEMP_NODE, DECLARE_TYPE);
  else
    REFERENCE_UPDATE(SYMBOL_TABLE_INDEX);
  end if;
end OPERAND_METRIC;

-- this procedure initializes the head node for the symbol table.

procedure INITIALIZE_OPERAND_LIST(DATA_FILE_NAME : in out string;
                                    HEAD_NODE : in out LINK) is
begin
  HEAD_NODE := new OPERAND_TYPE;
  HEAD_NODE.OPERAND := DATA_FILE_NAME;
  HEAD_NODE.NEXT_OPERAND := null;
  SYMBOL_TABLE_INDEX := 0;
  SCOPE_LEVEL := 0;
  LAST_ENTRY_INDEX := 0;
end INITIALIZE_OPERAND_LIST;

-- this procedure adds all of the information about a variable that
-- is initially parsed.

procedure ADD_SYMBOL(TEMP_NODE : in out LINK; DECLARE_TYPE : in integer) is
begin
  SYMBOL_TABLE(LAST_ENTRY_INDEX).LEXEME_ADDRESS := TEMP_NODE;
  if (DECLARATION) then
    SYMBOL_TABLE(LAST_ENTRY_INDEX).DECLARATION := DECLARE_TYPE;
  end if;
  SYMBOL_TABLE(LAST_ENTRY_INDEX).SCOPE := SCOPE_LEVEL;
end ADD_SYMBOL;
SYMBOL_TABLE(LAST_ENTRY_INDEX).REFERENCE := 0;
else -- a global variable
if (DECLARE_TYPE = CONSTANT_DECLARE) then
  SYMBOL_TABLE(LAST_ENTRY_INDEX).DECLARATION_TYPE := CONSTANT_DECLARE;
else
  SYMBOL_TABLE(LAST_ENTRY_INDEX).DECLARATION_TYPE := NO_DECLARE;
end if;
SYMBOL_TABLE(LAST_ENTRY_INDEX).SCOPE := 0;
SYMBOL_TABLE(LAST_ENTRY_INDEX).REFERENCE := 1;
end if;
LAST_ENTRY_INDEX := LAST_ENTRY_INDEX + 1;
end ADD_SYMBOL;

---

-- this function determines if the current operand is already in the
-- symbol table. If located, the symbol table index is set to the
-- appropriate position.
function DUPLICATE_LEXEME(TMP_NODE : in LINK) return boolean is
  TEST_NAME : string(1..LINESIZE);
  INPUT_LEXEME : string(1..LINESIZE);
  INPUT_SIZE : natural;
  TEST_SIZE : natural;
  LOCATED : boolean;
begin
  LOCATED := FALSE;
  INPUT_LEXEME := TMP_NODE.OPAND;
  INPUT_SIZE := TMP_NODE.SIZE;
  for I in 0..(LAST_ENTRY_INDEX-1) loop
    TEST_NAME := SYMBOL_TABLE(I).LEXEME_ADDRESS.OPAND;
    TEST_SIZE := SYMBOL_TABLE(I).LEXEME_ADDRESS.SIZE;
    if (ADJUST_LEXEME(TEMP NODE, INPUT_SIZE) =
        SYMBOL_TABLE_INDEX(TEMP NODE, INPUT_SIZE)) then
      LOCATED := TRUE;
      SYMBOL_TABLE_INDEX := I;
    end if;
    exit when LOCATED;
  end loop;
  return (LOCATED);
end DUPLICATE_LEXEME;

---

-- this procedure updates the reference count when an operand is parsed,
-- after initial entry into the symbol table.
procedure REFERENCE_UPDATE(SYMBOL_TABLE_INDEX : in out integer) is
begin
  SYMBOL_TABLE(SYMBOL_TABLE_INDEX).REFERENCE :=
  SYMBOL_TABLE(SYMBOL_TABLE_INDEX).REFERENCE + 1;
  TOTAL_OPERAND_COUNT := TOTAL_OPERAND_COUNT + 1;
end REFERENCE_UPDATE;

---

-- this procedure writes the Halstead operator count data to the
-- appropriate files.
procedure WRITE_OPERATOR_TABLE(OUTPUT1, OUTPUT2, OUTPUT3 : in out file_type) is
  OPERATORS_USED, OCCURENCES : integer := 0;
begin
  for I in TOKEN_AND..TOKEN_ASSIGNMENT loop
    if (OPERATOR_ARRAY(I) /= 0) then
      OPERATORS_USED := OPERATORS_USED + 1;
      OCCURENCES := OCCURENCES + OPERATOR_ARRAY(I);
    end if;
    put(OUTPUT1, OPERATOR_ARRAY(I), 5);
  end loop;
  new_line(OUTPUT1); 
  for I in IF_CONSTRUCT, CASE CONSTRUCT loop
  put(OUTPUT1, OPERATOR_ARRAY(I), 5);
  end loop;
end WRITE_OPERATOR_TABLE;
if (CONSTRUCT_COUNT(I) /= 0) then
  OPERATORS_USED := OPERATORS_USED + 1;
  OCCURRENCES := OCCURRENCES + CONSTRUCT_COUNT(I);
end if;
  put(OUTPUT1, CONSTRUCT_COUNT(I), 5);
end loop;

put(OUTPUT1, OPERATORS_USED, 5);
put(OUTPUT1, OCCURRENCES, 5);

put(OUTPUT2, TOTAL_LINES_INPUT, 5);
put(OUTPUT2, COMMENT_COUNT, 5);

put(OUTPUT3, OPERATORS_USED, 5);
put(OUTPUT3, OCCURRENCES, 5);

end WRITE_OPERATOR_TABLE;

---------------------------------------------------------------------

-- this procedure writes the Halstead operand information to the
-- appropriate files.
procedure WRITE_OPERAND_TABLE(OUTPUT1, OUTPUT2 : in out file_type) is
  NAME : string(1..LINESIZE);
  SIZE : integer;
begin
  put(OUTPUT1, LAST_ENTRY_INDEX - 1, 5);
  put(OUTPUT2, LAST_ENTRY_INDEX - 1, 5);
  put(OUTPUT1, TOTAL_OPERAND_COUNT, 5);
  put(OUTPUT2, TOTAL_OPERAND_COUNT, 5);
  new_line(OUTPUT1);
  for I in 0..(LAST_ENTRY_INDEX-1) loop
    NAME := SYMBOL_TABLE(I).LEXEME_ADDRESS.OPERAND;
    SIZE := SYMBOL_TABLE(I).LEXEME_ADDRESS.SIZE;
    put(OUTPUT1, SYMBOL_TABLE(I).SCOPE, 5);
    put(OUTPUT1, SYMBOL_TABLE(I).REFERENCE, 5);
    put(OUTPUT1, SYMBOL_TABLE(I).DECLARATION_TYPE, 5);
    CONVERT_UPPERCASE(NAME, SIZE);
    put(OUTPUT1, ADJUST_LEXEME(NAME, SIZE));
    new_line(OUTPUT1);
  end loop;
end WRITE_OPERAND_TABLE;

---------------------------------------------------------------------

-- this procedure maintains the maximum nesting level attained, the number
-- of times each nesting level is reached, and the number of times each
-- nesting construct is utilized.
procedure NESTING_METRIC(NEST_TYPE : in integer) is
begin
  case NEST_TYPE is
  when IF, LOOP, CASE, WHILE, CONSTRUCT =>
    FOR_CONSTRUCT CASE_CONSTRUCT
    => CONSTRUCT_COUNT(NEST_TYPE) := CONSTRUCT_COUNT(NEST_TYPE) + 1;
    NESTED_LEVEL_INCREASE := TRUE;
    CURRENT_NESTING_LEVEL := CURRENT_NESTING_LEVEL + 1;
    if (CURRENT_NESTING_LEVEL > MAXIMUM_NESTING_LEVEL) then
      MAXIMUM_NESTING_LEVEL := CURRENT_NESTING_LEVEL;
      NESTING_LINE_NUMBER := TOTAL_LINES_INPUT;
    end if;

  when IF_END LOOP_END CASE_END =>
    if (NESTED_LEVEL_INCREASE) then
      NESTED_COUNT(CURRENT_NESTING_LEVEL) :=
      NESTED_COUNT(CURRENT_NESTING_LEVEL) + 1;
      NESTED_LEVEL_INCREASE := FALSE;
    end if;
    CURRENT_NESTING_LEVEL := CURRENT_NESTING_LEVEL - 1;
  end case;
when others => null;
end case;
end NESTING_METRIC;

-- this procedure writes the nesting metric data to the appropriate file.
procedure WRITE_NESTING_TABLE(OUTPUT_FILE : in out file_type) is
begin
  newline(OUTPUT_FILE);
  for I in IF_CONSTRUCT..CASE_CONSTRUCT loop
    put(OUTPUT_FILE, CONSTRUCT_COUNT(I), 5);
  end loop;
  newline(OUTPUT_FILE);
  put(OUTPUT_FILE, MAXIMUM_NESTING_LEVEL, 5);
  newline(OUTPUT_FILE);
  put(OUTPUT_FILE, NESTING_LINE_NUMBER, 5);
  newline(OUTPUT_FILE);
  for I in FIRST_LEVEL_NEST..MAXIMUM_NESTING loop
    put(OUTPUT_FILE, NESTED_COUNT(I), 5);
  end loop;
end WRITE_NESTING_TABLE;

end HALSTEAD_METRIC;
with GLOBAL, GET_NEXT_CHARACTER;
use GLOBAL, GET_NEXT_CHARACTER;

package NUMERIC is
  procedure GET_NUMERIC_LITERAL(TOKEN_RECORD : in out TOKEN_RECORD_TYPE);
end NUMERIC;

package body NUMERIC is
  procedure GET_NUMERIC_LITERAL(TOKEN_RECORD : in out TOKEN_RECORD_TYPE) is
    DONE : boolean := FALSE;
    STATE : positive := 1;
    begin
      while not DONE loop
        -- store the character in the lexeme buffer
        -- and increment the lexeme pointer
        TOKEN_RECORD.LEXEME(LEXEME_LENGTH) := NEXT_CHARACTER;
        LEXEME_LENGTH := LEXEME_LENGTH + 1;
        -- each option in the case statement is a state in the finite
        -- state automata for determining numeric literals. Ada allows
        -- the use of the underscore to aid readability of long numeric literals
        case STATE is
          when 1 => if (LOOKAHEAD_ONE_CHARACTER in DIGITS_TYPE) then
            STATE := 1;
            GET_NEXT_CHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
          elsif (LOOKAHEAD_ONE_CHARACTER = '.') then
            STATE := 2;
            GET_NEXT_CHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
          elsif 
            ((LOOKAHEAD_ONE_CHARACTER = 'E') or
             (LOOKAHEAD_ONE_CHARACTER = 'e')) then
            STATE := 17;
            GET_NEXT_CHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
          elsif (LOOKAHEAD_ONE_CHARACTER = '_') then
            STATE := 9;
            GET_NEXT_CHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
          elsif 
            ((LOOKAHEAD_ONE_CHARACTER = '#') or
             (LOOKAHEAD_ONE_CHARACTER = ':')) then
            STATE := 10;
            GET_NEXT_CHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
          elsif 
            (LOOKAHEAD_ONE_CHARACTER not in UPPER_CASE_LETTER) and
            (LOOKAHEAD_ONE_CHARACTER not in LOWER_CASE_LETTER) and
            (LOOKAHEAD.ONE_CHARACTER /= '') and
            (LOOKAHEAD.ONE_CHARACTER /= '') then
            DONE := TRUE; -- universal integer accepted
          else
            ERROR_MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
          end if;
        end case;
      end while;
    end GET_NUMERIC_LITERAL;
end NUMERIC;
when 2 => if (LOOKAHEAD_ONE_CHARACTER == '.') then -- test for range dots
    TOKEN_RECORD.LEXEME:=(LEXEME LENGTH - 1):=(.)
    LEXEME LENGTH := LEXEME LENGTH - 1;
    NEXT BUFFER INDEX := CURRENT BUFFER INDEX;
    DONE := TRUE; -- universal integer preceded these
    -- range dots
    elsif (LOOKAHEAD ONE CHARACTER in DIGITS TYPE) then
        STATE := 3;
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE CHARACTER);
    else
        ERROR MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
        end if;
when 3 => if (LOOKAHEAD ONE CHARACTER in DIGITS TYPE) then
    STATE := 5;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE CHARACTER);
    elsif ((LOOKAHEAD ONE_CHARACTER = 'E' or
        (LOOKAHEAD ONE_CHARACTER = 'e')) then
        STATE := 4;
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE_CHARACTER);
    elsif (LOOKAHEAD ONE_CHARACTER = '_') then
        STATE := 5;
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE_CHARACTER);
    elsif ((LOOKAHEAD ONE CHARACTER not in UPPER_CASE LETTER) and
        (LOOKAHEAD ONE CHARACTER not in LOWER_CASE LETTER) and
        (LOOKAHEAD ONE_CHARACTER /= '') and
        (LOOKAHEAD ONE_CHARACTER /= ' ')) then
        DONE := TRUE; -- universal real accepted
    else
        ERROR MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
        end if;
when 4 => if ((LOOKAHEAD ONE CHARACTER = '+' or
        (LOOKAHEAD ONE_CHARACTER = '-') then
        STATE := 6;
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE_CHARACTER);
    elsif (LOOKAHEAD ONE CHARACTER in DIGITS TYPE) then
        STATE := 7;
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE_CHARACTER);
    else
        ERROR MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
        end if;
when 5 6 8 9 =>
    if (LOOKAHEAD ONE CHARACTER in DIGITS TYPE) then
        case STATE is
        when 5 => STATE := 3;
        when 6 8 => STATE := 7;
        when 9 => STATE := 1;
        when others => null;
        end case;
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE_CHARACTER);
    else
        ERROR MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
        end if;
when 7 => if (LOOKAHEAD ONE CHARACTER in DIGITS TYPE) then
    STATE := 7;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE CHARACTER);
    elsif (LOOKAHEAD ONE_CHARACTER == '_') then
        STATE := 8;
        GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD ONE_CHARACTER);
    elsif ((LOOKAHEAD ONE_CHARACTER not in UPPER_CASE LETTER) and
        (LOOKAHEAD ONE_CHARACTER not in LOWER_CASE LETTER) and
        (LOOKAHEAD ONE_CHARACTER /= '') and
        (LOOKAHEAD ONE_CHARACTER /= ' ')) then
        DONE := TRUE; -- integer or real in scientific notation
    else
        ERROR MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
        end if;
end if;
when 10 12 14 16 =>
if ((LOOKAHEAD_ONE_CHARACTER in DIGITS_TYPE) or
    (LOOKAHEAD_ONE_CHARACTER in UPPER_CASE_HEX) or
    (LOOKAHEAD_ONE_CHARACTER in LOWER_CASE_HEX)) then
    case STATE is
      when 10 12 => STATE := 11;
      when 14 16 => STATE := 15;
      when others => null;
    end case;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
else
    ERROR_MESSAGE( TOKEN_RECORD, TOKEN_TYPE);
end if;
when 11 => if ((LOOKAHEAD_ONE_CHARACTER in DIGITS_TYPE) or
    (LOOKAHEAD_ONE_CHARACTER in UPPER_CASE_HEX) or
    (LOOKAHEAD_ONE_CHARACTER in LOWER_CASE_HEX)) then
    STATE := 11;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
  elsif ((LOOKAHEAD_ONE_CHARACTER = 'z') or
    (LOOKAHEAD_ONE_CHARACTER = 'Z')) then
    STATE := 13;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
  elsif ((LOOKAHEAD_ONE_CHARACTER = 'w') or
    (LOOKAHEAD_ONE_CHARACTER = 'W')) then
    STATE := 13;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
else
    ERROR_MESSAGE( TOKEN_RECORD, TOKEN_TYPE);
end if;
when 13 => if ((LOOKAHEAD_ONE_CHARACTER = 'E') or
    (LOOKAHEAD_ONE_CHARACTER = 'e')) then
    STATE := 17;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
  elsif ((LOOKAHEAD_ONE_CHARACTER not in UPPER_CASE_LETTER) and
    (LOOKAHEAD_ONE_CHARACTER not in LOWER_CASE_LETTER) and
    (LOOKAHEAD_ONE_CHARACTER /= 'u')) then
    DONE := TRUE; -- based integer accepted
else
    ERROR_MESSAGE( TOKEN_RECORD, TOKEN_TYPE);
end if;
when 15 => if ((LOOKAHEAD_ONE_CHARACTER in DIGITS_TYPE) or
    (LOOKAHEAD_ONE_CHARACTER in UPPER_CASE_HEX) or
    (LOOKAHEAD_ONE_CHARACTER in LOWER_CASE_HEX)) then
    STATE := 15;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
  elsif ((LOOKAHEAD_ONE_CHARACTER = 'z') or
    (LOOKAHEAD_ONE_CHARACTER = 'Z')) then
    STATE := 16;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
  elsif ((LOOKAHEAD_ONE_CHARACTER = 'w') or
    (LOOKAHEAD_ONE_CHARACTER = 'W')) then
    STATE := 13;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
else
    ERROR_MESSAGE( TOKEN_RECORD, TOKEN_TYPE);
end if;
when 17 => if (LOOKAHEAD_ONE_CHARACTER in DIGITS_TYPE) then
    STATE := 7;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
  elsif ((LOOKAHEAD_ONE_CHARACTER = '+') or
    (LOOKAHEAD_ONE_CHARACTER = '+')) then
    STATE := 6;
    GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);
else
    ERROR_MESSAGE( TOKEN_RECORD, TOKEN_TYPE);
end if;
when others => null;
100
and case;
end loop;
end GET_NUMERIC_LIT;

end NUMERIC;
with PARSER-1, PARSER-2, PARSER-3, BYPASS_FUNCTION, HALSTEAD-METRIC, GLOBAL_PARSER, GLOBAL, TEXT_IO;
use PARSER-1, PARSER-2, PARSER-3, BYPASS_FUNCTION, HALSTEAD-METRIC, GLOBAL_PARSER, GLOBAL, TEXT_IO;

package PARSER_0 is
  function COMPILATION return boolean;
  function COMPILATION_UNIT return boolean;
  function CONTEXT_CLAUSE return boolean;
  function BASIC_UNIT return boolean;
  function LIBRARY_UNIT return boolean;
  function SECONDARY_UNIT return boolean;
  function LIBRARY_UNIT_BODY return boolean;
  function SUBUNIT return boolean;
end PARSER_0;

package body PARSER_0 is

  -- COMPILATION --> [COMPILATION_UNIT]*
  function COMPILATION return boolean is
    begin
      put("In compilation "); new_line;
      if (COMPILATION_UNIT) then
        while (COMPILATION_UNIT) loop
          null;
        end loop;
        return (TRUE);
      else
        return (FALSE);
      end if;
    end COMPILATION;

  -- COMPILATION_UNIT --> CONTEXT_CLAUSE BASIC_UNIT
  function COMPILATION_UNIT return boolean is
    begin
      if (CONTEXT_CLAUSE) then
        if (BASIC_UNIT) then
          return (TRUE);
        else
          return (FALSE);
        end if;
      else
        return (FALSE);
      end if;
    end COMPILATION_UNIT;

end PARSER_0;
function CONTEXT_CLAUSE return boolean is
  begin
    while (BYPASS(TOKEN_WITH)) loop
      if not (WITH_OR_USE_CLAUSE) then
        SYNTAX_ERROR("Context clause");
        end if;
    while (BYPASS(TOKEN_USE)) loop
      if not (WITH_OR_USE_CLAUSE) then
        SYNTAX_ERROR("Context clause");
      end if;
    end loop;
    end loop;
  return (TRUE);
end CONTEXT_CLAUSE;

function BASIC_UNIT return boolean is
  begin
    if (LIBRARY_UNIT) then
      return (TRUE);
    elsif (SECONDARY_UNIT) then
      return (TRUE);
    else
      return (FALSE);
    end if;
  end BASIC_UNIT;

function LIBRARY_UNIT return boolean is
  begin
    if (BYPASS(TOKEN_PROCEDURE)) then
      DECLARE_TYPE := PROCEDURE_DECLARE;
      if (PROCEDURE_UNIT) then
        return (TRUE);
      else
        SYNTAX_ERROR("Library unit");
        end if;
    elsif (BYPASS(TOKEN_FUNCTION)) then
      DECLARE_TYPE := FUNCTION_DECLARE;
      if (FUNCTION_UNIT) then
        return (TRUE);
      else
        SYNTAX_ERROR("Library unit");
      end if;
    elsif (BYPASS(TOKEN_PACKAGE)) then
      DECLARE_TYPE := PACKAGE_DECLARE;
      if (PACKAGE_DECLARATION) then
        return (TRUE);
      else
        SYNTAX_ERROR("Library unit");
      end if;
    elsif (BYPASS(TOKEN_GENERIC)) then
      if (GENERIC_DECLARATION) then
        return (TRUE);
      else
        SYNTAX_ERROR("Library unit");
      end if;
    else
      return (FALSE);
    end if;
  end LIBRARY_UNIT;
end if;
end LIBRARY_UNIT;

-- SECONDARY_UNIT --> LIBRARY_UNIT_BODY
-- --> SUBUNIT
function SECONDARY_UNIT return boolean is
begin
  if (LIBRARY_UNIT_BODY) then
    return (TRUE);
  elsif (SUBUNIT) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end SECONDARY_UNIT;

-- LIBRARY_UNIT_BODY --> procedure PROCEDURE_UNIT
-- --> function FUNCTION_UNIT
-- --> package PACKAGE_DECLARATION
-- --> generic GENERIC_DECLARATION
function LIBRARY_UNIT_BODY return boolean is
begin
  if (BYPASS(TOKEN_PROCEDURE)) then
    DECLARE_TYPE := PROCEDURE_DECLARE;
    if (PROCEDURE_UNIT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Library unit body");
      end if;
  elsif (BYPASS(TOKEN_FUNCTION)) then
    DECLARE_TYPE := FUNCTION_DECLARE;
    if (FUNCTION_UNIT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Library unit body");
      end if;
  elsif (BYPASS(TOKEN_PACKAGE)) then
    DECLARE_TYPE := PACKAGE_DECLARE;
    if (PACKAGE_DECLARATION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Library unit body");
      end if;
  else
    return (FALSE);
  end if;
end LIBRARY_UNIT_BODY;

-- SUBUNIT --> separate (NAME) PROPER_BODY
function SUBUNIT return boolean is
begin
  if (BYPASS(TOKEN_SEPARATE)) then
    if (BYPASS(TOKEN_LEFT_PAREN)) then
      if (NAME) then
        if (BYPASS(TOKEN_RIGHT_PAREN)) then
          if (PROPER_BODY) then
            return (TRUE);
          else
            SYNTAX_ERROR("Subunit");
            end if;
        else
          SYNTAX_ERROR("Subunit");
          end if;
      else
        SYNTAX_ERROR("Subunit");
        end if;
    else
      SYNTAX_ERROR("Subunit");
      end if;
  else
    SYNTAX_ERROR("Subunit");
    end if;
end SUBUNIT;
else
  SYNTAX_ERROR("Subunit");
  end if;
else
  SYNTAX_ERROR("Subunit");
  end if;
else
  return (FALSE);
  end if;
end SUBUNIT;
end PARSER_0;

-- if name statement
-- if bypass(token_left_paren)
-- if bypass(token_separate)
package PARSER_1 is
  function GENERIC_DECLARATION return boolean;
  function GENERIC_PARAMETER_DECLARATION return boolean;
  function GENERIC_FORMAL_PART return boolean;
  function PROCEDURE_UNIT return boolean;
  function SUBPROGRAM_BODY return boolean;
  function FUNCTION_UNIT return boolean;
  function FUNCTION_INIT_TAIL return boolean;
  function FUNCTION_BODY return boolean;
  function FUNCTION_BODY_TAIL return boolean;
  function TASK_DECLARATION return boolean;
  function TASK_BODY return boolean;
  function TASK_BODY_TAIL return boolean;
  function PACKAGE_DECLARATION return boolean;
  function PACKAGE_UNIT return boolean;
  function PACKAGE_BODY return boolean;
  function PACKAGE_BODY_TAIL return boolean;
  function PACKAGE_TAIL_END return boolean;
  function DECLARATIVE_PART return boolean;
  function BASIC_DECLARATIVE_ITEM return boolean;
  function BASIC_DECLARATION return boolean;
  function LATER_DECLARATIVE_ITEM return boolean;
  function PROPER_BODY return boolean;
  function SEQUENCE_OF_STATEMENTS return boolean;
  function STATEMENT return boolean;
  function COMPOUND_STATEMENT return boolean;
  function BLOCK_STATEMENT return boolean;
  function IF_STATEMENT return boolean;
  function CASE_STATEMENT return boolean;
  function CASE_STATEMENT_ALTERNATIVE return boolean;
  function LOOP_STATEMENT return boolean;
  function EXCEPTION_HANDLER return boolean;
  function ACCEPT_STATEMENT return boolean;
  function SELECT_STATEMENT return boolean;
  function SELECT_STATEMENT_TAIL return boolean;
  function SELECT_ALTERNATIVE return boolean;
  function SELECT_ENTRY_CALL return boolean;
end PARSER_1;

package body PARSER_1 is
  -- GENERIC_DECLARATION --> [GENERIC_PARAMETER_DECLARATION ?]
  -- GENERIC_FORMAL_PART
function GENERIC_DECLARATION return boolean is
begin
if (GENERIC_PARAMETER_DECLARATION) then
  null;
end if;
if (GENERIC_FORMAL_PART) then
  return(TRUE);
else
  return(FALSE);
end if;
end GENERIC_DECLARATION;

-- GENERIC_PARAMETER_DECLARATION --> IDENTIFIER_LIST : [MODE?] NAME
--                     [:= EXPRESSION?]
--                       --> type private [DISCRIMINANT_PART?]
--                       --> is PRIVATE_TYPE_DECLARATION
--                       --> is GENERIC_TYPE_DEFINITION
--                       --> with procedure PROCEDURE_UNIT
--                       --> with function FUNCTION_UNIT

function GENERIC_PARAMETER_DECLARATION return boolean is
begin
if (IDENTIFIER_LIST) then
  if (MODE(token:colon)) then
    if (NAME) then
      null;
    end if;
    if (BYPASS(token:assignment)) then
      if (EXPRESSION) then
        null;
      else
        SYNTAX_ERROR("Generic parameter declaration");
      end if;
    else
      SYNATX_ERROR("Generic parameter declaration");
    end if;
  else
    SYNTAX_ERROR("Generic parameter declaration");
  end if;
elsif (BYPASS(token:semicolon)) then
  return(TRUE);
else
  SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (BYPASS(token:type)) then
  if (BYPASS(token:identifier)) then
    if (DISCRIMINANT_PART) then
      null;
    end if;
  else
    SYNTAX_ERROR("Generic parameter declaration");
  end if;
elsif (BYPASS(token:procedure)) then
  if (BYPASS(token:function)) then
    if (DISCRIMINANT_PART) then
      null;
    end if;
  else
    SYNTAX_ERROR("Generic parameter declaration");
  end if;
elsif (BYPASS(token:private)) then
  if (BYPASS(token:procedure)) then
    if (DISCRIMINANT_PART) then
      null;
    end if;
  else
    SYNTAX_ERROR("Generic parameter declaration");
  end if;
elsif (BYPASS(token:private)) then
  if (BYPASS(token:procedure)) then
    if (DISCRIMINANT_PART) then
      null;
    end if;
  else
    SYNTAX_ERROR("Generic parameter declaration");
  end if;
else
  SYNTAX_ERROR("Generic parameter declaration");
end if;
else
    SYNTAX_ERROR("Generic parameter declaration");
    end if; -- if bypass(token_is)
else
    SYNTAX_ERROR("Generic parameter declaration");
    end if; -- if bypass(token_identifier)
elif (BYPASS(TOKEN_WITH)) then
    if (BYPASS(TOKEN_PROCEDURE)) then
        DECLARE_TYPE := PROCEDURE_DECLARE;
        if (PROCEDURE_UNIT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Generic parameter declaration");
            end if; -- if procedure_unit statement
    elsif (BYPASS(TOKEN_FUNCTION)) then
        DECLARE_TYPE := FUNCTION_DECLARE;
        if (FUNCTION_UNIT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Generic parameter declaration");
            end if; -- if function_unit statement
    else
        SYNTAX_ERROR("Generic parameter declaration");
        end if; -- if bypass(token_procedure)
else
    return (FALSE);
    end if; -- if identifier_list
end GENERIC_PARAMETER_DECLARATION;

function GENERIC_FORMAL_PART return boolean is
begin
    if (BYPASS(TOKEN_PROCEDURE)) then
        DECLARE_TYPE := PROCEDURE_DECLARE;
        if (PROCEDURE_UNIT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Generic formal part");
            end if; -- if procedure_unit statement
    elsif (BYPASS(TOKEN_FUNCTION)) then
        DECLARE_TYPE := FUNCTION_DECLARE;
        if (FUNCTION_UNIT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Generic formal part");
            end if; -- if function_unit statement
    elsif (BYPASS(TOKEN_PACKAGE)) then
        DECLARE_TYPE := PACKAGE_DECLARE;
        if (PACKAGE_DECLARATION) then
            return (TRUE);
        else
            SYNTAX_ERROR("Generic formal part");
            end if; -- if package_declaration
else
    return (FALSE);
end if;
end GENERIC_FORMAL_PART;

function PROCEDURE_UNIT return boolean is
begin
    identifier [FORMAL_PART ?] is SUBPROGRAM_BODY
    identifier [FORMAL_PART ?]
    identifier [FORMAL_PART ?] renames NAME;
    end GENERIC_PARAMETER_DECLARATION;

begin

DECLARATION := TRUE;
if (BYPASS(TOKEN_IDENTIFIER)) then
  SCOPE_LEVEL := SCOPE_LEVEL + 1;
if (FORMAL_PART) then
  null)
end if;
if (BYPASS(TOKEN_TS)) then
  if (SUBPROGRAM_BODY) then
    SCOPE_LEVEL := SCOPE_LEVEL - 1;
    return (TRUE);
  else
    SYNTAX_ERROR("Procedure unit");
  end if;
else
  SYNTAX_ERROR("Procedure unit");
end if;
elsif (BYPASS(TOKEN_SEMICOLON)) then
  SCOPE_LEVEL := SCOPE_LEVEL - 1;
  return (TRUE);
elsif (BYPASS(TOKEN_RENAMES)) then
  if (NAME) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      SCOPE_LEVEL := SCOPE_LEVEL - 1;
      return (TRUE);
    else
      SYNTAX_ERROR("Procedure unit");
    end if;
  else
    SYNTAX_ERROR("Procedure unit");
  end if;
else
  SYNTAX_ERROR("Procedure unit");
end if;
else
  return (FALSE);
end if;
end PROCEDURE_UNIT;

---

-- SUBPROGRAM_BODY --> new NAME [GENERIC.ACTUAL_PART ?] ;
-- -> separate ;
-- --> <> ;
-- --> [DECLARATIVE_PART ?] begin SEQUENCE_OF_STATEMENTS
-- --> [exception [EXCEPTION_HANDLER]+ ?] end [DESIGNATOR ?];
-- --> NAME ;
function SUBPROGRAM_BODY return boolean is
begin
  DECLARATION := TRUE;
  if (BYPASS(TOKEN_NEW)) then
    if (NAME) then
      if (GENERIC.ACTUAL_PART) then
        null;
      end if;
    else
      if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Subprogram body");
      end if;
    else
      SYNTAX_ERROR("Subprogram body");
    end if;
  else
    SYNTAX_ERROR("Subprogram body");
  end if;
elsif (BYPASS(TOKEN_SEPARATE)) then
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Subprogram body");
  end if;
else
  SYNTAX_ERROR("Subprogram body");
end if;
elself (BYPASS(TOKEN_BRACKETS)) then
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Subprogram body");
  end if;
elself (BYPASS(TOKEN/bootstrap) then
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Subprogram body");
  end if;
else
  SYNTAX_ERROR("Subprogram body");
end if;
elsif (DECLARATIVE_PART) then
  if (BYPASS(TOKEN_BEGIN)) then
    DECLARATION := FALSE;
    if (SEQUENCE_OF_STATEMENTS) then
      if (BYPASS(TOKEN_EXCEPTION)) then
        if (EXCEPTION_HANDLER) then
          while (EXCEPTION_HANDLER) loop
            null;
          end loop;
        else
          SYNTAX_ERROR("Subprogram body");
          end if;
        end if;
      end if;
    end if;
  else
    SYNTAX_ERROR("Subprogram body");
  end if;
end if;
else
  SYNTAX_ERROR("Subprogram body");
end if;
elsif (NAME) then
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Subprogram body");
    end if;
  end if;
else
  return (FALSE);
end if;
function FUNCTION_UNIT return boolean is
begin
  DECLARATION := TRUE;
  if (DESIGNATOR) then
    SCOPE_LEVEL := SCOPE_LEVEL + 1;
    if (FUNCTION_UNIT_TAIL) then
      SCOPE_LEVEL := SCOPE_LEVEL - 1;
      return (TRUE));
    else
      SYNTAX_ERROR("Function unit");
      end if;
    else
      return (FALSE);
      end if;
    end if;
end FUNCTION_UNIT;}
return (FALSE);
end if;
end FUNCTION_UNIT_TAIL;

-- FUNCTION_BODY --> is [FUNCTION_BODY_TAIL ?]
--
function FUNCTION_BODY return boolean is
begin
  if (BYPASS(TOKEN_IS)) then
    if (FUNCTION_BODY_TAIL) then
      null;
    end if;
    return (TRUE);
  elsif (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end FUNCTION_BODY;

-- FUNCTION_BODY_TAIL --> separate ;
--
function FUNCTION_BODY_TAIL return boolean is
begin
  if (BYPASS(TOKEN_SEPARATE)) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Function body tail");
    end if;
  elsif (SUBPROGRAM_BODY) then
    return (TRUE);
  elsif (NAME) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Function body tail");
    end if;
  else
    return (FALSE);
  end if;
end FUNCTION_BODY_TAIL;

-- TASK_DECLARATION --> body TASK_BODY;
--
function TASK_DECLARATION return boolean is
begin
  DECLARATION := TRUE;
  if (BYPASS(TOKEN_TYPE)) then
    null;
  end if;
  if (BYPASS(TOKEN_BODY)) then
    if (TASK_BODY) then
      if (BYPASS(TOKEN_SEMICOLON)) then

112
return (TRUE);
else
SYNTAX_ERROR("Task declaration");
end if;
else
SYNTAX_ERROR("Task declaration");
end if; -- if task_body statement
elsif (BYPASS(TOKEN_IDENTIFIER)) then
-- if bypass(token_identifier)
SCOPE_LEVEL := SCOPE_LEVEL + 1;
if (BYPASS(TOKEN_IS)) then
-- if bypass(token_is)
if (TASKBODYTAIL) then
SCOPE_LEVEL := SCOPE_LEVEL - 1;
return (TRUE);
else
SYNTAX_ERROR("Task body");
end if; -- if bypass(token_is)
else
return (FALSE);-- if bypass(token_body)
end if;
end TASKDECLARATION;
-- TASK_BODY --> identifier is TASK_BODY_TAIL
function TASK_BODY return boolean is
begin
if (BYPASS(TOKEN_IDENTIFIER)) then
SCOPE_LEVEL := SCOPE_LEVEL + 1;
if (BYPASS(TOKEN_IS)) then
if (TASK_BODY_TAIL) then
SCOPE_LEVEL := SCOPE_LEVEL - 1;
return (TRUE);
else
SYNTAX_ERROR("Task body");
end if; -- if bypass(token_is)
else
SYNTAX_ERROR("Task body");
end if;
else
return (FALSE);-- if bypass(token_identifier)
end if;
end TASK_BODY;
-- TASK_BODY_TAIL --> separate
-- [DECLARATIVE_PART ?] begin SEQUENCE_OF_STATEMENTS
-- [exception [EXCEPTION_HANDLER]+ ?] end [identifier ?]
function TASK_BODY_TAIL return boolean is
begin
null;
end TASK_BODY_TAIL;
DECLARATION := TRUE;
if (BYPASS(TOKEN_SEPARATE)) then
return (TRUE);
elsif (DECLARATIVE_PART) then
if (BYPASS(TOKEN_BEGIN)) then
DECLARATION := FALSE;
if (SEQUENCE_OF_STATEMENTS) then
if (BYPASS(TOKEN_EXCEPTION)) then
if (EXCEPTION_HANDLER) then
while (EXCEPTION_HANDLER) loop
null;
end loop;
else
SYNTAX_ERROR("Task body tail");
end if;
-- if exception_handler statement
end if;
if (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
DECLARATION := TRUE;
return (TRUE);
else
SYNTAX_ERROR("Task body tail");
end if;
-- if bypass(token_end)
else
SYNTAX_ERROR("Task body tail");
end if;
-- if sequence_of_statements
else
SYNTAX_ERROR("Task body tail");
end if;
-- if bypass(token_begin)
elsif (BYPASS(TOKEN_BEGIN)) then
DECLARATION := FALSE;
if (SEQUENCE_OF_STATEMENTS) then
if (BYPASS(TOKEN_EXCEPTION)) then
if (EXCEPTION_HANDLER) then
while (EXCEPTION_HANDLER) loop
null;
end loop;
else
SYNTAX_ERROR("Task body tail");
end if;
-- if exception_handler statement
end if;
if (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
DECLARATION := TRUE;
return (TRUE);
else
SYNTAX_ERROR("Task body tail");
end if;
-- if bypass(token_end)
else
SYNTAX_ERROR("Task body tail");
end if;
-- if sequence_of_statements
else
return (FALSE);
end if;
-- if bypass(token_sep)
end TASK_BODY_TAIL;

-- PACKAGE_DECLARATION --> body PACKAGE_BODY
-- --> identifier PACKAGE_UNIT
function PACKAGE_DECLARATION return boolean is
begin
DECLARATION := TRUE;
if (BYPASS(TOKEN_BODY)) then
if (PACKAGE_BODY) then
return (TRUE)
else
    SYNTAX_ERROR("Package declaration");
end if;

elsif (BYPASS(TOKEN_IDENTIFIER)) then
    SCOPE_LEVEL := SCOPE_LEVEL + 1;
    if (PACKAGE_UNIT) then
        SCOPE_LEVEL := SCOPE_LEVEL - 1;
        return (TRUE);
    else
        SYNTAX_ERROR("Package declaration");
    end if;
else
    return (FALSE);
end if;

-- if bypass(token_package)
end PACKAGE_DECLARATION;

-- PACKAGE_BODY --> identifier is PACKAGE_BODY_TAIL
function PACKAGE_BODY return boolean is
begin
    if (BYPASS(TOKEN_IDENTIFIER)) then
        SCOPE_LEVEL := SCOPE_LEVEL + 1;
        if (BYPASS(TOKEN_IS)) then
            if (PACKAGE_BODY_TAIL) then
                SCOPE_LEVEL := SCOPE_LEVEL - 1;
                return (TRUE);
            else
                SYNTAX_ERROR("Package body");
            end if;
        else
            SYNTAX_ERROR("Package body");
        end if;
    end if;
else
    return (FALSE);
end if;

-- if bypass(token_identifier)
end PACKAGE_BODY;

-- PACKAGE_BODY_TAIL --> separate
-- -- [DECLARATIVE_PART ?] [begin SEQUENCE_OF_STATEMENTS
-- -- [exception [EXCEPTION_HANDLER]+ ?] ?]
-- end [identifier ?];
function PACKAGE_BODY_TAIL return boolean is
begin
    DECLARATION := TRUE;
    if (BYPASS(TOKEN_SEPARATE)) then
        if (BYPASS(TOKEN_SEMICOLON)) then
            return (TRUE);
        else
            SYNTAX_ERROR("Package body tail");
        end if;
    elsif (DECLARATIVE_PART) then
        DECLARATION := FALSE;
        if (BYPASS(TOKEN_BEGIN)) then
            if (SEQUENCE_OF_STATEMENTS) then
                if (BYPASS(TOKEN_EXCEPTION)) then
                    if (EXCEPTION_HANDLER) then
                        while (EXCEPTION_HANDLER) loop
                            null;
                        end loop;
                    else
                        SYNTAX_ERROR("Package body tail");
                    end if;
                else
                    SYNTAX_ERROR("Package body tail");
                end if;
            end if;
        else
            SYNTAX_ERROR("Package body tail");
        end if;
    end if;
else
    if (BYPASS(TOKEN_END)) then
        if (BYPASS(TOKEN_IDENTIFIER)) then
            ...
null;
end if; -- if bypass(token_identifier)
if (BYPASS(TOKEN_SEMICOLON)) then
DECLARATION := TRUE;
return (TRUE);
else
SYNTAX_ERROR("Package body tail");
end if; -- if bypass(token_semicolon)
else
SYNTAX_ERROR("Package body tail");
end if; -- if bypass(token_end)
elsif (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
DECLARATION := TRUE;
return (TRUE);
else
SYNTAX_ERROR("Package body tail");
end if; -- if bypass(token_semicolon)
else
SYNTAX_ERROR("Package body tail");
end if; -- if bypass(token_begin)
elsif (BYPASS(TOKEN_BEGIN)) then
DECLARATION := FALSE;
if (SEQUENCE_OF_STATEMENTS) then
if (BYPASS(TOKEN_EXCEPTION)) then
if (EXCEPTION_HANDLER) then
while (EXCEPTION_HANDLER) loop
null;
end loop;
else
SYNTAX_ERROR("Package body tail");
end if; -- if exception_handler statement
end if;
if (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
DECLARATION := TRUE;
return (TRUE);
else
SYNTAX_ERROR("Package body tail");
end if; -- if bypass(token_semicolon)
else
SYNTAX_ERROR("Package body tail");
end if; -- if bypass(token_end)
elsif (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Package body tail");
end if; -- if bypass(token_semicolon)
else
return (FALSE);
end if;
if bypass(token_separate)
function PACKAGE_UNIT return boolean is
begin
if (BYPASS(TOKEN_IS)) then
  if (PACKAGE_TAIL_END) then
    return (TRUE);
  else
    SYNTAX_ERROR("Package unit");
  end if;
else
  if (BYPASS(TOKEN_RENAMES)) then
    if (NAME) then
      if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Package unit");
      end if;
    else
      SYNTAX_ERROR("Package unit");
    end if;
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE,");
    else
      SYNTAX_ERROR("Package unit");
    end if;
  else
    return (FALSE);
    if (BYPASS(TOKEN_IS)) then
      null;
    end if;
  end if;
end PACKAGE_UNIT;

function PACKAGE_TAIL_END return boolean is
begin
if (BYPASS(TOKEN_NEW)) then
  if (NAME) then
    if (GENERIC_ACTUAL_PART) then
      null;
    end if;
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Package tail end");
    end if;
  else
    SYNTAX_ERROR("Package tail end");
  end if;
else
  if (BASIC_DECLARATIVE_ITEM) then
    while (BASIC_DECLARATIVE_ITEM) loop
      null;
    end loop;
  end if;
else
  if (BYPASS(TOKEN_PRIVATE)) then
    while (BASIC_DECLARATIVE_ITEM) loop
      null;
    end loop;
end if;
else
  if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      null;
    end if;
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Package tail end");
    end if;
  else
    SYNTAX_ERROR("Package tail end");
  end if;
else
  if (BYPASS(TOKEN_PRIVATE)) then
    null;
  end if;
end if;
end PACKAGE_TAIL_END;
while (BASIC_DECLARATIVE_ITEM) loop
null;
end loop;
if (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Package tail end");
and if; -- if bypass(token_semicolon)
else
SYNTAX_ERROR("Package tail end");
and if; -- if bypass(token_end)
elsif (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Package tail end");
and if; -- if bypass(token_semicolon)
else
return (FALSE);
end if; -- if bypass(token_new)
end if;
-- BASIC_DECLARATIVE_ITEM -->
BASIC_DECLARATIVE
--
REPRESENTATION_CLAUSE
--
use WITH_OR_USE_CLAUSE
function BASIC_DECLARATIVE_ITEM return boolean is
begin
if (BASIC_DECLARATION) then
return (TRUE);
elif (REPRESENTATION_CLAUSE) then
return (TRUE);
elif (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Basic declarative item");
and if;
else
return (FALSE);
and if;
end if;
end BASIC_DECLARATIVE_ITEM;

-- DECLARATIVE_PART -->
[DECLARATIVE_PART]* [LATER_DECLARATIVE_ITEM]*
function DECLARATIVE_PART return boolean is
begin
while (BASIC_DECLARATIVE_ITEM) loop
null;
end loop;
while (LATER_DECLARATIVE_ITEM) loop
null;
end loop;
return (TRUE);
end DECLARATIVE_PART;

-- BASIC_DECLARATION -->
type TYPE_DECLARATION
--
subtype SUBTYPE_DECLARATION
function BASIC_DECLARATION return boolean is
begin
   if (BYPASS(TOKEN_TYPE)) then
      if (TYPE_DECLARATION) then
         return (TRUE);
      else
         SYNTAX_ERROR("Basic declaration");
      end if;
   elsif (BYPASS(TOKEN_SUBTYPE)) then
      if (SUBTYPE_DECLARATION) then
         return (TRUE);
      else
         SYNTAX_ERROR("Basic declaration");
      end if;
   elsif (BYPASS(TOKEN_PROCEDURE)) then
      if (PROCEDURE_UNIT) then
         return (TRUE);
      else
         SYNTAX_ERROR("Basic declaration");
      end if;
   elsif (BYPASS(TOKEN_FUNCTION)) then
      if (FUNCTION_UNIT) then
         return (TRUE);
      else
         SYNTAX_ERROR("Basic declaration");
      end if;
   elsif (BYPASS(TOKEN_PACKAGE)) then
      if (PACKAGE_DECLARATION) then
         return (TRUE);
      else
         SYNTAX_ERROR("Basic declaration");
      end if;
   elsif (BYPASS(TOKEN_GENERIC)) then
      if (GENERIC_DECLARATION) then
         return (TRUE);
      else
         SYNTAX_ERROR("Basic declaration");
      end if;
   elsif (IDENTIFIER_DECLARATION) then
      return (TRUE);
   elsif (BYPASS(TOKEN_TASK)) then
      DECLARE_TYPE := TASK_DECLARE;
      if (TASK_DECLARATION) then
         return (TRUE);
      else
         SYNTAX_ERROR("Basic declaration");
      end if;
   else
      return (FALSE);
   end if;
end BASIC_DECLARATION;

function LATER_DECLARATIVE_ITEM return boolean is
begin
   if (PROPER_BODY) then
      -- check for body declaration
   end if;
end LATER_DECLARATIVE_ITEM;
return (TRUE);
elsif (BYPASS(TOKEN_GENERIC)) then
  if (GENERIC_DECLARATION) then
    return (TRUE);
  else
    SYNTAX_ERROR("Later declarative item");
  end if;
elsif (BYPASS(TOKEN_USE)) then
  if (WITH_OR_USE_CLAUSE) then
    return (TRUE);
  else
    SYNTAX_ERROR("Later declarative item");
  end if;
else
  return (FALSE);
end if;
end LATER_DECLARATIVE_ITEM;

-- PROPER_BODY --> procedure PROCEDURE_UNIT
--                function FUNCTION_UNIT
--                package PACKAGE_DECLARATION
--                task TASK_DECLARATION
function PROPER_BODY return boolean is
begin
  if (BYPASS(TOKEN_PROCEDURE)) then
    DECLARE_TYPE := PROCEDURE_DECLARE;
    if (PROCEDURE_UNIT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Proper body");
    end if;
  elsif (BYPASS(TOKEN_FUNCTION)) then
    DECLARE_TYPE := FUNCTION_DECLARE;
    if (FUNCTION_UNIT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Proper body");
    end if;
  elsif (BYPASS(TOKEN_PACKAGE)) then
    DECLARE_TYPE := PACKAGE_DECLARE;
    if (PACKAGE_DECLARATION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Proper body");
    end if;
  elsif (BYPASS(TOKEN_TASK)) then
    DECLARE_TYPE := TASK_DECLARE;
    if (TASK_DECLARATION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Proper body");
    end if;
  else
    return (FALSE);
  end if;
end PROPER_BODY;

-- SEQUENCE_OF_STATEMENTS --> {STATEMENT};
function SEQUENCE_OF_STATEMENTS return boolean is
begin
  if (STATEMENT) then
    while (STATEMENT) loop
      null;
    end loop;
    return (TRUE);
  end if;
end SEQUENCE_OF_STATEMENTS;
else
    return (FALSE);
end if;
end SEQUENCE_OF_STATEMENTS;

-- STATEMENT --> [LABEL ?] SIMPLE_STATEMENT
-- --> [LABEL ?] COMPOUND_STATEMENT

function STATEMENT return boolean is
begin
    if (LABEL) then
        null;
    end if;
    if (SIMPLE_STATEMENT) then
        return (TRUE);
    elsif (COMPOUND_STATEMENT) then
        return (TRUE);
    else
        return (FALSE);
    end if;
end if;
end STATEMENT;

-- COMPOUND_STATEMENT --> if IF_STATEMENT
-- --> case CASE_STATEMENT
-- --> LOOP_STATEMENT
-- --> BLOCK_STATEMENT
-- --> accept ACCEPT_STATEMENT
-- --> select SELECT_STATEMENT

function COMPOUND_STATEMENT return boolean is
begin
    if (BYPASS(TOKEN_IF)) then
        NESTING_METRIC(IF_CONSTRUCT);
        if (IF_STATEMENT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Compound statement");
        end if;
    elsif (BYPASS(TOKEN_CASE)) then
        NESTING_METRIC(CASE_CONSTRUCT);
        if (CASE_STATEMENT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Compound statement");
        end if;
    elsif (LOOP_STATEMENT) then
        return (TRUE);
    elsif (BLOCK_STATEMENT) then
        return (TRUE);
    elsif (BYPASS(TOKEN_ACCEPT)) then
        if (ACCEPT_STATEMENT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Compound statement");
        end if;
    elsif (BYPASS(TOKEN_SELECT)) then
        if (SELECT_STATEMENT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Compound statement");
        end if;
    else
        return (FALSE);
    end if;
end if;
end COMPOUND_STATEMENT;
-- BLOCK_STATEMENT --> [identifier : ?] [declare DECLARATIVE_PART ?]
-- begin SEQUENCE_OF_STATEMENTS [exception
-- [EXCEPTION_HANDLER] ?] ? end [identifier ?] ;

function BLOCK_STATEMENT return boolean is

DECLARE_STATUS : boolean;
begin
if (DECLARATION) then
DECLARE_STATUS := TRUE;
else
DECLARATION := TRUE;
DECLARE_STATUS := FALSE;
end if;

DECLARE_TYPE := BLOCK_DECLARE;
if (BYPASS(TOKEN_IDENTIFIER)) then
SCOPE_LEVEL := SCOPE_LEVEL + 1;
if (BYPASS(TOKEN_COLON)) then
SCOPE_LEVEL := SCOPE_LEVEL - 1;
else
SYNTAX_ERROR("Block statement");
end if;
end if;
else
DECLARE_TYPE := VARIABLE_DECLARE;
end if;
if (BYPASS(TOKEN_DECLARE)) then
SCOPE_LEVEL := SCOPE_LEVEL + 1;
if (DECLARATIVE_PART) then
null;
else
SYNTAX_ERROR("Block statement");
end if;
end if;
if (BYPASS(TOKEN_BEGIN)) then
DECLARATION := FALSE;
if (SEQUENCE_OF_STATEMENTS) then
if (BYPASS(TOKEN_EXCEPTION)) then
if (EXCEPTION_HANDLER) then
while (EXCEPTION_HANDLER) loop
null;
end loop;
else
SYNTAX_ERROR("Block statement");
end if;
end if;
if (BYPASS(TOKEN_END)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
SCOPE_LEVEL := SCOPE_LEVEL - 1;
DECLARATION := TRUE;
return (TRUE);
else
SYNTAX_ERROR("Block statement");
end if;
else
SYNTAX_ERROR("Block statement");
end if;
else
SYNTAX_ERROR("Block statement");
end if;
end if;
end BLOCK_STATEMENT;
-- IF_STATEMENT --> EXPRESSION then SEQUENCE_OF_STATEMENTS
--
[elseif EXPRESSION then SEQUENCE_OF_STATEMENTS]
[else SEQUENCE_OF_STATEMENTS ?] and if;

function IF_STATEMENT return boolean is
begin
if (EXPRESSION) then
  if (BYPASS(TOKEN_THEN)) then
    if (SEQUENCE_OF_STATEMENTS) then
      while (BYPASS(TOKEN_ELSEIF)) loop
        if (EXPRESSION) then
          if (BYPASS(TOKEN_THEN)) then
            if not (SEQUENCE_OF_STATEMENTS) then
              SYNTAX_ERROR("If statement");
            end if;
            -- if not sequence_of_statements
          else
            SYNTAX_ERROR("If statement");
          end if;
          -- if bypass(token_then)
        else
          SYNTAX_ERROR("If statement");
        end if;
        -- if expression statement
      end loop;
    if (BYPASS(TOKEN_ELSE)) then
      if (SEQUENCE_OF_STATEMENTS) then
        null;
      else
        SYNTAX_ERROR("If statement");
      end if;
      -- if sequence_of_statements
    end if;
    -- if bypass(token_else)
  else
    SYNTAX_ERROR("If statement");
  end if;
  -- if bypass(token_end)
else
  SYNTAX_ERROR("If statement");
end if;
-- if bypass(token_if)
else
  SYNTAX_ERROR("If statement");
end if;
-- if bypass(token_end)
else
  SYNTAX_ERROR("If statement");
end if;
-- if sequence_of_statements
else
  SYNTAX_ERROR("If statement");
end if;
-- if bypass(token_then)
else
  return (FALSE);
end if;
-- if expression statement
end IF_STATEMENT;

-- CASE_STATEMENT --> EXPRESSION is [CASE_STATEMENT_ALTERNATIVE]+ end case;
function CASE_STATEMENT return boolean is
begin
if (EXPRESSION) then
  if (BYPASS(TOKEN_IS)) then
    if (CASE_STATEMENT_ALTERNATIVE) then
      while (CASE_STATEMENT_ALTERNATIVE) loop
        null;
      end loop;
    if (BYPASS(TOKEN_END)) then
      if (BYPASS(TOKEN_CASE)) then
        if (BYPASS(TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          SYNTAX_ERROR("If statement");
        end if;
        -- if bypass(token_if)
      end if;
      -- if bypass(token_else)
    else
      SYNTAX_ERROR("If statement");
    end if;
    -- if sequence_of_statements
  else
    SYNTAX_ERROR("If statement");
  end if;
  -- if bypass(token_end)
else
  SYNTAX_ERROR("If statement");
end if;
-- if bypass(token_if)
else
  SYNTAX_ERROR("If statement");
end if;
-- if bypass(token_end)
else
  SYNTAX_ERROR("If statement");
end if;
-- if sequence_of_statements
else
  SYNTAX_ERROR("If statement");
end if;
-- if bypass(token_then)
else
  return (FALSE);
end if;
-- if expression statement
end CASE_STATEMENT;
NESTING_METRIC(CASE_END);
return (TRUE);
else
SYNTAX_ERROR("Case statement");
end if;
else
SYNTAX_ERROR("Case statement");
end if;
else
SYNTAX_ERROR("Case statement");
end if;
else
SYNTAX_ERROR("Case statement");
end if;
else
return (FALSE);
end if;
expression statement
end CASE_STATEMENT;
------------------------------------------
-- CASE_STATEMENT_ALTERNATIVE --> when CHOICE [ CHOICE] =>
--
function CASE_STATEMENT_ALTERNATIVE return boolean is
begin
if (BYPASS(TOKEN_WHEN)) then
if (CHOICE) then
while (BYPASS(TOKEN_BAR)) loop
if not (CHOICE) then
SYNTAX_ERROR("Case statement alternative");
end if;
end loop;
if (BYPASS(TOKEN_ARROW)) then
if (SEQUENCE_OF_STATEMENTS) then
return (TRUE);
else
SYNTAX_ERROR("Case statement alternative");
end if;
else
SYNTAX_ERROR("Case statement alternative");
end if;
else
SYNTAX_ERROR("Case statement alternative");
end if;
else
return (FALSE);
end if;
end CASE_STATEMENT_ALTERNATIVE;
------------------------------------------
-- LOOP_STATEMENT --> [identifier :?] [ITERATION_SCHEME :?] loop
--
function LOOP_STATEMENT return boolean is
begin
if (BYPASS(TOKEN_IDENTIFIER)) then
if (BYPASS(TOKEN_COLON)) then
null;
else
SYNTAX_ERROR("Loop statement");
end if;
end if;
if (ITERATION_SCHEME) then
NO ITERATION := FALSE;
end if;
if (BYPASS(TOKEN_LOOP)) then
if (NO_ITERATION) then
  NESTING_METRIC(LOOP_CONSTRUCT);
else
  NO_ITERATION := TRUE;
  end if;
if (SEQUENCE_OF_STATEMENTS) then
  if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_LOOP)) then
      if (BYPASS(TOKEN_IDENTIFIER)) then
        null;
      end if;
      -- if bypass(token_identifier)
    else
      NESTING_METRIC(LOOP_END);
      return (TRUE);
    end if;
    SYNTAX_ERROR("Loop statement");
  end if;
  -- if bypass(token_loop)
else
  SYNTAX_ERROR("Loop statement");
  end if;
else
  SYNTAX_ERROR("Loop statement");
  end if;
  -- if sequence_of_statements
return (FALSE);
end if;
-- if bypass(token_loop)
end LOOP_STATEMENT;

-- EXCEPTION_HANDLER --> when EXCEPTION_CHOICE [ EXCEPTION_CHOICE]* =>
-- SEQUENCE_OF_STATEMENTS
function EXCEPTION_HANDLER return boolean is
begin
  if (BYPASS(TOKEN_WHEN)) then
    if (EXCEPTION_CHOICE) then
      while (BYPASS(TOKEN_BAR)) loop
        if not (EXCEPTION_CHOICE) then
          SYNTAX_ERROR("Exception handler");
        end if;
        -- if not exception_choice
      end loop;
      if (BYPASS(TOKEN_ARROW)) then
        if (SEQUENCE_OF_STATEMENTS) then
          return (TRUE);
        end if;
        -- if sequence_of_statements
      else
        SYNTAX_ERROR("Exception handler");
      end if;
      -- if bypass(token_arrow)
else
  SYNTAX_ERROR("Exception handler");
  end if;
else
  SYNTAX_ERROR("Exception handler");
  end if;-- if exception_choice statement
else
  return (FALSE);
  -- if bypass(token_when)
end EXCEPTION_HANDLER;

-- ACCEPT_STATEMENT --> identifier [(EXPRESSION) ?] [FORMAL_PART ?]
-- [do SEQUENCE_OF_STATEMENTS and identifier ?] ?)
function ACCEPT_STATEMENT return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    if (BYPASS(TOKEN_LEFT_PAREN)) then

if (EXPRESSION) then
  if (BYPASS(TOKEN_RIGHT_PAREN)) then
    null;
  else
    SYNTAX_ERROR("Accept statement");
  end if;
else
  SYNTAX_ERROR("Accept statement");
end if;

if (FORMAL_PART) then
  null;
else
  SYNTAX_ERROR("Accept statement");
end if;

if (SEQUENCE_OF_STATEMENTS) then
  if (BYPASS(TOKEN_END)) then
    null;
  else
    SYNTAX_ERROR("Accept statement");
  end if;
else
  SYNTAX_ERROR("Accept statement");
end if;

if (BYPASS(TOKEN_ID)) then
  null;
else
  SYNTAX_ERROR("Accept statement");
end if;

-- SELECT_STATEMENT --> SELECT_STATEMENT_TAIL SELECT_ENTRY_CALL and select :
begin
  if (SELECT_STATEMENT_TAIL) then
    if (SELECT_ENTRY_CALL) then
      if (BYPASS(TOKEN_END)) then
        if (BYPASS(TOKEN_SELECT)) then
          if (BYPASS(TOKEN_SEMICOLON)) then
            return (TRUE);
          else
            SYNTAX_ERROR("Select statement");
          end if;
        else
          SYNTAX_ERROR("Select statement");
        end if;
      else
        SYNTAX_ERROR("Select statement");
      end if;
    else
      SYNTAX_ERROR("Select statement");
    end if;
  else
    SYNTAX_ERROR("Select statement");
  end if;
else
  return (FALSE);
end if;

and ACCEPT_STATEMENT;

-- SELECT_STATEMENT_TAIL --> SELECT_ALTERNATIVE [or SELECT_ALTERNATIVE]*
-- NAME ] [SEQUENCE_OF_STATEMENTS ?]
begin
  if (SELECT_STATEMENT_TAIL) then
    if (SELECT_ENTRY_CALL) then
      if (BYPASS(TOKEN_END)) then
        if (BYPASS(TOKEN_SELECT)) then
          if (BYPASS(TOKEN_SEMICOLON)) then
            return (TRUE);
          else
            SYNTAX_ERROR("Select statement");
          end if;
        else
          SYNTAX_ERROR("Select statement");
        end if;
      else
        SYNTAX_ERROR("Select statement");
      end if;
    else
      SYNTAX_ERROR("Select statement");
    end if;
  else
    SYNTAX_ERROR("Select statement");
  end if;
end SELECT_STATEMENT_TAIL;
begin
  if (SELECT_ALTERNATIVE) then
    while (BYPASS(TOKEN_OR)) loop
      if not (SELECT_ALTERNATIVE) then
        SYNTAX_ERROR("Select statement tail");
      end if;
    end loop;
    return (TRUE);
  elsif (NAME) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      if (SEQUENCE_OF_STATEMENTS) then
        null;
      end if;
    else
      SYNTAX_ERROR("Select statement tail");
    end if;
  else
    return (FALSE);
  end if;
end SELECT_STATEMENT_TAIL;

-- SELECT_ALTERNATIVE --> [when EXPRESSION => ?] accept ACCEPT_STATEMENT
-- [SEQUENCE_OF_STATEMENTS ?]
-- --> [when EXPRESSION => ?] delay DELAY_STATEMENT
-- [SEQUENCE_OF_STATEMENTS ?]
-- --> [when EXPRESSION => ?] terminate ;

function SELECT_ALTERNATIVE return boolean is
begin
  if (BYPASS(TOKEN_WHEN)) then
    if (EXPRESSION) then
      if (BYPASS(TOKEN_ARROW)) then
        null;
      else
        SYNTAX_ERROR("Select alternative");
      end if;
    else
      SYNTAX_ERROR("Select alternative");
    end if;
  else
    return (FALSE);
  end if;
end SELECT_ALTERNATIVE;
function SELECT_ENTRY_CALL return boolean is
begin
  if (BYPASS(TOKEN_ELSE)) then
    if (SEQUENCE_OF_STATEMENTS) then
      return (TRUE);
    else
      SYNTAX_ERROR("Select entry call");
    end if;
  elsif (BYPASS(TOKEN_OR)) then
    if (BYPASS(TOKEN_DELAY)) then
      if (DELAY_STATEMENT) then
        null;
      end if;
    end if;
  else
    SYNTAX_ERROR("Select entry call");
  end if;
else
  SYNTAX_ERROR("Select entry call");
end if;
end SELECT_ENTRY_CALL;

end PARSER_1;
APPENDIX E

'ADAMEASURE' PROGRAM LISTING - PART 3

******************************************************************************

--
-- TITLE: AN ADA SOFTWARE METRIC
--
-- MODULE NAME: PACKAGE PARSER_2
--
-- DATE CREATED: 18 JUL 86
--
-- LAST MODIFIED: 04 OEC 86
--
-- AUTHORS: LCDR JEFFREY L. NIEDER
--
-- LT KARL S. FAIRBANKS, JR.
--
-- DESCRIPTION: This package contains thirty-three functions that are the middle
-- level productions for our top-down, recursive descent parser. Each function is preceded
-- by the grammar productions they are implementing.
--
******************************************************************************

with PARSER_3, PARSER_4, BYPASS_FUNCTION, BYPASS_SUPPORT_FUNCTIONS,
GLOBAL_PARSER, GLOBAL;
use PARSER_3, PARSER_4, BYPASS_FUNCTION, BYPASS_SUPPORT_FUNCTIONS,
GLOBAL_PARSER, GLOBAL;

package PARSER_2 is
    function GENERIC_ACTUAL_PART return boolean;
    function GENERIC_ASSOCIATION return boolean;
    function GENERIC_FORMAL_PARAMETER return boolean;
    function GENERIC_TYPE_DEFINITION return boolean;
    function PRIVATE_TYPE_DECLARATION return boolean;
    function TYPE_DECLARATION return boolean;
    function SUBTYPE_DECLARATION return boolean;
    function DISCRIMINANT_PART return boolean;
    function DISCRIMINANT_SPECIFICATION return boolean;
    function TYPE_DEFINITION return boolean;
    function RECORD_TYPE_DEFINITION return boolean;
    function COMPONENT_LIST return boolean;
    function COMPONENT_DECLARATION return boolean;
    function VARIANT_PART return boolean;
    function VARIANT return boolean;
    function WITH_OR_USE_CLAUSE return boolean;
    function FORMAL_PART return boolean;
    function IDENTIFIER_DECLARATION return boolean;
    function IDENTIFIER_DECLARATION_TAIL return boolean;
    function EXCEPTION_TAIL return boolean;
    function EXCEPTION_CHOICE return boolean;
    function CONSTANT_TERM return boolean;
    function LABEL return boolean;
    function ENTRY_DECLARATION return boolean;
    function PARAMETER_SPECIFICATION return boolean;
    function IDENTIFIER_LIST return boolean;
    function MODE return boolean;
    function DESIGNATOR return boolean;
    function SIMPLE_STATEMENT return boolean;
    function ASSIGNMENT_OR_PROCEDURE_CALL return boolean;
    function LABEL return boolean;
    function ENTRY_DECLARATION return boolean;
    function REPRESENTATION_CLAUSE return boolean;
    function RECORD_REPRESENTATION_CLAUSE return boolean;
end PARSER_2;

******************************************************************************
package body PARSER_2 is

-- GENERIC_ACTUAL_PART --> (GENERIC_ASSOCIATION [, GENERIC_ASSOCIATION])* 
function GENERIC_ACTUAL_PART return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (GENERIC_ASSOCIATION) then
      while (BYPASS(TOKEN_COMMA)) loop
        if not (GENERIC_ASSOCIATION) then
          SYNTAX_ERROR("Generic actual part");
        end if; -- if not generic_association
      end loop;
    end if; -- if bypass(token_left_paren)
  else
    SYNTAX_ERROR("Generic actual part");
  end if; -- if bypass(token_right_paren)
end if;
else
  if bypass(token_right_paren) then
    return (TRUE);
  else
    SYNTAX_ERROR("Generic actual part");
  end if;
  if bypass(token_left_paren) then
    SYNTAX_ERROR("Generic actual part");
  else
    return (FALSE);
  end if;
end GENERIC_ACTUAL_PART;

-- GENERIC_ASSOCIATION --> [GENERIC_FORMAL_PARAMETER ?] EXPRESSION
function GENERIC_ASSOCIATION return boolean is
begin
  if (GENERIC_FORMAL_PARAMETER) then
    null;
  end if;
  if (EXPRESSION) then
    -- if generic_formal_parameter statement
    return (TRUE);
  else
    return (FALSE);
  end if;
end GENERIC_ASSOCIATION;

-- GENERIC_FORMAL_PARAMETER --> identifier =>
-- --> string_literal =>
function GENERIC_FORMAL_PARAMETER return boolean is
begin
  LOOK_AHEAD_TOKEN := TOKEN_RECORD_BUFFER(TOKEN_ARRAY_INDEX + 1); 
  if (ADJUST_LEXEME(LOOK_AHEAD_TOKEN.LEXEME, 
    LOOK_AHEAD_TOKEN.LEXEME_SIZE - 1) = ">") then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      if (BYPASS(TOKEN_ARROW)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Generic formal parameter");
      end if; -- if bypass(token_arrow)
    elsif (BYPASS(TOKEN_STRING_LITERAL)) then
      if (BYPASS(TOKEN_ARROW)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Generic formal parameter");
      end if; -- if bypass(token_arrow)
    else
      SYNTAX_ERROR("Generic formal parameter");
    end if; -- if bypass(token_identifier)
  else
    SYNTAX_ERROR("Generic formal parameter");
  end if; -- if adjust_lexeme(lookahead_token)
-- GENERIC_TYPE_DEFINITION --> ( <> )
-- --> range <>
-- --> digits <>
-- --> delta <>
-- --> array ARRAY_TYPE_DEFINITION
-- --> access SUBTYPE_INDICATION

function GENERIC_TYPE_DEFINITION return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (BYPASS(TOKEN_BRACKETS)) then
      if (BYPASS(TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Generic type definition");
      end if;
    else
      SYNTAX_ERROR("Generic type definition");
    end if;
  elsif (BYPASS(TOKEN_RANGE)) or else (BYPASS(TOKEN_DIGITS))
or else (BYPASS(TOKEN_DELTA)) then
    if (BYPASS(TOKEN_BRACKETS)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Generic type definition");
    end if;
  elsif (BYPASS(TOKEN_ARRAY)) then
    if (ARRAY_TYPE_DEFINITION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Generic type definition");
    end if;
  elsif (BYPASS(TOKEN_ACCESS)) then
    if (SUBTYPE_INDICATION) then
      return (TRUE);
    else
      return (FALSE); -- if bypass(token_left_paren)
    end if;
  end if;
end GENERIC_TYPE_DEFINITION;

-- PRIVATE_TYPE_DECLARATION --> [limited ?] private
function PRIVATE_TYPE_DECLARATION return boolean is
begin
  if (BYPASS(TOKEN_LIMITED)) then
    null;
  end if;
  if (BYPASS(TOKEN_PRIVATE)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end PRIVATE_TYPE_DECLARATION;

-- SUBTYPE_DECLARATION --> identifier is SUBTYPE_INDICATION
function SUBTYPE_DECLARATION return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    if (BYPASS(TOKEN_IS)) then
      if (SUBTYPE_INDICATION) then
        if (BYPASS(TOKEN_SEMICOLON)) then
          null;
        end if;
      end if;
    end if;
  end if;
end SUBTYPE_DECLARATION;
return (TRUE));
else
  SYNTAX_ERROR("Subtype declaration");
end if;  -- if bypass(token_semicolon)
else
  SYNTAX_ERROR("Subtype declaration");
end if;  -- if subtype_indication statement
else
  SYNTAX_ERROR("Subtype declaration");
end if;  -- if bypass(token_is)
else
return (FALSE));
end if;  -- if bypass(token_identifier)
end SUBTYPE_DECLARATION;

-- TYPE_DECLARATION --> identifier [DISCRIMINANT_PART ?]
-- is SUBTYPE_INDICATION;
function TYPE_DECLARATION return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    if (DISCRIMINANT_PART) then
      null;
    end if;
    if (BYPASS(TOKEN_IS)) then
      null;
    end if;
    elseif (TYPE_DEFINITION) then
      null;
    else
      SYNTAX_ERROR("Type declaration");
    end if;
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Type declaration");
    end if;
  else
    return (FALSE);
  end if;
end TYPE_DECLARATION;

-- DISCRIMINANT_PART --> (DISCRIMINANTSPECIFICATION
-- is DISCRIMINANTSPECIFICATION)\n
function DISCRIMINANT_PART return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (DISCRIMINANTSPECIFICATION) then
      loop
        if (BYPASS(TOKEN_SEMICOLON)) then
          null;
        end if;
        if (BYPASS(TOKEN_RIGHT_PAREN)) then
          null;
        else
          SYNTAX_ERROR("Discriminant part");
        end if;
      end loop;
    else
      SYNTAX_ERROR("Discriminant part");
    end if;
  else
    return (FALSE);
  end if;
end DISCRIMINANT_PART;
-- DISCRIMINANT_SPECIFICATION --> IDENTIFIER_LIST : NAME [ := EXPRESSION ? ]

function DISCRIMINANT_SPECIFICATION return boolean is
begin
  if (IDENTIFIER_LIST) then
    if (BYPASS(TOKEN_COLON)) then
      if (NAME) then
        if (BYPASS(TOKEN_ASSIGNMENT)) then
          if (EXPRESSION) then
            null;
          else
            SYNTAX_ERROR("Discriminant specification");
        end if;
      else
        SYNTAX_ERROR("Discriminant specification");
      end if;
    end if;
  else
    SYNTAX_ERROR("Discriminant specification");
  end if;
end DISCRIMINANT_SPECIFICATION;

-- TYPE_DEFINITION --> ENUMERATION_TYPE_DEFINITION
-- --> INTEGER_TYPE_DEFINITION
-- --> digits / FLOATING_OR_FIXED_POINT_CONSTRAINT
-- --> delta FLOATING_OR_FIXED_POINT_CONSTRAINT
-- --> array ARRAY_TYPE_DEFINITION
-- --> record RECORD_TYPE_DEFINITION
-- --> access SUBTYPE_INDICATION
-- --> new SUBTYPE_INDICATION

function TYPE_DEFINITION return boolean is
begin
  if (ENUMERATION_TYPE_DEFINITION) then
    return (TRUE);
  elsif (INTEGER_TYPE_DEFINITION) then
    return (TRUE);
  elsif (BYPASS(TOKEN_DIGITS)) or else (BYPASS(TOKEN_DELTA)) then
    if (FLOATING_OR_FIXED_POINT_CONSTRAINT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Type definition");
    end if;
  -- floating_or_fixed_point_constraint
  elsif (BYPASS(TOKEN_ARRAY)) then
    if (ARRAY_TYPE_DEFINITION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Type definition");
    end if;
  -- array_type_definition
  elsif (BYPASS(TOKEN_RECORD_STRUCTURE)) then
    if (RECORD_TYPE_DEFINITION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Type definition");
    end if;
  -- record_type_definition
  elsif (BYPASS(TOKEN_ACCESS)) or else (BYPASS(TOKEN_NEW)) then
    if (SUBTYPE_INDICATION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Type definition");
    end if;
  -- subtype_indication
  else
    null;
  end if;
end TYPE_DEFINITION;

133
return (FALSE);
end if;
end TYPE_DEFINITION;

-- RECORD_TYPE_DEFINITION --> COMPONENT_LIST and record
function RECORD_TYPE_DEFINITION return boolean is
begin
if (COMPONENT_LIST) then
  if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_RECORD_STRUCTURE)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Record type definition");
    end if;
  else
    SYNTAX_ERROR("Record type definition");
  end if;
else
  return (FALSE);
end if;
end RECORD_TYPE_DEFINITION;

-- COMPONENT_LIST --> [COMPONENT_DECLARATION]* [VARIANT_PART ?]
-- --> null
function COMPONENT_LIST return boolean is
begin
  while COMPONENT_DECLARATION) loop
    null;
  end loop;
  if (VARIANT_PART) then
    null;
  elsif (BYPASS(TOKEN_NULL)) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      null;
    end if;
    return (TRUE);
else
  return (FALSE);
end if;
end COMPONENT_LIST;

-- COMPONENT_DECLARATION --> IDENTIFIER_LIST : SUBTYPE_INDICATION
-- [:= EXPRESSION ?]
function COMPONENT_DECLARATION return boolean is
begin
  if (IDENTIFIER_LIST) then
    if (BYPASS(TOKEN_COLON)) then
      if (SUBTYPE_INDICATION) then
        if (BYPASS(TOKEN_ASSIGNMENT)) then
          if (EXPRESSION) then
            if (BYPASS(TOKEN_SEMICOLON)) then
              return (TRUE);
            else
              SYNTAX_ERROR("Component declaration");
            end if;
          end if;
        else
          SYNTAX_ERROR("Component declaration");
        end if;
      end if;
    else
      SYNTAX_ERROR("Component declaration");
    end if;
  else
    return (TRUE);
  end if;
else
  return (FALSE);
end if;
end COMPONENT_DECLARATION;

134
case identifier is

function COMPONENT-DECLARATION return boolean is

begin

end COMPONENT-DECLARATION;

function VARIANT return boolean is

begin

end COMPONENT-DECLARATION;

function VARIANT return boolean is

begin

end COMPONENT-DECLARATION;

function VARIANT return boolean is

begin

end COMPONENT-DECLARATION;

function VARIANT return boolean is

begin

end COMPONENT-DECLARATION;

function VARIANT return boolean is

begin

end COMPONENT-DECLARATION;

function VARIANT return boolean is

begin

end COMPONENT-DECLARATION;
end if;                     -- if bypass(token_arrow)
else
  SYNTAX_ERROR("Variant");
end if;                     -- if choice statement
else
  return (FALSE);
end if;                     -- if bypass(token_when)
end VARIANT;

-- WITH_OR_USE_CLAUSE --> identifier [, identifier]*
function WITH_OR_USE_CLAUSE return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    while (BYPASS(TOKEN_COMMAS)) loop
      if (BYPASS(TOKEN_IDENTIFIER)) then
        SYNTAX_ERROR("With or use clause");
      end if;
    end loop;
  else
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("With or use clause");
    end if;
  end if;
  return (FALSE);
end if;                     -- if bypass(token_identifier)
end WITH_OR_USE_CLAUSE;

-- FORMAL_PART --> (PARAMETER_SPECIFICATION | PARAMETER_SPECIFICATION)*
function FORMAL_PART return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PARENT)) then
    if (PARAMETER_SPECIFICATION) then
      while (BYPASS(TOKEN_SEMICOLON)) loop
        if (PARAMETER_SPECIFICATION) then
          SYNTAX_ERROR("Formal part");
        end if;
      end loop;
    else
      if (BYPASS(TOKEN_RIGHT_PARENT)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Formal part");
      end if;
    end if;
  else
    if (BYPASS(TOKEN_RIGHT_PARENT)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Formal part");
    end if;
  end if;
  return (FALSE);
end if;                     -- if bypass(token_right paren) statement
end FORMAL_PART;

-- IDENTIFIER_DECLARATION --> IDENTIFIER_LIST : IDENTIFIER_DECLARATION_TAIL
function IDENTIFIER_DECLARATION return boolean is
begin
  if (IDENTIFIER_LIST) then
    if (BYPASS(TOKEN_COLON)) then
      if (IDENTIFIER_DECLARATION_TAIL) then
        return (TRUE);
      else
        SYNTAX_ERROR("Identifier declaration");
      end if;
    else
      SYNTAX_ERROR("Identifier declaration");
    end if;
  else
    SYNTAX_ERROR("Identifier declaration");
end if;

end if
else
    return(FALSE)
end if
and IDENTIFIER_DECLARATION;

-- IDENTIFIER_DECLARATION_TAIL --> exception EXCEPTION_TAIL
-- --> constant CONSTANT_TERM
-- --> array ARRAY_TYPE_DEFINITION
-- --> NAME IDENTIFIER_TAIL

function IDENTIFIER_DECLARATION_TAIL return boolean is
begin
    if (BYPASS(TOKEN_EXCEPTION)) then
        if EXCEPTION_TAIL then
            return(TRUE);
        else
            SYNTAX_ERROR("Identifier declaration tail");
        end if;
    elsif (BYPASS(TOKEN_CONSTANT)) then
        if CONSTANT_TERM then
            return(TRUE);
        else
            SYNTAX_ERROR("Identifier declaration tail");
        end if;
    elsif (BYPASS(TOKEN_ARRAY)) then
        if ARRAY_TYPE_DEFINITION then
            if BYPASS(TOKEN_ASSIGNMENT) then
                if EXPRESSION then
                    null;
                else
                    SYNTAX_ERROR("Identifier declaration tail");
                end if;
            else
                SYNTAX_ERROR("Identifier declaration tail");
            end if;
        else
            SYNTAX_ERROR("Identifier declaration tail");
        end if;
    elsif (NAME) then
        if IDENTIFIER_TAIL then
            return(TRUE);
        else
            SYNTAX_ERROR("Identifier declaration tail");
        end if;
    else
        return(FALSE);
    end if;
end IDENTIFIER_DECLARATION_TAIL;

-- EXCEPTION_TAIL -->
-- --> renames NAME
function EXCEPTION_TAIL return boolean is
begin
    if (BYPASS(TOKEN_SEMICOLON)) then
        return(TRUE);
    elsif (BYPASS(TOKEN_RENAMES)) then
        if (NAME) then
            if (BYPASS(TOKEN_SEMICOLON)) then
                return(TRUE);
            else
                SYNTAX_ERROR("Exception tail");
            end if;
        end if;
    else
        SYNTAX_ERROR("Exception tail");
    end if;
end EXCEPTION_TAIL;
end if;
else
    SYNTAX_ERROR("Exception tail");
end if;
else
    return (FALSE);
end if;
end EXCEPTION_TAIL;

-- EXCEPTION_CHOICE --> identifier
-- --> others
function EXCEPTION_CHOICE return boolean is
begin
    if (BYPASS(TOKEN_IDENTIFIER)) then
        return (TRUE);
    elsif (BYPASS(TOKEN_OTHERS)) then
        return (TRUE);
    else
        return (FALSE);
    end if;
end EXCEPTION_CHOICE;

-- CONSTANT_TERM --> array ARRAY_TYPE_DEFINITION [:= EXPRESSION ?] ;
-- --> ::= EXPRESSION ;
-- --> NAME IDENTIFIER_TAIL
function CONSTANT_TERM return boolean is
begin
    if (BYPASS(TOKEN_ARRAY)) then
        if (ARRAY_TYPEDEFINITION) then
            if (BYPASS(TOKEN_ASSIGNMENT)) then
                if (EXPRESSION) then
                    null;
                else
                    SYNTAX_ERROR("Constant term");
                end if;
            end if;
        else
            SYNTAX_ERROR("Constant term");
        end if;
    elseif (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
    else
        SYNTAX_ERROR("Constant term");
    end if;
    elsif (BYPASS(TOKEN_ASSIGNMENT)) then
        if (EXPRESSION) then
            if (BYPASS(TOKEN_SEMICOLON)) then
                return (TRUE);
            else
                SYNTAX_ERROR("Constant term");
            end if;
        elseif (NAME) then
            if (IDENTIFIER_TAIL) then
                return (TRUE);
            else
                SYNTAX_ERROR("Constant term");
            end if;
        else
            return (TRUE);
        end if;
    else
        SYNTAX_ERROR("Constant term");
    end if;
else
    if (NAME) then
        if (IDENTIFIER_TAIL) then
            return (TRUE);
        else
            SYNTAX_ERROR("Constant term");
        end if;
    else
        return (FALSE);
    end if;
end CONSTANT_TERM;

138
function IDENTIFIER_TAIL return boolean is
begin
  if (CONSTRAINT) then null; -- if constraint statement
  end if;
  if (BYPASS(TOKEN_RENAMES)) then
    if (NAME) then null;
    else
      SYNTAX_ERROR("Identifier tail"); -- if name statement
    end if;
    end if;
  if (BYPASS(TOKEN_ASSIGNMENT)) then
    if (EXPRESSION) then null;
    else
      SYNTAX_ERROR("Identifier tail");
    end if;
    end if;
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end IDENTIFIER_TAIL;

-- PARAMETER_SPECIFICATION --> IDENTIFIER_LIST : MODE NAME [::= EXPRESSION ?]
function PARAMETER_SPECIFICATION return boolean is
begin
  if (IDENTIFIER_LIST) then
    if (MODE) then
      if (NAME) then
        -- check for type_mark
        if (BYPASS(TOKEN_ASSIGNMENT)) then
          if (EXPRESSION) then null;
        else
          SYNTAX_ERROR("Parameter specification");
        end if;
        end if;
        return (TRUE);
      else
        SYNTAX_ERROR("Parameter specification");
      end if;
      end if;
    else
      SYNTAX_ERROR("Parameter specification");
    end if;
    end if;
  else
    SYNTAX_ERROR("Parameter specification");
  end if;
  end if;
else
  return (FALSE);
end IDENTIFIER_LIST statement
end PARAMETER_SPECIFICATION;

-- IDENTIFIER_LIST --> identifier [, identifier]*
function IDENTIFIER_LIST return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    while (BYPASS(TOKEN_COMMA)) loop
      if not (BYPASS(TOKEN_IDENTIFIER)) then
SYNTAX_ERROR("Identifier list");
end if;
-- if not bypass(token_identifier) statement
end loop;
return (TRUE);
else
return (FALSE);
end if;
-- if bypass(token_identifier) statement
end IDENTIFIER_LIST;

-- MODE --> [in ?]
-- --> in out
-- --> out
function MODE return boolean is
begin
if (BYPASS(TOKEN_IN)) then
if (BYPASS(TOKEN_OUT)) then
null;
end if;
elsif (BYPASS(TOKEN_OUT)) then
null;
end if;
return (TRUE);
end MODE;

-- DESIGNATOR --> identifier
-- --> string literal
function DESIGNATOR return boolean is
begin
if (BYPASS(TOKEN_IDENTIFIER)) then
return (TRUE);
elsif (BYPASS(TOKEN_STRING_LITERAL)) then
return (TRUE);
else
return (FALSE);
end if;
end DESIGNATOR;

-- SIMPLE_STATEMENT --> null ;
-- --> ASSIGNMENT_OR_PROCEDURE_CALL
-- --> exit EXIT_STATEMENT
-- --> return RETURN_STATEMENT
-- --> goto GOTO_STATEMENT
-- --> delay DELAY_STATEMENT
-- --> abort ABORT_STATEMENT
-- --> raise RAISE_STATEMENT
function SIMPLE_STATEMENT return boolean is
begin
if (BYPASS(TOKEN_NULL)) then
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Simple statement");
end if;
elif (ASSIGNMENT_OR_PROCEDURE_CALL) then
return (TRUE);
elsif (BYPASS(TOKEN_EXIT)) then
if (EXIT_STATEMENT) then
return (TRUE);
else
SYNTAX_ERROR("Simple statement");
end if;
elif (BYPASS(TOKEN_RETURN)) then
if (RETURN_STATEMENT) then
    return (TRUE);
else
    SYNTAX_ERROR("Simple statement");
    end if;
elsif (BYPASS_TOKEN_GOTO)) then
    if (GOTO_STATEMENT) then
        return (TRUE);
    else
        SYNTAX_ERROR("Simple statement");
        end if;
elsif (BYPASS_TOKEN_DELAY)) then
    if (DELAY_STATEMENT) then
        return (TRUE);
    else
        SYNTAX_ERROR("Simple statement");
        end if;
elsif (BYPASS_TOKEN_ABORT)) then
    if (ABORT_STATEMENT) then
        return (TRUE);
    else
        SYNTAX_ERROR("Simple statement");
    end if;
elsif (BYPASS_TOKEN_RAISE)) then
    if (RAISE_STATEMENT) then
        return (TRUE);
    else
        SYNTAX_ERROR("Simple statement");
        end if;
else
    return (FALSE);
    end if;
end SIMPLE_STATEMENT;

------------------------------------------------------------------------------------
-- ASSIGNMENT_OR_PROCEDURE_CALL --> NAME := EXPRESSION
-- --> NAME
function ASSIGNMENT_OR_PROCEDURE_CALL return boolean is
begin
    if (NAME) then
        if (BYPASS_TOKEN_ASSIGNMENT)) then
            if (EXPRESSION) then
                if (BYPASS_TOKEN_SEMICOLON)) then
                    return (TRUE);
                    -- parsed an assignment statement
                else
                    SYNTAX_ERROR("Assignment or procedure call");
                    end if;
                else
                    SYNTAX_ERROR("Assignment or procedure call");
                    end if;
                elsif (BYPASS_TOKEN_SEMICOLON)) then
                    return (TRUE);
                    -- parsed a procedure call statement
                else
                    SYNTAX_ERROR("Assignment or procedure call");
                    end if;
                else
                    return (FALSE);
                    end if;
    end ASSIGNMENT_OR_PROCEDURE_CALL;

------------------------------------------------------------------------------------
-- LABEL --> << identifier >>
function LABEL return boolean is
begin
    if (BYPASS_TOKEN_LEFT_BRACKET)) then
        if (BYPASS_TOKEN_IDENTIFIER)) then
            if (BYPASS_TOKEN_RIGHT_BRACKET)) then
                -- LABEL
return (TRUE);
else
  SYNTAX_ERROR("Label");
end if;
else
  SYNTAX_ERROR("Label");
end if;
else
  return (FALSE);
end ENTRY-DECLARATION;

function ENTRY-DECLARATION return boolean is
begin
  if (BYPASS_TOKEN_ENTRY) then
    if (BYPASS_TOKEN_IDENTIFIER) then
      if (DISCRETE_RANGE) then
        if (BYPASS_TOKEN_RIGHT_PAREN) then
          null;
        else
          SYNTAX_ERROR("Entry declaration");
        end if;
      end if;
    end if;
  end if;
else
  SYNTAX_ERROR("Entry declaration");
end if;

  if (BYPASS_TOKEN_SEMICOLON) then
    return (TRUE);
  else
    SYNTAX_ERROR("Entry declaration");
  end if;

end ENTRY-DECLARATION;

function REPRESENTATION_CLAUSE return boolean is
begin
  if (BYPASS_TOKEN_FOR) then
    if (NAME) then
      if (BYPASS_TOKEN_USE) then
        if (BYPASS_TOKEN_RECORD_STRUCTURE) then
          if (RECORD_REPRESENTATION_CLAUSE) then
            return (TRUE);
          else
            SYNTAX_ERROR("Representation clause");
          end if;
        end if;
      else
        SYNTAX_ERROR("Representation clause");
      end if;
    else
      if (BYPASS_TOKEN_AT) then
        if (SIMPLE_EXPRESSION) then
          if (BYPASS_TOKEN_SEMICOLON) then
            return (TRUE);
          else
            SYNTAX_ERROR("Representation clause");
          end if;
        else
          SYNTAX_ERROR("Representation clause");
        end if;
      else
        if (BYPASS_TOKEN_SEMICOLON) then
          return (TRUE);
        else
          SYNTAX_ERROR("Representation clause");
        end if;
      end if;
    else
      if (BYPASS_TOKEN_SEMICOLON) then
        return (TRUE);
      else
        SYNTAX_ERROR("Representation clause");
      end if;
    end if;
  end if;
end REPRESENTATION_CLAUSE;
else
  SYNTAX_ERROR("Representation clause");
end if;
-- if simple_expression statement
elsif (SIMPLE_EXPRESSION) then
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Representation clause");
  end if;
else
  SYNTAX_ERROR("Representation clause");
end if;
-- if bypass(token_record)
else
  SYNTAX_ERROR("Representation clause");
end if;
-- if bypass(token_use)
else
  SYNTAX_ERROR("Representation clause");
end if;
-- if name statement
else
  return (FALSE);
end if;
-- if bypass(token_for)
end REPRESENTATION_CLAUSE;

-- RECORD_REPRESENTATION_CLAUSE --> [at mod SIMPLE_EXPRESSION ?] [NAME at SIMPLE_EXPRESSION range RANGES]* end record

function RECORD_REPRESENTATION_CLAUSE return boolean is
begin
  if (BYPASS(TOKEN_AT)) then
    if (BYPASS(TOKEN_MOD)) then
      if (SIMPLE_EXPRESSION) then
        null;
      else
        SYNTAX_ERROR("Record representation clause");
      end if;
    else
      SYNTAX_ERROR("Record representation clause");
    end if;
  else
    SYNTAX_ERROR("Record representation clause");
  end if;
  -- if simple_expression
  while (NAME) loop
    if (BYPASS(TOKEN_AT)) then
      if (SIMPLE_EXPRESSION) then
        if (BYPASS(TOKEN_RANGE)) then
          if (RANGES) then
            null;
          else
            SYNTAX_ERROR("Record representation clause");
          end if;
        else
          SYNTAX_ERROR("Record representation clause");
        end if;
      else
        SYNTAX_ERROR("Record representation clause");
      end if;
    else
      SYNTAX_ERROR("Record representation clause");
    end if;
    -- if ranges statement
  end loop;
  if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_RECORD_STRUCTURE)) then
      if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Record representation clause");
      end if;
    else
      SYNTAX_ERROR("Record representation clause");
    end if;
  else
    SYNTAX_ERROR("Record representation clause");
  end if;
  -- if bypass(token_for)
end RECORD_REPRESENTATION_CLAUSE;
end if;
else
  return (FALSE);
end if;
end RECORD_REPRESENTATION_CLAUSE;
end PARSER_2;

-- if bypass(token_record_structure)
-- if bypass(token_end)
AN ADA SOFTWARE METRIC

MODULE NAME: PACKAGE PARSER_3

DATE CREATED: 22 JUL 86

LAST MODIFIED: 03 DEC 86

AUTHORS: LCDR JEFFREY L. NIEDER

LT KARL S. FAIRBANKS, JR.

DESCRIPTION: This package contains thirty-five functions that make up the baseline productions for our top-down, recursive descent parser. Each function is preceded by the grammar productions they are implementing.

with PARSER_4, BYPASS_FUNCTION, HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL;
use PARSER_4, BYPASS_FUNCTION, HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL;

package PARSER_3 is

  function SUBTYPE_INDICATION return boolean;
  function ARRAY_TYPE_DEFINITION return boolean;
  function CHOICE return boolean;
  function ITERATION_SCHEME return boolean;
  function LOOP_PARAMETER_SPECIFICATION return boolean;
  function EXPRESSION return boolean;
  function RELATION return boolean;
  function RELATION_TAIL return boolean;
  function SIMPLE_EXPRESSION return boolean;
  function SIMPLE_EXPRESSION_TAIL return boolean;
  function TERM return boolean;
  function FACTOR return boolean;
  function PRIMARY return boolean;
  function CONSTRAINT return boolean;
  function FLOATING_OR_FIXED_POINT_CONSTRAINT return boolean;
  function INDEX_CONSTRAINT return boolean;
  function RANGES return boolean;
  function AGGREGATE return boolean;
  function COMPONENT_ASSOCIATION return boolean;
  function ALLOCATOR return boolean;
  function NAME return boolean;
  function NAME_TAIL return boolean;
  function LEFT_PAREN_NAME_TAIL return boolean;
  function ATTRIBUTE_DESIGNATOR return boolean;
  function DISCRETE_RANGE return boolean;
  function EXIT_STATEMENT return boolean;
  function RETURN_STATEMENT return boolean;
  function GOTO_STATEMENT return boolean;
  function DELAY_STATEMENT return boolean;
  function ABORT_STATEMENT return boolean;
  function RAISE_STATEMENT return boolean;
end PARSER_3;

package body PARSER_3 is

  function SUBTYPE_INDICATION return boolean is
  begin
    if (NAME) then -- check for type_mark
      if (CONSTRAINT) then
        null;
      end if;
    end if;
    return (TRUE);
  end function;

end PARSER_3;
else
    return (FALSE);
end if;
and if;
end SUBTYPE_INDICATION;
---------------------------------------------------------------

-- ARRAY_TYPE_DEFINITION --> (INDEX_CONSTRAINT of SUBTYPE_INDICATION
-- this function parses both constrained and unconstrained arrays
function ARRAY_TYPE_DEFINITION return boolean is
begin
    if (BYPASS(TOKEN_LEFT_PAREN)) then
        if (INDEX_CONSTRAINT) then
            if (BYPASS(TOKEN_OF)) then
                if (SUBTYPE_INDICATION) then
                    return (TRUE);
                else
                    SYNTAX_ERROR("Array definition");
                end if;
            else
                SYNTAX_ERROR("Array definition");
            end if;
        else
            SYNTAX_ERROR("Array definition");
        end if;
    else
        SYNTAX_ERROR("Array definition");
    end if;
end ARRAY_TYPE_DEFINITION;
---------------------------------------------------------------

-- CHOICE --> EXPRESSION [..SIMPLE_EXPRESSION ?]
-- --> EXPRESSION [CONSTRAINT ?]
-- --> others
function CHOICE return boolean is
begin
    if (EXPRESSION) then
        if (BYPASS(TOKEN_RANGE_DOTS)) then
            -- check for discrete_range
            if (SIMPLE_EXPRESSION) then
                null;
            else
                SYNTAX_ERROR("Choice");
            end if;
        else
            SYNTAX_ERROR("Choice");
        end if;
        elsif (CONSTRAINT) then
            null;
        end if;
        return (TRUE);
    elsif (BYPASS(TOKEN_OTHERS)) then
        return (TRUE);
    else
        return (FALSE);
    end if;
end CHOICE;
---------------------------------------------------------------

-- ITERATION_SCHEME --> while EXPRESSION
-- --> for LOOP_PARAMETER_SPECIFICATION
function ITERATION_SCHEME return boolean is
begin
    if (BYPASS(TOKEN_WHILE)) then
        NESTING_METRIC(1);while_CONSTRUCT);
        if (EXPRESSION) then
            return (TRUE);
        else
            SYNTAX_ERROR("Iteration scheme");
        end if;
    elsif (BYPASS(TOKEN_FOR)) then
    end if;
end ITERATION_SCHEME;
NESTING_METRIC( FOR_CONSTRUCT );
if ( LOOP_PARAMETER_SPECIFICATION ) then
   return ( TRUE );
else
   SYNTAX_ERROR ( "Iteration scheme" );
   end if;
else
   return ( FALSE );
   end if;
end ITERATION_SCHEME;

-- LOOP_PARAMETER_SPECIFICATION --> identifier in [ reverse ? ] DISCRETE_RANGE

function LOOP_PARAMETER_SPECIFICATION return boolean is
begin
if ( BYPASS( TOKEN_IDENTIFIER ) ) then
   if ( BYPASS( TOKEN_IN ) ) then
      if ( BYPASS( TOKEN_REVERSE ) ) then
         null;
      end if;
   if ( DISCRETE_RANGE ) then
      -- if bypass( token_reverse )
      return ( TRUE );
   else
      SYNTAX_ERROR ( "Loop parameter specification" );
      end if;
   -- if discrete_range statement
else
   SYNTAX_ERROR ( "Loop parameter specification" );
   end if;
else
   null;
end if;
-- if bypass( token_in )
end LOOP_PARAMETER_SPECIFICATION;

-- EXPRESSION --> RELATION [ RELATION_TAIL ? ]

function EXPRESSION return boolean is
begin
if ( RELATION ) then
   if ( RELATION_TAIL ) then
      null;
   end if;
   -- if relation_tail statement
else
   return ( TRUE );
else
   return ( FALSE );
end if;
-- if relation statement
end EXPRESSION;

-- RELATION --> SIMPLE_EXPRESSION [ SIMPLE_EXPRESSION_TAIL ? ]

function RELATION return boolean is
begin
if ( SIMPLE_EXPRESSION ) then
   if ( SIMPLE_EXPRESSION_TAIL ) then
      null;
   end if;
   -- if simple_expression_tail statement
else
   return ( TRUE );
end if;
else
   return ( FALSE );
end if;
-- if simple_expression statement
end RELATION;

-- RELATION_TAIL --> [ and [ then ? ] RELATION]*
-- --> [ or [ else ? ] RELATION]*
-- --> [ xor RELATION]*
function RELATION_TAIL return boolean is
begin
  while (BYPASS(TOKEN_AND)) loop
    if (BYPASS(TOKEN_THEN)) then
      null;
    end if;
  end loop;
end RELATION_TAIL;

-- SIMPLE_EXPRESSION --> [+ ?] TERM [BINARY_ADDING_OPERATOR TERM]
-- --> [- ?] TERM [BINARY_ADDING_OPERATOR TERM]
function SIMPLE_EXPRESSION return boolean is
begin
  if (BYPASS(TOKEN_PLUS) or BYPASS(TOKEN_MINUS)) then
    if (TERM) then
      while (BINARY_ADDING_OPERATOR) loop
        if not (TERM) then
          SYNTAX_ERROR("Simple expression");
        end if;
      end loop;
      return (TRUE);
    else
      SYNTAX_ERROR("Simple expression");
    end if;
  else
    SYNTAX_ERROR("Simple expression");
  end if;
elsif (BYPASS(TOKEN_NOT)) then
  while (BINARY_ADDING_OPERATOR) loop
    if not (TERM) then
      SYNTAX_ERROR("Simple expression");
    end if;
  end loop;
  return (TRUE);
else
  return (FALSE);
end if;
end SIMPLE_EXPRESSION;

-- SIMPLE_EXPRESSION_TAIL --> RELATIONAL_OPERATOR SIMPLE_EXPRESSION
-- --> [not?] in RANGES
-- --> [not?] in NAME
function SIMPLE_EXPRESSION_TAIL return boolean is
begin
  if (RELATIONAL_OPERATOR) then
    if (SIMPLE_EXPRESSION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Simple expression tail");
    end if;
  elseif (BYPASS(TOKEN_NOT)) then
    null;
  end if;
end SIMPLE_EXPRESSION_TAIL;
if (BYPASS(TOKEN_IN)) then
  if (RANGES) then
    return (TRUE);
  elsif (NAME) then
    -- check for type_mark
    return (TRUE);
  else
    SYNTAX_ERROR("Simple expression tail");
    end if;
  else
    SYNTAX_ERROR("Simple expression tail");
    end if;
  elsif (BYPASS(TOKEN_IN)) then
    if (RANGES) then
      return (TRUE);
    elsif (NAME) then
      -- check for type_mark
      return (TRUE);
    else
      SYNTAX_ERROR("Simple expression tail");
      end if;
    else
      return (FALSE);
      end if;
    end if;
  end IF; -- if ranges statement
elsif (BYPASS(TOKEN_IN)) then
  if (RANGES) then
    return (TRUE);
  elsif (NAME) then
    -- check for type_mark
    return (TRUE);
  else
    SYNTAX_ERROR("Simple expression tail");
    end if;
  else
    return (FALSE);
    end if;
end TERMINATION

-- TERMINATION

-- TERM --> FACTOR [MULTIPLYING_OPERATOR FACTOR]
function TERMINATION return boolean is
begin
  if (FACTOR) then
    while (MULTIPLYING_OPERATOR) loop
      if not (FACTOR) then
        SYNTAX_ERROR("Term");
        end if;
      end loop;
      return (TRUE);
    else
      return (FALSE);
      end if;
    end if;
  end if;
end TERMINATION;

-- TERMINATION

-- FACTOR --> PRIMARY [## PRIMARY ?]
-- --> abs PRIMARY
-- --> not PRIMARY
function TERMINATION return boolean is
begin
  if (PRIMARY) then
    if (BYPASS(TOKEN_EXPONENT)) then
      if (PRIMARY) then
        null;
      else
        SYNTAX_ERROR("Factor");
        end if;
      end if;
    else
      SYNTAX_ERROR("Factor");
      end if;
    end if;
  else
    return (TRUE);
  end if;
else
  if (BYPASS(TOKEN_ABSOLUTE)) then
    if (PRIMARY) then
      return (TRUE);
    else
      SYNTAX_ERROR("Factor");
      end if;
    else
      SYNTAX_ERROR("Factor");
      end if;
    end if;
  else
    if (BYPASS(TOKEN_NOT)) then
      if (PRIMARY) then
        return (TRUE);
      else
        SYNTAX_ERROR("Factor");
        end if;
      else
        SYNTAX_ERROR("Factor");
        end if;
      end if;
    else
      return (FALSE);
      end if;
  end if;
end FACTORIZATION;

-- FACTORIZATION

-- PRIMARY --> abs PRIMARY
-- --> not PRIMARY
-- --> not PRIMARY
function TERMINATION return boolean is
begin
  if (PRIMARY) then
    if (BYPASS(TOKEN_EXPONENT)) then
      if (PRIMARY) then
        null;
      else
        SYNTAX_ERROR("Factor");
        end if;
      end if;
    else
      SYNTAX_ERROR("Factor");
      end if;
    end if;
  else
    return (TRUE);
  end if;
else
  if (BYPASS(TOKEN_ABSOLUTE)) then
    if (PRIMARY) then
      return (TRUE);
    else
      SYNTAX_ERROR("Factor");
      end if;
    else
      SYNTAX_ERROR("Factor");
      end if;
    end if;
  else
    if (BYPASS(TOKEN_NOT)) then
      if (PRIMARY) then
        return (TRUE);
      else
        SYNTAX_ERROR("Factor");
        end if;
      else
        SYNTAX_ERROR("Factor");
        end if;
      end if;
    else
      return (FALSE);
      end if;
  end if;
end PRIMARY;

-- PRIMARY

--abs PRIMARY --> abs PRIMARY
-- --> not PRIMARY
function TERMINATION return boolean is
begin
  if (abs PRIMARY) then
    if (BYPASS(TOKEN_EXPONENT)) then
      if (abs PRIMARY) then
        null;
      else
        SYNTAX_ERROR("Factor");
        end if;
      end if;
    else
      SYNTAX_ERROR("Factor");
      end if;
    end if;
  else
    return (TRUE);
  end if;
else
  if (BYPASS(TOKEN_ABSOLUTE)) then
    if (abs PRIMARY) then
      return (TRUE);
    else
      SYNTAX_ERROR("Factor");
      end if;
    else
      SYNTAX_ERROR("Factor");
      end if;
    end if;
  else
    if (BYPASS(TOKEN_NOT)) then
      if (primary) then
        return (TRUE);
      else
        SYNTAX_ERROR("Factor");
        end if;
      else
        SYNTAX_ERROR("Factor");
        end if;
      end if;
    else
      return (FALSE);
      end if;
  end if;
end abs PRIMARY;

--abs PRIMARY
else
    return (FALSE);
end if;
end FACTOR;

-- PRIMARY --> numeric_literal
-- --> null
-- --> string_literal
-- --> new ALLOCATOR
-- --> NAME
-- --> AGGREGATE

function PRIMARY return boolean is
begin
    if (BYPASS(TOKEN_NUMERIC_LITERAL)) then
        return (TRUE);
    elsif (BYPASS(TOKEN_NULL)) then
        return (TRUE);
    elsif (BYPASS(TOKEN_STRING_LITERAL)) then
        return (TRUE);
    elsif (BYPASS(TOKENALLOCATOR)) then
        return (TRUE);
    else
        SYNTAX_ERROR("Primary");
    end if;

elsif (NAME) then
    return (TRUE);
elsif (AGGREGATE) then
    return (TRUE);
else
    return (FALSE);
end if;
end PRIMARY;

-- CONSTRAINT --> range RANGES
-- --> range <=
-- --> digits FLOATING OR FIXED POINT CONSTRAINT
-- --> delta FLOATING OR FIXED POINT CONSTRAINT
-- --> INDEX CONSTRAINT

function CONSTRAINT return boolean is
begin
    if (BYPASS(TOKEN_RANGE)) then
        if (RANGES) then
            return (TRUE);
        elsif (BYPASS(TOKEN BRACKETS)) then
            -- check for <= when parsing
            return (TRUE); -- an unconstrained array
        else
            SYNTAX_ERROR("Constraint");
        end if;
    elsif (BYPASS(TOKEN_DIGITS)) or else (BYPASS(TOKEN_DELTA)) then
        if (FLOATING OR FIXED POINT CONSTRAINT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Constraint");
        end if;
    elsif (BYPASS(TOKEN_LEFT PAREN)) then
        if (INDEX CONSTRAINT) then
            return (TRUE);
        else
            SYNTAX_ERROR("Constraint");
        end if;
    else
        return (FALSE);
    end if;
end CONSTRAINT;
-- FLOATING_OR_FIXED_POINT_CONSTRAINT --> SIMPLE_EXPRESSION [range RANGES ?]
function FLOATING_OR_FIXED_POINT_CONSTRAINT return boolean is
begin
if SIMPLE_EXPRESSION then
  if (BYPASS(TOKEN_RANGE)) then
    if (RANGES) then
      null;
    else
      SYNTAX_ERROR("Floating or fixed point constraint");
      end if;
    -- if ranges statement
  end if;
  -- if bypass(token_range)
else
  return (TRUE);
end if;
-- if simple_expression statement
end FLOATING_OR_FIXED_POINT_CONSTRAINT;

-----------------------------------------------

-- INDEX_CONSTRAINT --> DISCRETE_RANGE [ , DISCRETE_RANGE]*
function INDEX_CONSTRAINT return boolean is
begin
if DISCRETE_RANGE then
  while (BYPASS(TOKEN_COMM)) loop
    if not (DISCRETE_RANGE) then
      SYNTAX_ERROR("Index constraint");
      end if;
    -- if not discrete_range
    end loop;
  if (BYPASS(TOKEN_RIGHT_PAREN)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Index constraint");
  end if;
else
  return (FALSE);
end if;
-- if discrete_range statement
end INDEX_CONSTRAINT;

-----------------------------------------------

-- RANGES --> SIMPLE_EXPRESSION [ ..SIMPLE_EXPRESSION ?]
function RANGES return boolean is
begin
if SIMPLE_EXPRESSION then
  if (BYPASS(TOKEN_RANGE_PROPS)) then
    if (SIMPLE_EXPRESSION) then
      null;
    else
      SYNTAX_ERROR("Ranges");
      end if;
    end if;
    -- if simple_expression statement
  else
    return (TRUE);
  end if;
else
  return (FALSE);
end if;
-- if simple_expression statement
end RANGES;

-----------------------------------------------

-- AGGREGATE --> (COMPONENT_ASSOCIATION [ , COMPONENT_ASSOCIATION]*)
function AGGREGATE return boolean is
begin
if (BYPASS(TOKEN_LEFT_PAREN)) then
  if (COMPONENT_ASSOCIATION) then
    while (BYPASS(TOKEN_COMM)) loop
      if not (COMPONENT_ASSOCIATION) then

SYNTAX_ERROR("Aggregate");
end if;
-- if not component association
end loop;
if (BYPASS(TOKEN_RIGHT_PAREN)) then
return (TRUE);
else
SYNTAX_ERROR("Aggregate");
end if;
else
SYNTAX_ERROR("Aggregate");
end if;
-- if bypass(token_right_paren)
else
return (FALSE);
end if;
-- if bypass(token_left_paren)
end AGGREGATE;
-- COMPONENT_ASSOCIATION --> [CHOICE [ CHOICE]# => ?] EXPRESSION
function COMPONENT_ASSOCIATION return boolean is
begin
if (CHOICE) then
while (BYPASS(TOKEN_BAR)) loop
if not (CHOICE) then
SYNTAX_ERROR("Component asociation");
end if;
end loop;
if (BYPASS(TOKEN_ARROW)) then
if (EXPRESSION) then
null;
else
SYNTAX_ERROR("Component asociation");
end if;
-- if expression statement
end if;
-- if bypass(token_arrow)
return (TRUE);
else
return (FALSE);
end if;
-- if choice statement
end COMPONENT_ASSOCIATION;
-- ALLOCATOR --> SUBTYPE_INDICATION ['AGGREGATE ?]
function ALLOCATOR return boolean is
begin
if (SUBTYPE_INDICATION) then
if (BYPASS(TOKEN_APOSTROPHE)) then
if (AGGREGATE) then
null;
else
SYNTAX_ERROR("Allocator");
end if;
-- if aggregate statement
end if;
-- if bypass(token_aPOSTROPHE)
return (TRUE);
else
return (FALSE);
end if;
-- if subtype_indication statement
end ALLOCATOR;
-- NAME --> identifier [NAME_TAIL ?]
-- --> character_literaL [NAME_TAIL ?]
-- --> string_literal [NAME_TAIL ?]
function NAME return boolean is
begin
if (BYPASS(TOKEN_IDENTIFIER)) then
if (NAME_TAIL) then
null;
end NAME;
end if;
return (TRUE);
else if (BYPASS(TOKEN_CHARACTER_LITERAL)) then
  if (NAME_TAIL) then
    null;
  end if;
  return (TRUE);
else if (BYPASS(TOKEN_STRING_LITERAL)) then
  if (NAME_TAIL) then
    null;
  end if;
  return (TRUE);
else
  return (FALSE);
end if;
end NAME;

-- NAME_TAIL --> (LEFT_PAREN_NAME_TAIL
  -- --> .SECTOR [NAME_TAIL]*
  -- --> `AGGREGATE [NAME_TAIL]*
  -- --> `ATTRIBUTE_DESIGNATOR [NAME_TAIL]*
function NAME_TAIL return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (LEFT_PAREN_NAME_TAIL) then
      return (TRUE);
    else
      return (FALSE);
    end if;
  elseif (BYPASS(TOKEN_APOSTROPHE)) then
    if (AGGREGATE) then
      while (NAME_TAIL) loop
        null;
      end loop;
      return (TRUE);
    elsif (ATTRIBUTE_DESIGNATOR) then
      while (NAME_TAIL) loop
        null;
      end loop;
      return (TRUE);
    else
      SYNTAX_ERROR("Name tail");
      end if;
  else
    return (FALSE);
  end if;
end NAME_TAIL;

-- LEFT_PAREN_NAME_TAIL --> [FORMAL_PARAMETER ?] EXPRESSION [...] EXPRESSION ?
  -- [, [FORMAL_PARAMETER ?] EXPRESSION [...] EXPRESSION ?]*
function LEFT_PAREN_NAME_TAIL return boolean is
begin
  if (FORMAL_PARAMETER) then
    -- check for optional formal parameter
    null;
  end if;
  -- before the actual parameter
  end if;
  -- if formal_parameter statement
if (EXPRESSION) then
  if (BYPASS(TOKEN_RANGE_DOTS)) then
    if not (EXPRESSION) then
      SYNTAX_ERROR("Left paren name tail");
    end if
    -- if not expression statement
  end if
  -- if bypass(token_range_dots)
while (BYPASS(TOKEN_COMMA)) loop
  if (FORMAL_PARAMETER) then
    null
  end if
  -- if formal_parameter statement
  if not (EXPRESSION) then
    SYNTAX_ERROR("Left paren name tail");
  end if
  -- if not expression statement
  if (BYPASS(TOKEN_RANGE_DOTS)) then
    if not (EXPRESSION) then
      SYNTAX_ERROR("Left paren name tail");
    end if
    -- if not expression statement
  end if
  -- if bypass(token_range_dots)
end loop;
if (BYPASS(TOKEN_RIGHT_PAREN)) then
  while (NAME_TAIL) loop
    null
  end loop;
  return (TRUE);
else
  return (FALSE);
end if;
elsif (DISCRETE_RANGE) then
  if (BYPASS(TOKEN_RIGHT_PAREN)) then
    while (NAME_TAIL) loop
      null
    end loop;
    return (TRUE);
  else
    SYNTAX_ERROR("Left paren name tail");
    end if
  -- if bypass(token_right_paren)
else
  return (FALSE);
end if;
end if
end LEFT_PAREN_NAME_TAIL

-- ATTRIBUTE DESIGNATOR --> identifier [[EXPRESSION] ?]
-- --> range [[EXPRESSION] ?]
-- --> digits [[EXPRESSION] ?]
-- --> delta [[EXPRESSION] ?]

function ATTRIBUTE DESIGNATOR return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) or else (BYPASS(TOKEN_RANGE)) then
    if (BYPASS(TOKEN_LEFT_PAREN)) then
      if (EXPRESSION) then
        if (BYPASS(TOKEN_RIGHT_PAREN)) then
          null
        else
          SYNTAX_ERROR("Attribute designator");
          end if
          -- if bypass(token_right_paren) statement
        else
          SYNTAX_ERROR("Attribute designator");
          end if
          -- if expression statement
        end if
        -- if bypass(token_left_paren) statement
        return (TRUE);
      else
        return (TRUE);
      end if
      else
        SYNTAX_ERROR("Attribute designator");
        end if
        -- if expression statement
      end if
      -- if bypass(token_right_paren) statement
    else
      return (TRUE);
    end if
    else
      (BYPASS(TOKEN_DIGITS)) or else (BYPASS(TOKEN_DELTA)) then
    if (BYPASS(TOKEN_LEFT_PAREN)) then
      if (EXPRESSION) then
        if (BYPASS(TOKEN_RIGHT_PAREN)) then
          null
        else
          SYNTAX_ERROR("Attribute designator");
          end if
          -- if bypass(token_right_paren) statement
        else
          return (TRUE);
        end if
      else
        return (TRUE);
      end if
    else
      SYNTAX_ERROR("Attribute designator");
      end if
      -- if expression statement
    end if
    -- if bypass(token_left_paren) statement
  end if
end function

154
end if; -- if bypass(token_right paren) statement
else
  SYNTAX_ERROR("Attribute designator");
end if;
end if; -- if expression statement
-- if bypass(token_left paren) statement
else
  return (FALSE);
end if; -- if bypass(token_identifier) statement

-- INTEGER_TYPE_DEFINITION --> range RANGES
function INTEGER_TYPE_DEFINITION return boolean is
begin
  if (BYPASS_TOKEN_RANGE) then
    if (RANGES) then
      return (TRUE);
    else
      SYNTAX_ERROR("Integer type definition");
    end if;
  else
    return (FALSE);
  end if;
end INTEGER_TYPE_DEFINITION;

-- DISCRETE_RANGE --> RANGES [CONSTRAINT ]
function DISCRETE_RANGE return boolean is
begin
  if (RANGES) then
    if (CONSTRAINT) then
      null;
    else
      return (TRUE);
    end if;
  else
    return (FALSE);
  end if;
end DISCRETE_RANGE;

-- EXIT_STATEMENT --> [NAME ?] [when EXPRESSION ?]
function EXIT_STATEMENT return boolean is
begin
  if (NAME) then
    null;
  end if;
  if (BYPASS_TOKEN_WHEN) then
    if (EXPRESSION) then null;
  else
    SYNTAX_ERROR("Exit statement");
  end if;
  if (BYPASS_TOKEN_SEMICOLON) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end EXIT_STATEMENT;

-- RETURN_STATEMENT --> [EXPRESSION ?]
function RETURN_STATEMENT return boolean is
begin
if (EXPRESSION) then
  null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
  return (TRUE);
else
  return (FALSE);
end if;
end RETURN_STATEMENT;

-- GOTO_STATEMENT --> NAME
function GOTO_STATEMENT return boolean is
begin
  if (NAME) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Goto statement");
    end if;
  else
    return (FALSE);
  end if;
end GOTO_STATEMENT;

-- DELAY_STATEMENT --> SIMPLE_EXPRESSION
function DELAY_STATEMENT return boolean is
begin
  if (SIMPLE_EXPRESSION) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Delay statement");
    end if;
  else
    return (FALSE);
  end if;
end DELAY_STATEMENT;

-- ABORT_STATEMENT --> NAME [, NAME]*/
function ABORT_STATEMENT return boolean is
begin
  if (NAME) then
    while (BYPASS(TOKEN_COMMA)) loop
      if not (NAME) then
        SYNTAX_ERROR("Abort statement");
      end if;
    end loop;
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Abort statement");
    end if;
  else
    return (FALSE);
  end if;
end ABORT_STATEMENT;

-- RAISE_STATEMENT --> [NAME ?]
function RAISE_STATEMENT return boolean is
begin
  if (NAME) then

null;
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
  return (TRUE);
else
  return (FALSE);
end if;
end RAISE_STATEMENT;
end PARSER_3;
package PARSER_4 is

  -- MULTIPLYING_OPERATOR --
  --  --> #
  --   --> /
  --   --> mod
  --   --> rem

  function MULTIPLYING_OPERATOR return boolean is
    begin
      if (BYPASS(TOKEN Asterisk)) then
        return (TRUE); 
      elsif (BYPASS(TOKEN Slash)) then
        return (TRUE); 
      elsif (BYPASS(TOKEN Hod)) then
        return (TRUE); 
      elsif (BYPASS(TOKEN REM)) then
        return (TRUE); 
      else
        return (FALSE); 
      end if; 
    end MULTIPLYING_OPERATOR;

  -- BINARY_ADDING_OPERATOR --
  --  --> +
  --   --> -
  --   --> &

  function BINARY_ADDING_OPERATOR return boolean is
    begin
      if (BYPASS(TOKEN PLUS)) then
        return (TRUE); 
      elsif (BYPASS(TOKEN MINUS)) then
        return (TRUE); 
      elsif (BYPASS(TOKEN AMPERSAND)) then
        return (TRUE); 
      else
        return (FALSE); 
      end if; 
    end BINARY_ADDING_OPERATOR;
return (FALSE);
end if;
end BINARY_ADDING_OPERATOR;

--- RELATIONAL_OPERATOR --> =
  -- --> /=
  -- --> <
  -- --> <=
  -- --> >
function RELATIONAL_OPERATOR return boolean is
begin
  if (BYPASS(TOKEN_EQUALS)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_NOT_EQUALS)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_LESS_THAN)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_LESS_THAN_EQUALS)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_GREATER_THAN)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_GREATER_THAN_EQUALS)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end RELATIONAL_OPERATOR;

--- ENUMERATION_TYPE_DEFINITION --> (ENUMERATION_LITERAL)
function ENUMERATION_TYPE_DEFINITION return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (ENUMERATION_LITERAL) then
      while (BYPASS(TOKEN_COMMA)) loop
        if not (ENUMERATION_LITERAL) then
          SYNTAX_ERROR("Enumeration type definition");
        end if;
      end loop;
    if (BYPASS(TOKEN_RIGHT_PAREN)) then
      return (TRUE);  -- if not enumeration_literal
    else
      SYNTAX_ERROR("Enumeration type definition");
    end if;
  else
    SYNTAX_ERROR("Enumeration type definition");
  end if;
else
  return (FALSE);
end if;
end ENUMERATION_TYPE_DEFINITION;

--- ENUMERATION_LITERAL --> identifier
  --> character_literal
function ENUMERATION_LITERAL return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_CHARACTER_LITERAL)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end ENUMERATION_LITERAL;
function FORMAL_PARAMETER return boolean is
begin
  LOOK_AHEAD_TOKEN := TOKEN_RECORD_BUFFER(TOKEN_ARRAY_INDEX + 1);
  if (ADJUST_LEXEME(LOOK_AHEAD_TOKEN.LEXEME, LOOK_AHEAD_TOKEN.LEXEME_SIZE - 1) = "=>") then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      if (BYPASS(TOKEN_ARROW)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Formal parameter");
      end if;
    else
      SYNTAX_ERROR("Formal parameter");
    end if;
  else
    return (FALSE);
  end if;
end FORMAL_PARAMETER;

function SELECTOR return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_CHARACTER_LITERAL)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_STRING_LITERAL)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_ALL)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end SELECTOR;

end PARSER_4;
-- TITLE: AN ADA SOFTWARE METRIC
--
-- MODULE NAME: PACKAGE SCANNER
--
-- DATE CREATED: 06 JUN 86
-- LAST MODIFIED: 04 NOV 86
--
-- AUTHORS: LCDR JEFFREY L. NIEDER
-- LT KARL S. FAIRBANKS, JR.
--
-- DESCRIPTION: This package reads each character from the
-- input buffer, determines its token class and calls
-- the appropriate procedure.
--
-- with LOW_LEVEL_SCANNER, NUMERIC, GET_NEXT_CHARACTER, GLOBAL;
use LOW_LEVEL_SCANNER, NUMERIC, GET_NEXT_CHARACTER, GLOBAL;

package SCANNER is
    procedure GET_NEXT_TOKEN(TOKEN_RECORD : in out TOKEN_RECORD_TYPE);
end SCANNER;

package body SCANNER is

    procedure GET_NEXT_TOKEN(TOKEN_RECORD : in out TOKEN_RECORD_TYPE) is
        begin
            LEXEME_LENGTH := 1;
            for I in 1..LINESIZE loop
                TOKEN_RECORD.LEXEME(I) := ' ';
            end loop;

            GETNEXTCHARACTER(NEXT_CHARACTER, LOOKAHEAD_ONE_CHARACTER);

            if ((NEXT_CHARACTER in UPPER_CASE_LETTER) or
                (NEXT_CHARACTER in LOWER_CASE_LETTER)) then
                TOKEN_RECORD.TOKEN_TYPE := IDENTIFIER;
                GETIDENTIFIER(TOKEN_RECORD);
            elsif ((NEXT_CHARACTER = '.') or
                (character'pos(NEXT_CHARACTER) in FORMATORS)) then
                TOKEN_RECORD.TOKEN_TYPE := SEPARATOR;
                FLUSH_SEPARATORS(TOKEN_RECORD);
            elsif (NEXT_CHARACTER in DIGITS_TYPE) then
                TOKEN_RECORD.TOKEN_TYPE := NUMERIC_LIT;
                GET_NUMERIC_LIT(TOKEN_RECORD);
            elsif ((NEXT_CHARACTER = '-') and (LOOKAHEAD_ONE_CHARACTER = '-')) then
                TOKEN_RECORD.TOKEN_TYPE := STRING_LIT;
                GET_STRING_LIT(TOKEN_RECORD);
            elsif (NEXT_CHARACTER = '"') then
                TOKEN_RECORD.TOKEN_TYPE := CHARACTER_LIT;
                GET_CHARACTER_LIT(TOKEN_RECORD);
            elsif (NEXT_CHARACTER = '-' or (NEXT_CHARACTER = ')') or
                (character'pos(NEXT_CHARACTER) in DELIMETERS1) or
                (character'pos(NEXT_CHARACTER) in DELIMETERS2)) then
                TOKEN_RECORD.TOKEN_TYPE := DELIMITER;
                GET_DELIMITER(TOKEN_RECORD);
            elsif (NEXT_CHARACTER = '"') then
                TOKEN_RECORD.TOKEN_TYPE := STRING_LIT;
                GET_STRING_LIT(TOKEN_RECORD);
            end if;
        end procedure;

end SCANNER;
elsif (NEXT_CHARACTER = '$') then
    TOKEN_RECORD.TOKEN_TYPE := SEPARATOR; -- input was a blank line
    TOKEN_RECORD.LEXEME(CURRENT_BUFFER_INDEX) := '$';
    NEXT_BUFFER_INDEX := REFILL_BUFFER_INDEX;
elsif (character'pos(NEXT_CHARACTER) = 0) then -- first character is null
    TOKEN_RECORD.TOKEN_TYPE := SEPARATOR;
    NEXT_BUFFER_INDEX := REFILL_BUFFER_INDEX; -- force buffer to refill
else
    -- first character read is not one of the legal characters
    TOKEN_RECORD.TOKEN_TYPE := ILLEGAL;
    ERROR_MESSAGE(TOKEN_RECORD.TOKEN_TYPE);
end if;

-- token value is an integer which corresponds to the token type's
-- position in the token list
TOKEN_RECORD.TOKEN_VALUE := 'TOKEN'pos(TOKEN_RECORD.TOKEN_TYPE);
TOKEN_RECORD.LEXEME_SIZE := LEXEME_LENGTH;
end GET_NEXT_TOKEN;
end SCANNER;}
LIST OF REFERENCES


INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Cameron Station
   Alexandria, Virginia 22304-6145

   National Technical Information Service
   Springfield, Virginia 22161

3. Department of Commerce, Office of
   Technology, Washington, D.C.

4. Post, Daniel Davis Center
   Department of Defense
   Washington, D.C.

5. DoD/OSDP, WACOM

6. NAIC

7. Dr. Robert Wilson
   Office of Secretary of Defense
   Washington, D.C.

8. Mr. Robert Wood
   Office of Naval Research
   Washington, D.C.

9. Mr. C. H.
   Naval Research Lab
   Washington, D.C.

10. LCDR Lewis
    NAIC
    3501 Wilson Blvd.
    Arlington, Virginia

11. LCDR Lewis
    NAIC
    3501 Wilson Blvd.
    Arlington, Virginia

12. Mr. J.
    STAP
    2800 Wilson Blvd.
    Arlington, Virginia

13. Mr. J.
    NAIC
    3501 Wilson Blvd.
    Arlington, Virginia
END

4 - 1 = 3