AN IMPLEMENTATION OF THE HALSTEAD AND HENRY "METRICS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P M HERZIG JUN 87
ADAMEASURE:
AN IMPLEMENTATION
OF THE HALSTEAD AND HENRY METRICS

by

Paul M. Herzig

June 1987

Thesis Advisor: Daniel L. Davis

Approved for public release; distribution is unlimited
**ADAMEASURE: AN IMPLEMENTATION OF THE HALSTEAD AND HENRY METRICS**

**PERSONAL AUTHOR(S):**
Herzig, Paul M.

**TYPE OF REPORT**
Master's Thesis

**DATE OF REPORT (Year Month Day)**
1987 June

**PAGE COUNT**
184

**ABSTRACT**
A software metric is a tool that should be used in the development of quality software. The properties that define good software vary but encompass reliability, complexity, efficiency, testability, understandability, and modifiability. The Henry metric measures the complexity of data flow within a module and the complexity of inter-module communication. This metric is an extension of a previous thesis titled 'Adameasure' that evaluated the Halstead metric. The present design and implementation is a tool that evaluates the Halstead and Henry metrics for Ada programs.

**SUBJECT TERMS**
Ada Metric; Data Flow Complexity; Ada Measure; Halstead; Henry

**REFERENCE Citations**

**ABSTRACT SECURITY CLASSIFICATION**
Unclassified

**UNCLASSIFIED SECURITY CLASSIFICATION OF THIS PAGE**
AdaMeasure
An Ada Software Metric
Implementation of the Henry Metric

by

Paul M. Herzig Jr.
Lieutenant Commander. United States Navy
B.S.E.E., University of New Mexico, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
from the

NAVAL POSTGRADUATE SCHOOL
June 1987

Author:

P.M. Herzig

Approved by:

Daniel L. Davis, Thesis Advisor

Bruce J. MacLennan. Second Reader

Vincent Y. Lum. Chairman. Department of
Department of Computer Science

Kneale T. Marshall
Dean of Information and Policy Sciences
ABSTRACT

A software metric is a tool that should be used in the development of quality software. The properties that define good software vary but encompass reliability, complexity, efficiency, testability, understandability, and modifiability. The Henry metric measures the complexity of data flow within a module and the complexity of inter-module communication. This thesis is an extension of a previous thesis titled 'AdaMeasure' that calculated the Halstead metric. The present design and implementation is a tool that computes the Halstead and Henry metrics for Ada programs.
# TABLE OF CONTENTS

I. INTRODUCTION AND BACKGROUND .............................................. 6
   A. DEFINITIONS ........................................................................ 6
   B. SALLIE HENRY’S METRIC ............................................... 6
   C. INFORMATION FLOW .......................................................... 7
   D. RELATIONS .......................................................................... 9
   E. INFORMATION FLOW STRUCTURE ........................................ 19
   F. INDICES OF MERIT ............................................................ 19
   G. MODULE ANALYSIS ............................................................ 21
   H. INTERFACE MEASUREMENTS ............................................ 22
   I. THEORY SUMMARY ............................................................ 24

II. DESIGN CRITERIA .................................................................... 25
   A. INTRODUCTION ................................................................. 25
   B. SPECIFIC ISSUES .............................................................. 28
   C. DESIGN ISSUES ............................................................... 28
   D. CONCLUSION ...................................................................... 30

III. DESIGN AND TESTING ........................................................... 31
   A. THE EMBEDDED CODE ..................................................... 31
   B. THE HENRY PACKAGE ..................................................... 31
I. INTRODUCTION AND BACKGROUND

A. DEFINITIONS

A metric is an assignment of indices of merit to programs in order to evaluate and predict software quality [Ref.1: p.6-2]. The qualities to measure are, at present, subjectively chosen but in general encompass reliability, complexity, efficiency, testability, understandability and modifiability [Ref.2: p.1-3]. The predictive nature of a metric allows it to be used to say "when" to proceed to the next phase in the software life cycle model. Another aspect of the predictive nature of a metric would be for it to provide management with a rough guess of the outcome of a particular path of development, provide an acceptance index, or provide an immediate feedback loop to the implementors while in the unit test phase [Ref.2: p.5]. How the metric is implemented will dictate its primary use from the above selections.

B. SALLIE HENRY'S METRIC

Sallie Henry's metric attempts to measure data flow complexity. It is intended to be used as a tool to establish a module's quality or to enforce particular modularization standards [Ref.2: p.6]. She argues that quality control of software is the result of software reliability and that reliability comes about through well designed modules that do not have complex data flow.
The hierarchical structure of a program should be layered modules. Each layer should function as a virtual machine and be composed of modules. This approach to modularization gives each module characteristics that can be exploited so that each module can be independently developed, more easily comprehended, assembled so that the system is more stable and designed so that the system is a great deal more flexible. This schema of development extols two primary tenets that are stated by D. Parnas in [Ref.3: p.339] and quoted here:

... provide the user of a module with all the information to use the module correctly, and nothing more. Provide the implementor of the module with all the information to implement the module correctly, and nothing more.

All this implies that a good design will have high module cohesion, good module strength and low module coupling [Ref.4: p.330].

C. INFORMATION FLOW

Information flow complexity is a twofold process the flow of data within a module and the flow of data external to the module. The measurement of these criteria is dependent on two premises: (1) that there is a capability to measure this data and (2) the data obtained can be used to evaluate software design. The seemingly obvious nature of the first premise runs into problems in implementation and applicability, but if it is accepted that the first deficiency can be surmounted, then the second part remains to be shown as reasonable. Applicability is a debated concept that is still not resolved. It revolves around whether the data gathered is related to the property under consideration. It is
further exacerbated by the human element that defines an environmental bubble and then programs within this bubble. How to measure this bubble without destroying its foundations is the problem of measuring human performance. The problem of what to measure is the problem of applicability.

The more specific the metric's application the less the applicability property is questioned, but, the problem of "what" to measure is still not clearly defined. This thesis will not argue the applicability question because the approach of Sallie Henry is reasonable and the results obtained from the metric appear to adequately encompass the area of data flow complexity. If the reader will accept that the properties measured are related to data flow complexity then the results obtained are also related to complexity.

The second premise is even more thorny. If the data is obtained and it seems reasonable can it be shown to be truly the result of the property under measurement? Any human endeavor will never be clearly and objectively quantified. Thus, the answer to the efficacy of the second premise is, proceed and maybe the amassing of results will eventually show the correlation.

The above analysis is far from a convincing argument to utilize metrics to measure programs however as this thesis was developed the applicability of measuring data flow complexity in order to determine code quality became more apparent although not proven. Nothing will be learned if no attempt is made to measure data flow complexity. This thesis attempts to measure data flow
complexity in the light of learning and the hope that the data gathered will prove the applicability of the process.

Consider first a simple module: a procedure in a structured language. Each procedure defines certain relations between itself and other procedures. These include:

- formal input/output parameters
- function call input and return data
- local data structures
- global data structures

These relations will generate a particular information flow structure similar to a hierarchical tree structure. This tree structure is peculiar to the procedure and will reflect its complexity of structure. It is reasonable to analyze this tree to determine derived calls, local data flow and global data flow.

D. RELATIONS

Some definitions are now in order. Global data flow exists from procedure 1 to procedure 2 if procedure 1 deposits data in the global data structure and then procedure 2 reads that data. Local data flow comprises direct and indirect species. A local direct flow, from procedure 1 to procedure 2, results when procedure 1 calls 2 passing parameters. An indirect data exchange from procedure 1 to procedure 2 exists if procedure 2 calls 1, which returns a value used by 2, or procedure 3 calls both 1 and 2, and passes an output value from 1 to 2.
Figure 1 represents data flow from procedure to procedure or from a procedure into a data structure. Parameter passing within this scheme is represented by the arrows. A hidden data exchange through modification of a variable is represented by the dashed flow arrow. Module A retrieves data from the data structure then calls B passing a parameter; module B updates the data structure. C calls D passing a parameter. D calls E with a parameter and E returns a value to D which is used by D and passed to F. The function of F updates the data structure.

Figure 1. Data Flow Structure
The direct data flows represented are:
A -> B, C -> D, D -> E, D -> F.

These are simply the calls.

The indirect local flows are:
E -> D, F -> A.

The global flows are:
B -> A, F -> A.

Both B and F update the data structure while A retrieves data from the structure.

The implications of data flow for procedure and function calls will be discussed later with derived calls.

The calling notation A(x) -> B() or A() -> B(x) is used to connote a data flow transmission from A to B either by direct parameter passing or side-effect. In the first condition the variable x is returned to procedure A and in the second example the variable x is sent to B. A condition that leads the Henry metric to not detect a procedure or function call's data flow (labeled a missed call) is for the condition where A(x) -> B() and variable x is a returned value from B not modified within procedure A's code. An example of this would be a conditional statement within A that depends on the returned value from function B. The data flow detection problem leads to two key ideas, effective parameters and data utilization.

Calls that are detected by information flow analysis are dependent upon how the information is passed. If the conditions A() -> B(x) exists where parameter x
is passed to B or condition A() -> B() where no parameters are exchanged then the calls will not be missed if B receives information in one of the following formats:

- a formal parameter
- a data structure
- a constant
- an actual parameter from a third procedure whose value is modified within A prior to the call to B

An effective parameter will define the call structure in such a way that the data flow will not be missed. It is a parameter that receives information from one of the calling procedure's parameters, a data structure, a constant, or a third procedure's returned actual parameter that is modified within the calling modules structure. What the effective parameter implies is that side-effect data flow is difficult to effectively analyze. Another construct that will cause a missed call is the condition A(z) -> B(x) where B is a function. This condition means A uses data from B. A uses data from B if (1) B updates a data structure used by A; (2) A receives a constant from B; (3) A receives an output parameter from B; or (4) B updates a return value to A. Thus information flow will be detected if A passes B an effective parameter or if A uses data from B.

Appendix A gives all the rules that are applicable to the data flow relationships. Some notation is now needed to simplify the descriptions that follow.
The form of a relation is \( L \leftarrow R_1, R_2, R_3, \ldots, R_n \); where \( L \) is the resultant from the application of the relationships \( R_1, R_2, \ldots, R_n \). An example would be:

\[ A.D3 \leftarrow A.D1, A.D2, A.constant. \]

This series notation represents the code line that begins with \( D3 \) below.

\[
A()
\begin{align*}
&\text{begin} \\
&D3 := D1 + D2 + 1; \\
&\text{end procedure } A;
\end{align*}
\]

In words, the \( A.D3 \) means procedure \( A \) updates data structure \( D3 \) by first applying relationship \( A.D1 \) then \( A.D2 \) and finally \( A.constant \). This format shows that data flows into procedure \( A \)'s data structure \( D3 \) from the noted relationships.

A thorough discussion of the notation for the relations is given in Appendix A but a short discussion follows to aide in the immediate understanding of Figure 2.

The notation \( B.I.I \) defines the first input parameter in the actual parameter list of procedure \( B \) and an \( O \) would refer to an output parameter. All possible data flow paths are considered even if a \( B.I.I \) parameter is not an input parameter. Thus, if procedure \( B \) has an output actual parameter in position \( B.1 \) and the Henry metric attempts to analyze this parameter as an input flow an error condition would result from the attempted evaluation (depicted as \( B.ERROR \)). \( B.NULL \) means that no relationship exits for this parameter or that there is no data flow into or out of the parameter being considered.

13
Figure 2. Data Flow With Call Structure

Code

A()
begin
X := D1 + 1;
Y := D2;
B(X,Y);
end;

B(P,Q)
begin
D3 := P + Q;
end;
Relations Set

A1 B.1.I <- A.D1, A.CONSTANT
A2 B.2.I <- A.D2
B1 B.1.O <- B.NULL
B2 B.2.0 <- B.NULL
B3 B.D3 <- B.1.I, B.2.I

The relation sets were derived by looking at the data flow into and out of procedure A. That is, since procedure A has no parameters there can only be local data flows into or out of the procedure. These flows are described in terms of the procedure call to B. B.1.I stands for procedure B's first input parameter. This parameter is fed from procedure A's data structure D1 and a constant. Analyzing procedure B's second input parameter yields the A2 relationship. Relationship B1 describes the first parameter in procedure B as an output parameter to procedure A that receives no data for transfer. Relationship B3 describes how the two input parameters to procedure B constitute the data flow to this data structure.

The data flow analysis deals primarily with the analysis of parameters which are direct data flow and indirect data flow as defined above. Modifying Figure 2 and incorporating some local variables will illustrate some more data flow analysis techniques as seen in Figure 3.
Figure 3. Data Flow And Inter-dependent Procedures

Code

A()
begin
  X := D1 + 1;
  Y := D2;
  B(X,Y)
end;

B(P,Q)
begin
  R := Q;
  C(P,R,S);
  D3 := S;
end;

C(I,J,K)
begin
  K := I + J;
  J := J + 1;
end;
Relation Set

A1, A2 same.

B1 B.1.O <-> C.1.O
B2 B.2.O <-> B.NULL
B3 C.1.I <-> B.1.I
B4 C.2.I <-> B.2.I
B5 C.3.I <-> B.ERROR
B6 B.D3 <-> C.3.O

C1 C.1.I <-> C.NULL
C2 C.2.O <-> C.2.I, C.CONSTANT
C3 C.3.O <-> C.1.I, C.2.I

In the relation set B1 receives data from procedure C's output parameter. B2 is the same. B3 through B5 describe the parameter list of procedure C. However B5 denotes an error or a condition that is not allowed. That is, the data direction was in error as variable S is an output from procedure C as indicated by relation B6. It should be noted that this relation set building considers all possible data flow paths without regard to the possibility that the parameters could be assigned only particular directions as Ada formal parameters are. Figure 4 shows the effects of a function call.

Code

A()
begin
X := D1 + 1;
Y := F(X);
B(X,Y)
end:
F(M) return integer;
begin
N := D2 * M;
return N;
end;

Relation Sets are changed as follows:

A1 F.1.I <- A.D1, A.CONSTANT
A2 B.1.I <- A.D1, A.CONSTANT, F.1.O
A3 B.2.I <- F.O

F1 F.1.O <- F.NULL
F2 F.O <- F.D2, F.1.I

Relation A1 has changed to reflect the analysis of the function call to F. The input to the function call is analyzed as well as its output and any possible modification of its input parameter. This analysis can be seen to cover all possibilities of hidden data transfers except the missed calls described earlier.
E. INFORMATION FLOW STRUCTURE

Once the relation set has been built the relations are sorted alphabetically and stored for future use in the Information Flow Structure (IFS). A recursive algorithm is employed to build the information tree structure for the flow analysis. The IFS is then analyzed to find the derived calls, the local flows and, finally, the global flows.

The IFS will have leaves that are data structures; the root is the initial call from the highest level procedure. Each node of the tree will have the relational form of X.DS, X.O, X.k.l, or X.k.O. See Appendix A for all the possibilities of derived calls. The local flows are described in Appendix A as derived calls. The global flows for a particular data structure are all the possible paths from leaf elements of the form A.DS to the root.

F. INDICES OF MERIT

The calculations of the indices of merit use the idea that the complexity of a module comprises the complexity of the code plus the complexity of the connections of the code to other modules. The formula describing the complexity of a module is

\[
\text{Complexity} = \text{length} \times (\text{fan-in} \times \text{fan-out})^{\text{code index}}.
\]

Length is defined as the number of executable statements. The expression fan-in \times fan-out represents all the combinatorial possibilities for each input to produce an output. The code index is an exponent that represents the code
difficulty. Nominal code difficulty for operating systems is 2. This index needs more data for other types of programming.

The purpose of this computation is to produce comparative numbers of merit that point out and isolate specific areas within the code that have the potential for problems. A high fan-in/fan-out implies a large interconnection to outside modules. This leads to the assessment that the code in question is most likely not properly modularized or, more succinctly, that the code has more than one function. The other form of data flow is global data flow to data structures. It is calculated as follows:

\[
\text{Global flow} = \text{write} \times (\text{read} + \text{read}_\text{write}) + \\
\text{read}_\text{write} \times (\text{read} + \text{read}_\text{write} - 1)
\]

The term write refers to a change to the data within the structure through an assignment statement and a read is an access to the data structure that does not change the data. The identifier read-write is the sum of reads and writes. A high global flow implies overworked data structures and represents a stress point in the program. A stress point is the weak link in the chain. The presence of high flow is not automatically an indicator of poor programming but it is a juncture in the program that is highly susceptible to problems. Once the metric has assembled all the different components, such as fan-in or global reads and calculated the above equations it performs module analysis.
G. MODULE ANALYSIS

Module analysis revolves on the outputs of each of the above equations and their respective components. The numbers generated are symptomatic of certain problems. The analysis is first conducted with the equations output defining the particular categories of problems then the components refine the analysis. Examples of the first level of analysis follow:

A high global flow calculation implies an overworked data structure. These structures are overworked because of the need for continuous accessing. This implies a better decentralized design is in order, that is, distribute the information to the procedures that it serves. A high module complexity index indicates not enough modularization. This number is to be treated with respect but should be analyzed in context with global data flow. Together these indices represent the in’s and out’s of the modules data. A corrective action based solely on complexity should be avoided. A procedure should be analyzed for singularity of purpose and non-duplication within a module. Simply put, a procedure should be in one place, have one purpose and have minimal external references. These properties are quantified by the Complexity and Global flow metric numbers.

Next the interim cases where one aspect is high and the other component is low. A module with high global flow and low complexity shows poor internal structure. This structure will most likely have excessive numbers of procedures with extensive use of data structures outside the module. Low global flow with
high module complexity implies either poor decomposition into procedures or extremely complicated interface.

H. INTERFACE MEASUREMENTS

Interface between procedures comprise protocol interface, coupling and binding of procedures. Protocol interface from module A to B is defined as those procedures that are not in any other module and which receive information from A for passing to B. Binding is the sensitivity measure between modules, that is, tightly bound modules have a high sensitivity. A tightly bound module is difficult to change without adversely affecting the other module. Coupling is the strength of binding. Figure 5 depicts the interface structure.

---

Figure 5. Interface Structure
Protocol interface, since it is not symmetrical between procedures, requires the construction of a tabular cross reference table with all possible procedures on both the X and Y axes. The internals of the table are the data flow complexity indexes for one way transmission from each procedure to the other.

Figure 5 shows that binding is sectioned into five components; the number of procedures sending information from A (NSP), the number of procedures receiving information from B (NRP), the number of procedures in the protocol interface (NPI), the number of paths to the interface from A (SPI), and the number of paths from the interface to B (PIR). The Direct Flow paths represented as the outer loops are data transmissions without the interim procedures. [Ref.2: p.85] lists the binding calculations as follows:

\[
\text{Binding} = (\text{NSP} + \text{NPI}) \times \text{SPI} + (\text{NPI} + \text{NRP}) \times \text{PIR}
\]

The term \((\text{NSP} + \text{NPI}) \times \text{SPI}\) is the coupling strength.

All the direct path binding is calculated by

\[
\text{DF Binding} = (\text{NSP} + \text{NRP}) \times \text{DF}
\]

Modules that are tightly bound are extremely difficult to maintain and modify. This difficulty stems from their lack of independence and the "ripple effect" of changes to one module flowing into the other.
I. THEORY SUMMARY

The purpose of the Henry metric is to provide designers and implementors with a method to quantify the quality of the code that they are developing. The goal is to produce reliable code that is interconnected in as logical a fashion as possible. Information flow complexity produces reliability through enforcement of design rules that lead to well connected code. The measurements will point out lack of functionality, improper modularization, poorly designed modules, poorly designed data structures, system stress points, inadequate refinement, strength of binding, modifiability, missing levels of abstraction and, will produce comparative indices to assess changes.
II. DESIGN CRITERIA

A. INTRODUCTION

The design of the implementation of the Henry metric was a two step process. First, a thorough understanding of the previous work by Neider and Fairbanks [Ref.5: pp.1-164] was undertaken to determine what data were available for importation into the Henry metric. The intention of the study and the basic design issues are (1) modify the output of the parser portion of their thesis to include the necessary data passes to the Henry metric (2) to initially analyze only the Ada package as a unit (3) encapsulate as much of the Henry metric into one Ada package as possible and (4) calculate the necessary Henry metric numbers transparently to the user but present the user with an output that is easily understood.

The underlying premise, of this program, is that the code presented for analysis has been successfully compiled. If the code does not compile and is presented to the parsers it will most likely fail to parse but in the event it does escape detection it will be erroneously analyzed.

The design criteria of encapsulation of the Henry metric was modified during the implementation due to the unwieldy length of the code. The division yielded three packages: one that holds all the global data, another for the analysis portion and a third for the interface to the user. Although this division violated one of
the basic design issues it was necessary in order to achieve solid data transfers between the Naval Post Graduate School computer and the Naval Weapons Station computers.

While the level of what was available from the parsers was being determined the data structures of the Henry metric were being layed out. The data structures and data gathering procedures for the Henry metric were designed to be as simple, yet as flexible as possible. Once the layout of the linked list data structure to hold the raw information for the Henry metric was decided upon, the tedious procedure of inserting the appropriate calls to the Henry package was undertaken. Basically, this reduced to exercising all the possible data flow characteristics from function or procedure calls that the Henry metric could expect to encounter and then ensuring that an appropriate call to the Henry data collection procedure was placed in the Neider/Fairbanks parser.

Next, the analysis procedure was designed. The analysis was separated from the display because computers have very different capabilities in their output devices. The first package analyzes the collected raw data and the second displays the finished, smooth data.

The program is menu driven. It initially gives the user a choice to parse a new program or view old data. If the parse choice is selected the parsers feed both the Henry and the Halstead packages with data. After a successful parse the user is presented with a choice of either viewing the Halstead or Henry metric data. This is when the analysis portion of the Henry metric is called. It is not
until the user decides which data to view is there a distinction made as to how to process the parsed data. This feature was designed for several reasons:

- Use the Halstead metric to refine the code
- then analyze the code via the Henry metric for data flow
- but the user should still have the option to select either metric

The data for both metrics are essentially different as are the purposes for gathering the data but the final goal of this dual metric system is to produce good code.

Finally, the presentation data module was designed. The Halstead and the Henry metric both produce numbers which are essentially meaningless unless a thorough understanding of the particular metric implementation is undertaken. Thus, both metrics, in differing fashions present "help" data to aid understanding of the metric output. These files are both verbally and graphically presented to the user.

The overriding design issue was to modularize the Henry metric as much as possible. The most significant exception to the Parnas' ideal [Ref.4: p.330] are the numerous calls to the data gathering procedure from the parser modules. These calls depend on details of how the data will be analyzed in their sequencing and data passing scheme. A conscientious effort was made to minimize global data and isolate procedures into nearly stand alone modules.
B. SPECIFIC ISSUES

The housekeeping routines of the global package are used to adjust the parsed data into a more palatable form for the analysis and presentation procedures. The output from the parser is stored as a linked list of raw data produced by procedures WRITE_HENRY_DATA and CREATE_NODE. This data is then analyzed by the ANALYSIS PACKAGE for the particular data constructs that represent a data flow. The output is an array of tabulated data that is a set of all the relations necessary for the Henry metric to detect local and global data flow.

The display module presents data in a tabular format or as a graphical representation for relative merit analysis. The intent was for the user to see the effect of changes, or to select a more verbose description of the meaning of the results. The modules that accomplish tabular and graphic displays are separated again because of the varying capabilities of machines. The purpose of these procedures is to provide some form of relative measure to the user so that the improvement or results of a change could be more objectively weighed. The overall purpose of the display modules is to show the data in such a fashion that an intelligent assessment is possible.

C. DESIGN ISSUES

Design issues encountered in the implementation of Henry's metric involve the efficient use of the Ada language's structures and data analysis techniques. Sallie Henry developed a metric process in which the constructs of a particular
language are ignored or not put to specific use. That is, some languages have extremely thorough type and range checking facilities. This is not considered in the basic design of her metric. These powerful features are incorporated in Ada and provide the application programmer more analysis capability.

This design issue concerns the collection of 'all possible paths' data for analysis of actual parameters. The approach taken by Henry is biased to a language where input, output, and combination input output parameters are treated as if they could be modified by the particular procedure regardless of their type. Ada is very picky about the manipulation of formal and actual parameters and goes to great lengths to ensure that parameter consistency is maintained by means of strong type/range checking. The explicit declaration of a parameter type was used to select which component of the complexity equation should be updated. The appropriate fan-in or fan-out number was also correctly updated from default declarations such as the undeclared default formal parameter.

The data analysis technique issue encountered was the need to analyze the data via the IFS. Henry’s IFS was designed so that a traversal of its nodes analyzing parent-child pairs will capture all the transitive relational data flows. The transitive flow analysis designed into the present parser will account for the first two layers. The reasoning behind this approach stems from a program review. This review, albeit not extensive, was conducted looking for the predominant use of transitive relations. The review revealed that transitivity is not often used and if used is at most two layers deep. There was little use of deep
transitive constructs. Thus, the design approach selected will detect the majority of the transitive data flow paths without the need for an extensive tree structure. The "normal" program has few transitive relations but the capability to analyze this style of program would add more accuracy to the metric.

Another design anomaly of the metric is the problem of detecting the difference between a function call outside the package declaration and a global data structure manipulation. Ada libraries or packages inhibits the proper analysis of a function call as opposed to a data structure read unless a full compiler's output is available. The present metric was designed so that local function calls (within the package being analyzed) are properly valued but the function calls outside the package are treated as data structure manipulations or more specifically as global data flows.

D. CONCLUSION

The design and implementation phase was driven by the analysis of the Neider/Fairbanks parser portion of their thesis followed by the modularization of the Henry metric. The tradeoffs considered were: the strong typing and range checking of the Ada language, the need for an information flow tree, the need for relative output for the user and, and most importantly the desire to incorporate all of the Henry metric into one Ada package.
III. DESIGN AND TESTING

A. THE EMBEDDED CODE

The previous work done by Neider/Fairbanks had to be modified to output the necessary data for the Henry metric. This was accomplished through embedding calls to the Write Henry Data procedure in Parser0, Parser1, Parser2, Parser3 and Bypass Function (See Appendix C). The writing of the Lexeme, or identifier's name, was controlled by a Boolean that was turned on or off according to the position of the parse of a particular package. The design criteria was to keep the data gathering as simple as possible. If time permitted, a more thorough and sophisticated scheme could be developed. The embedded code was thoroughly tested by two test harnesses that simulated a series of Current Token Records in the form of an input Ada package.

B. THE HENRY PACKAGE

The first package to be implemented was Henry.pkg. It was conceived to be a stand-alone construction that would initialize the data collection process, receive data from the other parsing packages and store the raw incoming data in a linked list. (See Appendix B). Minimal variables and foreign procedures from other packages are used. The Henry package's only "withed" packages are TEXT IO, HENRY GLOBAL, HENRY ANALYSIS and HENRY DISPLAY. This
approach was considered necessary so that the subsequent changes or upgrades would not affect other modules (ripple effect). The design was to implement a basic Henry metric first for Ada packages then to improve and more fully develop the Henry analysis techniques if time allowed. The Main Menu module sequences the user into the analysis and display support packages. The modularization was considered necessary because the analysis and display packages are separate entities and the separation will ensure maintainability.

The initialization is conducted by procedure Initialize_Henry and the declaration statements that assign initial values to various Boolean variables. Initialize_Henry creates two head nodes, one for the raw data linked list, the other for the procedure or function length records. The raw data linked list storage is a straight line of Henry records. These records have five fields that identify whether this is (1) local or global declaration, (2) the variable/procedure's name, (3) an action class, (4) a parameter class and (5) a pointer to the next record. The action class is comprised of various identifiers that range from procedure type to end parameters declare. Their purpose is to delineate the actions within the parsed program so that the Henry analysis package can look for the data flow. The parameter type field is used to define input, output or combination input output formal parameters. The variable Henry_Line_count is purposely initialized within this procedure to draw attention to it's initial value. The array of length records is initially a parallel construct not directly tied to the procedure or function it holds the data for. In the analysis package a sequential process is
produced where the records are linked to the data manipulation array. The purpose of the length record is to hold the begin and end line counts of each procedure or function. These line counts are used later to compute the specific modules length for inclusion in the complexity equation.

The receipt of incoming data is accomplished primarily by Write_Henry_Data. This procedure is supported by a boolean Write_Henry_Enable. This boolean turns on or off the recording of the incoming records from the Get_Current_Token_Record procedure. Specifically, the boolean will allow recording only selected data from the incoming record stream selected by the place within the recursive descent parser that the boolean is activated. This control is necessary to pick and choose the data that is critical and to ignore the remainder.

The procedures Create_Node and Clear_Henry_Lexeme support the data gathering scheme. The "in out" pointers within Create_Node serve the purpose of allowing a view of the last record in the incoming stream or to work on the current record. It is arranged so that New_Node points to the newly created blank record and Last_Record points to the just filled in trailing record. Procedure Clear_Henry_Lexeme is necessary because of the way Ada handles strings. Create_Line_Node procedure functions identically to Create_Node.

The incoming data is chosen from within the Bypass_Function and from Parser0 to Parser4 (Ref.5: pp.102-160) by where the calls to the Write_Henry_Data procedure is positioned. The purpose of this approach is to assure the Henry
metric receives sufficient information but more importantly that the records
written into the linked list are delimited in a particular fashion for ease of analysis.
There is still considerable data that can be collected for analysis from the parsers
but the Henry metric is not to the stage of development where it would be useful.
The added depth of information could be used in two areas: analysis and a more
informative output from both metrics.

The Write Henry Data procedure selectively enters the field data into the
raw data linked list records as dictated by the incoming actual parameters. That
is, the incoming data has default settings but if the data is to be ignored then the
"null setting" is passed as an actual parameter. This assists in the gathering
process. The design of the data gathering modules is such that modifications
could be easily implemented. This was purposely designed into them so that
upgrades would be fairly painless.

The Henry.pkg was constructed with modularization and maintainability in
mind. It was meant to be a stand alone entity that receives data from the
Neider/Fairbanks Bypass Function and Parser packages. It performs the
functions of initialization, data receipt and data storage besides defining the data
structures used throughout the Henry metric packages. There are a number of
improvements that could be added to the actual parameter analysis. These
improvements all concern the wealth of options Ada provides in parameter
passing schemes, such as, aggregates, dot notation to access hidden variables and
allocators. Further, the present Henry metric does not analyze the incoming
actual parameters for expressions but the variables are all considered for inclusion in the complexity calculation by the transitivity analysis.

C. HENRY ANALYSIS PACKAGE

The Henry Analysis package comprises three procedures to set up the raw linked list data and a fourth procedure to actually analyze the code for metric calculations. The Analysis procedures are called sequentially from the Henry package and function as support for the Henry package. They operate on the data in sequential discrete steps. They first determine the formal parameters, then search and identify procedures and variables and then determine the metric numbers. The approach used was to nibble each piece of the tremendously complex data flow calculation down into minute sub-steps until all that is left is to simply count the marks on each record for determination of the complexity or global flow metric numbers. This approach removed the necessity for an arduous single pass calculation.

The set up procedures are CLEAN UP HENRY DATA, SET UP HENRY ARRAY, and SPRUCE UP HENRY DATA. A support function, LOCAL NAME, assists in the setting-up process. These procedures' end product are two metrics, the complexity metric and the global flow metric.

The Clean Up procedure ensures that all parameter type records have all their fields properly filled. It scans for their parameter lists all the procedures and functions that are declared in the analyzed package. The field of most
The importances is the classification of either "in, out or in-out" type parameters. These fields are checked up to the colon delimiter within the formal parameter list and then entered into all parameter_type records correctly.

The Set_Up procedure scans through the entire linked list setting up another array of pointers to facilitate the analysis process. The Henry_Array records have identifier, beginning pointer and line_length record pointer entries. This procedure's purpose is to break up the long linked list into another array. It actually does not sub-divide the list it merely arranges an array of pointers into the linked list that delineate each function or procedure. The resulting array is called the Henry_Array. The line length record pointers are records that hold the stop and stop line numbers. These records are eventually used to compute procedure/function lengths.

The Spruce_Up procedure goes through the Henry array data and sorts out the local and global data flow paths. It does this through the use of the LOCAL_NAME function. This function searches either the Henry array for a particular procedure name or the package and appropriate procedure's declaration sections for the variable name in question. Its purpose is to sort out the local procedure or function calls from the global data structure manipulations. It cannot completely solve this problem but defers final resolution to the Calculation procedure.

The Calculate Metric procedure will again process the Henry array data looking for the final resolution to local procedure or function calls as opposed to a
global data structure manipulation. It proceeds in small increments to finally arrive at the complexity metric calculation and a global data flow calculation. The complexity metric number is arrived at by first considering all the in, out, in-out formal parameters to calculate the fan-in and fan-out numbers. After the initial cut the fan-in, fan-out numbers are incremented upward by the numbers of identified procedure actual parameters that feed these formal parameters and then by the Transitivity_In and Transitivity_Out functions.

An example of this process would be for procedure A with formal parameters X, Y. First process parameters X and Y for their explicit type adding 1 to fan-in if its an input parameter or 1 to fan-out if its an output parameter. Next process all the assignment expressions looking for a modification of the formal parameter. If procedure A modifies parameter X prior to a call to another function increment the fan-out count by the number of statements after the assignment delimiter. Then go through an analysis of transitivity incrementing fan-in or fan-out accordingly. Finally, call up the appropriate record of Henry_Line_count and calculate the length of the procedure or function in question.

The equation that the process is working toward solving is:

\[ \text{Complexity} = \text{length} \times (\text{fan-in} \times \text{fan-out})^2 \]

This equation represents the local data flows within the analyzed procedure. Sallie Henry set the exponent of the bracketed expression to 2 because of her experience with operating system code analysis. This program will continue with
this number until enough data can be compiled to support a change. Once this calculation is done then the global data flows are analyzed.

The global flows are arrived at by first eliminating all other possibilities. Then the remaining choices have to be foreign data flows. This process is started in the Spruce_Up procedure and completed within the Metric_Calculation procedure. The process is used to find whether the data structure is being read from or written to or both.

The equation that the analysis is striving to solve is:

\[ \text{Global flow} = \text{write} \times (\text{read} + \text{read-write}) + \text{read-write} \times (\text{read} + \text{read-write} - 1) \]

This equation represents how and by what means the global data structures are manipulated. The global data analysis procedure goes across procedure or function boundaries whereas the previous complexity metric calculations remain within the particular procedure or function under scrutiny. This across-the-border calculation is accomplished through the text file that is discussed next.

Within the calculation procedure the initial entries for the display package are started. This amounts to constructing a text file of descriptive terms and indices of merit for output in the Display Package. It also provide a temporary storage bank for the global data information. This across-boundary analysis of
global flows was necessary because of the implications of not being able to detect the difference within the Ada code of an access to a data structure or a function call to a "withed" package.

In summary, the Analysis package is a series of analytical steps. The purpose of these steps is to arrive at the complexity and global flow metrics. These indices and additional data are stored in a text file for output to the user within the Display package.

D. HENRY DISPLAY PACKAGE

The Henry Display package is the user interface portion of the metric program. It provides the user with four different aspects of viewing the analyzed data. The purpose of this package is to show the user the data flow characteristics of the particular parsed input program. The output data will be the fan-in, fan-out, length, complexity, and four global flow numbers. These numbers can be presented in a listing format, viewed with a help file of informative paragraphs or compared by means of the other portions of the analyzed package to gain a relative sense of merit.

The procedures that comprise the Display package are LIST METRIC DATA and WRITE RELATIVE DATA and GRAPH RELATIVE. The LIST METRIC DATA procedure will output the data file compiled while in the calculation portion of the previously discussed package. It will be a straight listing of information that will be grouped by each
element in the calculation of the complexity or global flow numbers, such as all
procedures are grouped under the head of FAN-IN. The purpose of this listing is
to show each procedure or functions component figure in the calculation of the
final complexity and global figures. If the programmer is in a compile, test, edit,
recompile mode of operation this will provide a spotlight on where to improve the
data flow "choke-points". These data flow critical points will be seen as either
high global flow or high complexity numbers. In short, the
LIST_METRIC_DATA is designed for a more sophisticated programmer wishing
to edit-and-run and see the results of particular programming style changes.

The WRITE_RELATIVE_DATA display will provide the same format of
data but the numbers will have been normalized. Accompanying each number set
will be a short narrative keyed to the relationships of the particular numbers.
That is, if the user sees a complexity number of 125 beside the procedure X he
will be provided with an explanation that that number is not too far out of line in
comparison to the other procedures or functions analyzed within this package.
The purpose of this approach is to normalize the output numbers to provide a
relative comparison for a more user friendly approach to the mysterious metric
number generation.

There is an additional procedure within the Display package that provides a
complete listing of the raw input data. This procedure will most likely be of no
use to anyone except those programmers who are extremely interested in the
factors that lead to the particular numbers presented.
The final package for viewing the data is the graphical presentation module. It takes the relative data and manipulates the floating point numbers to achieve a bar chart display.

In summary, the Display package will provide the user with information in a variety of formats so that he can reach a conclusion from relative merit or absolute input numbers. The data flow numbers will point out the critical data flow points within a procedure so that the programmer can better see where to improve or expend the most effort. The purpose of the output data is to show the user where to improve, not how to improve.

E. TESTING

The testing of the design was conducted as the modules were being built and at the integration step prior to the final product. This was accomplished through the use of test harnesses that simulated the particular module's inputs and through test input programs that were hand analyzed to verify the metric's outputs.

The testing of the Henry package was accomplished by gradually building a more thorough test harness as each previous test was successful. The final test harness encompassed over 200 input records that simulated a myriad of token record inputs. The testing of the Henry module presented some difficulty because it is so intimately tied to the parsers. This was overcome by simulating the Bypass Support package as a partial input and the test harness as the balance of
the test vehicle. The package performed well within the test harness and functions adequately within the context of the entire program.

The testing of the analysis package of the Henry metric again was an iterative build of the harness. The testing accomplished after all the Henry metric packages were integrated was accomplished on the same group of programs provided by the NWC programmers to test the Halstead metric integration. The harness testing was comprised of a 50 step program that simulated a package with three independent procedures/functions utilized within its scope. There were intentional references outside the scope of the test harness package to determine if the global call detection scheme functioned properly. In all, the test harness exercised every possible data flow scheme analyzable by the Henry metric including one that would be a missed call. The analysis package performs adequately within the scope of the harness. Testing revealed that the code within the analysis phase was non-reentrant which required the use of a boolean to define the status of the call to the package. This boolean will protect the data structure and effectively make the code reentrant.

The display module was tested with the same driver harness as the analysis package. The results were used to fine tune the package and to debug the problems. The process used was to call the analysis package from the display package and drive the display package with the test harness. This is also how the integrated program performs. The results were adequate from the stand point of the test harness but need some refinement when using the whole program.
The summary of testing would be extensive use of complicated test harnesses. Since a test harness has to simulate all the inputs to a module that the tested code could possibly see during integration, they are difficult to build much less debug. The debugging problem comes from the question 'is it the code or is it the harness?'. The test harness approach is quite fruitful from two orientations: (1) it forces the programmer into a thoroughly understanding his code and (2) the harness construction will lead the programmer into optimizing his code. Why doesn’t the programmer already understand his code? He does but the ramifications of a certain approach does not come surface until the design of a test harness is considered. The optimization is driven by the need to get accurate, fast results so that the troubleshoot-repair-compile-troubleshoot regimen can proceed fairly rapidly. This is a real concern with the tremendous
IV. CONCLUSIONS

A. IMPLEMENTATION

The Henry metric was implemented in as modularized a fashion as possible. The intent was to allow for improvements through a more thorough use of the parser's information in AdaMeasure's first revision. Also, certain aspects of the Henry metric were not implemented, but it is now felt that they would add depth to the analysis process. In particular, the first change should be that the Information Flow Structure be added. This tree-like structure will allow the analysis of hidden calls but will still not detect the missed call problem discussed earlier. The missed call problem will most likely only be solved through the use of the Program Counter Register, but this approach defeats the idea of a high level language. The final improvement would be to add analysis of the "withed" packages so that an interface table could be constructed.

The program was incorporated into the previous work by Neider and Fairbanks. Their work was extensive and deserves favorable mention because it made the implementation of the Henry metric considerably easier. The output of the program is still in need of sophistication and improvement. In particular, two improvements are needed: (1) explaining the theory behind the metric and (2) conveying the ideas to the user. For example, a high global data flow indicates an overworked global data structure. What should the metric present to the user?
The average programmer might not see the relevance of this and would miss the indication that a critical point in the data flow should probably be revised.

B. THE FUTURE OF METRICS

Metrics are tools. They point out areas of weakness. The metric will show a direction to proceed even in the absence of an absolute answer as to the correctness of the analyzed code.

The importance of metrics will grow as the size of programs grow. We do not know how important metrics will become but it does seem clear that there is a need for something that helps improve code quality and is fairly painless to use. The emphasis on "good" code will continue to be in the forefront of the Armed Service's concerns because of their intense involvement in real-time embedded programs. These programs present a real challenge for incorporation of changes, improvements or any other form of maintenance programming. The purpose of metrics in this environment would be to point the way to good modularized design.

The metric should be part of the test scenario besides being an integral member of the life cycle of the program. The metric will force quality control without the painful process of formal inspections. The formal process has its place but the metric tool could perform more than the inspection. The metric tool should be incorporated into the test cycle as a meter of improvement. This immediate feedback to the programmer will be beneficial. The manager could

45
also use the absolute number as a goal for acceptance. This approach will provide the manager with the data needed at decision points in the life cycle of a program. The absolute number could also be tied to the program throughout its life as a measure of improvement or degradation over time. The uses are many, as the reader can see. The importance of the metric cannot be overstated when the future holds programs that will span millions of lines of code.

Metrics are important. They hold out the hope of an automated tool that will guide, interpret, and assess progress for programmers and management alike. I hope that the work of this metric will assist in advancing metrics and the use of the Ada language.
APPENDIX A: INFORMATION FLOW MECHANISMS

MECHANISMS FOR INFORMATION FLOW ANALYSIS

As Sallie Henry so succinctly states:

The information flow analysis takes place in three phases. The first phase involves generating a set of relations indicating the flow of information through input parameters, output parameters, returned values from functions and data structures.

General Format of a Relation

The generation of relations is first prefaced with a quick review of relational format.

\[ L - R_1, R_2, ..., R_{\text{count}} \]

Where \( L \) may be in any one of the following forms:

1. \( P.DS \)  \( P \) is the name of a procedure and \( DS \) the data structure.
2. \( P.O \)  \( P \) is the procedure name and \( O \) is the return value.
3. \( P.j.O \)  \( P \) is the procedure \( j \) is an integer representing the formal parameter position, and \( O \) is the \( j \)th Output parameter.
4. \( P.j.I \)  \( P \) is the procedure, \( j \) is an integer representing the \( j \)th parameter, and \( I \) is the \( j \)th Input parameter.

\( R_i \) may be in one of the following forms:

1. \( S.DS \)  \( S \) is a procedure name and \( DS \) is the name of a data structure.
2. \( S.O \)  \( S \) is a procedure name and \( O \) is the returned value.
3. \( S.j.I \)  \( S \) is the procedure name, \( j \) is the \( j \)th parameter and \( I \) is the \( j \)th input parameter.
4. \( S.j.O \)  \( S \) is the procedure name, \( j \) is an integer representing the \( j \)th parameter in the list and \( O \) is the output parameter.
5. \( S.null \)  \( S \) is the procedure name, \( null \) represents no data.
6. \texttt{S.cons} - \(S\) is the procedure name and constant a value used within \(S\).

7. \texttt{S.error} - \(S\) is the procedure name and error represents an invalid flow of information through procedure \(S\).

\textbf{RULES}

1. \(L\) is of the form \(P \cdot DS\) then
   
   - This form is used only to generate the relations from procedure \(P\) that updates \(DS\) with \(R_i\).

2. \(L\) is of the form \(P \cdot O\) then
   
   - This is used only in generating the relations from procedure \(P\) that produce an output.

3. \(L\) is of the form \(P_j \cdot O\) then
   
   - This is used when generating the relations that produce an input of the \(j\)th parameter in the procedure's formal parameter list. There must be a unique relation for each of \(P\)'s parameters.

4. \(L\) is of the form \(P_j \cdot I\) then
   
   - This is used when generating the relations for procedure \(P\) that produce an input for the \(j\)th parameter. Another procedure \(T\) calls \(P\) to indicate that the \(j\)th parameter of \(P\) receives the input update.

5. \(R_i\) is of the form \(S \cdot DS\) then
   
   - Procedure \(S\) reads information from \(DS\); this format is used to indicate a read only.

6. \(R_i\) is of the form \(S \cdot O\) then
   
   - Relations are generated that come from procedure \(T\) that are return values to \(T\) from \(S\).

7. \(R_i\) is of the form \(S_j \cdot I\) then
   
   - For generating relations for procedure \(S\) that indicates \(S\)'s \(j\)th input parameter passes information to \(L\).

8. \(R_i\) is of the form \(S \cdot \text{cons}\).
   
   - Then \(S\) causes a constant number or string to flow to \(L\).

9. \(R_i\) is of the form \(S \cdot \text{Null}\) then
   
   - This is used to indicate when \(S\) does not update a parameter. That is, the parameter was strictly input only.

10. \(R_i\) is of the form \(S \cdot \text{error}\) then
    
    - \(S\) calls \(T\) and one of the parameters to \(T\) is an output only thus if \(S\) attempts to input a value this would be an error.
ANALYSIS OF CALLS

The following two procedures X and Y, exhibit all possible calling structures in the light of information flow analysis. NP stands for not possible, NC for no calls and the numbers beneath are used for later reference.

The pairs 1, 3, 5, 7, 13, and 15 cannot appear in a flow of data path because for D3's the only assignment and reads allowed are from procedures or functions. The other not possibles stem from input parameters not flowing into D3's and not flowing into output parameters. Entries 2 and 4 indicate X calling Y, receiving information from Y and using this information to update a D3. The rest of the possibilities can be reasoned in like manner except entries 10 and 12, which represent calls via a third procedure. Here procedure Z calls Y and passes the returned value from Y to X. This represents a no call between X and Y but there is a data flow.

TABLE 1.

<table>
<thead>
<tr>
<th>LOCAL CALL TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.DS</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Y.DS</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Y.O</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Y.k.I</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Y.k.O</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
All data flows from the highest calling structure, eventually being deposited in the data structures. Analysis of the table’s data confirms this premise.

MEMORYLESS PROCEDURES

Some procedures keep no record of their data passing or the data supplied. These procedures are used to do housekeeping for memory management, for example, but their analysis for data flow would produce a false amount of data transactions. Another area that these procedures appear in are arithmetic operations that are sometimes duplicated in hardware such as double precision math etc. This discussion leads to the problem that these procedures would be difficult to discern in an automated process. That is, if memoryless procedures are not to be considered in data flow analysis some form of human decision making is required. It should be noted that this is another premise that the automation of the Henry metric is based on. The absolute numbers for the Henry metrics would

<table>
<thead>
<tr>
<th>TABLE 2. GLOBAL CALL TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.DS</td>
</tr>
<tr>
<td>Y.DS</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Y.O</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Y.K.I</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Y.K.O</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
be inflated if memoryless procedures are not eliminated from the analysis. In short a memoryless procedure should be removed from the code to be analyzed if a more accurate assessment or if the absolute numbers produced are being used for a comparative study.
APPENDIX B: HENRY METRIC CODE

with GLOBAL, TEXT_IO;
use GLOBAL, TEXT_IO;

package HENRY_GLOBAL is

package INTEGER_IO is new TEXT_IO.INTEGER_IO(INTEGER);
use INTEGER_IO;

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

-- Real_IO produces floating point output

MAX_ARRAY_SIZE : constant integer := 50;
MAX_LINE_SIZE  : constant integer := 76;
DUMMY9s        : constant integer := 9999;
NULL_CHAR      : constant character := ' ';

-- DUMMY9s are used for false data input to the line length calculations

type DECLARED_TYPE is (BLANK, LOCAL_DECLARE, GLOBAL_DECLARE);

type ACTION_TYPE is (UNDEFINED, HENRY_HEAD_NODE, PACKAGE_TYPE, PROCEDURE_TYPE, FUNCTION_TYPE, PARAM_TYPE, ASSIGN_TYPE, IDENT_TYPE, DATA_STRUCTURE, FCNCALL_OR_DS).
PROCALL OR DS.
END PARAM DECLARE.
END ACTUAL PARAM.
END DECLARATIONS.
END ASSIGN TYPE.
END PACKAGE DECLARE.
END PACKAGE TYPE.
END FUNCTION TYPE.
END_PROCEDURE_CALL):

type PARAM_CLASS is (NONE, IN_TYPE, OUT_TYPE, IN_OUT_TYPE,
ACTUAL_PARAM):
subtype FORMAL_PARAM_CLASS is PARAM_CLASS range IN_TYPE..IN_OUT_TYPE;
subtype LEXEME_TYPE is string (1..MAX_LINE_SIZE);
subtype END_UNITS is ACTION_TYPE range
   END_FUNCTION_TYPE..END_PROCEDURE_CALL:

--Declared, action and parameter classes or types are used
--in the Henry record data collection process

type HENRY_RECORD:
type POINTER is access HENRY_RECORD;
type HENRY_RECORD is record
   IDENTITY : DECLARED_TYPE;
   NOMEN : LEXEME_TYPE;
   TYPE_DEFINE : ACTION_TYPE;
   PARAM_TYPE : PARAM_CLASS:
   NEXTI : POINTER;
end record:

--Henry record is the workhorse storage medium

type HENRY_LINE_COUNT_RECORD:
type LINE_POINTER is access HENRY_LINE_COUNT_RECORD;
type HENRY_LINE_COUNT_RECORD is record
   ID_NAME : LEXEME_TYPE;
   START_COUNT : INTEGER;
   STOP_COUNT : INTEGER;
   NEXT_REC : LINE_POINTER;
end record:

--Henry line count record is used to calculate the length of procedures
--or functions

type HENRY_DATA is record
   NAME_OF_DATA : LEXEME_TYPE;
   BEGIN_POINTER : POINTER;
   LINE_LENGTH_POINTER : LINE_POINTER;
end record:

--Henry data records are used to delineate the functions and procedures
--for easier data calculations

type HENRY_DATA_ARRAY is array (1..MAX_ARRAY_SIZE) of HENRY_DATA:

type OUTPUT_DATA is record
  TYPE OF ACTION TYPE := UNDEFINED;
  NAME OF LEXEME TYPE;
  TYPE FAN_IN : FLOAT := 0.0;
  TYPE FAN_OUT : FLOAT := 0.0;
  TYPE_COMPLEXITY : FLOAT := 0.0;
  TYPE_READ : FLOAT := 0.0;
  TYPE_WRITE : FLOAT := 0.0;
  TYPE_READ_WRITE : FLOAT := 0.0;
  TYPE FLOW : FLOAT := 0.0;
  CODE_LENGTH : INTEGER := 0;
end record,

--Output data records hold the final calculation numbers for storage into
--an output 'input file

type OUTPUT_ARRAY is array (1..MAX_ARRAY_SIZE) of OUTPUT_DATA:

NEXT HEN, LAST RECORD, NEW RECORD,
HEAD, NAME POINTER : POINTER;
HENRY_ARRAY : HENRY_DATA_ARRAY;
HENRY_LINE_COUNT : integer := 0;
OUTPUT_DATA : OUTPUT ARRAY;
LINE COUNT RECORD : HENRY LINE COUNT RECORD;
HEAD LINE, NEXT LINE, LAST LINE : LINE POINTER;
PACKAGE BODY DECLARE,
ASSIGN MARKER,
GLOBAL MARKER,
NAME TAIL SET,
ASSIGN STATEMENT,
FUNCTION PARAM DECLARE,
FORMAL PARAM DECLARE : BOOLEAN := FALSE;
FIRST HENRY CALL : BOOLEAN := TRUE;
DUMMY LEXEME : LEXEME,type;

procedure CREATE NODE(NEW NODE, LAST RECORD : in out POINTER);
procedure CREATE LINE COUNT NODE(NEXT LINE, LAST LINE : in out LINE POINTER);
procedure INITIALIZE HENRY(HEAD, in out POINTER;
  HEAD LINE, in out LINE POINTER);

procedure CLEAR HENRY LEXEME(HENRY LEXEME : in out LEXEME_TYPE);

end HENRY GLOBAL.
package body HENRY_GLOBAL is

procedure CREATE_NODE(NEW_NODE, LAST_RECORD : in out POINTER) is

TEMP_POINTER : POINTER;

begin
    put(result_file, "in create henry node"); new_line(result_file);
    TEMP_POINTER := new HENRY_RECORD;
    TEMP_POINTER.IDENTITY := BLANK;
    for I in 1..MAX_LINE_SIZE loop
        TEMP_POINTER.NOMEN(I) := NULL_CHAR;
    end loop;
    TEMP_POINTER.TYPE DEFINE := UNDEFINED;
    TEMP_POINTER.PARAM_TYPE := NONE;
    NEW_NODE.NEXT1 := TEMP_POINTER;
    LAST_RECORD := NEW_NODE;
    NEW_NODE := TEMP_POINTER;
end CREATE_NODE;

procedure CREATE_LINE_COUNT_NODE(NEXT_LINE,
                                  LAST_LINE : in out LINE_Pointer) is

TEMP_POINTER : LINE_Pointer;

begin
    put(result_file, "in henry create line node"); new_line(result_file);
    TEMP_POINTER := new HENRY_LINE_COUNT_RECORD;
    for I in 1..MAX_LINE_SIZE loop
        TEMP_POINTER.ID_NAME(I) := NULL_CHAR;
    end loop;
    TEMP_POINTER.START_COUNT := DUMMY9s;
    TEMP_POINTER.STOP_COUNT := DUMMY9s;
    NEXT_LINE.NEXT_REC := TEMP_POINTER;
    LAST_LINE := NEXT_LINE;
    NEXT_LINE := TEMP_POINTER;
end CREATE_LINE_COUNT_NODE;

procedure INITIALIZE_HENRY(HEAD : in out POINTER;
                           HEAD_LINE : in out LINE_Pointer) is

HEAD_STRING : STRING(1..9) := "HEAD NODE";
SIZE : INTEGER := 9;

begin
CREATE(HENRY_FILE, out_file. HENRY_FILE_NAME);
put(HENRY_FILE, "in INITIALIZE HENRY"). new_line(HENRY_FILE);
CREATE(HENRY_OUT, out_file. HENRY_OUT_NAME):
HEAD := new HENRY_RECORD;
HEAD NOMEN(1..SIZE) := HEAD STRING;
HEAD IDENTITY := BLANK.
HEAD TYPE DEFINE := HENRY_HEAD_NODE;
HEAD PARAM TYPE := NONE;
NEXT_HEN := HEAD;
CREATE_NODE(NEXT_HEN. LAST_RECORD);
HENRY_LINE_COUNT := 0;
DUMMY LEXEME(I) := NULL CHAR;
HEAD LINE := new HENRY_LINE_COUNT_RECORD;
HEAD LINE.ID NAME(1..SIZE) := HEAD STRING;
HEAD LINE.START COUNT := DUMMY9s;
HEAD LINE.STOP COUNT := DUMMY9s;
NEXT LINE := HEAD LINE;
CREATE_LINE COUNT_NODE(NEXT_LINE. LAST_LINE);
end INITIALIZE_HENRY;
------------------------------------------------------------------
-- clears the input string to null characters
procedure CLEAR_HENRY_LEXEME(HENRY_LEXEME in out LEXEME TYPE) is
begin
put(HENRY_FILE, "IN CLEAR HENRY LEXEME"). new_line(HENRY_FILE);
FOR I in 1..MAX LINE SIZE loop
HENRY_LEXEME(I) := NULL CHAR;
end loop;
END CLEAR_HENRY_LEXEME.
END HENRY GLOBAL;

------------------------------------------------------------------
-- TITLE: AN ADA SOFTWARE METRIC
-- MODULE NAME: PACKAGE HENRY METRIC
-- DATE CREATED: 06 APR 87
-- LAST MODIFIED: 15 MAY 87
-- AUTHORS: LCDR PAUL M HERZIG

56
DESCRIPTION: This package contains the Henry metric data collection and program control routines.

with GLOBAL, HENRY GLOBAL, HENRY ANALYSIS, HENRY DISPLAY, TEXT IO;
use GLOBAL, HENRY GLOBAL, HENRY ANALYSIS, HENRY DISPLAY, TEXT IO:

package HENRY is

procedure WRITE_HENRY_DATA(ID : in DECLARED_TYPE := BLANK;
IN NAME : in LEXEME_TYPE := DUMMY LEXEME;
DEFINE : in ACTION_TYPE := UNDEFINED;
PARAM : in PARAM_CLASS := NONE;
LINK : in POINTER);

procedure UPDATE_LINE_COUNT:

procedure WRITE_LINE_COUNT(IN NAME : in LEXEME_TYPE := DUMMY LEXEME;
FIRST_COUNT : in INTEGER := DUMMY9s;
LAST_COUNT : in INTEGER := DUMMY9s;
PTR : in LINE_POINTER);

end HENRY:

package body HENRY is

procedure WRITE_HENRY_DATA(ID : in DECLARED_TYPE := BLANK;
IN NAME : in LEXEME_TYPE := DUMMY LEXEME;
DEFINE : in ACTION_TYPE := UNDEFINED;
PARAM : in PARAM_CLASS := NONE;
LINK : in POINTER)
begin
print(result file, "in write henry data") new line(result file)
if ID BLANK then
    LINK IDENTIFY ID
end if

end WRITE_HENRY_DATA;
case ID as
  when LOCAL DECLARE - put(RESULT FILE, "Local declare")
  when GLOBAL DECLARE - put(RESULT FILE, "Global declare")
  when others - put(RESULT FILE, "Undefined")
end case
else put(RESULT FILE, "NO DECLARATION")
end if
new line(RESULT FILE)
if IN NAME(1) NULL CHAR then
  LINK NO NAME(1) MAX LINE SIZE) IN NAME(1) MAX LINE SIZE).
  put(RESULT FILE, "NAME")
else put(RESULT FILE, "NO NAME")
end if.
new line(RESULT FILE)
if DEFINE undefined then
  LINK TYPE DEFINE DEFINE.
  case DEFINE as
    when U NDEFINED put(RESULT FILE, "undefined")
    when HENRY HEAD NODE put(RESULT FILE, "Henry Head Node")
    when PACKAGE TYPE put(RESULT FILE, "Package declaration")
    when PROCEDURE TYPE put(RESULT FILE, "Procedure declaration")
    when FUNCTION TYPE put(RESULT FILE, "Function declaration")
    when PARAM TYPE put(RESULT FILE, "Parameter declaration")
    when ASSIGN TYPE put(RESULT FILE, "Assignment delimiter")
    when IDENT TYPE put(RESULT FILE, "Identifier")
    when DATA STRUCTURE put(RESULT FILE, "Data structure descriptor")
    when FUNCTION OR DS put(RESULT FILE, "Function or data descriptor")
    when PROCEDURE OR DS put(RESULT FILE, "Procedure or data descriptor")
    when END PARAM DECLARE put(RESULT FILE, "End parameter delimiter")
    when END ACTUAL PARAM put(RESULT FILE, "End actual parameter delimiter")
    when END DECLARATIONS put(RESULT FILE, "End declaration delimiter")
    when END ASSIGN TYPE put(RESULT FILE, "End assignment statement delimiter")
    when END PACKAGE DECLARE put(RESULT FILE, "End package declaration delimiter")
    when END PACKAGE TYPE put(RESULT FILE, "End package delimiter")
    when END FUNCTION TYPE put(RESULT FILE, "End function delimiter")
    when END PROCEDURE CALL put(RESULT FILE, "End procedure delimiter")
    when others put(RESULT FILE, "Unknown")
  end case
new line(RESULT FILE)
end if.

if PARAM = "" then
  LINK PARAM TYPE PARAM
  case PARAMS
    WHEN IN TYPE put(RESULT FILE, "IN PARAM")
    WHEN OUT TYPE put(RESULT FILE, "OUT PARAM")
    WHEN IN OUT TYPE put(RESULT FILE, "IN OUT PARAM")
    WHEN OTHERS put(RESULT FILE, "NONE")
  END CASE
end if.
new line(RESULT FILE)
WRITE HENRY DATA
-- increments the line count for eventual inclusion into
-- the calculation of a particular procedure's total length
-- the length number is used in the complexity calculation

procedure UPDATE LINE COUNT is

begin
  put(result_file, "in update line count");
  new_line(result_file);
  if not FORMAL PARAM DECLARE then
    HENRY LINE COUNT := HENRY LINE_COUNT + 1;
  end if;
end UPDATE LINE COUNT:

-- produces the records to hold the line count information
-- the records are not initially tied to a particular procedure
-- but are a parallel data structure until in the Hen_anal.pkg
-- where they are linked to the procedure that they hold the
-- data for

procedure WRITE LINE COUNT(IN NAME : in LEXEME TYPE: DUMMY LEXEME;
  FIRST COUNT : in INTEGER := DUMMY9s;
  LAST COUNT : in INTEGER := DUMMY9s;
  PTR in LINE_POINTER) IS

begin
  put(HENRY_FILE, "in WRITE LINE COUNT");
  new_line(HENRY_FILE);
  put(result_file, "in write line count");
  new_line(result_file);
  if NAME(1) = NULL CHAR then
    PTR NAME(1 MAX LINE SIZE) := IN NAME, end if;
  if FIRST COUNT = DUMMY9s then PTR START COUNT := FIRST COUNT, end if;
  if LAST COUNT = DUMMY9s then PTR STOP COUNT := LAST COUNT, end if;
end WRITE LINE COUNT:

end HENRY.
with GLOBAL GLOBAL PARSER BYPASS SUPPORT FUNCTIONS HENRY GLOBAL TEXT IO
use GLOBAL GLOBAL PARSER BYPASS SUPPORT FUNCTIONS HENRY GLOBAL TEXT IO;

package HENRY ANALYSIS 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.

PROC FUNC COUNT INTEGER 0
INDEX INTEGER
NAME POINTER POINTER

--PROC FUNC COUNT is the total number of procedures and functions in the
--analyzed package

type SELECTOR TYPE is (PROCEDURE FIND FUNCTION FIND
  VARIABLE FIND).

procedure CLEAN UP HENRY DATA(HEAD in POINTER).
procedure SET UP HENRY ARRAY(HEAD in POINTER
  HEAD LINE in LINE POINTER).

procedure SPRUCE UP HENRY DATA.
function LOCAL NAME(NAME POINTER in POINTER
  SELECTOR in SELECTOR TYPE,
  INDEX in INTEGER)
return BOOLEAN.

function CALCULATE LINE COUNT(WORK LINE LINE POINTER)
return INTEGER.

function FIND STRING SIZE(IN STRING LEXEME TYPE) RETURN INTEGER

function TRANSMISSION IN(IN NAME LEXEME TYPE
  BEGIN LOOP STOP LOOP POINTER)
return FLOAT.

function TRANSMISSION OUT(IN NAME LEXEME TYPE
  TOP POINTER)
return FLOAT.

procedure CALCULATE METRIC(HEAD in POINTER
  HEAD LINE in LINE POINTER).
package body HENRY ANALYSIS is

procedure CLEAN UP HENRY DATA(HEAD : IN POINTER) is

TEMP, TOP, BOTTOM : POINTER;

begin
  put(HENRY_FILE, "in CLEAN UP HENRY"); new_line(HENRY_FILE);
  CLEARSCREEN;
  put("Processing Henry data records ... please wait");
  TOP := HEAD;
  BOTTOM := TOP.NEXTI:

  LOOP
    EXIT WHEN TOP.TYPE.DEFINE = END_PACKAGE_DECLARE:
    TOP := BOTTOM;
    BOTTOM := TOP.NEXTI:
  END LOOP;

  LOOP
    EXIT WHEN BOTTOM.TYPE.DEFINE = END_PACKAGE_TYPE:
    if (BOTTOM.TYPE.DEFINE = PROCEDURE_TYPE) or
      (BOTTOM.TYPE.DEFINE = FUNCTION_TYPE) then
      PROC_FUNC_COUNT := PROC_FUNC_COUNT - 1:
    end if;

    TEMP := BOTTOM;
    BOTTOM := TEMP.NEXTI:
  end loop;
  BOTTOM := TOP;

  LOOP
    EXIT WHEN (TOP.TYPE.DEFINE = PROCEDURE_TYPE) OR
      (TOP.TYPE.DEFINE = FUNCTION_TYPE): TOP := BOTTOM.NEXTI:
  END LOOP;

  -- ensures all parameter records have a type defined

  FOR I in 1..PROC_FUNC_COUNT LOOP
    LOOP
      EXIT WHEN (TOP.TYPE.DEFINE = PROCEDURE_TYPE) OR
        (TOP.TYPE.DEFINE = FUNCTION_TYPE):
      TOP := BOTTOM.NEXTI:
    END LOOP;
  END LOOP;

end HENRY ANALYSIS.
BOTTOM TOP.
END LOOP.
TEMP = TOP NEXT1.
if TEMP TYPE DEFINE PARAM TYPE AND
TOP TYPE DEFINE FUNCTION TYPE then
LOOP
EXIT WHEN TEMP TYPE DEFINE END PARAM DECLARE.
if TEMP PARAM TYPE NOT IN FORMAL PARAM CLASS then
LOOP
EXIT WHEN (TEMP PARAM TYPE IN TYPE) OR
(TEMP PARAM TYPE OUT TYPE) OR
(TEMP PARAM TYPE IN OUT TYPE).
BOTTOM TEMP
TEMP BOTTOM NEXT1.
BOTTOM TEMP.
END LOOP.
BOTTOM TEMP.
TEMP TOP NEXT1.
TOP TEMP.
LOOP
EXIT WHEN (TOP PARAM TYPE IN TYPE) OR
(TOP PARAM TYPE OUT TYPE) OR
(TOP PARAM TYPE IN OUT TYPE).
TEMP PARAM TYPE BOTTOM PARAM TYPE.
BOTTOM TEMP.
TEMP TOP NEXT1.
TOP TEMP.
END LOOP.
else
TOP TEMP.
BOTTOM TEMP.
end if.
TEMP TOP NEXT1.
END LOOP.
functions usually invoke the default type parameter
insert this type if it is not defined.

else TOP TYPE DEFINE FUNCTION TYPE then
else TEMP TYPE DEFINE PARAM TYPE then
LOOP
EXIT WHEN TEMP TYPE DEFINE END PARAM DECLARE
TEMP PARAM TYPE IN TYPE
BOTTOM TEMP.
END LOOP.
end if.
end if.
TOP BOTTOM NEXT1.
BOTTOM TOP.
END LOOP FOR LOOP
end CLEAN UP HENRY DATA.
procedure SET UP HENRY ARRAY(HEAD in POINTER, HEAD LINE : in LINE POINTER)

WORK LINE TEMP LINE : LINE POINTER.
TEMP. TOP BOTTOM POINTER.

begin
put(HENRY FILE, "in SET UP HENRY") new line(HENRY FILE).
WORK LINE : HEAD LINE NEXT REC.
TEMP LINE : WORK LINE.
BOTTOM := HEAD;
TOP := BOTTOM;

-- GO PAST DECLARATIONS

LOOP
EXIT WHEN TOP TYPE DEFINE END PACKAGE DECLARE.
TOP := BOTTOM NEXT;
BOTTOM := TOP;
END LOOP.

-- set up the Henry array records so that their pointers are at the
-- top of each procedure or function

FOR 1 in 1 PROC FUNC COUNT LOOP
LOOP
EXIT WHEN (TOP TYPE DEFINE procedure TYPE) OR
(TOP TYPE DEFINE function TYPE).
TOP := BOTTOM NEXT;}
END LOOP: -- FOR LOOP
end SET UP HENRY ARRAY:

-- this procedure calculates the length of each procedure function
-- the results are fed into line length records

function CALCULATE LINE COUNT(WORK LINE, LINE POINTER)

   return INTEGER is

   DIFFERENCE INTEGER 0;
   INTEGER 1;

   begin
   put(HENRY FILE, "in CALCULATE LINE COUNT"); new line(HENRY FILE);
   DIFFERENCE WORK LINE STOP COUNT - WORK LINE START COUNT.
   RETURN (DIFFERENCE).
   end CALCULATE LINE COUNT.

   -- this function searches for local, within a procedure, and global-local,
   -- within a package, for variable name matches
   -- it is selectable for which name the search is conducted

function LOCAL NAME(NAME POINTER in POINTER,
                       SELECTOR in SELECTOR TYPE,
                       INDEX in INTEGER)

   return BOOLEAN is

   NAME, SOUGHT POINTER NAME, LEXEME TYPE,
   NAME SIZE, POINTER SIZE, INTEGER, MAX LINE SIZE,
   RESULT BOOLEAN FALSE.
   TEMP, TEMP1, INT, INTEGER 1.

   begin
   put(HENRY FILE, "in LOCAL NAME") new line(HENRY FILE);
   NAME, SOUGHT(1 NAME SIZE) NAME, POINTER Nomen(1 NAME SIZE).
   CONVERT UPPER CASE(NAME SOUGHT, NAME SIZE).
   if SELECTOR, PROCEDURE FIND, or (SELECTOR FUNCTION FIND)
   and PROC FUNC COUNT 0) then
   LOOP
   POINTER NAME(1 POINTER SIZE)
   HENRY ARRAY(H) NAME OF DATA(1 POINTER SIZE).
   CONVERT UPPER CASE(POINTER NAME, POINTER SIZE)
   RESULT (NAME SOUGHT(1 NAME SIZE)
   POINTER NAME(1 POINTER SIZE)).
   EXIT when (1 PROC FUNC COUNT) or (RESULT)
   loop
   end loop
I := 1:

--if it is a variable name search first within the package declarations, next within the procedure declarations

elsif SELECTOR = VARIABLE.Find then
    TEMP := HEAD.Next;
    LOOP
        EXIT WHEN (TEMP.Type Define = END PACKAGE DECLARATIONS) OR (RESULT);
        if TEMP.Type Define = IDENT Type then
            Pointer_Name(1..Pointer_Size) := TEMP.Nomen(1..Pointer_Size);
            Convert Upper Case(Pointer_Name(Pointer_Size);
            RESULT := (NAME SOUGHT(1..NAME Size) = Pointer_Name(1..Pointer_Size));
            end if;
        TEMP1 := TEMP.Next;
        TEMP := TEMP1;
        END LOOP;
    end if;
end if:
RETURN (RESULT);
end LOCAL NAME;

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

--finishes polishing the Henry records, the data can now be analyzed
--for local global data and starts the actual number crunching

procedure SPRUCE_UP HENRY DATA is
TEMP, TEMP1, TEMP2 : POINTER.

begin
put(HENRY_FILE, "in SPRUCE_UP_HENRY"); new_line(HENRY_FILE);
FOR I in 1..PROC_FUNC_COUNT LOOP
    TEMP1 := HENRY_ARRAY(I).BEGIN_POINTER;

--loop past parameters

    LOOP
        EXIT WHEN TEMP1 TYPE DEFINE = END_DECLARATIONS;
        TEMP2 := TEMP1.NEXT1;
        TEMP1 := TEMP2;
    END LOOP;
    TEMP := TEMP1.NEXT1;

--first analyze identifier types (variables) for local or global
--significance. Update the record if it is not local

    LOOP
        --LOOK FOR IDENT TYPES
        EXIT WHEN (TEMP.TYPE DEFINE = END FUNCTION TYPE) OR
            (TEMP.TYPE DEFINE = END PROCEDURE_CALL);
        if TEMP.TYPE DEFINE = IDENT.TYPE then
            if TEMP.IDENTITY = BLANK then
                if LOCAL NAME(TEMP, VARIABLE FIND, I) then
                    TEMP.IDENTITY := LOCAL DECLARE;
                else TEMP.IDENTITY := GLOBAL DECLARE;
                end if;
            end if;
        end if;
        TEMP1 := TEMP.NEXT1;
        TEMP := TEMP1;
    END LOOP;

--now go through the Henry records looking for unresolved
--procedure or function calls update the Henry records
--to reflect procedure types or function types or data structures

    TEMP1 := HENRY_ARRAY(I).BEGIN_POINTER;
    TEMP := TEMP1.NEXT1;

--get past declarations

    LOOP
        EXIT WHEN TEMP.TYPE DEFINE = END_DECLARATIONS;
        TEMP1 := TEMP.NEXT1;
        TEMP := TEMP1;
    END LOOP

66
looking for procedure or function calls

LOOP
EXIT WHEN (TEMP TYPE DEFINE = END FUNCTION TYPE) OR
   (TEMP TYPE DEFINE = END PROCEDURE CALL);

if TEMP TYPE DEFINE = PROCALL OR DS then
   TEMP1 := TEMP;
   LOOP -- MOVE PAST THE PARAMETERS
       EXIT WHEN TEMP1 TYPE DEFINE = END ACTUAL PARAM;
   TEMP2 := TEMP1;
   TEMP1 := TEMP2.NEXT1;
END LOOP;
if (LOCAL NAME(TEMP, PROCEDURE FIND, 1)) then
   TEMP TYPE DEFINE := PROCEDURE TYPE;
else
   TEMP2 := TEMP1.NEXT1;
   if TEMP2 TYPE DEFINE = ASSIGN_TYPE then
      TEMP TYPE DEFINE := DATA STRUCTURE;
   -- IF NOT IT IS A PROCEDURE CALL ONLY
   TEMP1 := TEMP2.NEXT1;
   LOOP
       EXIT WHEN TEMP1 TYPE DEFINE = END ASSIGN_TYPE;
       if (TEMP1 TYPE DEFINE = FuncALL OR DS) then
          if NOT LOCAL NAME(TEMP1, FUNCTION FIND, 1) then
             TEMP1 TYPE DEFINE := DATA STRUCTURE;
          else TEMP1 TYPE DEFINE := FUNCTION TYPE;
          end if;
       end if;
   end if;
   TEMP2 := TEMP1;
   TEMP1 := TEMP2.NEXT1;
END LOOP;
else TEMP TYPE DEFINE := PROCEDURE TYPE;
end if;
end if.

... only function calls that cannot be resolved into a local name are
... specified as data structures

elsif TEMP TYPE DEFINE = FuncALL OR DS then
   TEMP1 := TEMP;
   LOOP -- LOOKING FOR FUNCTIONS
       EXIT WHEN TEMP TYPE DEFINE = END ASSIGN TYPE;
       if TEMP TYPE DEFINE = FuncALL OR DS then
          if NOT LOCAL NAME(TEMP FUNCTION FIND, 1) then
             TEMP TYPE DEFINE := FUNCTION TYPE;
          else TEMP TYPE DEFINE := DATA STRUCTURE;
          end if;
       end if;
   end if;
   TEMP1 := TEMP2.NEXT1

67
TEMP := TEMP1;
END LOOP;
end if:
TEMP1 := TEMP;
TEMP := TEMP1.NEXT;
END LOOP; -- PROCALL OR_DS LOOP
END LOOP; -- FOR LOOP

end SPRUCE_UP_HENRY_DATA:

function FIND_STRING_SIZE(IN STRING : LEXEME_TYPE) RETURN INTEGER is
SIZE : INTEGER := 0;
BEGIN
PUT(HENRY_FILE, "IN FIND STRING SIZE"); NEW_LINE(HENRY_FILE);
FOR I IN 1..MAX_LINE_SIZE LOOP
IF IN_STRING(I) = NULL_CHAR THEN
SIZE := SIZE + 1;
END IF:
END LOOP:
RETURN SIZE;
END FIND_STRING_SIZE;

-- transitivity is detected by searching the right hand side of
-- assignment statements for a name match of the actual
-- parameters from a function or procedure call

function TRANSITIVITY_IN(IN NAME : LEXEME_TYPE;
BEGIN_LOOP, STOP LOOP : POINTER)
RETURN FLOAT is
ASSIGN_MARK.
PROCEDURE_MARK : BOOLEAN := FALSE;
TRANS_COUNT : FLOAT := 0.0;
TEMP, TEMP1 : POINTER := BEGIN_LOOP;
T1, T2 : POINTER.
MAX : INTEGER := MAX_LINE_SIZE;
BEGIN
-- stop loop is determined by where in the parameter list you are
LOOP
EXIT WHEN TEMP = STOP LOOP.
if TEMP TYPE DEFINE ASSIGN_TYPE THEN
ASSIGN MARK := TRUE;
elsi Temp TYPE DEFINE = END ASSIGN TYPE THEN
ASSIGN MARK := FALSE:
end if:

-- mark whether you've passed an assignment

if (Temp.NOMEN(1..MAX) = IN_NAME(1..MAX)) AND
(Not ASSIGN MARK) THEN
TRANS_COUNT := TRANS_COUNT - 1.0;

-- if you have detected a name match count the number of assignment
-- variables as transitive feed into the actual parameter
-- note functions have already been calculated the same for
-- data structures so skip these counts

T1 := Temp; T2 := T1.NEXT1:
if (T1 TYPE DEFINE : IDENT TYPE) AND
(T2 TYPE DEFINE : ASSIGN TYPE) then
LOOP
EXIT WHEN T2 TYPE DEFINE - END ASSIGN TYPE:
if T2 TYPE DEFINE : IDENT TYPE THEN
TRANS_COUNT := TRANS_COUNT - 1.0:
end if:
T1 := T2;
T2 := T1.NEXT1:
END LOOP:
end if:
end if:
TEMP := TEMP1.NEXT1:
TEMP1 := TEMP:
END LOOP:
RETURN(TRANS_COUNT):
END TRANSITIVITY IN:

-- if detect a name match on the right hand side of an assignment
-- statement have a transitive relation on this variable but
-- there is no need to count the rest of the assignment
-- variables because the most it can account for is 1

function TRANSITIVITY OUT(IN NAME : LEXEME TYPE;
TOP : POINTER)
RETURN FLOAT is
ASSIGN MARK : BOOLEAN := FALSE.
TRANS_COUNT : FLOAT := 0.0;
TEMP, TEMP1 : POINTER := TOP.
MAX : INTEGER := MAX LINE SIZE;
BEGIN
LOOP
EXIT WHEN (TEMP TYPE DEFINE END PROCEDURE CALL) OR (TEMP TYPE DEFINE END FUNCTION TYPE).
IF TEMP TYPE DEFINE ASSIGN TYPE THEN
ASSIGN MARK := TRUE.
ELSIF TEMP TYPE DEFINE END ASSIGN TYPE THEN
ASSIGN MARK := FALSE.
ELSE IF
IF (TEMP.NOMEN[1 MAX] IN NAME[1 MAX]) AND (ASSIGN MARK)
THEN
TRANS COUNT := TRANS COUNT - 10.
ELSE IF
TEMP := TEMP1.NEXT1.
TEMP1 := TEMP.
END LOOP:
RETURN(TRANS COUNT):
END TRANSITIVITY OUT:

--finishes polishing the data and with the transitivty functions calculates
--the fan in fan out of data besides the global data structures

procedure CALCULATE METRIC (HEAD in POINTER. 
HEAD LINE in LINE POINTER) is
TEMP LINE : LINE POINTER.
TEMP. TOP TEMP1 TEMP2 POINTER
PROC PTR
PARAM PTR POINTER
FAN IN FAN OUT FLOAT.
LENGTH : INTEGER 0.
MAX : INTEGER MAX LINE SIZE.
CODE EXPONENT : INTEGER 2.
COMPLEXITY
GLOBAL FLOW.
GLOBAL READ.
GLOBAL WRITE.
GLOBAL READ WRITE FLOAT.

--global flow represents the whole picture of global data flow
--the equation is below and encompasses both read and write to
--global data structures
--note: global data structures could be external function calls
--there is no means to determine the difference

NEW NAME : STRING MAX LINE SIZE.
NAME OF LENNAME TYPE.
ASSIGN MARK
GLOBAL MARK BOOLEAN FALSE.
SIZE INTEGER MAX LINE SIZE
NEW SIZE INTEGER 0
TEMP NAME STRING MAX SIZE

begin
PUT HENRY FILE IN CALCULATE METRIC NEW LINE HENRY FILE
end

IF FIRST HENRY CALL then
CLEAN UP HENRY DATA HEAD
SET UP HENRY ARRAY HEAD HEAD LINE
SPREAD UP HENRY DATA
FOR EACH PROCEDURE COUNT LOOP
GLOBAL READ
GLOBAL WRITE
FAN IN
FAN OUT
COMPUTATION
GLOBAL FLOW LENGTH
TEMP HENRY ARRAY HEAD IN POINTER
CLEAR HENRY NAME TEMP NAME
TEMP NAME INTEGER MAX LINE SIZE HENRY ARRAY NAME OF DATA
SIZE FIND STRING SIZE NAME
CLEAR HENRY NAME NEW NAME
CONVERT UPPER CASE TEMP NAME SIZE

initialize the variables for each procedure by setting the counters

TEMP TYPE DEFINE PROCEDURE TYPE CASE
OUT PUT DATA TYPE CASE PROCEDURE TYPE
NEW SIZE INTEGER 0
NEW NAME INTEGER 0
NEW NAME (NEW SIZE TEMP NAME + SIZE)
OUT PUT DATA NAME OF NEW SIZE NEW NAME NEW SIZE
PUT HENRY OUT
NEW LINE HENRY OUT
PUT HENRY OUT NEW NAME
NEW LINE HENRY OUT

TEMP TYPE DEFINE FUNCTION TYPE CASE
OUT PUT DATA TYPE CASE FUNCTION TYPE
NEW SIZE INTEGER 0
NEW NAME INTEGER 0
NEW NAME (NEW SIZE TEMP NAME + SIZE)
OUT PUT DATA NAME OF NEW SIZE NEW NAME NEW SIZE
PUT HENRY OUT
NEW LINE HENRY OUT
PUT HENRY OUT NEW NAME

71
NEW LINE HENRY OUT

END

BEGIN a pretty name for the function.

TEMP1 TEMP NEXT1
TEMP2 TEMP

IF GLOBAL increase global flow metric

THEN WHEN (TEMP TYPE DEFINE END FUNCTION TYPE) OR
      (TEMP TYPE DEFINE END PROCEDURE CALL)
      THEN TYPE DEFINE ASSIGN TYPE THEN
      ASSIGN MARKER TRUE
      ELSE TYPE DEFINE END ASSIGN TYPE THEN
      ASSIGN MARKER FALSE
      GLOBAL MARKER FALSE

END IDENTIFY GLOBAL DECLARE AND (ASSIGN MARKER)

GLOBAL READ GLOBAL READ TO
GLOBAL MARKER TRUE
GLOBAL READ WRITE GLOBAL READ WRITE TO

END IDENTIFY GLOBAL DECLARE AND
ASSIGN MARKER THEN
GLOBAL WRITE GLOBAL WRITE TO
GLOBAL MARKER TRUE

TEMP1 TEMP NEXT1
TEMP2 TEMP

END
look for procedure and function type actual parameters

```
TEMP  TEMP1
TEMP1 TEMP2 NEXT1
TEMP  TEMP1
LOOP
EXIT WHEN (TEMP TYPE DEFINE END FUNCTION TYPE OR
(TEMP TYPE DEFINE END PROCEDURE TYPE))
 TEMP1 TEMP
TEMP1 TEMP1 NEXT1
if TEMP TYPE DEFINE ASSIGN TYPE cases
ASSIGN MARKER TRUE
else TEMP TYPE DEFINE END ASSIGN TYPE cases
ASSIGN MARKER FALSE
GLOBAL MARKER FALSE
end if
if TEMP TYPE DEFINE PROCEDURE TYPE cases
 TEMP1 TEMP1 NEXT1
LOOP
EXIT WHEN TEMP TYPE DEFINE END ACTUAL PARAM
FAN OUT FAN OUT TO
TEMP2 TEMP1
TEMP1 TEMP2 NEXT1
END LOOP
else TEMP TYPE DEFINE FUNCTION TYPE THEN

count the function parameters

```
```
TEMP  TEMP1
LOOP
EXIT WHEN TEMP TYPE DEFINE END ACTUAL PARAM
FAN OUT FAN OUT TO
TEMP2 TEMP1
TEMP1 TEMP2 NEXT1
END LOOP
FAN IN FAN IN TO RETURN FROM FUNCTION
else TEMP TYPE DEFINE DATA STRUCTURE cases
(NOT ASSIGN MARK) THEN
GLOBAL MARK TRUE
GLOBAL WRITE GLOBAL WRITE
else TEMP TYPE DEFINE DATA STRUCTURE cases
(AND ASSIGN MARK) THEN
GLOBAL MARK THEN
GLOBAL READ WRITE GLOBAL WRITE
else NOT GLOBAL MARK THEN
GLOBAL READ GLOBAL READ
else
```
```
new week for transitivity in the actual parameters.

TOP  HEAD NEXTHT
TEMP  TOP
END

EXCEPTION TOP TYPE DEFINE END PACKAGE DECLARE.
TOP  HEAD NEXTHT
TEMP  TOP
END

EXCEPTION PROCEDURE COUNT LOOP.
TEMP  HENRY ARRAY BEGIN POINTER
PROC PTR  TEMP NEXTHT
TEMP  PROC PTR
END

EXCEPTION Proc PTR TYPE DEFINE PROCEDURE TYPE OR
PROC PTR TYPE DEFINE FUNCTION TYPE OR
PROC PTR TYPE DEFINE END PROCEDURE CALL OR
PROC PTR TYPE DEFINE END FUNCTION TYPE.
Proc PTR  HEAD NEXTHT
TEMP  PROC PTR
END

EXCEPTION Proc PTR TYPE DEFINE END PROCEDURE CALL AND
PROC PTR TYPE DEFINE END FUNCTION TYPE; THEN
PARAM PTR  PROC PTR NEXTHT
END

EXCEPTION WHEN PARAM PTR TYPE DEFINE END ACTUAL PARAM.
PARAM PTR  MAX  PARAM PTR MAX NEMEN MAX
PARAM PTR  HENRY TRANS-HIVITY IN NAME OF
PARAM PTR  PROC PTR
PARAM PTR  HENRY OUT TRANS-HIVITY OUT NAME OF
PARAM PTR.
PARAM PTR  PARAM PTR
PARAM PTR  HEAD NEXTHT
END

PARAM PTR  PROC PTR TYPE DEFINE END PROCEDURE CALL OR
PROC PTR TYPE DEFINE END FUNCTION TYPE OR
PROC PTR TYPE DEFINE END PACKAGE TYPE.
PROC PTR  PARAM PTR NEXTHT
END

PARAM PTR  PROC PTR TYPE DEFINE END PROCEDURE CALL OR
PROC PTR TYPE DEFINE END FUNCTION TYPE OR
PROC PTR TYPE DEFINE END PACKAGE TYPE.
PROC PTR  PARAM PTR NEXTHT
END

PARAM PTR  PROC PTR TYPE DEFINE END PROCEDURE CALL OR
PROC PTR TYPE DEFINE END FUNCTION TYPE OR
PROC PTR TYPE DEFINE END PACKAGE TYPE.
PROC PTR  PARAM PTR NEXTHT
END

PARAM PTR  PROC PTR TYPE DEFINE END PROCEDURE CALL OR
PROC PTR TYPE DEFINE END FUNCTION TYPE OR
PROC PTR TYPE DEFINE END PACKAGE TYPE.
GLOBAL FLOW ::= GLOBAL_WRITE
    (GLOBAL_READ - GLOBAL_READ_WRITE) -
    GLOBAL_READ_WRITE
    (GLOBAL_READ - GLOBAL_READ_WRITE - 10).
put(HENRY_OUT, "NUMBER OF LINES = ");
put(HENRY_OUT, LENGTH);
OUT_PUT_DATA(I).CODE_LENGTH := LENGTH;
NEW_LINE(HENRY_OUT);
put(HENRY_OUT, "FAN_IN = ");
put(HENRY_OUT, FAN_IN);
OUT_PUT_DATA(I).TYPE_FAN_IN := FAN_IN;
NEW_LINE(HENRY_OUT);
put(HENRY_OUT, "FAN_OUT = ");
put(HENRY_OUT, FAN_OUT);
OUT_PUT_DATA(I).TYPE_FAN_OUT := FAN_OUT;
NEW_LINE(HENRY_OUT);
put(HENRY_OUT, "COMPLEXITY = ");
put(HENRY_OUT, COMPLEXITY);
OUT_PUT_DATA(I).TYPE_COMPLEXITY := COMPLEXITY;
NEW_LINE(HENRY_OUT);
put(HENRY_OUT, "GLOBAL_READ = ");
put(HENRY_OUT, GLOBAL_READ);
OUT_PUT_DATA(I).TYPE_READ := GLOBAL_READ;
NEW_LINE(HENRY_OUT);
p(LHENRY_OUT, "GLOBAL_WRITE = ");
OUT_PUT_DATA(I).TYPE_WRITE := GLOBAL_WRITE;
NEW_LINE(HENRY_OUT);
p(LHENRY_OUT, "GLOBAL_READ_WRITE = ");
OUT_PUT_DATA(I).TYPE_READ_WRITE := GLOBAL_READ_WRITE;
NEW_LINE(HENRY_OUT);
p(LHENRY_OUT, "GLOBAL_FLOW ");
OUT_PUT_DATA(I).TYPE_FLOW := GLOBAL_FLOW;
NEW_LINE(HENRY_OUT.
END LOOP.
put(HENRY_OUT, "-------------------------");
end if. --FIRST HENRY CALL.
FIRST_HENRY_CALL := FALSE;
END_CALCULATE_METRIC.
END_HENRY_ANALYSIS.

--------------------------------------------------------
TITLE ANADA SOFTWARE METRICS

7.
IN ROW, WIDTH in INTEGER) is

SCREEN WIDTH INTEGER = 76,
CENTER POS INTEGER = 0,
TEMP NAME ROW STRING TYPE,

begin
FOR I IN 1..30 LOOP
TEMP NAME(I) = NULL CHAR;
END LOOP,
TEMP NAME(1..WIDTH) = NAME(1..WIDTH);
CENTER POS = SCREEN WIDTH 2 - WIDTH 2;
SET CURSOR POS(CENTER POS, IN ROW);
PUT(TEMP NAME);
NEW LINE,
end CENTER STRING,

procedure SET UP SCREEN(IN STRING in ROW STRING TYPE,
STRING SIZE in INTEGER) is

begin
CLEARSCREEN
SET REVERSE(ON)
CENTER STRING(IN STRING, 1..STRING SIZE)
SET REVERSE(OFF)
PUT(""-------------------------------"")
NEW LINE(2)
END SET UP SCREEN,

lets the entire record stream of the Henry meter data

procedure LIST HENRY DATA is

SHORT NAME SIZE STRING SIZE INTEGER 4
NAME POINTED POINTED HEAD
TEMP NAME END NAME TYPE
TEMP NAME STRING SHORT NAME END
NAME IN INTEGER
CENTER POINT END INTEGER
TEMP END END INTEGER
IDENT STRING ROW STRING 4
NAME POINTED END
begin
HEADER_STRING | HEADER_SIZE | "LIST OF HENRY RECORDS"
PUT HENRY FILE "IN LIST HENRY DATA" NEW LINE HENRY FILE
LOOP EXIT WHEN DONE
SET UP SCREEN | HEADER STRING | HEADER SIZE

LOOP
put "DECLARATION"
case TEMP POINTER IDENTITY is
when LOCAL DECLARE put "local declare"
when GLOBAL DECLARE put "global declare"
when there put "undeclared"
end case
new line
put NAME
put "NAME"
TEMP POINTER | NAME | MAX LINE SIZE
NAME = TEMP POINTER | NAME | MAX LINE SIZE
SHORT NAME = TEMP NAME | SHORT NAME SIZE
NAME = TEMP NAME | SHORT NAME SIZE
SHORT NAME = NAME | NULL CHAR
END LOOP

A DEC 00
TEMP POINTER | DEC NAME | MAX LINE SIZE
NAME = TEMP POINTER | DEC NAME | MAX LINE SIZE
SHORT NAME = TEMP NAME | SHORT NAME SIZE
NAME = TEMP NAME | SHORT NAME SIZE
SHORT NAME = NAME | NULL CHAR
END LOOP
PARAMETER TYPE
XTEMP POINTER PARAMETER
:
IN TYPE INTEGER

OUT TYPE STRING

FUNCTION PARAMETERS

RETURN VALUE

PRINTSCREEN ON EXIT ONE
PRINTSCREEN ON EXIT AND NOT DONE THEN PRINTSCREEN ON EXIT
SET SCREEN HEADER STRING HEADER SIZE

PARAMETER NEXT
XTEMP CONTINUE
WHEN NEW HEADER COMPLETE START OVER

IF ONE THEN
EXIT

XTEMP ONE

PRINTSCREEN ON EXIT ONE

PRINTSCREEN ON EXIT

START DATA

START DATA

STRING TO

INTEGER TO

POINTER TO HEAD NEXT

INTEGER TO

POINTER TO HEAD

INTEGER TO

POINTER TO STRING

BOOLEAN TO

BOOLEAN TO

BOOLEAN TO

BOOLEAN TO

INTEGER TO

INTEGER TO

NEWLINE HENRY HILL
NEWLINE STRING HEAD SIZE NEWLINE HENRY METRIC VALUES
NEWLINE HENRY OUT
NEWLINE MOD HENRY OUT
RESET (HENRY OUT, IN FILE).
end if
else OPEN (HENRY OUT, IN FILE, HENRY OUT NAME);
end if
SET UP SCREEN (HEADER STRING, HEADER SIZE);
IN STRING (1:8) "PACKA GE";
IN STRING (9:10) INPUT FILE NAME (1:4).
end IN STRING;
NEW LINE 2
LOOP
EXIT WHEN (END OF FILE (HENRY OUT) OR DONE);
FOR J IN 1 TO LOOP
IN STRING (J) NULL CHAR.
END LOOP;
GET LINE (HENRY OUT IN STRING, NUMBER OF);
PUT LINE (IN STRING).
PAUSE PRINT (STOP RUNNING COUNT, DONE);
IF RUNNING COUNT = 0 AND NOT DONE THEN
RUNNING COUNT = 1.
SET UP SCREEN (HEADER STRING, HEADER SIZE);
end if
END LOOP
IF NOT DONE THEN
STOP; 1 RUNNING COUNT -- 1;
PAUSE PRINT (STOP RUNNING COUNT, DONE);
end if
CLOSE (HENRY OUT);

LIST METRIC DATA.

This listing
- identifies each procedure function analyzed with its
  name and number.
- lists the relative comparison metric data for each function

structure WRITE RELATIVE DATA IS

INDICATOR(1)
INDICATOR(2)
INDICATOR(3) FLOAT 0.0
UPPER LIMIT constant FLOAT 1.0
LOWER LIMIT constant FLOAT 0.25
TEMP HOLDER STRING 10
STOP RUNNING INTEGER 1
HEADER STRING ROW STRING, TYPE
ROW STRING ROW STRING, TYPE
SIZE INTEGER
DONE BOOLEAN FALSE
HEADER SIZE INTEGER 25
...

84
HEADER STRING(1..HEADER SIZE) := "THE RELATIVE PERFORMANCE DATA".
SET UP SCREEN(HEADER STRING, HEADER SIZE);
if PROC FUNC COUNT < 16 THEN STOP := PROC FUNC COUNT;
else STOP := 16:
end if:
PUT[HENRY FILE, "IN WRITE RELATIVE DATA"]: NEW_LINE(HENRY_FILE):

--name the outer loop so that can exit gracefully when the user
--wants to quit

OUTER LOOP
FOR J IN 1..7 LOOP
CASE J is
when 1 => ROW STRING(1..6) := "FAN IN";
SIZE := 6;
when 2 => ROW STRING(1..7) := "FAN OUT";
SIZE := 7;
when 3 => ROW STRING(1..10) := "COMPLEXITY";
SIZE := 10;
when 4 => ROW STRING(1..11) := "GLOBAL READ";
SIZE := 11;
when 5 => ROW STRING(1..12) := "GLOBAL WRITE";
SIZE := 12;
when 6 => ROW STRING(1..17) := "GLOBAL READ WRITE";
SIZE := 17;
when 7 => ROW STRING(1..11) := "GLOBAL FLOW";
SIZE := 11;
when others => null:
end case.
CENTER STRING(ROW STRING, 4, SIZE);
FOR I IN 1..PROC FUNC COUNT LOOP
SET CURSOR POS(1, 1 - 5):
REL ARRAY(I).NAME OF := OUTPUT DATA(I).NAME OF:
PUT(REL ARRAY(I).NAME OF); SET CURSOR POS(42, 1 - 5), PUT(" : ");
end loop:

--set up the names before write the data

CASE J is
when 1 => put(REL ARRAY(I).TYPE FAN IN),
when 2 => put(REL ARRAY(I).TYPE FAN OUT),
when 3 => put(REL ARRAY(I).TYPE COMPLEXITY),
when 4 => put(REL ARRAY(I).TYPE READ),
when 5 => put(REL ARRAY(I).TYPE WRITE),
when 6 => put(REL ARRAY(I).TYPE READ WRITE),
when 7 => put(REL ARRAY(I).TYPE FLOW),
when others => null
end case.
NEW_LINE.
PAUSE PRINT(STOP RUNNING DONE)

boolean done is set true by user answering the query to put
EXIT OUTER LOOP WHEN DONE:
if (RUNNING = 0) AND (STOP = 16) THEN
    STOP := PROC FUNC COUNT - 17;
elsif RUNNING = 0 THEN
    SET UP SCREEN(HEADER STRING, HEADER SIZE):
    RUNNING := 1:
end if;
end loop:
end loop OUTER LOOP:

-- set up to loop again once have cycled through to first stop
-- count. This means have filled the screen once
STOP := 1; RUNNING := 1;
PAUSE_PRINT(STOP, RUNNING, DONE):
CLEARSCREEN:
PUT("The following are the maximums for each calculation: ");
new line:
put("-----------------------------------------------------");
new line:
put("Fan In "); put(REL ARRAY(MAX_FAN IN).NAME OF); new line;
put("Fan Out "); put(REL ARRAY(MAX_FAN_OUT).NAME OF); new line;
put("Complexity "); put(REL ARRAY(MAX_COMPLEXITY).NAME OF); NEW LINE:
put("Global Read "); put(REL ARRAY(MAX_READ).NAME OF); NEW LINE:
put("Global Write "); put(REL ARRAY(MAX_WRITE).NAME OF); NEW LINE:
put("Global Read Write "); put(REL ARRAY(MAX_READ_WRITE).NAME OF); NEW LINE:
put("Global Flow "); put(REL ARRAY(MAX_FLOW).NAME OF); NEW LINE:
new line:
put("-----------------------------------------------------");
new line:
STOP := 1; RUNNING := 1;
PAUSE_PRINT(STOP, RUNNING, DONE);
SET UP SCREEN(HEADER STRING, HEADER SIZE):

-- calculate the indicator numbers so that can determine the relative
-- performance of each procedure function within each category

FOR I IN 1..PROC FUNC COUNT LOOP
    if REL_ARRAY(I).TYPE FLOW = 0.0 THEN
        INDICATOR1 := REL ARRAY(I).TYPE COMPLEXITY /
        REL ARRAY(I).TYPE FLOW;
    else INDICATOR1 := REL ARRAY(I).TYPE COMPLEXITY:
    end if:
    if REL ARRAY(I).TYPE FAN OUT = 0.0 THEN
        INDICATOR2 := REL ARRAY(I).TYPE FAN IN /
        REL ARRAY(I).TYPE FAN OUT:
    else INDICATOR2 := REL ARRAY(I).TYPE FAN IN:
    end if:
    if REL ARRAY(I).TYPE WRITE = 0.0 THEN
        INDICATOR3 := REL ARRAY(I).TYPE READ /
        REL ARRAY(I).TYPE WRITE.
    end if:
else INDICATOR3  REL ARRAY(I) TYPE READ
end if
PUT REL ARRAY(I) NAME OF(I) put ") new line.

--put out the results of the indicator analysis

IF INDICATOR1  UPPER LIMIT THEN
  PUT("- Has significant complexity compared to global data flow ")
  new line
  if INDICATOR2  UPPER LIMIT THEN
    put("- This implies poor internal code structure ")
    new line
    put("- Consider remodularization ")
    new line
  elsif INDICATOR2  LOWER LIMIT THEN
    PUT("- This implies an extremely complex interface ")
    new line
  end if
ELSE IF INDICATOR1  LOWER LIMIT THEN
  PUT("- Has significant global data flow compared to complexity ")
  new line
  if INDICATOR3  UPPER LIMIT THEN
    put("- This implies an overworked data structure ")
    new line
    put("- or a considerable number of function calls ")
    new line
    put("- Consider redistributing the data structure into this module")
    new line
  elsif INDICATOR3  LOWER LIMIT THEN
    PUT("- This implies a program stress point").
    new line
    put("- or a critical data flow point")
    new line
    put("- Consider reorganizing the data structure ")
    new line
  end if
ELSE
  TEMP HOLDER(1 10)  REL ARRAY(I) NAME OF(I 10).
  put("- Is a fairly well balanced "), put(TEMP HOLDER).
  new line
  put("- This implies good modularization ").
  new line
END IF.
STOP 1 RUNNING : 1.
PAUSE PRINT(STOP RUNNING DONE).
EXIT WHEN DONE.
end loop.
if NOT DONE THEN
STOP 1 RUNNING : 1.
PAUSE PRINT(STOP RUNNING DONE).
end if
WRITE RELATIVE DATA

DECLARE MODULAR "WRITE RELATIVE DATA"

DECLARE GRAPH "WRITE RELATIVE DATA"

DECLARE LOOP CNT INTEGER
DECLARE ROW STRING, ROW STRING TYPE
DECLARE HEADER STRING, ROW STRING TYPE
DECLARE SIZE INTEGER
DECLARE STOP RUNNING INTEGER 1
DECLARE REM ON INTEGER
DECLARE STATE INTEGER

DECLARE NUM LOOP CNT INTEGER
DECLARE REM ON INTEGER
DECLARE HEADER SIZE INTEGER 3

DECLARE NUM LOOP CNT PROCEDURE OF NT
DECLARE REM ON PROCEDURE OF NT

END PROCEDURE

NUM LOOP CNT = PROCEDURE OF NT
REM ON = PROCEDURE OF NT

loop: count is the number of screens need to display
rem on- rem on is the partial screen that is left over

HEADER STRING(1) = "THE GRAPHICAL PERFORMANCE DATA"
if NUM LOOP CNT = 1 THEN STOP 5
done STOP REM ON
end if

PUT HENRY FILE "IN WRITE RELATIVE DATA", NEW LINE(HENRY FILE)
SET UP SCREEN(HEADER STRING, HEADER SIZE)

set up to exit gracefully when the user wants to quit

GRAPH LOOP
FOR J IN 1 TO LOOP
CASE J is
when 1 = ROW STRING(1-6) "FAN IN"
SIZE 6.
when 2 = ROW STRING(1-7) "FAN OUT"
SIZE 7.
when 3 = ROW STRING(1-10) "COMPlexity"
SIZE 10.
when 4 = ROW STRING(1-11) "GLOBAL READ"
SIZE 11.
when 5 = ROW STRING(1-12) "GLOBAL WRITE"
SIZE 12.
when 6 = ROW STRING(1-17) "GLOBAL READ WRITE"
SIZE 17.
when 7 = ROW STRING(1-11) "GLOBAL FLOW"
SIZE 11.

85
I...
FOR INI PROCEDURE CONTROL LOOP
IF OUT PUT DATA IN TYPE FAN IN
  OUT PUT DATA MAX FAN IN TYPE FAN IN THEN
  MAX FAN IN = 1
END IF
IF OUT PUT DATA IN TYPE FAN OUT
  OUT PUT DATA MAX FAN OUT TYPE FAN OUT THEN
  MAX FAN OUT = 1
END IF
IF OUT PUT DATA IN TYPE COMPLEXITY
  OUT PUT DATA MAX COMPLEXITY TYPE COMPLEXITY THEN
  MAX COMPLEXITY = 1
END IF
IF OUT PUT DATA IN TYPE READ
  OUT PUT DATA MAX READ TYPE READ THEN
  MAX READ = 1
END IF
IF OUT PUT DATA IN TYPE WRITE
  OUT PUT DATA MAX WRITE TYPE WRITE THEN
  MAX WRITE = 1
END IF
APPENDING MODIFIED PARSERS

BYPASS TOKEN ARRAY ENTRY CODE result

ENTRY CODE FOR EACH WORD TO BE CONSUMED
BY BYPASS FUNCTION

ENTRY CODE FOR EACH WORD TO BE CONSUMED
BY BYPASS FUNCTION

 jumping BYPASS FUNCTION

The function compares the lexeme of the current token with the
lexeme currently being sought by the parser. If the current token
matches the lexeme, then a test is conducted to ensure it is not
a reserved word.

function BYPASS(TOKEN ARRAY ENTRY CODE integer) return boolean as
CONSUME boolean FALSE
LEXEME string 1 LINESIZE
SIZE natural
HISTORY LEXEME string 1 MAX LINESIZE

begin
GET CURRENT TOKEN RECORDCURRENT TOKEN RECORD LEXEME LEXEME.
LEXEME CURRENT TOKEN RECORD LEXEME.
SIZE CURRENT TOKEN RECORD LEXEME SIZE + 1

90
when TOKEN NUMBER LITERAL
if (CURRENT TOKEN RECORD TOKEN TYPE = NUMBER LITERAL) then
CONSUME TRUE
DECLARE TYPE CONSTANT DECLARE
OPERAND NODE CURRENT TOKEN RECORD DECLARE TYPE
DECLARE TYPE VARIABLE DECLARE
if HENRY WRITE ENABLE then
WRITE HENRY DATA LOCAL DECLARE HENRY ENUM MEMBER TYPE
NONE NEXT HEN
CREATE NODE(NEXT HEN LAST RECORD)
HENRY WRITE ENABLE FALSE.
end if
end if

when TOKEN CHARACTER LITERAL
if (CURRENT TOKEN RECORD TOKEN TYPE = CHARACTER LITERAL) then
if HENRY WRITE ENABLE then
WRITE HENRY DATA LOCAL DECLARE HENRY ENUM MEMBER TYPE
NONE NEXT HEN
CREATE NODE(NEXT HEN LAST RECORD)
HENRY WRITE ENABLE FALSE.
end if
CONSUME TRUE.
end if

when TOKEN STRING LITERAL
if (CURRENT TOKEN RECORD TOKEN TYPE = STRING LITERAL) then
when \textsc{token} \textsc{begin} \\
 if (\textsc{adjust lexeme}(\textsc{lexeme}, \textsc{size}) == \textsc{"for"}) then \textsc{consume} \textsc{true} \\
 when \textsc{token} \textsc{others} \\
 if (\textsc{adjust lexeme}(\textsc{lexeme}, \textsc{size}) == \textsc{"others"}) then \textsc{consume} \textsc{true} \\
 when \textsc{token} \textsc{return} \\
 if (\textsc{adjust lexeme}(\textsc{lexeme}, \textsc{size}) == \textsc{"return"}) then \textsc{consume} \textsc{true} \\
 when \textsc{token} \textsc{exit} \\
 if (\textsc{adjust lexeme}(\textsc{lexeme}, \textsc{size}) == \textsc{"exit"}) then \textsc{consume} \textsc{true} \\
 when \textsc{token} \textsc{procedure} \\
 if (\textsc{adjust lexeme}(\textsc{lexeme}, \textsc{size}) == \textsc{"procedure"}) then \textsc{consume} \textsc{true} \\
 when \textsc{token} \textsc{function} \\
 if (\textsc{adjust lexeme}(\textsc{lexeme}, \textsc{size}) == \textsc{"function"}) then \textsc{consume} \textsc{true} \\
 when \textsc{token} \textsc{with} \\
 if (\textsc{adjust lexeme}(\textsc{lexeme}, \textsc{size}) == \textsc{"with"}) then \textsc{consume} \textsc{true} \\
 when \textsc{token} \textsc{use} \\
 if (\textsc{adjust lexeme}(\textsc{lexeme}, \textsc{size}) == \textsc{"use"}) then \textsc{consume} \textsc{true}
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 =&gt;
  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 := TRUE;
  end if;

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;
ADAMEASURE: AN IMPLEMENTATION OF THE HALSTEAD AND HENRY METRICS
NAVAL POSTGRADUATE SCHOOL MONTEREY CA
P M HERZIG JUN 87
UNCLASSIFIED F/G 12/5 NL
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_NONE =>
  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;
  OPERATOR_MTRIC(TOKEN_AND, CONSUME, RESERVE_WORD_TEST);

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

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

when TOKEN_XOR =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "xor") then
    CONSUME := TRUE;
  end if;
  OPERATOR_MTRIC(TOKEN_XOR, CONSUME, RESERVE_WORD_TEST);
when TOKEN_MOD =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "mod") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_MOD, CONSUME, RESERVE_WORD_TEST);
when TOKEN REM =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "rem") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN REM, CONSUME, RESERVE_WORD_TEST);
when TOKEN_ABSOLUTE =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "abs") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_ABSOLUTE, CONSUME, RESERVE_WORD_TEST);
when TOKEN_ASTERISK =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "**") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_ASTERISK, CONSUME, RESERVE_WORD_TEST);
when TOKEN_SLASH =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "/") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_SLASH, CONSUME, RESERVE_WORD_TEST);
when TOKEN_EXPONENT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "**") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_EXPONENT, CONSUME, RESERVE_WORD_TEST);
when TOKEN_PLUS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "+") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_PLUS, CONSUME, RESERVE_WORD_TEST);
when TOKEN_MINUS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "-") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_MINUS, CONSUME, RESERVE_WORD_TEST);
when TOKEN_AMPERSAND =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "&") then
        CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_AMPERSAND, CONSUME, RESERVE_WORD_TEST);
end if;
OPERATOR_METRIC(TOKEN_AMPERSAND, CONSUME, RESERVE_WORD_TEST);

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

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

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

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

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

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

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

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

when TOKEN_SEMICOLON =>
if [ADJUST_LEXEME(LEXEME, SIZE) = ";") then

99
UPDATE_LINE_COUNT;
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_APOSTROPHE =>
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;
end if;

when TOKEN RIGHT BRACKET =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "\n") 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_PARSER
-- DATE CREATED: 09 OCT 86
-- LAST MODIFIED: 30 MAY 87
--
-- AUTHORS:    LCDR JEFFREY L. NIEDER
--             LT KARL S. FAIRBANKS, JR.
--             LCDR PAUL M. HERZIG
-- DESCRIPTION: This package contains eight functions that
--    make up the highest level productions for our top-down.
--    recursive descent parser.
--
-- ******************************************************
with PARSER_1, PARSER_2, PARSER_3, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION, HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL, TEXT_IO;
use PARSER_1, PARSER_2, PARSER_3, HENRY_GLOBAL, HENRY, 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;
    put(REsULT_FILE, "In compilation "); new_line(REsULT_FILE);
    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
    put(REsULT_FILE, "In compilation_unit "); new_line(REsULT_FILE);
    if (CONTEXT_CLAUSE) then
      if (BASIC_UNIT) then
        return (TRUE);
      else
        return (FALSE);
      end if:
    else
      return (FALSE);
    end if:
  end COMPILATION_UNIT:
function CONTEXT_CLAUSE return boolean is
begin
  put(RESULT_FILE, "In context clause "); new_line(RESULT_FILE);
  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; -- inner while loop
  end loop; -- outer while loop
  return (TRUE);
end CONTEXT_CLAUSE;

function BASICUNIT return boolean is
begin
  put(RESULT_FILE, "In basic-unit "); new_line(RESULT_FILE);
  if (LIBRARY_UNIT) then
    return (TRUE);
  elsif (SECONDARY_UNIT) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end BASICUNIT;

function LIBRARYUNIT return boolean is
begin
  put(RESULT_FILE, "In library unit "); new_line(RESULT_FILE);
  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
    return (TRUE);
  else
    SYNTAX_ERROR("Library unit");
  end if; -- if procedure_unit statement
end LIBRARYUNIT;
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:
else
    return (FALSE);
end if:
end LIBRARY_UNIT;

-- SECONDARY_UNIT --> LIBRARY_UNIT_BODY
-- -- --> SUBUNIT
function SECONDARY_UNIT return boolean is
begin
    put(RESULT_FILE, "In secondary_unit"); new_line(RESULT_FILE);
    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
    put(RESULT_FILE, "In library_unit_body"); new_line(RESULT_FILE):
    if (BYPASS(TOKEN_PROCEDURE)) then
        DECLARE TYPE := PROCEDURE_DECLARE;
        if (PROCEDURE_UNIT) then

104
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;
HENRY_WRITE_ENABLE := TRUE;
put(result_file, "true"); new line(result_file);
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
put(RESULT_FILE, "In subunit "); new line(RESULT_FILE);
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;
else
SYNTAX_ERROR("Subunit");
end if;
else
SYNTAX_ERROR("Subunit");
end if;
else
return (FALSE);
end if;
end SUBUNIT;
end SUBUNIT;

end PARSER_0;

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_UNIT_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_BOD 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;

with PARSER_2, PARSER_3, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION, HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL, TEXT_10;
use PARSER_2, PARSER_3, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION, HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL, TEXT_10;
package body PARSER_1 is

-- GENERIC DECLARATION --> GENERIC PARAMETER DECLARATION ?
-- GENERIC FORMAL PART
function GENERIC DECLARATION return boolean is
begin
put(RESULT_FILE, "In generic declaration "); new_line(RESULT_FILE):
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 ;
-- => type private DISCRIMINANT PART :
-- is GENERIC TYPE DEFINITION :
-- => with procedure PROCEDURE UNIT
-- => with function FUNCTION UNIT
function GENERIC PARAMETER DECLARATION return boolean is
begin
put(RESULT_FILE, "In generic parameter declaration "); new_line(RESULT_FILE):
if (IDENTIFIER LIST) then
if (BYPASS(TOKEN COLON)) then
if (MODE) then

107
null;
end if;
-- if mode statement
if (NAME) then
-- check for type_mark
if (BYPASS(TOKEN_ASSIGNMENT)) then
if (EXPRESSION) then
null;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
-- if expression statement
end if;
-- if bypass(token_assignment)
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
-- if bypass(token_semicolon)
elsif (BYPASS(TOKEN_TYPE)) then
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (GENERIC_TYPE_DEFINITION) then
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (PRIVATE_TYPE_DEFINITION) then
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (GENERIC_TYPE_DEFINITION) then
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (PRIVATE_TYPE_DEFINITION) then
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (BYPASS(TOKEN_IS)) then
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (BYPASS(TOKEN_IS)) then
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (BYPASS(TOKEN_TYPE)) then
if (BYPASS(TOKEN_IS)) then
DECLARE_TYPE := PROCEDURE_DECLARE;
if (PROCEDURE_UNIT) then

return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if:
elsif (BYPASS(TOKEN_FUNCTION)) then
DECLARE_TYPE := FUNCTION_DECLARE;
if (FUNCTION_UNIT) then
return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if:
else
SYNTAX_ERROR("Generic parameter declaration");
end if:
else
return (FALSE);
end if;
end GENERIC_PARAMETER_DECLARATION;

-- GENERIC FORMAL PART --> procedure PROCEDURE_UNIT
--> function FUNCTION_UNIT
--> package PACKAGE_DECLARATION

function GENERIC_FORMAL_PART return boolean is
begin
put(RESULT_FILE, "In generic formal part "); new_line(RESULT_FILE);
if (BYPASS(TOKEN_PROCEDURE)) then
DECLARE_TYPE := PROCEDURE_DECLARE;
if (PROCEDURE_UNIT) then
return (TRUE);
else
SYNTAX_ERROR("Generic formal part");
end if:
elsif (BYPASS(TOKEN_FUNCTION)) then
DECLARE_TYPE := FUNCTION_DECLARE;
if (FUNCTION_UNIT) then
return (TRUE);
else
SYNTAX_ERROR("Generic formal part");
end if:
elsif (BYPASS(TOKEN_PACKAGE)) then
DECLARE_TYPE := PACKAGE_DECLARE;
if (PACKAGE_DECLARATION) then
return (TRUE);
else
SYNTAX_ERROR("Generic formal part");
end if:
else
SYNTAX_ERROR("Generic formal part");
end if:
else
return (FALSE);
end if;
end GENERIC_FORMAL_PART:
PROCEDURE_UNIT --> identifier [FORMAL_PART ?] is SUBPROGRAM_BODY
-- --> identifier FORMAL_PART ?]
-- --> identifier FORMAL_PART ? renames NAME;
function PROCEDURE_UNIT return boolean is
begin
put(RESULT_FILE, "In procedure unit "); new_line(RESULT_FILE);
DECLARATION := TRUE;
HENRY_WRITE_ENABLE := TRUE;
if (BYPASS(TOKEN_IDENTIFIER)) then
if PACKAGE_BODY DECLARE then
WRITE_HENRY_DATA(LOCAL_DECLARE, DUMMY_LEXEME,
PROCEDURE_TYPE, NONE, LAST_RECORD);
end if:
SCOPE_LEVEL := SCOPE_LEVEL - 1;
if (FORMAL_PART) then
null;
end if:
-- if formal part statement
if (BYPASS(TOKEN_IS)) then
WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PARAM_DECLARE,
NONE, NEXT_HEN);
CREATE_NODE(NEXT_HEN, LAST_RECORD);
WRITE_LINE_COUNT(LAST_RECORD.NOMEN, HENRY_LINE_COUNT,
DUMMY9s, NEXT_LINE);
if (SUBPROGRAM_BODY) then
WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, ENDPROCEDURECALL,
NONE, NEXT_HEN);
CREATE_NODE(NEXT_HEN, LAST_RECORD);
WRITE_LINE_COUNT(DUMMY_LEXEME, DUMMY9s, HENRY_LINE_COUNT.
NEXT_LINE);
CREATE_LINE_COUNT_NODE(NEXT_LINE, LAST_LINE);
SCOPE_LEVEL := SCOPE_LEVEL - 1;
return (TRUE);
else
SYNTAX_ERROR("Procedure unit");
end if:
-- if subprogram body statement
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:
-- if bypass(token_semicolon)
else
SYNTAX_ERROR("Procedure unit");
end if:
-- if name statement
110
end if:  -- if bypass(token_is)
else
    return (FALSE):
end if:  -- if bypass(token_identifier)
end PROCEDURE_UNIT:

function SUBPROGRAM_BODY return boolean is
    NAME_POINTER : POINTER;
    begin
        put(RESULT_FILE, "In subprogram body "); new_line(RESULT_FILE);
        NAME_POINTER := NEXTHEN;
        DECLARATION := TRUE;
        if (BYPASS(TOKEN_NEW)) then
            HENRY_WRITE_ENABLE := FALSE;
        end if;
        if (NAME) then
            if (GENERIC_ACTUAL_PART) then
                null:
            end if;
        end if;
        if (BYPASS(TOKEN_SEMICOLON)) then
            return (TRUE);
        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:
elsif (DECLARATIVE_PART) then
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_DECLARATIONS, NONE, NEXTHEN);
    CREATE_NODE(NEXTHEN, LAST_RECORD);
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("Subprogram body");  
    end if;  
  else  
    SYNTAX_ERROR("Subprogram body");  
  end if;  
end if;  
-- if bypass(token exception)  
if (BYPASS(TOKEN END)) then  
  HENRY_WRITE_ENABLE := FALSE;  
  if (DESIGNATOR) then  
    null:  
  end if;  
  if (BYPASS(TOKEN SEMICOLON)) then  
    DECLARATION := TRUE;  
    return (TRUE);  
  else  
    SYNTAX_ERROR("Subprogram body");  
  end if;  
else  
  SYNTAX_ERROR("Subprogram body");  
end if;  
-- if sequence of statements  
else  
  SYNTAX_ERROR("Subprogram body");  
end if;  
-- if bypass(token end)  
elsif (BYPASS(TOKEN BEGIN)) then  
  DECLARATION := FALSE;  
  WRITE HENRY DATA(BLANK, DUMMY_LEXEME. END DECLARATIONS.  
  NONE. NEXT HEN);  
  CREATE_NODE(NEXT HEN, LAST_RECORD);  
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;  
  else  
    SYNTAX_ERROR("Subprogram body");  
  end if;  
-- if bypass(token exception)  
if (BYPASS(TOKEN END)) then  
  HENRY_WRITE_ENABLE := FALSE;  
  if (DESIGNATOR) then  
    null:  
  end if;  
-- if designator statement  
end if;  
-- if sequence of statements  
else  
  SYNTAX_ERROR("Subprogram body");  
end if;  
-- if bypass(token end)
if (BYPASS(TOKEN_SEMICOLON)) then
    DECLARATION := TRUE;
    return (TRUE);
else
    SYNTAX_ERROR("Subprogram body");
    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;

else
    return (FALSE);
end if;
end SUBPROGRAM_BODY;

FUNCTION UNIT --> DESIGNATOR FUNCTION_UNIT_TAIL

function FUNCTION_UNIT return boolean is
begin
    putForResultFile("In function unit ").new lineForResultFile;
    DECLARATION := TRUE;
    HENRY_WRITE_ENABLE := TRUE;
    if (DESIGNATOR) then
        if PACKAGE_BODY_DECLARE then
            WRITE_HENRY_DATA(LOCAL_DECLARE, DUMMY_LEXEME, FUNCTION_TYPE, NONE, LAST_RECORD);
        end if;
    SCOPE_LEVEL := SCOPE_LEVEL - 1:
    if (FUNCTION_UNIT_TAIL) then
        SCOPE_LEVEL := SCOPE_LEVEL - 1;
        return (TRUE);
    else
        SYNTAX_ERROR("Function unit");
        end if;
    end if;
end FUNCTION_UNIT:

113
-- FUNCTION UNIT TAIL --> is new NAME [GENERIC_ACTUAL_PART ?] ;
-- --> [FORMAL_PART ?] return NAME FUNCTION BODY

function FUNCTION UNIT TAIL return boolean is
begin
  put (RESULT_FILE, "In function unit tail"); new_line (RESULT_FILE);
  if (BYPASS (TOKEN _IS)) then
    FUNCTION PARAM DECLARE := TRUE;
  end if;
  if (BYPASS (TOKEN_NEW)) then
    if (NAME) then
      if (GENERIC_ACTUAL_PART) then null;
    else
      if (GENERIC _ACTUAL PART) then
        null;
    end if;
  end if;
  if (BYPASS (TOKEN _SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR ("Function unit tail");
    end if;
  end if;
  if (NAME) then
    if (GENERIC _ACTUAL PART) then
      null;
    else
      SYNTAX_ERROR ("Function unit tail");
    end if;
  end if;
  if (FORMAL_PART) then
    FUNCTION PARAM DECLARE := FALSE;
    if (BYPASS (TOKEN RETURN)) then
      if (NAME) then
        if (FUNCTION BODY) then
          return (TRUE);
        else
          SYNTAX_ERROR ("Function unit tail");
        end if;
      else
        SYNTAX_ERROR ("Function unit tail");
      end if;
    else
      SYNTAX_ERROR ("Function unit tail");
    end if;
  end if;
  if (BYPASS (TOKEN RETURN)) then
    if (NAME) then
      if (FUNCTION BODY) then
        return (TRUE);
      else
        SYNTAX_ERROR ("Function unit tail");
      end if;
    else
      SYNTAX_ERROR ("Function unit tail");
    end if;
  end if;
  return (FALSE);
end FUNCTION UNIT TAIL:
function FUNCTION_BODY return boolean is
begin
put(RESULT_FILE, "In function body "); new_line(RESULT_FILE);
if (BYPASS(TOKEN IS)) then
  WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PARAM_DECLARE, NONE, NEXT PARAM DECLARE,
  CREATE_NODE(NEXT_HEN, LAST_RECORD);
  if (FUNCTION_BODY_TAIL) then
    WRITE_LINE_COUNT(DUMMY_LEXEME, DUMMY9s, HENRY_LINE_COUNT, NEXT_LINE);
    CREATE_LINE_COUNT_NODE(NEXT_LINE, LAST_LINE);
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_FUNCTION_TYPE, NONE, NEXT_HEN);
    CREATE_NODE(NEXT_HEN, LAST_RECORD);
  end if;
return (TRUE);
elsif (BYPASS(TOKEN SEMICOLON)) then
  return (TRUE);
else
  return (FALSE);
end if;
end FUNCTION_BODY;

function FUNCTION_BODY_TAIL return boolean is
begin
put(RESULT_FILE, "In function body tail "); new_line(RESULT_FILE);
if (BYPASS(TOKEN SEPARATE)) then
  if (BYPASS(TOKEN SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Function body tail");
  end if;
else
  SYNTAX_ERROR("Function body tail");
end if;
elsif (BYPASS(TOKEN BRACKETS)) 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:
--                    --> [type ? identifier is [ENTRY DECLARATION *
--                                                      REPRESENTATION_CLAUSE]* end [identifier ? ?];

function TASK_DECLARATION return boolean is
begin
    put(RESULT_FILE, "In task declaration "); new_line(RESULT_FILE);
    DECLARATION := TRUE;
    if (BYPASS(TOKEN TYPE)) then
        null:
    end if:
    if (BYPASS(TOKEN BODY)) then
        if (TASK BODY) then
            if (BYPASS(TOKEN SEMICOLON)) then
                return (TRUE):
            else
                SYNTAX_ERROR("Task declaration");
            end if:
        else
            SYNTAX_ERROR("Task declaration");
        end if:
    elsif (BYPASS(TOKEN IDENTIFIER)) then
        SCOPE_LEVEL := SCOPE_LEVEL - 1;
        if (BYPASS(TOKEN IS)) then
            while (ENTRY DECLARATION) loop
                null:
            end loop;
            while (REPRESENTATION_CLAUSE) 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("Task declaration");
            end if:
        else
            SYNTAX_ERROR("Task declaration");
        end if:
    else
        SYNTAX_ERROR("Task declaration");
    end if:
end if:
116
elsif (BYPASS(TOKEN_SEMICOLON)) then
  SCOPE_LEVEL := SCOPE_LEVEL - 1;
  return (TRUE);
else
  SYNTAX_ERROR("Task declaration");
end if;
else
  SYNTAX_ERROR("Task body");
end if;
end TASKDECLARATION;

-- TASK BODY --> identifier is TASK_BODY_TAIL
function TASK_BODY return boolean is
begin
  put(RESULT_FILE, "In task body "); new_line(RESULT_FILE);
  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;
  else
    SYNTAX_ERROR("Task body");
  end if;
else
  SYNTAX_ERROR("Task body");
end if;
else
  return (FALSE);
end if;
end TASK_BODY;

-- TASK BODY_TAIL --> separate
--    --> DECLARATIVE PART ?; begin SEQUENCE OF STATEMENTS
--        --> EXCEPTION HANDLER - ?; end identifier ?
function TASK_BODY_TAIL return boolean is
begin
  put(RESULT_FILE, "In task body tail"); new_line(RESULT_FILE);
  DECLAREATION := TRUE;
  if (BYPASS(TOKEN_SEPARATE)) then
    return (TRUE);
  elsif (DECLARATIVE_PART) then
    if (BYPASS(TOKEN_BEGIN)) then
      DECLAREATION := FALSE;
      SEQUENCEOF_STATEMENTS;
      if (BYPASS(TOKEN_EXCEPTION)) then
        EXCEPTION HANDLER;
        while (EXCEPTION HANDLER) loop
          117
null;
end loop;
else
SYNTAX_ERROR("Task body tail");
end if;  -- if exception_handler statement
end if;  -- if bypass(token_exception)
if (BYPASS(TOKEN_END)) then
  if (BYPASS(TOKEN_IDENTIFIER)) then
    null;
  end if;
  -- if bypass(token_identifier)
  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_end)
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_exception)
  if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      null;
    end if;
    -- if bypass(token_identifier)
    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_separate)
end TASK_BODY_TAIL:

118
-- PACKAGE DECLARATION --> body PACKAGE BODY

- PACKAGE DECLARATION --> identifier PACKAGE UNIT

function PACKAGE DECLARATION return boolean is
begin
put(RESULT FILE, "In package declaration "); new line(RESULT FILE);
DECLARATION := TRUE;
HENRY WRITE ENABLE := TRUE;
if (BYPASS(TOKEN BODY)) then
PACKAGE BODY DECLARE := TRUE;
HENRY WRITE ENABLE := FALSE;
if (PACKAGE BODY) then
return (TRUE);
else
SYNTAX_ERROR("Package declaration");
end if;
elsif (BYPASS(TOKEN IDENTIFIER)) then
WRITE HENRY DATA(LOCAL DECLARE, DUMMY LEXEME, PACKAGE TYPE, NONE, LAST RECORD);
SCOPE LEVEL := SCOPE LEVEL - 1;
if (PACKAGE UNIT) then
SCOPE LEVEL := SCOPE LEVEL - 1;
return (TRUE);
else
SYNTAX_ERROR("Package declaration");
end if;
else
SYNTAX_ERROR("Package body");
end if;
end PACKAGE DECLARATION;

-- PACKAGE BODY --> identifier is PACKAGE BODY TAIL

function PACKAGE BODY return boolean is
begin
put(RESULT FILE, "In package body "); new line(RESULT FILE):
if (BYPASS(TOKEN IDENTIFIER)) then
SCOPE LEVEL := SCOPE LEVEL - 1;
if (BYPASS(TOKEN IS)) then
if (PACKAGE BODY TAIL) then
WRITE HENRY DATA(BLANK, DUMMY LEXEME, END PACKAGE TYPE, NONE, NEXT HEN);
SCOPE LEVEL := SCOPE LEVEL - 1;
return (TRUE);
else
SYNTAX_ERROR("Package body");
end if;
else
SYNTAX_ERROR("Package body");
end if;
else
SYNTAX_ERROR("Package body");
end if;
end PACKAGE BODY:
return (FALSE);
eend if: -- if bypass(token_identifier)
eend 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
puts(RESULT_FILE, "In package_body_tail "); new_line(RESULT_FILE);
DECLAREATION := 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
  DECLAREATION := 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:
      end if:
    end if:
    -- if exception_handler statement
  end if:
  -- if bypass(token_exception)
else
  if (BYPASS(TOKEN_END)) then
    HENRY_WRITE_ENABLE := FALSE;
    if (BYPASS(TOKEN_IDENTIFIER)) then
      null;
    end if:
    -- if bypass(token_identifier)
  else
    DECLAREATION := TRUE;
  end if:
  -- if bypass(token_semicolon)
else
  SYNTAX_ERROR("Package body tail");
end if:
-- if bypass(token_end)
else
  SYNTAX_ERROR("Package body tail");
end if:
-- if sequence_of_statements
elsif (BYPASS(TOKEN_END)) then
  HENRY_WRITE_ENABLE := FALSE;
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;
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:
    -- exception handler state-
  end if:
else
  SYNTAX_ERROR("Package body tail");
end if;  -- if sequence_of_statements
end if;
if (BYPASS(TOKEN END)) then
  HENRY_WRITE_ENABLE := FALSE;
  if (BYPASS(TOKEN IDENTIFIER)) then
    null:
  end if;  -- if bypass(token_identifier)
else
  SYNTAX_ERROR("Package body tail");
end if;  -- if bypass(token_semicolon)
else
  SYNTAX_ERROR("Package body tail");
end if;  -- if bypass(token_end)
else
  SYNTAX_ERROR("Package body tail");
end if;  -- if sequence_of_statements
elsif (BYPASS(TOKEN END)) then
  HENRY_WRITE_ENABLE := FALSE;
  if (BYPASS(TOKEN IDENTIFIER)) then
    null:
  end if;  -- if bypass(token_identifier)
else
  SYNTAX_ERROR("Package body tail");
end if;  -- if bypass(token_semicolon)
else

121


return (FALSE);
end if:  -- if bypass(token_separate)
end PACKAGE_BODY_TAIL:

-- PACKAGE_UNIT --> is PACKAGE_TAIL_END
   --> renames NAME;
function PACKAGE_UNIT return boolean is
begin
   put(RESULT_FILE, "In package_unit "); new_line(RESULT_FILE);
   if (BYPASS(TOKEN_IS)) then
      if (PACKAGE_TAIL_END) then
         return (TRUE);
      else
         SYNTAX_ERROR("Package unit");
      end if:
   elsif (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:
   else
      SYNTAX_ERROR("Package unit");
   end if:  -- if name statement
else
   return (FALSE):  -- if bypass(token_is)
end if:
end PACKAGE_UNIT:

-- PACKAGE_TAIL_END --> new NAME [GENERIC_ACTUAL_PART ?];
   --> [BASIC_DECLARATIVE_ITEM]* 'private'
   --> [BASIC_DECLARATIVE_ITEM]* end 'identifier ?';
function PACKAGE_TAIL_END return boolean is
begin
   put(RESULT_FILE, "In package_tail_end "); new_line(RESULT_FILE);
   if (BYPASS(TOKEN_NEW)) then
      if (NAME) then
         if (GENERIC_ACTUAL_PART) then
            null;
         end if:  -- if generic_actual_part statement
      else
         if (BYPASS(TOKEN_SEMICOLON)) then
            return (TRUE);
         else
            SYNTAX_ERROR("Package tail end");
         end if:
      else
         SYNTAX_ERROR("Package tail end");
      end if:
else
      SYNTAX_ERROR("Package tail end").
elsif (BASIC DECLARATIVE ITEM) then
while (BASIC DECLARATIVE ITEM) loop
null;
end loop;
if (BYPASS(TOKEN PRIVATE)) then
while (BASIC DECLARATIVE ITEM) loop
null;
end loop;
end if;
if (BYPASS(TOKEN_END)) then
HENRY WRITE ENABLE := FALSE;
if (BYPASS(TOKEN_IDENTIFIER)) then
null:
end if:
if (BYPASS(TOKEN SEMICOLON)) then
WRITE HENRY_DATA(BLANK, DUMMY LEXEME, END PACKAGE DECLARE. NONE, NEXT HEN);
CREATE_NODE(NEXT HEN, LAST_RECORD);
return (TRUE);
else
SYNTAX_ERROR("Package tail end");
end if:
if bypass(token _semicolon)
else
SYNTAX_ERROR("Package tail end");
end if:
if bypass(token _end)
elsif (BYPASS(TOKEN_END)) then
HENRY WRITE ENABLE := FALSE;
if (BYPASS(TOKEN IDENTIFIER)) then
null:
end if:
if (BYPASS(TOKEN SEMICOLON)) then
WRITE HENRY_DATA(BLANK, DUMMY LEXEME, END PACKAGE DECLARE. NONE, NEXT HEN):
CREATE_NODE(NEXT HEN, LAST_RECORD):
return (TRUE);
else
SYNTAX_ERROR("Package tail end");
end if:
if bypass(token _semicolon)
else
SYNTAX_ERROR("Package tail end");
end if:
if bypass(token end)
elsif (BYPASS(TOKEN_END)) then
HENRY WRITE ENABLE := FALSE;
if (BYPASS(TOKEN IDENTIFIER)) then
null:
end if:
if (BYPASS(TOKEN SEMICOLON)) then

WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PACKAGE_DECLARE, 
NONE, NEXT_HEN); 
CREATE_NODE(NEXT_HEN, LAST_RECORD):
   return (TRUE);
else
   SYNTAX_ERROR("Package tail end");
   end if;
   -- if bypass(token_semicolon)
else
   return (FALSE);
end if;
-- if bypass(token_new)
end PACKAGE_TAIL_END;

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

-- BASIC_DECLARATIVE_ITEM -> BASIC_DECLARATIVE
--                             -> REPRESENTATION_CLAUSE
--                             -> USE WITH OR USE_CLAUSE

function BASIC_DECLARATIVE_ITEM return boolean is
begin
   put(RESULT_FILE, "In basicdeclarative item "); new_line(RESULT_FILE);
   HENRY_WRITE_ENABLE := TRUE;
   if (BASIC_DECLARATION) then
      HENRY_WRITE_ENABLE := FALSE;
      return (TRUE);
   elsif (REPRESENTATION_CLAUSE) then
      return (TRUE);
   elsif (BYPASS(TOKEN_USE)) then
      if (WITH OR USE_CLAUSE) then
         return (TRUE);
      else
         SYNTAX_ERROR("Basic declarative item");
      end if;
   else
   end if;
   return (FALSE);
end BASIC_DECLARATIVE_ITEM;

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

-- DECLARATIVE_PART --> [BASIC_DECLARATIVE_ITEM * LATER DECLARATIVE_ITEM *
function DECLARATIVE_PART return boolean is
begin
   put(RESULT_FILE, "In declarative part "); new_line(RESULT_FILE);
   while (BASIC_DECLARATIVE_ITEM) loop
      null;
   end loop;
   while (LATER_DECLARATIVE_ITEM) loop
      null;
   end loop;
   return (TRUE);
end DECLARATIVE_PART;

124
-- 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

function BASIC_DECLARATION return boolean is
begin
put(RESULT_FILE, "In basic declaration "); new_line(RESULT_FILE);
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
  DECLARE TYPE := PROCEDURE_DECLARE;
  if (PROCEDURE_UNIT) then
    HENRY_WRITE_ENABLE := FALSE;
    return (TRUE);
  else
    SYNTAX_ERROR("Basic declaration");
  end if;
elsif (BYPASS(TOKEN_FUNCTION)) then
  DECLARE TYPE := FUNCTION_DECLARE;
  if (FUNCTION_UNIT) then
    HENRY_WRITE_ENABLE := FALSE;
    return (TRUE);
  else
    SYNTAX_ERROR("Basic declaration");
  end if;
elsif (BYPASS(TOKEN_PACKAGE)) then
  DECLARE TYPE := PACKAGE_DECLARE;
  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;
else
  SYNTAX_ERROR("Basic declaration");
end if;
end if:
end function;
SYNTAX_ERROR("Basic declaration");
end if;
elseif (IDENTIFIER_DECLARATION) then
    HENRY_WRITE_ENABLE := FALSE;
    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;

-- LATER_DECLARATIVE_ITEM --> PROPER BODY
-- --> generic GENERIC DECLARATION
-- --> use WITH OR USE CLAUSE

function LATER_DECLARATIVE_ITEM return boolean is
begin
    put(RESULT_FILE, "In later declarative item "). new_line(RESULT_FILE);
    if (PROPER_BODY) then
        -- check for body declaration
        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

126
put(RESULT FILE, "In proper body "); new line(RESULT FILE);
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
  put(RESULT FILE, "In sequence of statements "); new line(RESULT FILE);
  if (STATEMENT) then
    while (STATEMENT) loop
      null;
    end loop;
    return (TRUE);
  else
    return (FALSE);
  end if;
end SEQUENCE OF STATEMENTS.
-- STATEMENT --> LABEL " SIMPLE STATEMENT  
-- --> LABEL " COMPOUND STATEMENT

function STATEMENT return boolean is
begin
  put(RESULT FILE, "In statement "); new line(RESULT FILE);
  if (LABEL) then
    null;
  end if;
  if (SIMPLE STATEMENT) then
    return (TRUE);
  elsif (COMPOUND STATEMENT) then
    return (TRUE);
  else
    return (FALSE);
  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
  put(RESULT FILE, "In compound statement "); new line(RESULT FILE);
  if (BYPASS(TOKEN IF)) then
    NESTING METRIC(IF CONSTRUCT);
    if (IF STATEMENT) then
      return (TRUE);
    else
      SYNTAX ERROR("Compound statement");
    end if;
  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;
  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;
  else
    SYNTAX ERROR("Compound statement");
  end if;
end if;

128
elsif (BYPASS(TOKEN SELECT)) then
  if (SELECT STATEMENT) then
    return (TRUE);
  else
    SYNTAX ERROR("Compound statement");
  end if;
else
  return (FALSE);
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
  put(RESULT FILE, "In block statement "); new_line(RESULT FILE);
  if (DECLARATION) then
    DECLARE STATUS : TRUE;
  else
    DECLARE : 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:
  if (SEQUENCE OF STATEMENTS) then
    if (BYPASS(TOKEN EXCEPTION)) then
      if (EXCEPTION HANDLER) then
        while (EXCEPTION HANDLER) loop
          null:
        end loop:
      else
        null:
      end if:
    else
      null:
    end if:
  end if:
  null:
end if:
end loop.
else
SYNTAX ERROR("Block statement").
end if.
end if.
end if.
if (BYPASS(TOKEN END)) then
if (BYPASS(TOKEN IDENTIFIER)) then
null.
end if.
end if.
end if.
end if.
end if.
end if.
else
SYNTAX ERROR("Block statement").
end if.
end if.
else
SYNTAX ERROR("Block statement").
end if.
else
SYNTAX ERROR("Block statement").
end if.
else
if not (DECLARE STATUS) then
DECLARATION FALSE:
end if:
return (FALSE).
else
if bypass(token begin)
end if.
end BLOCK STATEMENT.

-- IF STATEMENT -- . EXPRESSION then SEQUENCE OF STATEMENTS
-- . else EXPRESSION then SEQUENCE OF STATEMENTS .
-- . else SEQUENCE OF STATEMENTS . end if .

function IF STATEMENT return boolean is
begin
put(RESULT FILE, "In if statement "); new line(RESULT FILE).
if (EXPRESSION) then
if (BYPASS(TOKEN THEN)) then
if (SEQUENCE OF STATEMENTS) then
while (BYPASS(TOKEN ELSIF)) loop
if (EXPRESSION) then
if (BYPASS(TOKEN THEN)) then
if (SEQUENCE OF STATEMENTS) then
while (BYPASS(TOKEN ELSE)) loop
if (EXPRESSION) then
if (BYPASS(TOKEN THEN)) then
if not (SEQUENCE OF STATEMENTS) then
SYNTAX ERROR("If statement").
end if.
else
SYNTAX ERROR("If statement").
end if.
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)
if (BYPASS(TOKEN END)) then
  if (BYPASS(TOKEN IF)) then
    if (BYPASS(TOKEN SEMICOLON)) then
      NESTING METRIC(IF END):
      return (TRUE);
    else  
SYNTAX ERROR("If statement");
end if.  -- if bypass(token semicolon)
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
put(RESULT FILE, "In case statement "). new_line(RESULT FILE);
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
        NESTING METRIC(CASE END).
        return (TRUE);
      else

SYNTAX_ERROR("Case statement");
end if: -- if bypass(token _semicolon)
else
  SYNTAX_ERROR("Case statement");
end if: -- if bypass(token _case)
else
  SYNTAX_ERROR("Case statement");
end if: -- if case_statement_alternative
else
  SYNTAX_ERROR("Case statement");
end if: -- if bypass(token _is)
else
  return (FALSE);
end if: -- if expression statement
end CASE STATEMENT:

-- CASE STATEMENT_ALTERNATIVE --> when CHOICE | CHOICE * -->
-- SEQUENCE_OF_STATEMENTS
function CASE_STATEMENT_ALTERNATIVE return boolean is
  begin
    put (RESULT(FILE, "In case_statement_alternative "); new_line (RESULT_FILE);
    if (BYPASS (TOKEN WHEN)) then
      if (CHOICE) then
        while (BYPASS (TOKEN_BAR)) loop
          if not (CHOICE) then -- if not choice statement
            SYNTAX_ERROR("Case statement alternative");
          end if:
        end loop;
      end if: -- if choice statement
      if (BYPASS (TOKEN_ARROW)) then
        if (SEQUENCE_OF_STATEMENTS) then
          return (TRUE);
        else
          SYNTAX_ERROR("Case statement alternative");
        end if: -- if sequence of statements
      else
        SYNTAX_ERROR("Case statement alternative");
      end if: -- if bypass(token _arrow)
    else
      SYNTAX_ERROR("Case statement alternative");
    end if: -- if choice statement
    else
      return (FALSE);
    end if: -- if bypass(token _when)
  end CASE_STATEMENT_ALTERNATIVE;
-- LOOP_STATEMENT --> identifier : ? ITERATION_SCHEME ? | loop
-- SEQUENCE_OF_STATEMENTS end loop identifier ? ;
function LOOP_STATEMENT return boolean is
begin
  put(RESULT_FILE, "In loop statement "); new_line(RESULT_FILE);
  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:
        end if;
        if (BYPASS(TOKEN SEMICOLON)) then
          NESTING_METRIC(LOOP_END);
          return (TRUE);
        else
          SYNTAX_ERROR("Loop state_ment");
        end if:
      else
        SYNTAX_ERROR("Loop statement ");
      end if:
    else
      SYNTAX_ERROR("Loop statement ");
    end if:
  else
    if (SEQUENCE_OF_STATEMENTS) then
      if (BYPASS(TOKEN LOOP)) then
        SYNTAX_ERROR("Loop statement ");
      else
        SYNTAX_ERROR("Loop statement ");
      end if:
    else
      SYNTAX_ERROR("Loop statement ");
    end if:
  else
    return (FALSE);
  end if:
end LOOP_STATEMENT;

-- EXCEPTION_HANDLER --> when EXCEPTION_CHOICE EXCEPTION_CHOICE * 
-- SEQUENCE_OF_STATEMENTS
function EXCEPTION_HANDLER return boolean is
  133
begin
put(RESULT_FILE, "In exception handler "); new_line(RESULT_FILE);
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);
else
SYNTAX_ERROR("Exception handler");
end if;
-- if sequence_of_statements
else
SYNTAX_ERROR("Exception handler");
end if;
-- if bypass(token_arrow)
else
SYNTAX_ERROR("Exception handler");
end if;
-- if exception_choice statement
else
return (FALSE);
-- if bypass(token-when)
end if:
end EXCEPTION_HANDLER:

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

-- ACCEPT_STATEMENT --> identifier (EXPRESSION) ?| FORMAL_PART ?
-- do SEQUENCE_OF_STATEMENTS end identifier ? ? ;
function ACCEPT_STATEMENT return boolean is
begin
put(RESULT_FILE, "In accept statement "); new_line(RESULT_FILE);
if (BYPASS(TOKEN IDENTIFIER)) then
if (BYPASS(TOKEN LEFT_PAREN)) then
if (EXPRESSION) then
if (BYPASS(TOKENRIGHT_PAREN)) then
null;
else
SYNTAX_ERROR("Accept statement");
end if;
-- if bypass(token right paren)
else
SYNTAX_ERROR("Accept statement");
end if;
-- if expression statement
end if:
-- if bypass(token left paren)
if (FORMAL_PART) then
null:
end if:
-- if formal_part statement
if (BYPASS(TOKEN DO)) then
if (SEQUENCE_OF_STATEMENTS) then
if (BYPASS(TOKEN END)) then
if (BYPASS(TOKEN IDENTIFIER)) then

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

134
null:
end if; -- if bypass(token_identifier)
else
SYNTAX_ERROR("Accept statement");
end if; -- if bypass(token_end)
else
SYNTAX_ERROR("Accept statement");
end if; -- if sequence_of_statements
end if;
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Accept statement");
end if; -- if bypass(token_semicolon)
else
return (FALSE);
end if; -- if bypass(token_identifier)
end ACCEPT_STATEMENT;

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

-- SELECT STATEMENT --> SELECT STATEMENT TAIL SELECT_ENTRY_CALL end select :
function SELECT_STATEMENT return boolean is
begin
put(RESULT_FILE, "In select_statement "); new_line(RESULT_FILE);
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; -- if bypass(token_end)
else
  SYNTAX_ERROR("Select statement");
end if; -- if select_entry_call statement
der
else
return (FALSE);
end if; -- if select_statement_tail
end SELECT_STATEMENT;

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

-- SELECT STATEMENT TAIL --> SELECT ALTERNATIVE or SELECT ALTERNATIVE
-- NAME: SEQUENCE OF STATEMENTS ?
function SELECT_STATEMENT_TAIL return boolean is
begin
put(RESULT_FILE, "In select_statement_tail "); new_line(RESULT_FILE);
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)
-- check for entry call statement
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
--
-- --> when EXPRESSION => ? delay DELAY_STATEMENT
--
-- --> when EXPRESSION => ? terminate ;
function SELECT_ALTERNATIVE return boolean is
begin
put(RESULT_FILE, "In select_alternative "); new_line(RESULT_FILE);
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
SYNTAX_ERROR("Select alternative");
end if;
end SELECT_ALTERNATIVE;

-- SEQUENCE OF STATEMENTS
--
-- SEQUENCE OF STATEMENTS

data type SELECT_ALTERNATIVE is
end SELECT_ALTERNATIVE;
SYNTAX_ERROR("Select alternative");
end if; -- if accept_statement
elsif (BYPASS(TOKEN_DELAY)) then
if (DELAY_STATEMENT) then
if (SEQUENCE_OF_STATEMENTS) then
null;
end if; -- if sequence_of_statements
return (TRUE);
else
SYNTAX_ERROR("Select alternative");
end if; -- if delay_statement
elsif (BYPASS(TOKEN_TERMINATE)) then
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Select alternative");
end if; -- if bypass(token_semicolon)
else
return (FALSE);
end if; -- if bypass(token_accept)
end SELECT_ALTERNATIVE;

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

-- SELECT ENTRY_CALL --> else SEQUENCE_OF_STATEMENTS
-- --> or delay DELAY_STATEMENT SEQUENCE_OF_STATEMENTS ?

function SELECT_ENTRY_CALL return boolean is
begin
put(RESULT_FILE, "In select entry_call "); new_line(RESULT_FILE);
if (BYPASS(TOKEN_ELSE)) then
if (SEQUENCE_OF_STATEMENTS) then
return (TRUE);
else
SYNTAX_ERROR("Select entry call");
end if; -- if sequence_of_statements
elsif (BYPASS(TOKEN_OR)) then
if (BYPASS(TOKEN_DELAY)) then
if (DELAY_STATEMENT) then
if (SEQUENCE_OF_STATEMENTS) then
null;
end if; -- if sequence_of_statements
return (TRUE);
else
SYNTAX_ERROR("Select entry call");
end if; -- if delay_statement
else
SYNTAX_ERROR("Select entry call");
end if; -- if bypass(token_delay)
else
return (FALSE);
end if; -- if bypass(token_else)
end SELECT_ENTRY_CALL.
package PARSER_2 is
    IDENT DECLARE : BOOLEAN := FALSE:
    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;
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:
else
SYNTAX_ERROR("Generic actual part");
end if:
end loop:
if (BYPASS(TOKEN RIGHT PAREN)) then
return (TRUE):
else
SYNTAX_ERROR("Generic actual part");
end if:
else
SYNTAX_ERROR("Generic actual part");
end if:
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 generic formal parameter statement
if (EXPRESSION) then
-- check for generic actual parameter
return (TRUE):
else

end if:
------------------------------------------------------------------------------

139
return (FALSE);
end if:
-- if expression
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:
elsif (BYPASS(TOKEN STRING LITERAL)) then
  if (BYPASS(TOKEN ARROW)) then
    return (TRUE):
  else
    SYNTAX ERROR("Generic formal parameter");
  end if:
else
  SYNTAX ERROR("Generic formal parameter");
end if:
else
  SYNTAX ERROR("Generic formal parameter");
end if:
return (FALSE):
end GENERIC FORMAL PARAMETER:

-- 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:
else
  SYNTAX ERROR("Generic type definition");
end if:

140
end if: -- if bypass(token brackets)
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: -- if bypass(token brackets)
elsif (BYPASS(TOKEN ARRAY)) then
 if (ARRAY TYPE DEFINITION) then
 return (TRUE);
 else
 SYNTAX ERROR("Generic type definition");
 end if: -- if array type definition
elsif (BYPASS(TOKEN_ACCESS)) then
 if (SUBTYPE INDICATION) then
 return (TRUE);
 else
 SYNTAX ERROR("Generic type definition");
 end if: -- if subtype indication
else
 return (FALSE);
 end if: -- if bypass(token_left_paren)
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
     return (TRUE);
    else
     SYNTAX ERROR("Subtype declaration");
    end if:
   else
    SYNTAX ERROR("Subtype declaration");
   end if:
  end if:
end 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 discriminant_part statement
if (BYPASS(TOKEN IS)) then -- declaration is full_type if 'is'
if (PRIVATE_TYPE_DECLARATION) then
null:
elsif (TYPE_DEFINITION) then -- present, otherwise incomplete type
null:
else
SYNTAX_ERROR("Type declaration");
end if; -- if type_definition statement
end if: -- if bypass(token is)
if (BYPASS(TOKEN SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Type declaration");
end if: -- if bypass(token semicolon)
else
return (FALSE);
end if: -- if bypass(token identifier)
end TYPE_DECLARATION:

-- DISCRIMINANT_PART --> (DISCRIMINANT_SPECIFICATION
-- DISCRIMINANT_SPECIFICATION.*)
function DISCRIMINANT_PART return boolean is
begin
if (BYPASS(TOKEN LEFT_PAREN)) then
if (DISCRIMINANT_SPECIFICATION) then
while (BYPASS(TOKEN SEMICOLON)) loop
if not (DISCRIMINANT_SPECIFICATION) then
SYNTAX_ERROR("Discriminant part");
end if.
end if.
end if.
end if; -- if not discriminant specification
end loop:
if (BYPASS(TOKEN_RIGHT_PAREN)) then
  return (TRUE):
else
  SYNTAX_ERROR("Discriminant part");
  end if; -- if bypass(token_right_paren)
else
  SYNTAX_ERROR("Discriminant part");
  end if:
  if discriminant_specification
else
  return (FALSE);
  end if; -- if bypass(token_left_paren)
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
      -- check for type_mark
      if (BYPASS(TOKEN_ASSIGNMENT)) then
        if (EXPRESSION) then
          null:
        else
          SYNTAX_ERROR("Discriminant specification");
          end if:
        end if; -- if bypass(token_assignment)
      else
        SYNTAX_ERROR("Discriminant specification");
      end if:
    else
      SYNTAX_ERROR("Discriminant specification");
    end if:
  else
    SYNTAX_ERROR("Discriminant specification");
  end if:
else
  SYNTAX_ERROR("Discriminant specification");
else
  SYNTAX_ERROR("Discriminant specification");
else
  return (FALSE);
  end if; -- if identifier_list statement
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; -- if array type definition
  elsif (BYPASS(TOKEN RECORD STRUCTURE)) then
    if (RECORD TYPE DEFINITION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Type definition");
    end if; -- 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; -- if subtype indication
  else
    return (FALSE);
  end if;
end TYPE DEFINITION:

-- RECORD TYPE DEFINITION --> COMPONENT LIST end 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; -- if bypass(token record-structure)
    else
      SYNTAX_ERROR("Record type definition");
    end if; -- if bypass(token end)
  else
    return (FALSE);
  end if;
end RECORD TYPE DEFINITION:
end if. -- if component list statement
end RECORD TYPE DEFINITION.

-- COMPONENT LIST -- COMPONENT DECLARATION * VAARIANT 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.
end if.
return (TRUE).
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.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
else
SYNTAX_ERROR("Component declaration").
end if.
end if. -- if bypass(token colon)
else
  return (FALSE).
end if. -- if identifier list statement
end COMPONENT DECLARATION:

-- VARIANT\ PART -- -> case identifier is \VAR -- end case :
function VARIANT\ PART return boolean is
begin
  if (BYPASS(TOKEN CASE)) then
    if (BYPASS(TOKEN IDENTIFIER)) then
      if (BYPASS(TOKEN IS)) then
        if (VARIANT) then
          while (VARIANT) loop
            null;
          end loop:
        end if:
        if (BYPASS(TOKEN END)) then
          if (BYPASS(TOKEN CASE)) then
            if (BYPASS(TOKEN SEMICOLON)) then
              return (TRUE);
            else
              SYNTAX\ ERROR("Variant part");
            end if:
          end if;
        else
          SYNTAX\ ERROR("Variant part");
        end if:
      end if:
    else
      SYNTAX\ ERROR("Variant part");
    end if:
  end if:
else
  SYNTAX\ ERROR("Variant part");
end if:
end VARIANT\ PART;

-- VARIANT\ -- \ when \CHOICE \ CHOICE \* -> COMPONENT\ LIST
function VARIANT return boolean is
begin
  if (BYPASS(TOKEN WHEN)) then
    if (CHOICE) then
while (BYPASS(TOKEN BAR)) loop
    if not (CHOICE) then
        SYNTAX ERROR("Variant");
        end if; -- if not choice statement
    end loop;
    if (BYPASS(TOKEN_ARROW)) then
        if (COMPONENT_LIST) then
            return (TRUE);
        else
            SYNTAX ERROR("Variant");
            end if; -- if component_list statement
        else
            SYNTAX ERROR("Variant");
            end if; -- if bypass(token_arrow)
        else
            return (FALSE);
            end if; -- if bypass(token_when)
    end if:
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 COMMA)) loop
            if not (BYPASS(TOKEN IDENTIFIER)) then
                SYNTAX ERROR("With or use clause");
            end if:
        end loop:
        if (BYPASS(TOKEN SEMICOLON)) then
            return (TRUE);
        else
            SYNTAX ERROR("With or use clause");
            end if: -- if bypass(token_semicolon)
        else
            return (FALSE);
            end if: -- if bypass(token_identifier)
    end if:
end WITH OR USE_CLAUSE.

-- FORMAL PART --> (PARAMETER SPECIFICATION : PARAMETER SPECIFICATION * ) :
function FORMAL PART return boolean is
begin
    if (BYPASS(TOKEN LEFT_PAREN)) then
        FORMAL PARAM DECLARE TRUE;
        if (PARAMETER_SPECIFICATION) then
            while (BYPASS(TOKEN SEMICOLON)) loop

147
if not (PARAMETER_SPECIFICATION) then
SYNTAX_ERROR("Formal part");
end if; -- if not parameter specification statement
end loop;
if (BYPASS(TOKEN_RIGHT_PAREN)) then
if PACKAGE_BODY_DECLARE then
WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PARAM_DECLARE,
NONE, NEXTHEN);
CREATE_NODE(NEXTHEN, LAST_RECORDER);
end if;
FORMAL_PARAM_DECLARE := FALSE;
return (TRUE);
else
SYNTAX_ERROR("Formal part");
end if; -- if bypass(token_right_paren) statement
else
SYNTAX_ERROR("Formal part");
end if: -- if parameter specification statement
else
return (FALSE);
end if: -- if bypass(token_left_paren) statement
end FORMAL_PART:

-- IDENTIFIER DECLARATION --> IDENTIFIER_LIST IDENTIFIER DECLARATION TAIL
function IDENTIFIER_DECLARATION return boolean is
begin
put(RESULT_FILE, "IN IDENTIFIER DECLARATION"); NEW_LINE(RESULT_FILE);
HENRY_WRITE_ENABLE := TRUE;
IDENT_DECLARE := TRUE;
if (IDENTIFIER_LIST) then
if (BYPASS(TOKEN_COLON)) then
if (IDENTIFIER_DECLARATION_TAIL) then
HENRY_WRITE_ENABLE := FALSE;

return (TRUE);
else
SYNTAX_ERROR("Identifier declaration");
end if: -- if identifier list statement
else
SYNTAX_ERROR("Identifier declaration");
end if: -- if bypass(token Colon)
else
return (FALSE);
end if: -- if identifier list statement
end IDENTIFIER_DECLARATION;

-- IDENTIFIER DECLARATION TAIL --> exception EXCEPTION TAIL
-- constant CONSTANT_TERM

148
function IDENTIFIER DECLARATION TAIL return boolean is
begin
put(RESULT_FILE. "IN IDENTIFIER DECLARATION TAIL"). NEW LINE(RESULT_FILE):
if (BYPASS(TOKEN EXCEPTION)) then
  if (EXCEPTION TAIL) then
    return (TRUE):
  else
    SYNTAX_ERROR("Identifier declaration tail");
  end if:
else if (BYPASS(TOKEN CONSTANT)) then
  if (CONSTANT TERM) then
    return (TRUE):
  else
    SYNTAX_ERROR("Identifier declaration tail");
  end if:
else if (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:
    end if:
  end if:
else
  SYNTAX_ERROR("Identifier declaration tail");
end if:
else if (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

149
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:
    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_TYPE_DEFINITION) then
            if (BYPASS(TOKEN_ASSIGNMENT)) then
                if (EXPRESSION) then
                    null;
                else
                    SYNTAX_ERROR("Constant term");
                end if:
            else
                SYNTAX_ERROR("Constant term");
            end if:
        else
            SYNTAX_ERROR("Constant term");
        end if:
    else
        SYNTAX_ERROR("Constant term");
    end if:
if (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:
end if:
else
SYNTAX_ERROR("Constant term");
end if:
elsif (NAME) then
if (IDENTIFIER_TAIL) then
return (TRUE);
else
SYNTAX_ERROR("Constant term");
end if:
else
return (FALSE);
end if:
end CONSTANT_TERM:

IDENTIFIER_TAIL

constraint? := EXPRESSION? ;

renames NAME? ;

function IDENTIFIER_TAIL return boolean is
begin
put(RESULT_FILE, "IN IDENTIFIER TAIL"); NEW_LINE(RESULT_FILE):
if (CONSTRAINT) then
null;
end if:
if (BYPASS(TOKEN_RENAMES)) then
if (NAME) then
null:
else
SYNTAX_ERROR("Identifier tail");
end if:
end if:
else
SYNTAX_ERROR("Identifier tail");
end if:
if (BYPASS(TOKEN_ASSIGNMENT)) then
if (EXPRESSION) then
null;
else
SYNTAX_ERROR("Identifier tail");
end if:
end if:
else
SYNTAX_ERROR("Identifier tail");
end if:
else
return (TRUE);
end if:
end IDENTIFIER_TAIL;
return (FALSE);
end if:
-- if bypass(token_semicolon)
end IDENTIFIER_TAIL:

-- PARAMETER SPECIFICATION --> IDENTIFIER_LIST: MODE NAME := EXPRESSION ?;
function PARAMETER_SPECIFICATION return boolean is
begin
put(RESULT_FILE, "IN PARAMETER SPECIFICATION"), NEW_LINE(RESULT_FILE);
HENRY_WRITE_ENABLE := TRUE; -- to capture first parameter
if (IDENTIFIER_LIST) then
  if (BYPASS(TOKEN_COLON)) 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:
      else
        SYNTAX_ERROR("Parameter specification");
      end if:
    else
      SYNTAX_ERROR("Parameter specification");
    end if:
  else
    SYNTAX_ERROR("Parameter specification");
  end if:
else
  SYNTAX_ERROR("Parameter specification");
end if:
-- if bypass(token_colon)
else
  return (FALSE);
end if:
-- if identifier_list statement
end PARAMETER_SPECIFICATION:

-- IDENTIFIER_LIST --> identifier . identifier *
function IDENTIFIER_LIST return boolean is
begin
put(RESULT_FILE, "IN IDENTIFIER LIST"), NEW_LINE(RESULT_FILE);
if (BYPASS(TOKEN_IDENTIFIER)) then
  if (FORMAL PARAM_DECLARE AND PACKAGE BODY DECLARE) then
    WRITE HENRY_DATA(LOCAL DECLARE, DUMMY LEXEME, IDENT TYPE, LOCAL, IDENT LAST_RECORD);
  elsif NOT PACKAGE BODY DECLARE then
    WRITE HENRY_DATA(LOCAL DECLARE, DUMMY LEXEME, IDENT TYPE, LOCAL, IDENT LAST_RECORD);
  end if:
while (BYPASS(TOKEN_COMMA)) loop
  if (IDENT DECLARE) OR (FORMAL PARAM_DECLARE AND PACKAGE BODY DECLARE)
end if:
while (BYPASS(TOKEN_COMMA)) loop
  if (IDENT DECLARE) OR (FORMAL PARAM_DECLARE AND PACKAGE BODY DECLARE)
then
HENRY_WRITE_ENABLE := TRUE;
end if:
if FORMA: PARAM DECLARE AND PACKAGE BODY DECLARE then
WRITE_HENRY_DATA(BLANK, DUMMYLEXEME, PARAM_TYPE,
NONE, NEXT_HEN);
elsif (NOT FORMAL PARAM DECLARE) then
WRITE_HENRY_DATA(LOCAL DECLARE, DUMMY_lexeme, IDENT_TYPE,
NONE, NEXT_HEN);
end if:
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
put(Result_FILE, "IN PARAMETER MODE"); NEW_LINE(Result_FILE);
if (BYPASS(TOKEN IN)) then
if PACKAGE BODY DECLARE then
WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, IN_TYPE, LAST_RECORD
end if:
if (BYPASS(TOKEN OUT)) then
if PACKAGE BODY DECLARE then
WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE,
IN_OUT_TYPE, LAST_RECORD);
end if:
end if:
else if (BYPASS(TOKEN OUT)) then
if PACKAGE BODY DECLARE then
WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE,
OUT_TYPE, LAST_RECORD);
end if:
end if:
if (LAST_RECORD.TYPE_DEFINE = PARAM_TYPE)
AND (LAST_RECORD.PARAM_TYPE = NONE) THEN
WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, IN_TYPE, LAST_RECORD,
end if:
return (TRUE);
end MODE:

---------------------------------------------
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;

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;
    elsif (ASSIGNMENT OR PROCEDURE_CALL) then -- includes a check for a
      return (TRUE); -- code statement and an
    elsif (BYPASS(TOKEN_EXIT)) then
      if (EXIT_STATEMENT) then
        return (TRUE);
      else
        SYNTAX_ERROR("Simple statement");
      end if;
    elsif (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;
    else
      SYNTAX_ERROR("Simple statement");
    end if;
  end SIMPLE_STATEMENT;
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

ASSIGN_Pointer, FUNCALL_Pointer : POINTER;

begin
  put(result_file, "in assign or procedure call"); new_line(result_file);
  HENRY_WRITE_ENABLE := TRUE;
  ASSIGN_Pointer := NEXT_HEN;
  if (NAME) then
    if (BYPASS(TOKEN_ASSIGNMENT)) then
      ASSIGN_STATEMENT := TRUE;
    write HENRY_DATA(BLANK, DUMMY_LEXEME, ASSIGN_TYPE, NONE, NEXT_HEN);
    create_node(NEXT_HEN, LAST_RECORD);
    if NAME TAIL_SET then
      write HENRY_DATA(BLANK, DUMMY_LEXEME, PROCALL_OR_DS, NONE, ASSIGN_Pointer);
    end if:
    FUNCALL_Pointer := NEXT_HEN;
    HENRY_WRITE_ENABLE := TRUE;
    if (EXPRESSION) then
      if (BYPASS(TOKEN_SEMICOLON)) then
        NAME_TAIL_SET := FALSE;
        ASSIGN_STATEMENT := FALSE;
        write HENRY_DATA(BLANK, DUMMY_LEXEME, END_ASSIGN_TYPE.
NONE, NEXT_HEN;
CREATE_NODE(NEXT_HEN, LAST_RECORD);
HENRY_WRITE_ENABLE := FALSE;
return (TRUE); -- parsed an assignment statement
else
SYNTAX_ERROR("Assignment or procedure call");
end if;
else
SYNTAX_ERROR("Assignment or procedure call");
end if;
else
SYNTAX_ERROR("Assignment or procedure call");
end if;
eendif;
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
return (TRUE);
end if;
else
SYNTAX_ERROR("Label");
end if;
else
SYNTAX_ERROR("Label");
end if;
else
return (FALSE);
end if;
end LABEL;

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

-- ENTRY DECLARATION --> entry identifier (DISCRETE_RANGE) ?
-- FORMAL PART ? :
function ENTRY_DECLARATION return boolean is
begin
if (BYPASS(TOKEN_ENTRY)) then
if (BYPASS(TOKEN_IDENTIFIER)) then
if (BYPASS(TOKEN_LEFT_PAREN)) then

156
if (DISCRETE_RANGE) then
  if (BYPASS(TOKEN_RIGHT_PAREN)) then
    null;
  else
    SYNTAX_ERROR("Entry declaration");
  end if;
else
  SYNTAX_ERROR("Entry declaration");
end if;

if (FORMAL_PART) then
  null;
end if:
else
  SYNTAX_ERROR("Entry declaration");
end if;

if (BYPASS(TOKENSEMICOLON)) then
  return (TRUE);
else
  SYNTAX_ERROR("Entry declaration");
end if:

else
  return (FALSE);
end if:

end ENTRY_DECLARATION;

-- REPRESENTATION_CLAUSE --> for NAME use record RECORD_REPRESENTATION_CLAUSE
--> for NAME use at ? SIMPLE_EXPRESSION;

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;
        else
          SYNTAX_ERROR("Representation clause");
        end if;
      else
        SYNTAX_ERROR("Representation clause");
      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
      SYNTAX_ERROR("Representation clause");
    end if;
  else
    SYNTAX_ERROR("Representation clause");
  end if:
else
  SYNTAX_ERROR("Representation clause");
end if:
elif (SIMPLE_EXPRESSION) then
else if (BYPASS(TOKEN_SEMICOLON)) then
  return (TRUE);
else
  SYNTAX_ERROR("Representation clause");
end if:
elseif (SIMPLE_EXPRESSION) then
else
  SYNTAX_ERROR("Representation clause");
end if:
ego
if (BYPASS(TOKEN SEMICOLON)) then
    return (TRUE);
else
    SYNTAX_ERROR("Representation clause");
end if: -- if bypass(token_semicolon)
else
    SYNTAX_ERROR("Representation clause");
end if: -- if bypass(token_record)
else
    SYNTAX_ERROR("Representation clause");
end if: -- if bypass(token_use)
else
    return (FALSE);
end if: -- if bypass(token_for)
end REPRESENTATION_CLAUSE:

-- RECORD_REPRESENTATION_CLAUSE --> \mathbf{\texttt{at mod SIMPLE_EXPRESSION ?}}
-- \mathbf{\texttt{NAME at SIMPLE_EXPRESSION range RANGES \texttt{.}}}
--
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: -- if simple_expression
    else
        SYNTAX_ERROR("Record representation clause");
end if: -- if bypass(token_mod)
end if: -- if bypass(token_at)
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: -- if ranges statement
    else
        SYNTAX_ERROR("Record representation clause");
    end if: -- if bypass(token_range)
else
    SYNTAX_ERROR("Record representation clause");
end if: -- if simple_expression
end if:
end record;
else
  SYNTAX_ERROR("Record representation clause").
end if. -- if bypass(token_at)
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: -- if bypass(token_semicolon)
  else
    SYNTAX_ERROR("Record representation clause").
  end if:
else
  return (FALSE);
end if: -- if bypass(token_end)
end RECORD_REPRESENTATION_CLAUSE;
end PARSER_2;

---

-- TITLE: AN ADA SOFTWARE METRIC
--
-- MODULE NAME: PACKAGE PARSER_3
-- DATE CREATED: 22 JUL 86
-- LAST MODIFIED: 30 MAY 87
--
-- AUTHORS: LCDR JEFFREY L. NIEDER
-- LT KARL S. FAIRBANKS, JR.
-- LCDR PAUL M. HERZIG
-- 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, HENRY GLOBAL, HENRY, BYPASS_FUNCTION, HALSTEAD METRIC,
GLOBAL_PARSER, GLOBAL, TEXT IO;
use PARSER_4, HENRY GLOBAL, HENRY, BYPASS_FUNCTION, HALSTEAD METRIC,
GLOBAL_PARSER, GLOBAL, TEXT IO;

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;
  
end PARSER_3;
package body PARSER_3 is

-- SUBTYPE INDICATION --> NAME CONSTRAINT?
function SUBTYPE_INDICATION return boolean is begin
    if (NAME) then; -- check for type mark
        if (CONSTRAINT) then
            null;
        end if;
        return (TRUE);
    else
        return (FALSE);
    end 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:
      -- if subtype indication
    else
      SYNTAX_ERROR("Array definition");
    end if:
    -- if bypass(token_of)
  else
    SYNTAX_ERROR("Array definition");
  end if:
  -- if index_constraint statement
else
  return (FALSE);
end if:
-- if bypass(token_left_paren)
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:
      -- if simple_expression statement
    elsif (CONSTRAINT) then
      null:
    end if:
    -- if bypass token_range_dots
    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

end
NESTING_METRIC(WHILE_CONSTRUCT):
if (EXPRESSION) then
  return (TRUE);
else
  SYNTAX_ERROR("Iteration scheme");
end if;
elsif (BYPASS(TOKEN FOR)) then
  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 bypass(token_reverse)
    if (DISCRETE_RANGE) then
      return (TRUE);
    else
      SYNTAX_ERROR("Loop parameter specification");
    end if;
    -- if discrete_range statement
  else
    SYNTAX_ERROR("Loop parameter specification");
  end if;
  -- if bypass(token_in)
else
  return (FALSE);
end if;
-- if bypass(token_identifier)
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
  return (TRUE);
else

162
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
      return (TRUE):
    else
      return (FALSE):
    end if:
  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: -- if bypass(token then)
    end loop:
  while (BYPASS(TOKEN OR)) loop
    if (BYPASS(TOKEN ELSE)) then
      null:
      end if: -- if bypass(token else)
    end loop:
  while (BYPASS(TOKEN XOR)) loop
    if not (RELATION) then
      SYNTAX_ERROR("Relation tail");
      end if: -- if not relation statement
    end loop:
  return (TRUE);
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 (TERM) 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 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;
else
SYNTAX_ERROR("Simple expression tail");
end if;
ellif (BYPASS(TOKEN_NOT)) then
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;
else
SYNTAX_ERROR("Simple expression tail");
end if;
end if;
end if;
end if;
end SIMPLE_EXPRESSION_TAIL;

164
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:
  -- if ranges statement
else
  return (FALSE):
end if:
-- if relational_operator statement
end SIMPLE_EXPRESSION_TAIL:

-- TERM --> FACTOR MULTIPLYING_OPERATOR FACTOR;*
function TERM return boolean is
begin
  if (FACTOR) then
    while (MULTIPLYING_OPERATOR) loop
      if not (FACTOR) then
        SYNTAX_ERROR("Term");
      end if:
      -- if not factor statement
    end loop:
    return (TRUE):
  else
    return (FALSE):
  end if:
  -- if factor statement
end TERM:

-- FACTOR --> PRIMARY ** PRIMARY ?
-- --> abs PRIMARY
-- --> not PRIMARY
function FACTOR return boolean is
begin
  if (PRIMARY) then
    if (BYPASS(TOKEN_EXPONENT)) then
      if (PRIMARY) then
        null:
      else
        SYNTAX_ERROR("Factor");
      end if:
      -- if primary statement
    end if:
    -- if bypass(token_exponent) statement
    return (TRUE):
  elsif (BYPASS(TOKEN_ABSOLUTE)) then
    if (PRIMARY) then
      return (TRUE):
    else
      SYNTAX_ERROR("Factor");
    end if:
    -- if primary(abs) statement
  else

elsif (BYPASS(TOKEN NOT)) then
    if (PRIMARY) then
        return (TRUE);
    else
        SYNTAX ERROR("Factor");
    end if;
else
    return (FALSE);
end if:
end FACTOR;

-- PRIMARY --> numeric literal
-- --> null
-- --> string literal
-- --> new ALLOCATOR
-- --> NAME
-- --> AGGREGATE

function PRIMARY return boolean is
begin
    HENRY WRITE ENABLE - TRUE:
    if (BYPASS(TOKEN NUMERIC_LITERAL)) then
        WRITE HENRY DATA(BLANK, DUMMY LEXEME, IDENT_TYPE, NONE, LAST_RECORD);
        return (TRUE);
    elsif (BYPASS(TOKEN NULL)) then
        return (TRUE);
    elsif (BYPASS(TOKEN STRING_LITERAL)) then
        WRITE HENRY DATA(BLANK, DUMMY LEXEME, IDENT_TYPE, NONE, LAST_RECORD);
        return (TRUE);
    elsif (BYPASS(TOKEN NEW)) then
        if (ALLOCATOR) 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);
      else
        if (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:
      end if:
    end if:
  end if:
  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 COMMA)) 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; -- if bypass(token_right Paren)
else
    return (FALSE);
end if; -- if discrete_range statement
end INDEXCONSTRAINT;

-- RANGES -- · SIMPLE EXPRESSION SIMPLE EXPRESSION?
function RANGES return boolean is
begin
    if (SIMPLE EXPRESSION) then
        if (BYPASS(TOKEN RANGE DOTS)) then
            if (SIMPLE EXPRESSION) then
                null:
            else
                SYNTAX_ERROR("Ranges");
            end if; -- if simple_expression statement
        end if:
        if (BYPASS(TOKEN RANGE DOTS)) then
            end if; -- if bypass(token range dots)
        else
            return (TRUE); -- if simple_expression statement
        end if:
    else
        return (FALSE);
end if:
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 COMMA)) loop
                if not (COMPONENT ASSOCIATION) then
                    SYNTAX_ERROR("Aggregate");
                end if; -- if not component association
            end loop:
        end if:
        if (BYPASS(TOKEN RIGHT_PAREN)) then
            return (TRUE);
        else
            SYNTAX_ERROR("Aggregate");
        end if; -- if bypass(token_right Paren)
    else
        return (TRUE);
    end if:

    -- INDEXCONSTRAINT:

    -- RANGES -- · SIMPLE EXPRESSION SIMPLE EXPRESSION?
    function RANGES return boolean is
    begin
        if (SIMPLE EXPRESSION) then
            if (BYPASS(TOKEN RANGE DOTS)) then
                if (SIMPLE EXPRESSION) then
                    null:
                else
                    SYNTAX_ERROR("Ranges");
                end if; -- if simple_expression statement
            end if:
            if (BYPASS(TOKEN RANGE DOTS)) then
                end if; -- if bypass(token range dots)
            else
                return (TRUE); -- if simple_expression statement
            end if:
        else
            return (FALSE);
        end if:
    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 COMMA)) loop
                    if not (COMPONENT ASSOCIATION) then
                        SYNTAX_ERROR("Aggregate");
                    end if; -- if not component association
                end loop:
            end if:
            if (BYPASS(TOKEN RIGHT_PAREN)) then
                return (TRUE);
            else
                SYNTAX_ERROR("Aggregate");
            end if; -- if bypass(token_right Paren)
        else
            return (TRUE);
        end if:

    -- INDEXCONSTRAINT:

    -- RANGES -- · SIMPLE EXPRESSION SIMPLE EXPRESSION?
    function RANGES return boolean is
    begin
        if (SIMPLE EXPRESSION) then
            if (BYPASS(TOKEN RANGE DOTS)) then
                if (SIMPLE EXPRESSION) then
                    null:
                else
                    SYNTAX_ERROR("Ranges");
                end if; -- if simple_expression statement
            end if:
            if (BYPASS(TOKEN RANGE DOTS)) then
                end if; -- if bypass(token range dots)
            else
                return (TRUE); -- if simple_expression statement
            end if:
        else
            return (FALSE);
        end if:
    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 COMMA)) loop
                    if not (COMPONENT ASSOCIATION) then
                        SYNTAX_ERROR("Aggregate");
                    end if; -- if not component association
                end loop:
            end if:
            if (BYPASS(TOKEN RIGHT_PAREN)) then
                return (TRUE);
            else
                SYNTAX_ERROR("Aggregate");
            end if; -- if bypass(token_right Paren)
        else
            return (TRUE);
        end if:

    -- INDEXCONSTRAINT:

    -- RANGES -- · SIMPLE EXPRESSION SIMPLE EXPRESSION?
    function RANGES return boolean is
    begin
        if (SIMPLE EXPRESSION) then
            if (BYPASS(TOKEN RANGE DOTS)) then
                if (SIMPLE EXPRESSION) then
                    null:
                else
                    SYNTAX_ERROR("Ranges");
                end if; -- if simple_expression statement
            end if:
            if (BYPASS(TOKEN RANGE DOTS)) then
                end if; -- if bypass(token range dots)
            else
                return (TRUE); -- if simple_expression statement
            end if:
        else
            return (FALSE);
        end if:
    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 COMMA)) loop
                    if not (COMPONENT ASSOCIATION) then
                        SYNTAX_ERROR("Aggregate");
                    end if; -- if not component association
                end loop:
            end if:
            if (BYPASS(TOKEN RIGHT_PAREN)) then
                return (TRUE);
            else
                SYNTAX_ERROR("Aggregate");
            end if; -- if bypass(token_right Paren)
        else
            return (TRUE);
        end if:
SYNTAX ERROR("Aggregate");
end if: -- if component association statement
else
  return (FALSE): -- if bypass(token left paren)
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 association");
      end if:
      end loop:
    if (BYPASS(TOKEN ARROW)) then
      if (EXPRESSION) then
        null:
      else
        SYNTAX ERROR("Component association");
      end if:
    end if: -- if expression statement
  end if: -- if bypass(token arrow)
  return (TRUE);
else
  return (FALSE): -- if choice statement
end if: -- if bypass(token arrow)
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:
    end if: -- if aggregate statement
  end if: -- if bypass(token apostrophe)
  return (TRUE);
else
  return (FALSE): -- if aggregate statement
end if: -- if bypass(token apostrophe)
end ALLOCATOR:
begin
put(result_file, "in name"); new line(result_file);
if (BYPASS(TOKEN IDENTIFIER)) then
     NAME POINTER := LAST RECORD;
if (NAME TAIL) then
     null;
end if;
return (TRUE);
HENRY WRITE ENABLE := TRUE;
elif (BYPASS(TOKEN CHARACTER_LITERAL)) then
     if (NAME TAIL) then
         null;
     end if;
     return (TRUE);
elif (BYPASS(TOKEN STRING_LITERAL)) then
     if (NAME TAIL) then
         null;
     end if;
     return (TRUE);
else
     return (FALSE);
end if:
end NAME:

begin
put(result_file, "in name tail"); new line(result_file);
if (BYPASS(TOKEN LEFT PAREN)) then
     NAME TAIL SET := TRUE;
HENRY WRITE ENABLE := TRUE;
if ASSIGN STATEMENT then
     WRITE HENRY DATA(BLANK, DUMMY LEXEME, FUNCALL OR DS,
                        NONE, NAME POINTER);
else
     WRITE HENRY DATA(BLANK, DUMMY LEXEME, PROCALL OR DS,
                        NONE, NAME POINTER);
end if:
if (LEFT PAREN NAME TAIL) then
     return (TRUE);
else
     return (FALSE);
end if: -- if left paren name tail
elsif (BYPASS(TOKEN PERIOD)) then
  if (SELECTOR) then
    while (NAME TAIL) loop
      null:
      end loop:
      return (TRUE):
  else
    SYNTAX_ERROR("Name tail");
  end if:
elsif (BYPASS(TOKEN APOSTROPHE)) then
  if (AGGREGATE) then
    while (NAMETAIL) loop
      null:
      end loop:
      return (TRUE):
  elsif (ATTRIBUTEDESIGNATOR) then
    while (NAME TAIL) loop
      null:
      end loop:
      return (TRUE):
  else
    SYNTAX_ERROR("Name tail");
  end if:
else
  return (FALSE);
end if:
-- if bypass(token left paren)
end NAME TAIL:

function LEFT PAREN NAME TAIL return boolean is
  begin
    put(result file, "in left paren name tail"): new line(result file):
    if (FORMAL PARAMETER) then -- check for optional formal parameter
      null:
      HENRY WRITE ENABLE := TRUE;
      if (EXPRESSION) then
        if NAME TAIL SET then
          WRITE HENRY DATA(BLANK, DUMMY LEXEME, PARAM TYPE, ACTUAL PARAM. 
           LAST_RECORD);
        end if:
      end if:
    end if: if (BYPASS(TOKEN RANGE DOTS)) then
    if not (EXPRESSION) then
      SYNTAX ERROR("Left paren name tail");
    end if:
    if bypass(token_range_dots)
      while (BYPASS(TOKEN COMMA)) loop
        171
if (FORMAL_PARAMETER) then
  null:
end if; -- if formal parameter statement

HENRY_WRITE_ENABLE := TRUE;
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)
if NAME_TAIL_SET then
  WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, ACTUAL_PARAM,
                   LAST_RECORD);
end if;
end loop;
if (BYPASS(TOKEN_RIGHT_PAREN)) then
  WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_ACTUAL_PARAM,
                   ACTUAL_PARAM, NEXT_HEN);
CREATE_NODE(NEXT_HEN, LAST_RECORD);
NAME_TAIL_SET := FALSE:
while (NAME_TAIL) loop
  null:
end loop;
return (TRUE);
else
  return (FALSE); -- if bypass(token_right_paren)
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;
else
  return (FALSE); -- if bypass(token_right_paren)
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_right_paren) statement
else
  SYNTAX_ERROR("Attribute designator");
end if:

if expression statement
else
  SYNTAX_ERROR("Attribute designator");
end if:

if bypass(token_left_paren) statement
return (TRUE);
else
  return (FALSE);
end if:

if bypass(token_identifier) statement
end ATTRIBUTE DESIGNATOR:

-- 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

  173
if (CONSTRAINT) then
  null:
end if:
return (TRUE);
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 name statement
  if (BYPASS(TOKEN WHEN)) then
    if (EXPRESSION) then
      null:
    else
      SYNTAX ERROR("Exit statement");
    end if:
  end if:
  -- if expression statement
  if (BYPASS(TOKEN SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if:
  -- if bypass(token semicolon)
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

175
if (NAME) then
  null;
end if;
if (BYPASS(TOKEN SEMICOLON)) then
  return (TRUE);
else
  return (FALSE);
end if;
end RAISE STATEMENT;
end PARSER 3;

---

**TITLE:** AN ADA SOFTWARE METRIC

**MODULE NAME:** PACKAGE PARSER 4

**DATE CREATED:** 23 JUL 86

**LAST MODIFIED:** 30 MAY 87

**AUTHORS:** LCDR JEFFREY L. NIEDER

LT KARL S. FAIRBANKS, JR.

LCDR PAUL M. HERZIG

**DESCRIPTION:** This package contains seven functions that
are the lowest level productions for our top-down,
recursive descent parser. Each function is preceded
by the grammar productions they are implementing

---

with BYPASS FUNCTION, BYPASS SUPPORT FUNCTIONS, GLOBAL PARSER, GLOBAL TEXT
use BYPASS FUNCTION, BYPASS SUPPORT FUNCTIONS, GLOBAL PARSER, GLOBAL TEXT

package PARSER 4 is
  function MULTIPLYING OPERATOR return boolean;
  function BINARY ADDING OPERATOR return boolean;
  function RELATIONAL OPERATOR return boolean;
  function ENUMERATION TYPE DEFINITION return boolean;
  function ENUMERATION LITERAL return boolean;
  function FORMAL PARAMETER return boolean;
  function SELECTOR return boolean;
end PARSER 4;

package body PARSER 4 is

---

176
MULTIPLYING OPERATOR

function MULTIPLYING OPERATOR return boolean is
begin
put(RESULT FILE, "In multiplying operator "); new line(RESULT FILE);
if (BYPASS(TOKEN Asterisk)) then
return (TRUE);
elsif (BYPASS(TOKEN Slash)) then
return (TRUE);
elsif (BYPASS(TOKEN Mod)) 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
put(RESULT FILE, "In binary-adding operator "); new line(RESULT FILE);
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;

RELATIONAL OPERATOR

function RELATIONAL OPERATOR return boolean is
begin
put(RESULT FILE, "In relational operator "); new line(RESULT FILE);
if (BYPASS(TOKEN Equals)) then
return (TRUE).
end if;
end RELATIONAL OPERATOR;
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
      . ENUMERATION LITERAL *)
function ENUMERATION TYPE DEFINITION return boolean is
begin
    put(RESULT FILE. "In enumeration type definition "). new _line(RESULT FILE).
    if (BYPASS(TOKEN LEFT PAREN)) then
        HENRY_WRITE_ENABLE := TRUE:
        if (ENUMERATION LITERAL) then
            while (BYPASS(TOKEN COMMA)) loop
                HENRY_WRITE_ENABLE := TRUE:
                if not (ENUMERATION LITERAL) then
                    SYNTAX_ERROR("Enumeration type definition");
                end if:
            end loop;
        end if:
    end if:
    if (BYPASS(TOKEN RIGHT PAREN)) then
        return (TRUE);
    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
    put(RESULT FILE. "In enumeration literal "). new _line(RESULT FILE):
    if (BYPASS(TOKEN IDENTIFIER)) then
return (TRUE);
elsif (BYPASS(TOKEN CHARACTER_LITERAL)) then
  return (TRUE);
else
  return (FALSE);
end if;
end ENUMERATION_LITERAL;

-- FORMAL_PARAMETER -> identifier ->
function FORMAL_PARAMETER return boolean is
begin
  put(RESULT_FILE, "In formal parameter "); new line(RESULT_FILE);
  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
      return (TRUE):
    else
      SYNTAX_ERROR ("Formal parameter");
    end if:
  else
    SYNTAX_ERROR ("Formal parameter");
  end if:
  -- if bypass(token_arrow)
  return (FALSE);
end FORMAL_PARAMETER;

-- SELECTOR -> identifier
  -- character_literal
  -- string_literal
  -- all
function SELECTOR return boolean is
begin
  put(RESULT_FILE, "In selector "); new line(RESULT_FILE);
  if (BYPASS(TOKEN IDENTIFIER)) then
    return (TRUE);
  elsif (BYPASS(TOKEN CHARACTER_LITERAL)) then
    return (TRUE);
  elsif (BYPASS(TOKEN STRING_LITERAL)) then
    return (TRUE);
  end if:
end SELECTOR;
elsif (BYPASS(TOKEN, ALL)) then
   return (TRUE);
else
   return (FALSE);
end if;
end SELECTOR:

end PARSER 4:
LIST OF REFERENCES


<table>
<thead>
<tr>
<th>No.</th>
<th>Initial Distribution List</th>
</tr>
</thead>
</table>
| 1.  | Defense Technical Information Center  
Cameron Station  
Alexandria, Virginia  22304-6145 | 2 |
| 2.  | Library, Code 0142  
Naval Postgraduate School  
Monterey, California  93943-5002 | 2 |
| 3.  | Dr. Daniel L. Davis Code 52DV  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California  93943 | 2 |
| 4.  | Dr. Bruce J. MacClennan Code 52ML  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California  93943 | 1 |
| 5.  | Lieutenant Commander Paul Herzig  
578-C Schubrick Rd  
Monterey California  93940 | 1 |
| 6.  | Chairman (Code 52)  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California  93943 | 1 |
| 7.  | Computer Technology Curricular Officer (Code 37)  
Naval Postgraduate School  
Monterey, California  93943 | 1 |
| 8.  | Chief of Naval Operations  
Director, Information Systems (OP 945)  
Navy Department  
Washington, D.C.  20350-2000 | 1 |
9. Mr. Carl Hall  
Software Missile Branch. Code 3922  
Naval Weapons Center  
China Lake, California 93555-6001  

10. Mr. Robert Westbrook  
CMDR 31C  
Naval Weapons Center  
China Lake, California 93555-6001  

No. Copies  
1  1
END
10-87
DTIC