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 (1)

**Personal Author(s):** Herz, Paul W.

**Type of Report:** Master's Thesis

**Time Covered:** FROM ______ TO ______

**Date of Report:** 1987 June

**Source of Funding Numbers:**

<table>
<thead>
<tr>
<th>Program Element No.</th>
<th>Project No.</th>
<th>Task No.</th>
<th>Work Unit Accession No.</th>
</tr>
</thead>
</table>

**Subject Terms:** Ada Metric; Data Flow Complexity; Ada Measure; Halstead; Henry

A software metric is a tool that should be used in the development of quality software. The properties that define good software vary but generally include reliability, complexity, efficiency, testability, understandability, and modifiability. The Henry metric measures the complexity of the flow within a module and the complexity of inter-module communication. This report is an extension of a previous thesis titled 'AdaMeasure' that evaluated the Halstead metric. The present design and implementation is a tool that compares the Halstead and Henry metrics for Ada programs.
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
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C. THE HENRY ANALYSIS PACKAGE</td>
<td>35</td>
</tr>
<tr>
<td>D. THE HENRY DISPLAY PACKAGE</td>
<td>39</td>
</tr>
<tr>
<td>E. TESTING</td>
<td>41</td>
</tr>
<tr>
<td>IV. CONCLUSIONS</td>
<td>44</td>
</tr>
<tr>
<td>A. IMPLEMENTATION</td>
<td>44</td>
</tr>
<tr>
<td>B. THE FUTURE OF METRICS</td>
<td>45</td>
</tr>
<tr>
<td>APPENDIX A: INFORMATION FLOW MECHANISMS</td>
<td>47</td>
</tr>
<tr>
<td>APPENDIX B: HENRY METRIC CODE</td>
<td>52</td>
</tr>
<tr>
<td>APPENDIX C: MODIFIED PARSERS</td>
<td>90</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>181</td>
</tr>
<tr>
<td>INITIAL DISTRIBUTION LIST</td>
<td>182</td>
</tr>
</tbody>
</table>
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.

![Data Flow Structure](image_url)

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 $L \leftarrow R_1, R_2, R_3, ..., R_n$; where $L$ is the resultant from the application of the relationships $R_1, R_2, ..., R_n$. 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.

```pseudo
A()
begin
  
  D3 := D1 + D2 + 1;
end procedure A;
```

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.
Code

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

B(P,Q)
begin
D3 := P + Q;
end;
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.I, 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

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

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:

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

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

---

22
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 laid 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
importance 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 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 37
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 standpoint 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 an 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
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.

\[ L \rightarrow R_1, R_2 \ldots R_{count} \]

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 jth Output parameter.
4. \( P.j.I \) \( P \) is the procedure, \( j \) is an integer representing the jth parameter, and \( I \) is the jth 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 jth parameter and \( I \) is the jth input parameter.
4. \( S.j.O \) \( S \) is the procedure name, \( j \) is an integer representing the jth parameter in the list and \( O \) is the output parameter.
5. \( S.null \) \( S \) is the procedure name, null represents no data.
6. $S\text{ cons.}$ $S$ is the procedure name and constant a value used within $S$.

7. $S$ error $S$ is the procedure name and error represents an invalid flow of information through procedure $S$.

RULES

1. $L$ is of the form $P. DS$ 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 $P. O$ then
   This is used only in generating the relations from procedure $P$ that produce an output.

3. $L$ is of the form $P. j. O$ 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 $P. j. I$ 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 $S. DS$ then
   Procedure $S$ reads information from $DS$ this format is used to indicate a read only.

6. $R_i$ is of the form $S. O$ then
   Relations are generated that come from procedure $T$ that are return values to $T$ from $S$.

7. $R_i$ is of the form $S. j. I$ 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 $S. \text{cons.}$
   Then $S$ causes a constant number or string to flow to $L$.

9. $R_i$ is of the form $S. \text{Null}$ 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 $S. \text{error}$ 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 DS's the only assignment and reads allowed are from procedures or functions. The other not possibles stem from input parameters not flowing into DS'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 DS. 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></th>
<th>X.DS</th>
<th>X.O</th>
<th>X.K.I</th>
<th>X.K.O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y.DS</td>
<td>NP</td>
<td>NP</td>
<td>Y calls X</td>
<td>NP</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>Y.O</td>
<td>X calls Y</td>
<td>X calls Y</td>
<td>NC</td>
<td>X calls Y</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>Y.K.I</td>
<td>NP</td>
<td>NP</td>
<td>Y calls X</td>
<td>NP</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Y.K.O</td>
<td>X calls Y</td>
<td>X calls Y</td>
<td>NC</td>
<td>X calls Y</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</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>----------</td>
</tr>
<tr>
<td>Y.DS</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Y.O</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>Y.K.I</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>Y.K.O</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

50
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

---------------------------------------------
-- TITLE: AN ADA SOFTWARE METRIC
-- MODULE NAME: PACKAGE HENRY_GLOBAL
-- DATE CREATED: 09 MAY 87
-- LAST MODIFIED: 19 MAY 87
--
-- AUTHOR: LCDR PAUL M. HERZIG
--
-- DESCRIPTION: This package contains the data declarations
and basic procedures used throughout the Henry metric.
--
---------------------------------------------

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,
FUNCALL_OR_DS,

52
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 PARAMCLASS is (NONE, IN_TYPE, OUT_TYPE, IN_OUT_TYPE,
                   ACTUAL_PARAM);
subtype FORMAL_PARAM_CLASS is PARAMCLASS 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 := FALSE;
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 creates Henry record nodes for data storage

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

-- sets all of the variables to their initial values besides
-- creating the first Henry record and line count record
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;
HENRY_LINE := new HENRY_LINE_COUNT_RECORD;
HENRY_LINE.ID_NAME(I..SIZE) := HEAD_STRING;
HENRY_LINE.START_COUNT := DUMMY9s;
HENRY_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);

data HENRY:

package body HENRY is

-- produces the written data records from the parser inputs
-- data is only written if it is something other than the
-- null settings

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) IS

begin
    -- print file "write henry data" new line(result file)
    if ID := BLANK then
        LINK IDENTITY := ID

end HENRY;
case ID as
  when LOCAL DECLARE      put(RESULT FILE, "Local declare")
  when GLOBAL DECLARE     put(RESULT FILE, "Global declare")
  when others             put(RESULT FILE, "Undeclared")
end case
else put(RESULT FILE, "NO DECLARATION")
end if

new line(RESULT FILE)
if IN NAME(t) = NULL CHAR then
  LINK NOMEN(MAX LINE SIZE) IN NAME(MAX LINE SIZE).
  put(RESULT FILE IN NAME).
else put(RESULT FILE, "NO NAME")
end if.

new line(RESULT FILE)
if DEFINE UNDEFINED then
  LINK TYPE DEFINE DEFINE.
end if

new line(RESULT FILE)
end if

PARAM NONE 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
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;

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 IN NAME(1) NULL CHAR then
PTR.ID NAME(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 TRANSITIVITY IN(IN NAME LEXEME TYPE)
        BEGIN LOOP
            STOP LOOP.
        RETURN FLOAT.

    function TRANSITIVITY OUT(IN NAME LEXEME TYPE
        TOP in POINTER)
        RETURN FLOAT.

    procedure CALCULATE METRIC(HEAD in POINTER
        HEAD LINE in LINE POINTER).

60
end HENRY ANALYSIS.

package body HENRY ANALYSIS is

-- starts the process of setting up the raw Henry records into
-- decipherable data, it also counts the numbers of procedures
-- functions and fills in empty parameter type fields in the
-- Henry records

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.NEXT1;

-- move past package declarations

LOOP
   EXIT WHEN TOP.TYPE.DEFINE = END_PACKAGE_DECLARE;
   TOP := BOTTOM;
   BOTTOM := TOP.NEXT1;
END LOOP;

-- count the number of procedures functions

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.NEXT1;
end loop;
BOTTOM := TOP;

-- ensure 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.NEXT1;
   end loop;
end loop;

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.
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.
TEMP = TOP NEXT1.
TOP = TEMP.
END LOOP
else
TOP = TEMP.
BOTTOM  TEMP.
end if
TEMP = TOP NEXT1.
END LOOP

functions usually invoke the default in type parameter
convert this type if it is not defined

elsif TOP TYPE DEFINE  FUNCTION TYPE THEN
elsif TEMP TYPE DEFINE  PARAM TYPE THEN
LOOP
EXIT WHEN TEMP TYPE DEFINE  END PARAM DECLARE.
TEMP PARAM TYPE = IN TYPE
TEMP = BOTTOM NEXT1.
BOTTOM  TEMP.
END LOOP
end if
end if
TOP = BOTTOM NEXT1.
BOTTOM  TOP.
END LOOP FOR LOOP.
sets up the Henry data records to mark the beginning of each function or procedure. It also ties the procedure line length records to its proper procedure function.

```haskell
procedure SET UP HENRY ARRAY(HEAD : in POINTER, HEAD LINE : in LINE POINTER) is

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 NEXT1.
  BOTTOM : = TOP.
END LOOP.

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

FOR I in 1 Proc Func Count Loop
LOOP
  EXIT WHEN (TOP TYPE DEFINE END PROCEDURE TYPE) OR (TOP TYPE DEFINE FUNCTION TYPE).
  TOP : = BOTTOM NEXT1.
  BOTTOM : = TOP.
END LOOP.
HENRY ARRAY(I) NAME OF DATA(I MAX LINE SIZE)
  TOP Nomen(I MAX LINE SIZE).
HENRY ARRAY(I) BEGIN POINTER TOP.
LOOP
  EXIT WHEN (BOTTOM TYPE DEFINE END FUNCTION TYPE) OR (BOTTOM TYPE DEFINE END PROCEDURE CALL).
  TEMP : = BOTTOM NEXT1.
  BOTTOM : = TEMP.
END LOOP.

-- set up the array records to their related procedure function.

LOOP
  BOTTOM NEXT1.
  BOTTOM : = TOP.
HENRY ARRAY(I) LINE LENGTH POINTER WORK LINE.
WORK LINE TEMP LINE NEXT REC.
TEMP LINE WORK LINE.
```

63
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;
    I 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 SIZE POINTER NAME, LEXEME TYPE,
    NAME SIZE, POINTER SIZE INTEGER, MAX LINE SIZE,
    RESULT BOOLEAN := FALSE,
    TEMP, TEMP1, POINTER,
    I INTEGER := 1;

    begin

        put(HENRY_FILE, "in LOCAL NAME"); new line(HENRY_FILE);
        NAME, Sought 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[1..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);
                    I := I + 1;
                END LOOP
--if it is a variable name search first within the package
--declarations, next within the procedure declarations

elsif SELECTOR = VARIABLE FIND then

  TEMP := HEAD.NEXT1:

  LOOP
    EXIT WHEN (TEMP.TYPE DEFINE - END PACKAGE DECLARE) 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.NEXT1:
    TEMP := TEMP1:
  END LOOP;

  --did not find the variable within the package declarations
  --search the specified procedures declarations

  if NOT RESULT then
    TEMP := HENRY ARRAY(INDEX).BEGIN POINTER:
    LOOP
      --DID NOT FIND NAME IN PACKAGE DECLARATIONS
      EXIT WHEN (TEMP.TYPE DEFINE - END 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.NEXT1:
      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
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
      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 LOOP;
    TEMP1 := TEMP1.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 := TEMP1.NEXT1;
    TEMP := TEMP1;
  END LOOP
looking for procedure or function calls

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

if TEMP TYPE DEFINE = PROCALL OR DS then
  TEMPI := TEMP;
  LOOP -- MOVE PAST THE PARAMETERS
  EXIT WHEN TEMPI TYPE DEFINE = END ACTUAL_PARAM;
  TEMP2 := TEMPI;
  TEMPI := TEMP2.NEXTI;
END LOOP;
if (LOCAL NAME TEMP PROCEDURE FIND 1)) then
  TEMPI TYPE DEFINE := PROCEDURE_TYPE;
else
  TEMPI := TEMP1.NEXTI;
  if TEMPI TYPE DEFINE = ASSIGN_TYPE then
    TEMPI TYPE DEFINE := DATA_STRUCTURE;
    --IF NOT IT IS A PROCEDURE CALL ONLY
    TEMPI := TEMP2.NEXTI;
  LOOP
  EXIT WHEN TEMPI TYPE DEFINE = END ASSIGN_TYPE;
  if (TEMP TYPE DEFINE = FUNCALL OR DS) then
    if NOT LOCAL NAME TEMP FUNCTION FIND 1) then
      TEMPI TYPE DEFINE := DATA_STRUCTURE;
    else TEMPI TYPE DEFINE := FUNCTION_TYPE;
    end if;
  end if;
  end if;
  TEMPI := TEMP1;
  TEMPI := TEMP2.NEXTI;
END LOOP;
else TEMPI TYPE DEFINE := PROCEDURE_TYPE;
end if;
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
  TEMPI := TEMP;
  LOOP -- LOOKING FOR FUNCTIONS
  EXIT WHEN TEMPI TYPE DEFINE = END ASSIGN_TYPE;
  if TEMP TYPE DEFINE = FUNCALL OR DS then
    if NOT LOCAL NAME TEMP FUNCTION FIND 1) then
      TEMPI TYPE DEFINE := FUNCTION_TYPE;
    else TEMPI TYPE DEFINE := DATA_STRUCTURE;
    end if;
  end if;
  end if;
  TEMPI := TEMP1;
  TEMPI := TEMP2.NEXTI;
```

67
TEMP := TEMP1;
END LOOP;
end if:
TEMP1 := TEMP;
TEMP := TEMPI.NEXT1;
END LOOP; -- PROCALL OR_DS LOOP
END LOOP; -- FOR LOOP

end SPRUCE_UP_HENRY_DATA;

--this function only works for Ada language strings that identify
--a variable

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. TEMPI : 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;
elsif 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:
    end loop;
end if:

T1 := T2;
T2 := T1.NEXT1;
END LOOP;
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.
END IF.
IF (TEMP NOME (1 MAX) IN NAME (1 MAX)) AND (ASSIGN MARK) THEN
TRANS COUNT := TRANS COUNT - 10.
END IF.
TEMP := TEMP1.NEXT1.
TEMP1 := TEMP.
END LOOP.
RETURN TRANS COUNT.
END TRANSIVITY OUT:

--finishes polishing the data and with the transivity 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 TEMPI 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 (1 MAX LINE SIZE).
NAME OF LENEME TYPE.
ASSIGN MARK.
GLOBAL MARK BOOLEAN FALSE.
SIZE INTEGER MAX LINE SIZE
NEW SIZE INTEGER 10
TEMP NAME STRING 1 SIZE)

BEGIN
PUT HENRY FILE "IN CALCULATE METRIC" NEW HENRY FILE

first HENRY file is open so that the data can be reloaded

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

initialize the variables for the procedure and for the temporary

TEMP TYPE DEFINE PROCEDURE TYPE OUT PUT DATA TYPE PROCEDURE TYPE
NEW SIZE NAME 10
NEW NAME LIST PROCEDURE
NEW NAME (H NEW SIZE TEMP NAME + SIZE)
OUT PUT DATA AT NAME OF 1 NEW SIZE NEW NAME NEW SIZE
PUT HENRY OUT NEW LINE HENRY OUT NEW NAME
PUT HENRY OUT NEW NAME
PUT HENRY OUT
OUT PUT DATA AT NAME OF 1 NEW SIZE NEW NAME NEW SIZE
PUT HENRY OUT NEW LINE HENRY OUT
OUT TEMP TYPE DEFINE FUNCTION TYPE OUT PUT DATA TYPE FUNCTION TYPE
NEW SIZE
NEW NAME LIST
NEW NAME (H NEW SIZE TEMP NAME + SIZE)
OUT PUT DATA AT NAME OF 1 NEW SIZE NEW NAME NEW SIZE
PUT HENRY OUT NEW LINE HENRY OUT NEW NAME
PUT HENRY OUT NEW NAME
OUT TEMP TYPE
NEW LINE HENRY OUT
end

DEF F fn def name for the function

TEMP1 TEMP2 NEXT
TEMP2 TEMP1

function to create increase global flow metric

END WHEN TEMP1 TYPE DEFINE END FUNCTION TYPE) OR
TEMP1 TYPE DEFINE END PROCEDURE CALL)
- TEMP1 TYPE DEFINE ASSIGN TYPE then
  ASSIGN MARKER TRUE
- TEMP1 TYPE DEFINE END ASSIGN TYPE then
  ASSIGN MARKER FALSE
  GLOBAL MARKER FALSE

- TEMP IDENTIFY GLOBAL DECLARE AND (ASSIGN MARKER)
  GLOBAL DECLARE GLOBAL READ TO
  GLOBAL MARKER TRUE
  GLOBAL READ WRITE GLOBAL READ WRITE TO

- TEMP IDENTIFY GLOBAL DECLARE AND
  ASSIGN MARKER then
  GLOBAL WRITE GLOBAL WRITE TO
  GLOBAL MARKER TRUE

TEMP1 TEMP2 NEXT
TEMP2 TEMP1

END WHEN TEMP1 TYPE DEFINE PARAM TYPE then
- TEMP1 TYPE DEFINE PARAM DECLARE END PARAM DECLARE
  TEMP1 PARAM TYPE IN TYPE THEN
  TEMP1 PARAM TYPE OUT TYPE THEN
  TEMP1 PARAM TYPE IN OUT TYPE THEN
  END TEMP1 PARAM IN OUT
  END TEMP1 PARAM IN OUT

END TEMP1 PARAM
END TEMP1
look for procedure and function type actual parameters

TEMPI  TEMPI
TEMPI  TEMPI NEXT1
TEMPI  TEMPI
LOOP
EXIT WHEN (TEMPI TYPE DEFINE  END FUNCTION TYPE OR
(TEMPI TYPE DEFINE  END PROCEDURE TYPE)

TEMPI  TEMPI
TEMPI  TEMPI NEXT1
if TEMPI TYPE DEFINE  ASSIGN TYPE then
ASSIGN MARKER  TRUE
else if TEMPI TYPE DEFINE  END ASSIGN TYPE then
ASSIGN MARKER  FALSE
GLOBAL MARKER  FALSE
end if
if TEMPI TYPE DEFINE  PROCEDURE TYPE then
TEMPI  TEMPI NEXT1
LOOP
EXIT WHEN TEMPI TYPE DEFINE  END ACTUAL PARAM
FAN OUT  FAN OUT to
TEMPI2  TEMPI
TEMPI  TEMPI2 NEXT1
END LOOP
else if TEMPI TYPE DEFINE  FUNCTION TYPE then

count the function parameters

TEMPI  TEMPI NEXT1
LOOP
EXIT WHEN TEMPI TYPE DEFINE  END ACTUAL PARAM
FAN IN  FAN IN to
RETURN FROM FUNCTION
else if TEMPI TYPE DEFINE  DATA STRUCTURE then
(NOT ASSIGN MARK) then
GLOBAL MARK  TRUE
GLOBAL WRITE  GLOBAL WRITE
else if TEMPI TYPE DEFINE  DATA STRUCTURE then
GLOBAL MARK then
GLOBAL READ WRITE  GLOBAL WRITE
else NOT GLOBAL MARK then
GLOBAL READ  GLOBAL WRITE
end if
END if
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);
put(HENRY_OUT, "GLOBAL WRITE = ");
put(HENRY_OUT, GLOBAL_WRITE);
OUT_PUT_DATA(I).TYPE_WRITE := GLOBAL_WRITE;
NEW_LINE(HENRY_OUT);
put(HENRY_OUT, "GLOBAL READ_WRITE = ");
put(HENRY_OUT, GLOBAL_READ_WRITE);
OUT_PUT_DATA(I).TYPE_READ_WRITE := GLOBAL_READ_WRITE;
NEW_LINE(HENRY_OUT);
put(HENRY_OUT, "GLOBAL FLOW ");
put(HENRY_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.
GRAPH RELATIVE DATA

PAUSE PRINT STOP COUNT in INTEGER
RUNNING COUNT in out INTEGER
DONE in out BOOLEAN

SET UP SCREEN IN STRING in ROW STRING TYPE
STRING SIZE in INTEGER

ENTER STRING NAME in ROW STRING TYPE
IN ROW WIDTH in integer

M S H BINARY DISPLAY

M S BINARY DISPLAY

M S BINARY DISPLAY

PAUSE PRINT STOP COUNT in INTEGER
RUNNING COUNT in out INTEGER
DONE in out BOOLEAN

H BINARY DISPLAY

H BINARY DISPLAY

H BINARY DISPLAY
IN ROW, WIDTH in INTEGER) is

SCREEN WIDTH INTEGER : 76.
CENTER POS INTEGER : 0.
TEMP NAME STRING TYPE.

begin
FOR I IN 1..30 LOOP
   TEMP NAME(I) : NULL CHAR.
END LOOP.
TEMP NAME(I-WIDTH) NAME(I-WIDTH).
CENTER POS (SCREW 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, STRING SIZE).
SET REVERSE(OFF).
PUT("".........................""
NEW LINE.
END SET UP SCREEN.

procedure LIST HENRY DATA is

SHORT NAME SIZE in INTEGER.
NAME POINTER IN POINTER HEAD.
TEMP NAME STRING TYPE.
NAME NAME STRING SIZE SHORT NAME OUT
WIDTH INTEGER.
CENTER POS STRING INTEGER.
DIR STRING SCREW SDNC.
WIDTH 100.
}
begin
HEADER STRING(1 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 others put "Unidentified"
end case
new line
put "NAME"
TEMP POINTER NAME = TEMP NAME SIZE
put line
also
TEMP NAME = MAX LINE SIZE
TEMP POINTERS NAME = MAX LINE SIZE
SHORT NAME = SHORT NAME SIZE
TEMP NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
put line
also
NAME = DEF NAME
TEMP POINTERS NAME = 0 NAME
NAME = SHORT NAME
NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
put line
also
NAME = SHORT NAME
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
put line
also
NAME = SHORT NAME
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
SHORT NAME = SHORT NAME SIZE
PARAMETER TYPE

TEMP POINTER PARAM TYPES

IN TYPE

OUT TYPE

IN OUT TYPE

EXIT

PRINT

COUNT DONE

COUNT

SET SCREEN HEADER STRING HEADER SIZE

NEXT

COUNT

EXIT

NEW LINE

NEXT

EXIT

EXIT

EXIT

EXIT

EXIT

EXIT

EXIT
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) "PACKAGE;  
IN STRING(9:10) INPUT FILE NAME(1:41);  
on IN STRING;  
NEW LINE 2;
LOOP
EXIT WHEN (END OF FILE HENRY OUT OR DONE)
FOR J 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
contains each procedure function analyzed with for example the
mean number. It also gives a verbal report for each function
structure.

procedure WRITE RELATIVE DATA IS

INDICATOR1
INDICATOR2
INDICATION FLOAT 0.0
UPPER LIMIT constant FLOAT 4.0
LOWER LIMIT constant FLOAT -2.5
TEMP HOLDER STRING ID
STOP RUNNING INTEGER 1
HEADER STRING ROW STRING TYPE
ROW STRING ROW STRING TYPE
SIZE INTEGER
DONE BOOLEAN FALSE
HEADER SIZE INTEGER 25
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, I - 5):
REL ARRAY(I).NAME OF := OUT_PUT_DATA(I).NAME OF:
PUT(REL ARRAY(I).NAME OF, SET_CURSOR_POS(42, I - 5), PUT(" : ")):
--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 PROC Func COUNT LOOP
if REL ARRAY(I).TYPE FLOW = 0.0 THEN
    INDICATORI := REL ARRAY(I).TYPE COMPLEXITY - REL ARRAY(I).TYPE FLOW;
else INDICATORI := REL ARRAY(I).TYPE COMPLEXITY:
end if:
if REL ARRAY(I).TYPE FAN OUT = 0.0 THEN
    INDICATOR2 := REL ARRAY(I).TYPE FAN IN:
else INDICATOR2 := REL ARRAY(I).TYPE FAN OUT:
end if:
if REL ARRAY(I).TYPE WRITE = 0.0 THEN
    INDICATOR3 := REL ARRAY(I).TYPE READ:
else INDICATOR3 := 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
      else if INDICATOR2  LOWER LIMIT THEN
         PUT(" - This implies an extremely complex interface")
         new line
      end if
      ELSEIF INDICATOR1  LOWER LIMIT THEN
         PUT(" - Has significant global data flow compared 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
      else if 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(I 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

84
WRITE RELATIVE DATA

GRAPH RELATIVE DATA

LOOP ONE INTEGER
ROW STRING ROW STRING TYPE
HEADER STRING ROW STRING TYPE
SIZE INTEGER
STOP RUNNING INTEGER 1
DONE BOOLEAN FALSE
STATE INTEGER
NUM LOOP ONE
REMAIN INTEGER
HEADER SIZE INTEGER

Case:
NUM LOOP ONE PROCEDURE ON NOT
REMAIN ONE PROCEDURE ON NOT
loop: number of screens need to display
remain = number of the partial screen that is left over

HEADER STRING "THE GRAPHICAL PERFORMANCE DATA"
if NUM LOOP CNT = 1 THEN STOP
else STOP REM CNT
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 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.

85
FOOTPRINT START

END LOOP

NEW LINE

PAUSE PRINT STOP RUNNING, DONE

END GRAPH LOOP WHEN DONE

END IF

RUNNING, 0 AND STOP 0 THEN

if PROC FUNC COUNT 5 THEN

STOP 0

end if

SET UP SCREEN HEADER STRING HEADER SIZE

RUNNING 1

elseif RUNNING 10 AND STOP 5 THEN

if PROC FUNC COUNT 5 THEN

SET UP SCREEN HEADER STRING HEADER SIZE

RUNNING 1

elseif PROC FUNC COUNT 5 THEN

NUM LOOP CNT NUM LOOP CNT 1

if NUM LOOP CNT 2 THEN STOP 5 RUNNING 1

elseif REM CNT 0 THEN

STOP REM CNT RUNNING 1

else SET UP SCREEN HEADER STRING HEADER SIZE

RUNNING 1

end if

end if
if OUT PUT DATA(MAX WRITE) TYPE WRITE 0 0 THEN
REL ARRAY(I) TYPE WRITE NORMALIZER
(OUT PUT DATA(MAX WRITE) TYPE WRITE)
else REL ARRAY(I) TYPE WRITE 0 0
end if

if OUT PUT DATA(MAX READ WRITE) TYPE READ WRITE 0 0 THEN
REL ARRAY(I) TYPE READ WRITE NORMALIZER
(OUT PUT DATA(I) TYPE READ WRITE)
else REL ARRAY(I) TYPE READ WRITE 0 0
end if
APPENDING MODIFIED PARSERS

BYPASS-FUNCTIONS

The function compares the lexeme of the current token with the
empty string currently being sought by the parser. If the current token
matches either token, 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
LENEME string length(LINESIZE)
SIZE integer natural
HENRY LENEME string LENGTH(LINESIZE)

BEGIN
GET CURRENT TOKEN RECORD(CURRENT TOKEN RECORD LENEME LENGTH)
LENEME = CURRENT TOKEN RECORD LENEME,
SIZE = CURRENT TOKEN RECORD LENEME SIZE - 1

90
### WHEN TOKEN NUMERIC LITERAL

```
CURRENT_TOKEN RECORD TOKEN TYPE NUMBER лит Hình

CONSUME TRUE

DECLARE TYPE CONSTANT DECLARE

OPRAND NODE CURRENT TOKEN RECORD DECLARE TYPE DECLARE

IF HENRY WRITE ENABLE THEN
  WRITE HENRY DATA LOCAL DECLARE HENRY LEXEME IDENT TYPE NONE NEXT HEN
  CREATE NODE NEXT HEN LAST RECORD
  HENRY WRITE ENABLE FALSE.

END IF

END IF
```

### WHEN TOKEN CHARACTER LITERAL

```
CURRENT_TOKEN RECORD TOKEN TYPE CHARACTER лит Hình

IF HENRY WRITE ENABLE THEN
  WRITE HENRY DATA LOCAL DECLARE HENRY LEXEME IDENT TYPE NONE NEXT HEN.
  CREATE NODE NEXT HEN LAST RECORD
  HENRY WRITE ENABLE FALSE.

CONSUME TRUE.

END IF
```

### WHEN TOKEN STRING LITERAL

```
CURRENT_TOKEN RECORD TOKEN TYPE STRING лит Hình
```

91
when TOKEN "for"
  if (ADJUST LEAXEME(LEXEME, SIZE) = "for") then
  CONSUME TRUE.
end if.

when TOKEN "others"
  if (ADJUST LEAXEME(LEXEME, SIZE) = "others") then
  CONSUME TRUE.
end if.

when TOKEN "return"
  if (ADJUST LEAXEME(LEXEME, SIZE) = "return") then
  CONSUME TRUE.
end if.

when TOKEN "exit"
  if (ADJUST LEAXEME(LEXEME, SIZE) = "exit") then
  CONSUME TRUE.
end if.

when TOKEN "procedure"
  if (ADJUST LEAXEME(LEXEME, SIZE) = "procedure") then
  CONSUME TRUE.
end if.

when TOKEN "function"
  if (ADJUST LEAXEME(LEXEME, SIZE) = "function") then
  CONSUME TRUE.
end if.

when TOKEN "with"
  if (ADJUST LEAXEME(LEXEME, SIZE) = "with") then
  CONSUME TRUE.
end if.

when TOKEN "use"
  if (ADJUST LEAXEME(LEXEME, SIZE) = "use") then
  CONSUME TRUE.
end if.
when TOKEN PACKAGE:
  if (ADJUST LEXEME(LEXEME. SIZE) = "package") then
    CONSUME := TRUE.
  end if:

when TOKEN BODY:
  if (ADJUST LEXEME(LEXEME. SIZE) = "body") then
    CONSUME := TRUE.
  end if:

when TOKEN RANGE :=>
  if (ADJUST LEXEME(LEXEME. SIZE) = "range") then
    CONSUME := TRUE.
  end if:

when TOKEN IN :=>
  if (ADJUST LEXEME(LEXEME. SIZE) = "in") then
    CONSUME := TRUE.
  end if:

when TOKEN OUT :=>
  if (ADJUST LEXEME(LEXEME. SIZE) = "out") then
    CONSUME := TRUE.
  end if:

when TOKEN SUBTYPE :=
  if (ADJUST LEXEME(LEXEME. SIZE) = "subtype") then
    CONSUME := TRUE.
  end if:

when TOKEN TYPE :=
  if (ADJUST LEXEME(LEXEME. SIZE) = "type") then
    CONSUME := 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(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA
P M HERZIG JUN 87

UNCLASSIFIED

F/G 12/5

NL
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A
when TOKEN ENTRY =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "entry") then
    CONSUME := TRUE;
  end if;

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

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

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

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

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

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

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

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

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

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

when TOKEN DO =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "do") then
    CONSUME := TRUE;
  end if;
when TOKEN GOTO =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "goto") then
    CONSUME := TRUE;
  end if;
when TOKEN OF =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "of") then
    CONSUME := TRUE;
  end if;
when TOKEN ALL =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "all") then
    CONSUME := TRUE;
  end if;
when TOKEN PRAGMA =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "pragma") then
    CONSUME := TRUE;
  end if;
when TOKEN AND =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "and") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN AND, CONSUME, RESERVE WORD TEST);
when TOKEN OR =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "or") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN OR, CONSUME, RESERVE WORD TEST);
when TOKEN NOT =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "not") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN NOT, CONSUME, RESERVE WORD TEST);
when TOKEN XOR =>
  if (ADJUST LEXEME(LEXEME, SIZE) = "xor") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(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 TOKENASTERISK =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "*"n) then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKENASTERISK, 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;
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

UPDATE_LINE_COUNT;
CONSUME := TRUE;
end if;

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

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

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

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

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

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

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

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

when TOKEN RIGHT BRACKET =>
  if (ADJUST LEXEME(LEXEME, SIZE) = ">>") then
    CONSUME := TRUE;
  end if;

when others => null;
end case;

ADJUST_TOKEN_BUFFER(CONSUME, RESERVE_WORD_TEST);

return (CONSUME);
end BYPASS;

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(RESERED 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;

-- 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(RESERED 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.
-- ********************************************************************************---
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:
-*-- CONTEXT_CLAUSE --> with WITH_OR_USE_CLAUSE | use WITH_OR_USE_CLAUSE --*
* 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;
end loop: -- inner while loop
end loop: -- outer while loop
return (TRUE);
end CONTEXT_CLAUSE;

-- BASIC_UNIT --> LIBRARY_UNIT
-- --> SECONDARY_UNIT
function BASIC_UNIT 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 BASIC_UNIT;

-- LIBRARY_UNIT --> procedure PROCEDURE_UNIT
-- --> function FUNCTION_UNIT
-- --> package PACKAGE_DECLARATION
-- --> generic GENERIC_DECLARATION
function LIBRARY_UNIT 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;
else
  if procedure_unit statement
endif;
elsif (BYPASS(TOKEN_FUNCTION)) then
103
DECLARE TYPE := FUNCTION DECLARE;
if (FUNCTION UNIT) then
  return (TRUE);
else
  SYNTAX_ERROR("Library unit");
end if:
elsif (BYPASS(TOKENPACKAGE)) then
DECLARE TYPE := PACKAGE DECLARE;
if (PACKAGE DECLARATION) then
  return (TRUE);
else
  SYNTAX_ERROR("Library unit");
end if:
elsif (BYPASS(TOKEN GENERIC)) then
  if (GENERIC DECLARATION) then
    return (TRUE);
  else
    SYNTAX_ERROR("Library unit");
  end if:
else
  return (FALSE);
end if:
end LIBRARY UNIT;

-- 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

return (TRUE);
else
SYNTAX_ERROR("Library unit body");
end if;
elsif (BYPASS(TOKEN FUNCTION))
DECLARE_TYPE := FUNCTION_DECLARE;
if (FUNCTION_UNIT) then
return (TRUE);
else
SYNTAX_ERROR("Library unit body");
end if;
elsif (BYPASS(TOKEN PACKAGE))
DECLARE_TYPE := PACKAGE_DECLARE;
HENRY_WRITE_ENABLE := TRUE;
pout(result_file, "true"); newline(result_file);
if (PACKAGE_DECLARATION) then
return (TRUE);
else
SYNTAX_ERROR("Library unit body");
end if;
else
return (FALSE);
end if;
---

-- SUBUNIT -- separate (NAME) PROPER_BODY
function SUBUNIT return boolean is
begin
put(RESULT_FILE, "In subunit "); newline(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
return (FALSE);
end if;
---
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 BODY TAIL return boolean;
function PACKAGE DECLARATION return boolean;
function PACKAGE UNIT return boolean;
function PACKAGE BODY return boolean;
function PACKAGE BODY TAIL return boolean;
function PACKAGE_TAIL END return boolean;
function DECLARATIVE PART return boolean;
function BASIC DECLARATIVE ITEM return boolean;
function BASIC DECLARATION return boolean;
function LATER DECLARATIVE ITEM return boolean;
function PROPER BODY return boolean;
function SEQUENCE_OF_STATEMENTS return boolean;
function STATEMENT return boolean;
function COMPOUND_STATEMENT return boolean;
function BLOCK_STATEMENT return boolean;
function IF_STATEMENT return boolean;
function CASE_STATEMENT return boolean;
function CASE_STATEMENT_ALTERNATIVE return boolean;
function LOOP_STATEMENT return boolean;
function EXCEPTION_HANDLER return boolean;
function ACCEPT_STATEMENT return boolean;
function SELECT_STATEMENT return boolean;
function SELECT_STATEMENT_TAIL return boolean;
function SELECT_ALTERNATIVE return boolean;
function SELECT_ENTRY_CALL return boolean;
end PARSER_1;

package body PARSER_1 is

end PARSER_1;
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 IDENTIFIER)) then
if (DISCRIMINANT_PART) then
null;
end if;
-- if discriminant_part
if (BYPASS(TOKEN IS)) then
if (PRIVATE_TYPE_DECLARATION) 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;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
-- if private_type_declaration
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
-- if bypass(token_is)
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
elsif (BYPASS(TOKEN_WITH)) then
if (BYPASS(TOKEN_PROCEDURE)) then
DECLARE TYPE := PROCEDURE DECLARE;
if (PROCEDURE_UNIT) then
return (TRUE);
else
SYNTAX_ERROR("Generic parameter declaration");
end if;
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 (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, END_PROCEDURE_CALL,
  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;
elsif (BYPASS(TOKEN SEMICOLON)) then
SCOPE_LEVEL := SCOPE_LEVEL - 1;
return (TRUE);
elsif (BYPASS(TOKEN_RENAMES)) then
if (NAME) then
  if (BYPASS(TOKEN SEMICOLON)) then
    SCOPE_LEVEL := SCOPE_LEVEL - 1;
    return (TRUE);
  else
    SYNTAX_ERROR("Procedure unit");
  end if;
else
  SYNTAX_ERROR("Procedure unit");
end if;
end if;
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:

-- SUBPROGRAM_BODY --> new NAME |GENERIC_ACTUAL_PART? ;
 -- --> separate ;
 -- --> <> ;
 -- --> DECLARATIVE_PART? begin SEQUENCE_OF_STATEMENTS
 -- |exception |EXCEPTION_HANDLER ; ? end |DESIGNATOR ? ;
 -- --> NAME :
function SUBPROGRAM_BODY return boolean is

NAME_POINTER : POINTER;

begin
put(Result_FILE, "In subprogram body "); new_line(Result_FILE);
NAME_POINTER := NEXT_HEN;
DECLARATION := TRUE;
if (BYPASS(TOKEN_NEW)) then
  HENRY_WRITE_ENABLE := FALSE;
if (NAME) then
  if (GENERIC_ACTUAL_PART) then
    null:
  end if; -- if generic actual part
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Subprogram body");
  end if; -- if bypass(token_semicolon)
else
  SYNTAX_ERROR("Subprogram body");
end if; -- if name statement
elsif (BYPASS(TOKEN_SEPARATE)) then
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Subprogram body");
  end if; -- if bypass(token_semicolon)
elsif (BYPASS(TOKEN_BRACKETS)) then
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Subprogram body");
  end if; -- if bypass(token_semicolon)
elsif (DECLARATIVE_PART) then
  WRITE_HENRY_DATA(BlANK, DUMMY_LEXEME, END_DECLARATIONS, 
    NONE, NEXT_HEN);
  CREATE_NODE(NEXT_HEN, LAST_RECORD);
if (BYPASS(TOKEN_BEGIN)) then
  DECLARATION := FALSE;
else
  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;
end if;
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;
else
  SYNTAX_ERROR("Subprogram body");
end if;
if (BYPASS(TOKEN_END)) then
  HENRY WRITE ENABLE := FALSE;
else
  if (DESIGNATOR) then
    null;
  end if;
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;
else
  SYNTAX_ERROR("Subprogram body");
end if;
eif (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;
else
  SYNTAX_ERROR("Subprogram body");
end if;
else
  SYNTAX_ERROR("Subprogram body");
end if;
eif (BYPASS(TOKEN_END)) then
  HENRY WRITE ENABLE := FALSE;
else
  if (DESIGNATOR) then
    null;
  end if;
eif (BYPASS(TOKEN BEGIN)) then
  else
    SYNTAX_ERROR("Subprogram body");
  end if;
else
  SYNTAX_ERROR("Subprogram body");
end if;
eif (BYPASS(TOKEN_END)) then
  HENRY WRITE ENABLE := FALSE;
i
if (BYPASS(TOKEN_SEMICOLON)) then
    DECLARATION := TRUE;
    return (TRUE);
else
    SYNTAX_ERROR("Subprogram body");
    end if;
    -- if bypass(token_semicolon)
else
    SYNTAX_ERROR("Subprogram body");
    end if;
    -- if bypass(token_end)
else
    SYNTAX_ERROR("Subprogram body");
    end if;
    -- if sequence of statements
elsif (NAME) then
    if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
    else
        SYNTAX_ERROR("Subprogram body");
        end if;
        -- if bypass(token_semicolon)
    else
        return (FALSE);
        end if;
        -- if bypass(token_new)
end if;
end SUBPROGRAM_BODY;

-- FUNCTION_UNIT --> DESIGNATOR FUNCTION_UNIT TAIL
function FUNCTION_UNIT return boolean is
begin
    put(RESULT_FILE, "In function unit "); new_line(RESULT_FILE);
    DECLARATION := TRUE;
    HENRY_WRITE ENABLE := TRUE;
    if (DESIGNATOR) then
        if PACKAGE BODY DECLARE then
            WRITEHENRYATA(LOCAL DECLARE, DUMMY_lexeme, FUNCTION_TYPE,
                NONE, LAST_RECORD);
            WRITE_LINE_COUNT(LAST_RECORD NOMEN, HENRY LINE COUNT,
                DUMMY9s, NEXT_LINE);
        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;
        else
            return (FALSE);
            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;
if (BYPASS(TOKEN_NEW)) then
  if (NAME) then
    if (GENERIC_ACTUAL_PART) then
      null;
    end if;
    if (BYPASS(TOKEN SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Function unit tail");
    end if;
  else
    SYNTAX_ERROR("Function unit tail");
  end if;
elsif (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 FUNCTION_UNIT_TAIL;
-- FUNCTION BODY --> is FUNCTION BODY 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)
      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
    SYNTAX ERROR("Function body tail");
    end if:
  if (BYPASS(TOKEN SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE).
  end if:
end FUNCTION BODY:

-- FUNCTION BODY TAIL -- separate :
-- 
-- SUBPROGRAM_BODY
-- NAME :
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:
      endif;
    elsif (BYPASS(TOKEN SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX ERROR("Function body tail");
      end if:
    endforeach:
    elsif (SUBPROGRAM BODY) then
      return (TRUE);
    elsif (NAME) then
      if (BYPASS(TOKEN SEMICOLON)) then
        return (TRUE);
      endif:
    endif:
  end function:

115
else
SYNTAX_ERROR("Function body tail");
end if: -- if bypass(token_semicolon)
else
return (FALSE);
end if: -- if bypass(token_separate)
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_type)
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: -- if task body statement
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_identifier)
if (BYPASS(TOKEN_SEMICOLON)) then
SCOPE LEVEL := SCOPE LEVEL - 1;
return (TRUE);
else
SYNTAX_ERROR("Task declaration");
end if: -- if bypass(token_semicolon)
else
SYNTAX_ERROR("Task declaration");
end if: -- if bypass(token_end)
elsif (BYPASS(TOKEN_SEMICOLON)) then
    SCOPE_LEVEL := SCOPE_LEVEL - 1;
    return (TRUE);
else
    SYNTAX_ERROR("Task declaration");
end if;
if bypass(token_is) then
    SCOPE_LEVEL := SCOPE_LEVEL - 1;
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;
if bypass(token_is) then
    return (FALSE);
else
    SYNTAX_ERROR("Task body");
end if;
end TASK_BODY;

-- TASK_BODY_TAIL --> separate
-- --> DECLARATIVE PART ? begin SEQUENCE OF STATEMENTS
-- --> exception EXCEPTION_HANDLER + ? end identifier ?
function TASK_BODY_TAIL return boolean is
begin
put(RESULT_FILE, "In task_body_tail "); new_line(RESULT_FILE);
DECLAARATION := TRUE;
if (BYPASS(TOKEN_SEPARATE)) then
    return (TRUE);
elsif (DECLARATIVE PART) then
    if (BYPASS(TOKEN_BEGIN)) then
        DECLARATION := FALSE;
    end if;
elsif (SEQUENCE OF STATEMENTS) then
    if (BYPASS(TOKEN_EXCEPTION)) then
        if (EXCEPTION HANDLER) then
            while (EXCEPTION_HANDLER) loop
            end loop;
        end if;
    end if;
else
    SYNTAX_ERROR("Task body");
end if;
if bypass(token_identifier) then
    return (FALSE);
else
    SYNTAX_ERROR("Task body");
end if;
null;
end loop;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
end if;
end if;
end loop;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;
else
  SYNTAX_ERROR("Task body tail");
end if;

118
-- PACKAGE DECLARATION --> body PACKAGE BODY

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
-- if package unit tail statement
end if:
-- if bypass(token_package)
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. ENDPACKAGE 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
-- if bypass(token is)
end if:
119
return (FALSE);
end if;  -- if bypass(token_identifier)
end PACKAGE_BODY;

-- PACKAGE_BODY_TAIL --> separate ;
-- --> DECLARATIVE_PART ? [begin SEQUENCE_OF_STATEMENTS
-- exception EXCEPTION_HANDLER - ?; ?]
-- end identifier ?;
function PACKAGE_BODY_TAIL return boolean is
begin
put(RESULT_FILE, "In package_body_tail "); new_line(RESULT_FILE);
DECLARATION := TRUE;
if (BYPASS(TOKEN_SEPARATE)) then
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Package body tail");
  end if;
elsif (DECLARATIVE_PART) then
  DECLARATION := FALSE;
  if (BYPASS(TOKEN_BEGIN)) then
    if (SEQUENCE_OF_STATEMENTS) then
      if (BYPASS(TOKEN_EXCEPTION)) then
        if (EXCEPTION_HANDLER) then
          while (EXCEPTION_HANDLER) loop
            null;
          end loop;
        else
          SYNTAX_ERROR("Package body tail");
        end if;
      end if;
    end if;
    -- if exception_handler statement
  end if;
  -- if bypass(token_semicolon)
if (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;
end if;
else
  SYNTAX_ERROR("Package body tail");
end if;
else
  SYNTAX_ERROR("Package body tail");
end if;
else
  SYNTAX_ERROR("Package body tail");
end if;
else
  SYNTAX_ERROR("Package body tail");
end if;
else
  SYNTAX_ERROR("Package body tail");
end if;
else
  SYNTAX_ERROR("Package body tail");
end if;
i
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:
    else
      SYNTAX_ERROR("Package body tail");
    end if;
    -- if exception_handler statement
  end if:
  -- if bypass(token_exception)
  if (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;
else
  SYNTAX_ERROR("Package body tail");
end if;
else
  SYNTAX_ERROR("Package body tail");
end if;
else
  SYNTAX_ERROR("Package body tail");
end if;
endif
else
  SYNTAX_ERROR("Package body tail");
endif
else
  SYNTAX_ERROR("Package body tail");
endif
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:
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;
end if;
else
SYNTAX_ERROR("Package tail end");
end if:
-- if bypass(token_semicolon)
else
SYNTAX_ERROR("Package tail end");
end if:
122
end if; -- if name statement
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;
elsif (BYPASS(TOKEN_END)) then
  while (BASIC_DECLARATIVE_ITEM) loop
    null;
  end loop;
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;
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;
return (FALSE);
end if;
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 basic_declarative_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
return (FALSE);
end if;
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:
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;
elif (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;
elif (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;
elif (BYPASS(TOKEN_PACKAGE)) then
DECLARE TYPE := PACKAGE DECLARE;
if (PACKAGE_DECLARATION) then
return (TRUE);
else
SYNTAX_ERROR("Basic declaration");
end if;
elif (BYPASS(TOKEN_GENERIC)) then
if (GENERIC_DECLARATION) then
return (TRUE);
else
SYNTAX_ERROR("Basic declaration");
end if;
elif (BYPASS(TOKEN_TASK)) then
DECLARE TYPE := TASK_DECLARE;
if (TASK_DECLARATION) then
return (TRUE);
else
SYNTAX_ERROR("Basic declaration");
end if;
else
end if;
elif (BYPASS(TOKEN_TYPE)) then
if (TYPE_DECLARATION) then
return (TRUE);
else
SYNTAX_ERROR("Basic declaration");
end if;
elif (BYPASS(TOKEN_SUBTYPE)) then
if (SUBTYPE_DECLARATION) then
return (TRUE);
else
SYNTAX_ERROR("Basic declaration");
end if;
elif (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;
elif (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;
elif (BYPASS(TOKEN_PACKAGE)) then
DECLARE TYPE := PACKAGE DECLARE;
if (PACKAGE_DECLARATION) then
return (TRUE);
else
SYNTAX_ERROR("Basic declaration");
end if;
else
end if;
elif (BYPASS(TOKEN_GENERIC)) then
if (GENERIC_DECLARATION) then
return (TRUE);
else
end if;
else

SYNTAX_ERROR("Basic declaration");
end if:
elsif (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

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;
else (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 --
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.
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;

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;
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;
end if;
elsif (BYPASS(TOKEN SELECT)) then
  if (SELECT STATEMENT) then
    return (TRUE);
  else
    SYNTAX ERROR("Compound statement");
  end if;
else
  return (FALSE);
end if;
end 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 STATUS := 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 bypass(token colon)
else
  DECLARE TYPE := VARIABLE DECLARE;
end if;
-- if bypass(token identifier)
if (BYPASS(TOKEN DECLARE)) then
  SCOPE LEVEL := SCOPE LEVEL - 1;
if (DECLARATIVE PART) then
  null:
else
  SYNTAX_ERROR("Block statement");
end if;
-- if declarative part statement
else
  DECLARE TYPE := BLOCK DECLARE;
endif: -- if bypass(token declare)
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 if;
end if;
  null:
end if;
end loop.
else
  SYNTAX ERROR("Block statement").
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)

if (BYPASS(TOKEN SEMICOLON)) then
  SCOPE LEVEL := SCOPE LEVEL - 1.
  DECLARATION := TRUE.
  return (TRUE).
else
  SYNTAX ERROR("Block statement").
end if. -- if bypass(token semicolon)
else
  SYNTAX ERROR("Block statement").
end if. -- if bypass(token end)
else
  SYNTAX ERROR("Block statement").
end if. -- if sequence of statements
else
  if not (DECLARE STATUS) then
    DECLARATION := FALSE.
  end if.
  return (FALSE). -- if bypass(token begin)
end if.

end BLOCK STATEMENT.

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. -- if not sequence of statements
                    else
                      SYNTAX ERROR("If statement").
                    end if.
                  end if.
                end while.
              else
                SYNTAX ERROR("If statement").
              end if.
            end if.
          end if.
        end while.
      else
        SYNTAX ERROR("If statement").
      end if.
    else
      SYNTAX ERROR("If statement").
    end if.
  end if.
end function.
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.
else
  if (BYPASS(TOKEN ELSE)) then
    if (SEQUENCE OF STATEMENTS) then
      null.
    else
      SYNTAX ERROR("If statement").
    end if.
  end if.
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 ;)
else
  SYNTAX ERROR("If statement").
end if.
else
  NESTING METRIC(IF ELSE):
  return (TRUE);
else
  null.
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 ELSE)) then
    if (CASE STATEMENT ALTERNATIVE) then
      while (CASE STATEMENT ALTERNATIVE) loop
      end loop;
    end if.
  end loop;
end if.
else
  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 :)
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:
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
SYNTAX_ERROR("Case statement alternative");
end if: -- if not choice statement
end loop;
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;
  else
    SYNTAX_ERROR("Loop statement");
  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;
        else
          SYNTAX_ERROR("Loop statement");
        end if:
      else
        SYNTAX_ERROR("Loop statement");
      end if;
    else
      SYNTAX_ERROR("Loop statement");
    end if:
  else
    SYNTAX_ERROR("Loop statement");
  end if:
else
  return (FALSE);
end LOOP_STATEMENT;

-- EXCEPTION_HANDLER -- 
-- when EXCEPTION_CHOICE EXCEPTION_CHOICE *
-- SEQUENCE_OF_STATEMENTS
function EXCEPTION_HANDLER return boolean is
begin
put(RESULT_FILE, "In exception handler "); new_line(RESULT_FILE);
if (BYPASS(TOKEN WHEN)) then
begin
if (EXCEPTION_CHOICE) then
begin
while (BYPASS(TOKEN BAR)) loop
if not (EXCEPTION_CHOICE) then
SYNTAX_ERROR("Exception handler");
end if;
end loop;
end if;
end if;
end if;
end if;
end if;
end if:
end EXCEPTION_HANDLER:

-- ACCEPT_STATEMENT --> identifier (EXPRESSION) ? | FORMAL_PART ?
--
doseQUENCE_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
begin
if (BYPASS(TOKEN LEFT_PAREN)) then
begin
if (EXPRESSION) then
begin
if (BYPASS(TOKEN_RIGHT_PAREN)) then
null;
else
SYNTAX_ERROR("Accept statement");
end if;
end if:
end if:
end if;
end if;
else
SYNTAX_ERROR("Accept statement");
end if:
else
SYNTAX_ERROR("Accept statement");
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
end if:
begin
  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
            return (FALSE);
          end if;
        else
          SYNTAX ERROR("Select statement");
        end if;
      else
        end if:
      end if;
    end if;
  end SELECT STATEMENT:
  -- SELECT STATEMENT TAIL
  -- SELECT ALTERNATIVE or SELECT ALTERNATIVE NAME
  SEQUENCE OF STATEMENTS
  -- SELECT STATEMENT TAIL
end
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) then
-- check for entry call statement
if (BYPASS(TOKEN SEMICOLON)) then
if (SEQUENCE_OF_STATEMENTS) then
null;
end if;
-- if sequence_of_statements
return (TRUE);
else
SYNTAX_ERROR("Select statement tail");
end if;
-- if bypass(token_semicolon)
else
return (FALSE);
-- if select_alternative statement
end if;
end SELECT_STATEMENT_TAIL;

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

-- SELECT_ALTERNATIVE --> [when EXPRESSION => ?] accept ACCEPT_STATEMENT
-- --> when EXPRESSION => ? SEQUENCE_OF_STATEMENTS
-- --> when EXPRESSION => ? delay DELAY_STATEMENT
-- --> when EXPRESSION => ? SEQUENCE_OF_STATEMENTS
-- --> 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;
-- if bypass(token_arrow)
else
SYNTAX_ERROR("Select alternative");
end if;
-- if expression statement
end if;
-- if bypass(token_when)
if (BYPASS(TOKEN ACCEPT)) then
if (ACCEPT_STATEMENT) then
if (SEQUENCE_OF_STATEMENTS) then
null;
end if;
-- if sequence_of_statements
return (TRUE);
else
end if;
end if:
end SELECT_ALTERNATIVE;
SYNTAX_ERROR("Select alternative");
end if;

elsif (BYPASS(TOKEN_DELAY)) then
if (DELAY_STATEMENT) then
if (SEQUENCE_OF_STATEMENTS) then
null;
end if;
return (TRUE);
else
SYNTAX_ERROR("Select alternative");
end if;
else
SYNTAX_ERROR("Select alternative");
end if;
else (BYPASS(TOKEN_TERMINATE)) then
if (BYPASS(TOKEN_SEMICOLON)) then
return (TRUE);
else
SYNTAX_ERROR("Select alternative");
end if;
else
return (FALSE);
end if;
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;
else
SYNTAX_ERROR("Select entry call");
end if;
elsif (BYPASS(TOKEN_OR)) then
if (BYPASS(TOKEN_DELAY)) then
if (DELAY_STATEMENT) then
if (SEQUENCE_OF_STATEMENTS) then
null;
end if;
return (TRUE);
else
SYNTAX_ERROR("Select entry call");
end if;
else
SYNTAX_ERROR("Select entry call");
end if;
else
SYNTAX_ERROR("Select entry call");
end if;
else
return (FALSE);
end if;
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:
    end loop:
  end if:
end if:
if (BYPASS(TOKEN_RIGHT_PAREN)) then
  return (TRUE);
else
  SYNTAX_ERROR("Generic actual part");
end if:
end GENERIC_ACTUAL_PART.

-- GENERIC_ASSOCIATION --> GENERIC FORMAL PARAMETER ? EXPRESSION

function GENERIC_ASSOCIATION return boolean is
begin
if (GENERIC_FORMAL_PARAMETER) then
  null;
end if:
if (EXPRESSION) then
  -- check for generic actual parameter
  return (TRUE);
else
end if:
end GENERIC_ASSOCIATION.

end PARSER_2;
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):
    elsif (BYPASS(TOKEN_STRING_LITERAL)) then
      if (BYPASS(TOKEN_ARROW)) then
        return (TRUE):
      else
        return (FALSE):
      end if:
    else
      return (FALSE):
    end if:
  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
  return (FALSE);
end if:
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:
end GENERIC_TYPE_DEFINITION:
end if: -- if bypass(token brackets)
elif (BYPASS(TOKEN RANGE)) or else (BYPASS(TOKEN DIGITS))
or else (BYPASS(TOKEN DELTA)) then
if (BYPASS(TOKEN BRACKETS)) then
  return (TRUE);
else
  SYNTAX ERROR("Generic type definition");
end if:
elsif (BYPASS(TOKEN ARRAY)) then
if (ARRAY TYPE DEFINITION) then
  return (TRUE);
else
  SYNTAX ERROR("Generic type definition");
end if:
elif (BYPASS(TOKEN ACCESS)) then
if (SUBTYPE INDICATION) then
  return (TRUE);
else
  SYNTAX ERROR("Generic type definition");
end if:
else
  return (FALSE);
end if:
end GENERIC TYPE DEFINITION:
---------------------------------------------------------------------
-- PRIVATE TYPE DECLARATION --> limited ? private
function PRIVATE TYPE DECLARATION return boolean is
begin
  if (BYPASS(TOKEN LIMITED)) then
    null:
  end if:
  if (BYPASS(TOKEN PRIVATE)) then
    return (TRUE);
  else
    return (FALSE);
  end if:
end PRIVATE TYPE DECLARATION:
---------------------------------------------------------------------
-- SUBTYPE DECLARATION --> identifier is SUBTYPE INDICATION :
function SUBTYPE DECLARATION return boolean is
begin
  if (BYPASS(TOKEN IDENTIFIER)) then
    if (BYPASS(TOKEN IS)) then
      if (SUBTYPE INDICATION) then
        if (BYPASS(TOKEN SEMICOLON)) then
          return (TRUE);
        else
          SYNTAX ERROR("Subtype declaration");
        end if:
      else
        SYNTAX ERROR("Subtype declaration");
      end if:
    else
      SYNTAX ERROR("Subtype declaration");
    end if:
  else
    SYNTAX ERROR("Subtype declaration");
end if:
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 function;

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

-----------------------------------------------
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
      end if;
      if (BYPASS(TOKEN_ASSIGNMENT)) then
        if (EXPRESSION) then
          null:
        else
          SYNTAX_ERROR("Discriminant specification");
        end if;
      end if;
      else
        SYNTAX_ERROR("Discriminant specification");
      end if;
    end if;
    return (TRUE);
  else
    SYNTAX_ERROR("Discriminant specification");
  end if; -- if name statement
else
  SYNTAX_ERROR("Discriminant specification");
end if; -- if bypass(token_colon)
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 \* VARIANT PART ?
\ -- \ -- \ null.
function COMPONENT LIST return boolean is
begin
while (COMPONENT DECLARATION) loop
null.
end loop.
if (VARIANT PART) then
null.
elsif (BYPASS(TOKEN NULL)) then
if (BYPASS(TOKEN SEMICOLON)) then
null.
end if
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: -- if bypass(token semicolon)
else
SYNTAX ERROR("Component declaration");
end if: -- if expression statement
end if: -- if bypass(token assignment)
if (BYPASS(TOKEN SEMICOLON)) then
return (TRUE);
else
SYNTAX ERROR("Component declaration");
end if: -- if bypass(token semicolon)
else
SYNTAX ERROR("Component declaration");
end if: -- if subtype indication statement
else
SYNTAX ERROR("Component declaration");
end if: -- if bypass(token semicolon)
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.

 component declaration:

  -- VARIANT PART  -- >  case identifier is  'VARIANT  - 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_is)
          end if:  -- if bypass(token_identifier)
        end if:  -- if bypass(token_case)
      end if:  -- if variant statement
    end if:  -- if bypass(token_case)
else
  SYNTAX_ERROR("Variant part");
end if:

end VARIANT PART;

  -- VARIANT -- >  when CHOICE  
  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:
    end loop;
    if (BYPASS(TOKEN_ARROW)) then
        if (COMPONENT_LIST) then
            return (TRUE);
        else
            SYNTAX ERROR("Variant");
        end if:
    else
        SYNTAX ERROR("Variant");
    end if:
else
    SYNTAX ERROR("Variant");
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:
else
    return (FALSE);
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

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, NEXT_HEN);
CREATE_NODE( NEXT_HEN, LAST_RECORD);
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
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:
elsif (BYPASS(TOKEN CONSTANT)) then
if (CONSTANT TERM) then
return (TRUE);
else
SYNTAX_ERROR("Identifier declaration tail");
end if:
elsif (BYPASS(TOKEN ARRAY)) then
if (ARRAY TYPE DEFINITION) then
if (BYPASS(TOKEN ASSIGNMENT)) then
if (EXPRESSION) then
null:
else
SYNTAX_ERROR("Identifier declaration tail");
end if:
elsif (BYPASS(TOKEN SEMICOLON)) then
return (TRUE):
else
SYNTAX_ERROR("Identifier declaration tail");
end if:
elsif (NAME) then
if (IDENTIFIER TAIL) then
return (TRUE);
else
SYNTAX_ERROR("Identifier declaration tail");
end if:
elsif (IDENTIFIER TAIL) then
return (TRUE):
else
SYNTAX_ERROR("Identifier declaration tail");
end if:
elsif (NAME) then
if (IDENTIFIER TAIL) then
return (TRUE):
else
SYNTAX_ERROR("Identifier declaration tail");
end if:
elsif (NAME) then
if (IDENTIFIER TAIL) then
return (TRUE):
else
SYNTAX_ERROR("Identifier declaration tail");
end if:
elsif (NAME) then
return (FALSE);
end if:
eend 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:
         end if:
      end if:
   else
      SYNTAX_ERROR("Constant term"): end if:
   end if:
   if (BYPASS(TOKEN_SEMICOLON)) then
      null;
return (TRUE):
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:
elsif (BYPASS(TOKEN_ASSIGNMENT)) then
  if (EXPRESSION) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE):
    else
      SYNTAX_ERROR("Constant term");
    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:
  else
    SYNTAX_ERROR("Identifier tail");
  end if:
  if (BYPASS(TOKEN_RENAMES)) then
    if (NAME) then
      null:
    else
      SYNTAX_ERROR("Identifier tail");
    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:
  if (BYPASS(TOKEN_SEMICOLON)) then
    null:
  else
    SYNTAX_ERROR("Identifier tail");
  end if:
end if:
else
  return (TRUE):
end if:
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
  return (TRUE);
end if:
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
    HENRY_WRITE_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, NONE, LAST_RECORD);
  elsif (NOT PACKAGE_BODY_DECLARE) then
    HENRY_WRITE_DATA(LOCAL_DECLARE, DUMMY_LEXEME, IDENT_TYPE,
                      NONE, LAST_RECORD);
  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, DUMMY_LEXEME, 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_IDENTIFICke)) 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:

.ReactNode

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;
    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;
  end if;
  elsif (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 DESIGNSATOR 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 DESIGNSATOR;

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
    return (TRUE);
  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;
elif (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:
elsif (BYPASS(TOKEN_SEMICOLON)) then
  WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PROCALL_OR_DS, NONE, ASSIGN_POINTER);
  CREATE_NODE(NEXT_HEN, LAST_RECORD);
  return (TRUE): -- parsed a procedure call statement
else
  SYNTAX_ERROR("Assignment or procedure call");
end if:
else
  return (FALSE): -- if name statement
end if;
end ASSIGNMENT_OR_PROCEDURE_CALL:

-- LABEL --> << identifier >>
function LABEL return boolean is
begin
  if (BYPASS(TOKEN_LEFT_BRACKET)) then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      if (BYPASS(TOKEN_RIGHT_BRACKET)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Label");
      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) ?
function ENTRY_DECLARATION return boolean is
begin
  if (BYPASS(TOKEN_ENTRY)) then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      if (BYPASS(TOKEN_LEFT_PAREN)) then
        IS
      end if:
    end if:
  end if:
end ENTRY_DECLARATION;
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;
else
    SYNTAX_ERROR("Entry declaration");
end if;

if (BYPASS(TOKEN SEMICOLON)) then
    return (TRUE);
else
    SYNTAX_ERROR("Entry declaration");
end if;

if (BYPASS(TOKEN IDENTIFIER)) then
    return (FALSE);
else
    SYNTAX_ERROR("Entry declaration");
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;
            elsif (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;
end REPRESENTATION_CLAUSE;
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
    SYNTAX_ERROR("Representation clause");
end if; -- if name statement
else
    return (FALSE);
end if: -- if bypass(token_for)
end REPRESENTATION_CLAUSE:

-- RECORD_REPRESENTATION_CLAUSE --> 'at mod SIMPLE_EXPRESSION ?
-- ~NAME at SIMPLE_EXPRESSION range RANGES *
-- end record ;
function RECORD_REPRESENTATION_CLAUSE return boolean is
  begin
    if (BYPASS(TOKEN_AT)) then
      if (BYPASS(TOKEN_MOD)) then
        if (SIMPLE_EXPRESSION) then
          null:
        else
          SYNTAX_ERROR("Record representation clause");
        end if;
      else
        SYNTAX_ERROR("Record representation clause");
      end if;
    else
        SYNTAX_ERROR("Record representation clause");
    end if; -- if 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;
        else
          SYNTAX_ERROR("Record representation clause");
        end if; -- if ranges statement
      else
        SYNTAX_ERROR("Record representation clause");
      end if:
    else
        SYNTAX_ERROR("Record representation clause");
    end if;
  else
    SYNTAX_ERROR("Record representation clause");
  end if;
  return (FALSE);
end if: -- if bypass(token_for)
end RECORD_REPRESENTATION_CLAUSE:
else
  SYNTAX_ERROR("Record representation clause").
end if. -- if bypass(token_end)
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
  SYNTAX_ERROR("Record representation clause").
end if: -- if bypass(token_record_structure)
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_SCHEMA return boolean;
  function LOOP_PARAMETER_SPECIFICATION return boolean;
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:
      end if:
    else
      SYNTAX_ERROR("Array definition");
      end if:
    else
      SYNTAX_ERROR("Array definition");
      end if:
    end if:
  else
    SYNTAX_ERROR("Array definition");
  end if:
else
  return (FALSE):
end if:
end ARRAY TYPE DEFINITION:

-- CHOICE --> EXPRESSION .. SIMPLE EXPRESSION ?
--       --> EXPRESSION CONSTRAINT ?
--            --> others

function CHOICE return boolean is
begin
  if (EXPRESSION) then
    if (BYPASS(TOKEN RANGE DOTS)) then -- check for discrete range
      if (SIMPLE_EXPRESSION) then
        null:
      else
        SYNTAX_ERROR("Choice");
        end if:
      end if:
    elsif (CONSTRAINT) then
      null:
      end if:
    else
      -- if bypass token_range_dots
      return (TRUE);
    elsif (BYPASS(TOKEN OTHERS)) then
      return (TRUE):
    else
      return (FALSE):
    end if:
  end if:
end CHOICE:

-- ITERATION SCHEME -- while EXPRESSION
--                     -- for LOOP PARAMETER SPECIFICATION

function ITERATION SCHEME return boolean is
begin
  if (BYPASS(TOKEN WHILE)) then
NESTING METRIC(WHILE CONSTRUCT):
if (EXPRESSION) then
  return (TRUE);
else
  SYNTAX_ERROR("Iteration scheme");
end if:
elseif (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)
    end if:
    -- if token_in
    if (DISCRETE RANGE) then
      return (TRUE);
    else
      SYNTAX_ERROR("Loop parameter specification");
    end if:
    -- if discrete_range
  else
    SYNTAX_ERROR("Loop parameter specification");
  end if:
  -- if bypass(token_identifier)
else
  return (FALSE);
end if:
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
    return (TRUE);
  else
    return (TRUE);
  end if:
end EXPRESSION.
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 RELATION:

-- RELATION TAIL --> and then ? RELATION *
-- --> or else ? RELATION *
-- --> xor RELATION *
function RELATION TAIL return boolean is
begin
  while (BYPASS(TOKEN AND)) loop
    null;
  end if; -- if bypass(token then)
  if not (RELATION) then
    SYNTAX_ERROR("Relation tail");
  end if; -- if not relation statement
end loop:
  while (BYPASS(TOKEN OR)) loop
    null;
  end if; -- if bypass(token else)
  if not (RELATION) then
    SYNTAX_ERROR("Relation tail");
  end if; -- if not relation statement
end loop:
  while (BYPASS(TOKEN XOR)) loop
    null;
  end if; -- if bypass(token xor)
  if not (RELATION) then
    SYNTAX_ERROR("Relation tail");
  end if; -- if not relation statement
end loop:
return (TRUE);
end RELATION TAIL:

163
-- 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;
-- if not term statement
end loop;
return (TRUE);
else
SYNTAX_ERROR("Simple expression");
end if;
-- if term statement
elsif (TERM) then
while (BINARY_ADDING_OPERATOR) loop
if not (TERM) then
SYNTAX_ERROR("Simple expression");
end if;
-- if not term statement
end loop;
return (TRUE);
else
return (FALSE);
end if;
-- if bypass(token_plus) et al statement
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;
-- if simple expression statement
elsif (BYPASS(TOKEN_NOT)) then
if (BYPASS(TOKEN_IN)) then
if (RANGES) then
return (TRUE);
elseif (NAME) then
-- check for type mark
return (TRUE);
else
SYNTAX_ERROR("Simple expression tail");
end if;
-- if ranges statement
else
SYNTAX_ERROR("Simple expression tail");
end if;
-- if bypass(token_in) statement
end if;
else
SYNTAX_ERROR("Simple expression tail");
end if;
-- if bypass(token_not) statement
end if;
return (FALSE);
end if;
-- if bypass(token_not) statement
end SIMPLE_EXPRESSION_TAIL;
elsif (BYPASS(TOKEN_IN)) then
  if (RANGES) then
    return (TRUE);
  elsif (NAME) then
    -- check for type_mark
    return (TRUE);
  else
    SYNTAX_ERROR("Simple expression tail");
  end if;
else
  return (FALSE);
end if:
end 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;
    end loop;
    return (TRUE);
  else
    return (FALSE);
  end if:
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:
    end if:
    return (TRUE);
  elsif (BYPASS(TOKEN Absolute)) then
    if (PRIMARY) then
      return (TRUE);
    else
      SYNTAX_ERROR("Factor");
    end if:
  end if:
end FACTOR:
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

166
function CONSTRAINT return boolean is
begin
if (BYPASS(TOKEN RANGE)) then
  if (RANGES) then
    return (TRUE);
  elsif (BYPASS(TOKEN BRACKETS)) then
    -- check for < > when parsing
    return (TRUE);
  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;
else
  return (TRUE);
end if:
else
  return (FALSE).
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)
end if;
else
  return (FALSE); -- if discrete_range statement
end if
end INDEX_CONSTRAINT;

-- 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:
  end if:
  else
    return (TRUE); -- if bypass(token range dots)
  end if;
  else
    return (FALSE); -- if simple_expression statement
  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;
    if (BYPASS(TOKEN RIGHT_PAREN)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Aggregate");
    end if; -- if bypass(token_right_paren)
  else
  end if;
end AGGREGATE;
SYNTAX ERROR("Aggregate"):  
end if:  -- if component association statement  
else  
return (FALSE):  -- if bypass(token left paren)  
end if:  
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  
nul:  
else  
SYNTAX ERROR("Component association");  
end if:  
end if:  
return (TRUE);  
else  
return (FALSE):  -- if choice statement  
end if:  
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  
nul:  
else  
SYNTAX ERROR("Allocator").  
end if:  
end if:  
return (TRUE);  
else  
return (FALSE):  -- if subtype indication statement  
end if:  
end ALLOCATOR:
function NAME_TAIL return boolean is

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;
end NAME_TAIL;
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:
end NAME_TAIL:

-- LEFT PAREN NAME TAIL --> FORMAL PARAMETER? EXPRESSION _ EXPRESSION?
  FORMAL PARAMETER _ EXPRESSION? _ EXPRESSION? _
  ) 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: -- before the actual parameter
  end if: -- if formal parameter statement
  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:
    if [BYPASS(TOKEN RANGE DOTS)] then
      if not (EXPRESSION) then
        SYNTAX_ERROR("Left paren name tail"):;
      end if:
      -- if not expression statement
    end if:
    -- if bypass(token_range_dots)
    while (BYPASS(TOKEN COMMA)) loop
  end if: [BYPASS(TOKEN COMMA)]
if (FORMAL_PARAMETER) then
  null: -- 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 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;
  else
    SYNTAX_ERROR("Attribute designator");
  end if;
end if:
-- if bypass(token_right_paren) statement
else
  SYNTAX_ERROR("Attribute designator");
end if;
-- if expression statement
end if:
-- if bypass(token_left_paren) statement
return (TRUE);
elsif (BYPASS(TOKEN_DIGITS)) or else (BYPASS(TOKEN_DELTA)) then
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (EXPRESSION) then
      if (BYPASS(TOKEN_RIGHT_PAREN)) then
        null:
      else
        SYNTAX_ERROR("Attribute designator");
      end if;
    else
      SYNTAX_ERROR("Attribute designator");
    end if;
  else
    SYNTAX_ERROR("Attribute designator");
  end if:
  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:
-- if constraint statement
return (TRUE):
else
  return (FALSE):
end if:
-- if ranges statement
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 bypass(token when)
  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

174
if (BYPASS(TOKEN SEMICOLON)) then
    return (TRUE);
else
    SYNTAX ERROR("Goto statement");
end if:  -- if bypass(token semicolon)
else
    return (FALSE);
end if:  -- if name statement
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:  -- if bypass(token semicolon)
    else
        return (FALSE);
    end if:  -- if simple_expression statement
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:  -- if not name statement
        end loop;
        if (BYPASS(TOKEN SEMICOLON)) then
            return (TRUE);
        else
            SYNTAX ERROR("Abort statement");
        end if:  -- if bypass(token semicolon)
    else
        return (FALSE);
    end if:  -- if name statement
end ABORT STATEMENT.

-- RAISE STATEMENT --> NAME .
function RAISE STATEMENT return boolean is
begin
if (NAME) then
    null.
end if;
if (BYPASS(TOKEN SEMICOLON)) then
    return (TRUE);
else
    return (FALSE);
end if;
end RAISE STATEMENT:
end PARSER 3;

package PARSER_4
is
    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
-- 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:
    end if;
    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;
    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);
    elsif (BYPASS(TOKEN_ARROW)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Formal parameter");
    end if:
  else
    SYNTAX_ERROR("Formal parameter");
  end if:
  return (FALSE);
end if;
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);
  else
    return (TRUE);
  end if;
end if;
end SELECTOR;
elsif (BYPASS(TOKEN, ALL)) then
    return (TRUE);
else
    return (FALSE);
end if:
end SELECTOR:

end PARSER 4:
LIST OF REFERENCES


181
<table>
<thead>
<tr>
<th>No.</th>
<th>Copy</th>
<th>Name and Address</th>
</tr>
</thead>
</table>
| 1.  | 2    | Defense Technical Information Center  
Cameron Station  
Alexandria, Virginia 22304-6145 |
| 2.  | 2    | Library, Code 0142  
Naval Postgraduate School  
Monterey, California 93943-5002 |
| 3.  | 2    | Dr. Daniel L. Davis, Code 52DV  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California 93943 |
| 4.  | 1    | Dr. Bruce J. MacClennan, Code 52ML  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California 93943 |
| 5.  | 1    | Lieutenant Commander Paul Herzig  
578-C Schubrick Rd  
Monterey California 93940 |
| 6.  | 1    | Chairman (Code 52)  
Department of Computer Science  
Naval Postgraduate School  
Monterey, California 93943 |
| 7.  | 1    | Computer Technology Curricular Officer (Code 37)  
Naval Postgraduate School  
Monterey, California 93943 |
| 8.  | 1    | Chief of Naval Operations  
Director, Information Systems (OP 945)  
Navy Department  
Washington, D.C. 20350-2000 |
<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Address</th>
<th>Postal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.</td>
<td>Mr. Carl Hall</td>
<td>Software Missile Branch. Code 3922</td>
<td>93555-6001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Naval Weapons Center</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>China Lake, California</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>Mr. Robert Westbrook</td>
<td>CMDR 31C</td>
<td>93555-6001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Naval Weapons Center</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>China Lake, California</td>
<td></td>
</tr>
</tbody>
</table>

No. Copies: 1
END

10-81

DTIC