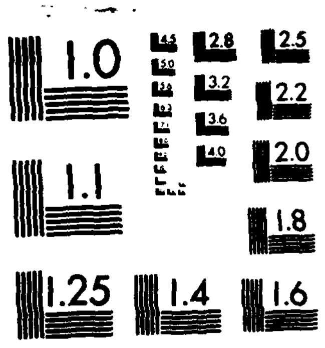
AD-A128 857 A DIANA-DRIVEN PRETTY-PRINTER FOR ADA(U) TARTAN LABS INC PITTSBURGH PA K J BUTLER ET AL. 22 FEB 83 TL-83-3 MDA903-82-C-0148 1/2

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

11

AD A128857

TARTAN LABORATORIES INCORPORATED

A Diana-Driven Pretty-Printer for Ada

Kenneth J. Butler
Arthur Evans Jr.

Prepared for

Ada Joint Program Office
801 North Randolph Street
Arlington Virginia 22203

Contract Number MDA903-82-C-0148

Prepared by

TARTAN Laboratories Incorporated
477 Melwood Avenue
Pittsburgh PA 15213

1983 February 22

TL 83-3

The effort which is reported herein was performed by Tartan Laboratories Inc. for the Ada Joint Program Office of the Department of Defense under contract number MDA903-82-C-0148 (expiration date: 28 February 1983). The Project Director for Tartan Laboratories is Arthur Evans, Jr.

83 06 01 004

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TL 83-3 ✓	2. GOVT ACCESSION NO. AD-A128857	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A DIANA-DRIVEN PRETTY PRINTER FOR ADA		5. TYPE OF REPORT & PERIOD COVERED Contract deliverable 0002AD
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kenneth J. Butler, Arthur Evans Jr.		8. CONTRACT OR GRANT NUMBER(s) MDA903-82-C-0148
9. PERFORMING ORGANIZATION NAME AND ADDRESS Tartan Laboratories Inc. 477 Melwood Ave. Pittsburgh PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office 801 North Randolph Street Arlington VA 22203		12. REPORT DATE 1983 Feb 22
		13. NUMBER OF PAGES viii + 108
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) DCASMA Pittsburgh 1610-S Federal Building 1000 Liberty Avenue Pittsburgh PA 15222		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Diana, Ada, programming language, pretty-printer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The design of a program PrettyPrint whose function is to pretty-print Ada programs is described. PrettyPrint takes as input a Diana representation of an Ada program. The intent of the design was to stress the Diana design.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The design of a program PRETTYPRINT whose function is to pretty-print ADA programs is described. PRETTYPRINT takes as input a DIANA representation of an ADA program. The intent of the design was to stress the DIANA design.

PREFACE

As part of its efforts in support of the ADA[1] programming language, the ADA Joint Program Office (AJPO) is deeply involved with the production of tools that will support ADA programmers. An ADA Programming Support Environment (APSE) is intended to include a rich collection of tools such as compilers, editors, pretty-printers, cross-reference generators, verifiers, and so on, to assist the programmer using the APSE and to ease the difficult task of creating the complex software required for DoD applications.

Early in the design of ADA compilers, the contractors involved chose DIANA as the notation to be used for communication between the components of the compilers they were building. DIANA is an abstract data type, designed with the intention that any object of the type represents *all* the information in an ADA source program, along with the results of lexical analysis, syntax analysis, and static semantic analysis. Further, AJPO has long realized that ADA is not the proper *lingua franca* to be used as a means of communication between the tools in an APSE and that DIANA is in fact an excellent notation for the purpose.

DIANA, like ADA, did not spring forth perfect from the pens of its creators. ADA has had the benefit of a long process of informed feedback from interested computer specialists to bring to its present state of excellence; DIANA requires similar nurturing and care for it to grow into a mature tool that properly meets the widely varying needs of its user community. AJPO has therefore contracted with TARTAN Laboratories to maintain DIANA and to improve it.

One issue which we at TARTAN have addressed in this effort has been to determine DIANA's suitability for an application other than a compiler. Although DIANA's original design contemplated such non-compiler applications, essentially all of the existing experience has been with compilers, so only compiler applications have had the opportunity to influence DIANA's continual development. Thus we have designed a program named PRETTYPRINT which is a pretty-printer for ADA that uses as input a DIANA representation of an ADA program. It is the design of PRETTYPRINT that is described in this document.

---

[1]Ada is a registered Trademark of the Ada Joint Program Office, Department of Defense, United States Government.

A pretty-printer, a tool that belongs in the tool chest of any group writing large amounts of code, enforces standards of layout on the page of programs written in the language. In the usual case, the pretty-printer reads one version of the program to be formatted and writes a new version, properly formatted. However, in an environment such as an APSE, it is more appropriate for the pretty-printer to take as its input a DIANA representation of an ADA program, rather than textual ADA.

It is important for the reader to keep in mind that the purpose of the design has been to exercise various aspects of DIANA, and *not* to build a better pretty-printer. This point is addressed more fully in Section 1.1 of this document.

This document assumes that the reader is familiar with the programming language ADA, as it is defined in Draft Revised MIL-STD 1815, the ADA Language Reference Manual dated July 1982 [2]. It also assumes knowledge of DIANA as defined in the DIANA Reference Manual (hereafter referred to as the DRM) dated February, 1983 [1].

The work reported herein has been supported by the ADA Joint Program Office under contract MDA903-82-C-0148. We thank AJPO for supporting the DIANA revision effort of which the the design of PRETTYPRINT is a part. We thank in particular Lt. Colonel Larry Druffel, the director of AJPO. Valuable assistance as Contracting Officer's Technical Representative was provided first by Lt. Commander Jack Kramer and later by Lt. Commander Brian Schaar; we are pleased to acknowledge them.

CHAPTER 1 INTRODUCTION

This report presents the design of PRETTYPRINT, a program designed to accept as input a DIANA representation of an ADA program and to produce as output a properly formatted textual version of the same ADA program.

This introductory chapter contains in succeeding sections the goals of the PRETTYPRINT design project, an overview of the design, some comments about DIANA, some comments about the design of PRETTYPRINT, and an overview of the rest of the document.

1.1. Goals of the Design of PRETTYPRINT

As stated in the Preface, the purpose of this design exercise has not been to investigate pretty-printers as such but rather to test and stress various aspects of the design of DIANA. Thus the design has been strongly influenced by the two goals of investigating DIANA's suitability for a non-compiler application and of stressing the design of the DIANA interface package. We have regarded these goals as being so important that we have sacrificed considerations of elegance of design and of efficiency and compactness in order to achieve them. They are considered in detail in the next two subsections.

1.1.1. Non-Compiler Application

The primary purpose for this design has been to exercise DIANA in an application domain other than a compiler. Most current users of DIANA employ it in compiler writing. However, inasmuch as DIANA was intended also to be a useful intermediate form for many of the tools to be found in an APSE, it is imperative that its design be stressed by using it in such an application. PRETTYPRINT is clearly one non-compiler tool that uses DIANA.

Crucial to the use of DIANA for driving any tool such as PRETTYPRINT is the DIANA design principle (presented in Section 1.1.1 of the DRM[1]) that DIANA must preserve the structure of the original source program. Our success in designing

[1]As stated in the Preface, we use DRM to refer to the Diana Reference Manual.

PRETTYPRINT has made it adequately clear that the source structure is preserved.

1.1.2. Exercise the DIANA Package

PRETTYPRINT should exercise the DIANA package. Since DIANA is an abstract data type, for which many possible concrete representations can be defined in any reasonable programming language, defining a specific concrete representation in ADA is a practical idea. In ADA such a definition would consist of a package whose specification contains

- the relevant type definitions (some private), and
- specifications of subprograms to access objects of the type.

The body of the package would contain the bodies of those subprograms and, likely, other types and subprograms.

Chapter 4 of the DRM contains the public part of a specification for such a package, named *Diana*; the private part of the specification and the package body are missing. The design of PRETTYPRINT described herein is specified by using ADA as a Program Design Language,[2] with the intention that the implementation will be written in ADA using that package.

The designers of the *Diana* package provided two distinct methods for traversing DIANA structures. As a deliberate policy, this design uses both such ways so as to insure that both are properly designed and adequate for the purpose.

- The first tree walk, *WALK1*, uses the general tree traversing operations: *ARITY, SON1, etc.* The function *ARITY* applied to a DIANA node returns the structure of the node, which is (essentially) the number of offspring of the node. The SON_k subprograms then provide access to the relevant offspring. These operations permit a program to traverse a tree structure without taking specific cognizance of the nature (other than number of offspring) of each node traversed.

- The second tree walk, *WALK2*, uses the attribute-specific operations such as *AS_ACTUAL, AS_ALIGNMENT, etc.* In this method, on reaching a node one must determine (with function *KIND*) the nature of the node, and then use the appropriate attribute-accessing functions to explore the children.

The intent has been to test the adequacy of the design of package *Diana*.

[2] *This use of Ada as a Program Design Language* (PDL) is discussed in Section 1.4.4 on page 23.

As it turns out, we found it appropriate for this project to augment DIANA, in a manner anticipated in the DRM. Thus the package used here is *PP_Diana*, rather than *Diana* of Chapter 4 of the DRM. See Section 1.2.2.3 on page 12 of this document for details of what was done and why.

1.2. Design Overview

We have found it convenient to think separately about two aspects of pretty-printing: *reconstruction*, and *formatting*. The first aspect involves reconstructing the characters that make up the source text, without concern for how the characters are to be laid out on the page; the second aspect involves making all decisions concerning page layout and then carrying them out. Although this distinction is often useful in the discussions in this document, it turns out that it is not very apparent in the code itself, whose modularity is designed from a different viewpoint.

All of PRETTYPRINT's work is performed by the three major modules of the program: *WALK1*, *WALK2*, and *FORMAT*, described briefly in the following subsections and then in more detail in the rest of this report. Part of the structure of the so-called "main program" and the specification parts of modules *WALK1* and *WALK2* are shown in Figure 1-1 on page 8; the specification of *FORMAT* is in Figure 1-4 on page 16. The modules are listed in the order in which they are discussed in this chapter; of course, they would have to be presented to an ADA compiler in a different order.

Entries in the package *PP_Diana* are used for all accessing of the DIANA structure; this package is described in Section 1.2.2 on page 9.

1.2.1. The Main Program

For the sake of convenience, we have assumed a main program, here the procedure *Main*, which is called somehow by "the operating system", a concept not further discussed. We show only that part of *Main* that calls the routines that perform pretty-printing.

We assume that a specific instance of a DIANA structure is specified as being the one to be pretty-printed, again *via* some means not here discussed. The otherwise-unspecified function *Get_PP_Tree* returns this structure. It is important to note that the program as written assumes that the tree returned is one whose

```
-- Main program.  Assume that this is called by "the operating system".

with Get_PP_Tree; use Get_PP_Tree;      -- function that reads PP_Diana
with PP_Diana; use PP_Diana;            -- DIANA package, for pretty printing
with WALK1, WALK2;                      -- routines to walk the trees

procedure Main is                       -- main program
    T: TREE;                            -- DIANA tree to be pretty-printed
begin
    -- Negotiate with user to determine specific tree to be printed.
    T := Get_PP_Tree(...);              -- The tree to be printed is in T.
    WALK1.WALK1(T);                     -- perform first walk
    WALK2.WALK2(T);                     -- perform second walk
end Main;

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

-- Package to perform the first walk over the tree.

with PP_Diana; use PP_Diana;            -- DIANA package, for pretty printing

package WALK1 is
    procedure WALK1(T: in out TREE);
end WALK1;

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

-- Package to perform the second walk over the tree.

with PP_Diana; use PP_Diana;            -- DIANA package, for pretty printing

package WALK2 is
    procedure WALK2(T: in TREE);
end WALK2;
```

Figure 1-1: Top Level ADA Modules

type is *PP_Diana.Tree*, and not *Diana.Tree*[3]. There are of course other possibilities for acquiring the program's input, such as the following:

- The program could acquire (somehow) a structure of type *Diana.Tree* rather than *PP_Diana.Tree*, and then invoke a suitable subprogram to transform it to the desired type.

- If the DIANA structure to be pretty-printed exists in DIANA external form, the DIANA reader that transforms external DIANA into internal might be modified to create the additional attributes, though without values.

- *WALK1* could be reformulated to take an in argument of type *Diana.Tree* and produce an out argument of type *PP_Diana.Tree*.

We do not pursue this point further as it is not relevant to the purpose of th project.

Procedure *WALK1* is called with one argument: a tree of type *PP_Diana.Tree*. It walks over that structure and modifies it, calculating and storing values for certain attributes specific to pretty-printing.

Next procedure *WALK2* walks over the resulting tree, emitting the desired output as it goes. It calls entries in package *FORMAT* to perform storing of output. All formatting decisions are made in *WALK2*.

1.2.2. The DIANA Structure

As suggested above, the input to the program is not an object of type *Diana.Tree* but rather one of type *PP_Diana.Tree*. The distinction between these is now presented. Section 1.2.2.1 describes first the ADA package *Diana*, the package described in Chapter 4 of the DRM. Next Section 1.2.2.2 addresses the issues involved in using IDL to describe a new structure in terms of one already defined. Finally, in Section 1.2.2.3 the special version of DIANA relevant to pretty-printing is presented.

1.2.2.1. The Package *Diana*

Chapter 4 of the DRM provides the specification part of an ADA package that provides access to a concrete representation of DIANA. Omitting most details, a

[3] This distinction is clarified in Section 1.2.2

sketch of the package is shown in Figure 1-2 on page 11[4]. It defines and makes available the following names:

type TREE	An object of this private type is a node of the DIANA structure.
type SEQ_TYPE	An object of this private type is a sequence of nodes of the same class.
type NODE_NAME	This is an enumeration type providing an enumeration literal for each kind of DIANA node.
function MAKE	This function creates and returns a DIANA node of the kind which is its argument. Note that it is overloaded so as also to be able to create an empty list.
procedure DESTROY	This procedure indicates that a node is no longer required.
function KIND	Given a node, this function returns its node-kind.
type ARITIES	This enumeration type provides a literal for each number of structural children a node might have.
function SON_k	For k = 1, 2, 3, each such function returns the k^{th} offspring of a node.
procedure SON_k	For k = 1, 2, 3, each such procedure stores a new k^{th} offspring of the node.
list processing	A collection of functions and procedures implement the usual list-processing primitives.
attributes	For each possible attribute, there is a function to return the value of that attribute at a node, and a procedure to store a new value for the attribute.

Although subprograms *MAKE* and *DESTROY* just listed and subprograms *INSERT* and *APPEND* mentioned in Figure 1-2 are not used by program PRETTYPRINT, for completeness they are listed in this discussion.

---

[4]For the sake of completeness, the package lists several subprograms that are not used by PrettyPrint. Such subprograms are MAKE (which appears twice, overloaded), DESTROY, INSERT, and APPEND.

```
package Diana is
    type Tree is private;              -- a Diana node
    type SEQ_TYPE is private;          -- sequence of nodes

    type NODE_NAME is                  -- enumeration class for node names
     (  ...                            -- about 160 different node types
      );

    -- Tree constructors.
    function  MAKE     (c: in NODE_NAME)      return TREE;
    procedure DESTROY  (t: in TREE);

    function  KIND     (t: in TREE)           return NODE_NAME;

    -- Tree traversers from the Ada Formal Definition.

    type ARITIES is (nullary, unary, binary, ternary, arbitrary);

    function  ARITY    (t: in TREE)           return ARITIES;
    function  SON1     (t: in TREE)           return TREE;
    procedure SON1     (t: in out TREE; v: in TREE);
    function  SON2     (t: in TREE)           return TREE;
    procedure SON2     (t: in out TREE; v: in TREE);
    function  SON3     (t: in TREE)           return TREE;
    procedure SON3     (t: in out TREE; v: in TREE);

    -- Handling of list constructs.
    function  HEAD    (l: in SEQ_TYPE)      return TREE;       -- LISP CAR
    function  TAIL    (l: in SEQ_TYPE)      return SEQ_TYPE;   -- LISP CDR
    function  MAKE                          return SEQ_TYPE;
                                                  -- return empty list
    function  IS_EMPTY (l: in SEQ_TYPE)     return BOOLEAN;
    function  INSERT (l: in out SEQ_TYPE;
                      i: in TREE)           return SEQ_TYPE;
                                                -- inserts i at start of l
    function  APPEND (l: in out SEQ_TYPE;
                      i: in TREE)           return SEQ_TYPE;
                                             -- inserts i at end of l

    -- Handling of LIST attribute of list constructs.
    procedure  LIST     (t: in out TREE; v: in SEQ_TYPE);
    function   LIST     (t: in TREE)        return SEQ_TYPE;

    -- Structural Attributes.

    procedure AS_ACTUAL             (t: in out TREE; v: in TREE);
    function  AS_ACTUAL             (t: in TREE) return TREE ;      -- assoc
    ...
    -- followed by functions and procedures for about 100 attributes .....

private

    -- To be filled in...

end Diana;
```

Figure 1-2: Sketch of the DIANA Package

1.2.2.2. IDL and Refinements

DIANA is defined in a notation called IDL [4], a notation designed expressly for describing structures to be used as interfaces between software components. The designers of IDL foresaw that users of an abstract type (such as DIANA) might require another abstract type that was *almost* the same as the first, but slightly different. IDL therefore provides the concepts of *refinement* and derivation, methods for defining such a structure by listing only the differences from some already defined IDL structure.

Details about refinements and derivations are to be found in Section 2.3 of the IDL Reference Manual [4]. However, enough information about the concept for present purposes may be found in Appendix II of the DRM, in which the Abstract Parse Tree (APT) is defined by derivation from the DIANA structure. Derivation is a more general process than the refinement used here, as derivation permits both additions and deletions whereas refinement permits only additions.

1.2.2.3. The Package *PP_Diana*

For the purposes of the design of PRETTYPRINT, it is useful to have three additional attributes at some of the nodes. These provide a place to record data gathered during the first tree walk so that they are available during the second.

In the present case, we define a new abstraction, *PP_Diana*, by refinement of DIANA. This new type is like *DIANA* but has added to it three new attributes, listed below. Then the ADA package *PP_Diana* is just like the ADA package *Diana* except that the enumeration type *NODE_NAME* contains three new names and there are three new functions and three new procedures to deal with the new attributes.

The three new attributes are as follows:

pp_chars	This attribute holds the number of characters required to print the complete ADA structure at the node and its descendants. The computation ignores all considerations of formatting, assuming that all the code fits on one line and that lexemes are separated (when necessary) by a single space.
pp_max_chars	Present on any node that is a list, this attribute holds the maximum number of characters required to print any element of the list. The calculation follows the same conven-

tions as for *pp_chars*.

pp_indent	This attribute holds the total number of extra indentation levels required to print this node. For all leafs the value is zero. For a procedure body, for example, it is one greater than the maximum required for any declaration, statement, or exception in the body.

A complete listing of the refinement that specifies *PP_Diana* is in Appendix C.

1.2.3. The Package *WALK1*

Note from Figure 1-1 on page 8 that the package *WALK1* makes available externally only a single procedure, also named *WALK1*. This procedure walks over the structure that is its input, calculating and storing values for the three attributes that are needed by PRETTYPRINT, as described in the preceding section. The process is described in detail in Section 4.2 on page 68.

It computes and stores at each node the indentation level required to print that node, based (essentially) on the nesting depth of such ADA control structures as packages and procedures and compound statements whose bodies are indented from the surrounding text.

It computes and stores at each node the number of characters required to print the node, ignoring formatting requirements. These data are needed for the second pass over the tree producing output.

For each sequence node (*i.e.*, a DIANA "Seq of" node), it computes and stores the maximum number of characters required by any element of the sequence.

The body of *WALK1*, considered in detail in Section 4.2 on page 68, consists of first some declarations of interest, then functions and procedures to do the work, and finally the body of procedure *WALK1*. A sketch of the package body for *WALK1* is in Figure 1-3 on page 14. The pass walks the tree using the general tree traversing operations of the package *PP_DIANA*.

1.2.4. The Package *WALK2*

Package *WALK2*, like package *WALK1*, exports only a single procedure. Again, it is a procedure that performs a single walk over the structure. Procedure *WALK2* walks over the tree, calling entries in package *FORMAT* to perform

```
-- Package to perform the first walk over the tree.

package body WALK1 is

    Son_Count : ARITIES range unary .. ternary;

    Nest: constant array (NODE_NAME,Son_Count) of NATURAL
          := ( ... );
    Char : constant array (NODE_NAME) of NATURAL
          := ( .. );

function Max(X: in NATURAL; Y: in NATURAL)
       returns NATURAL is separate;

procedure ListWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL;
                   MaxLength : out NATURAL);

procedure NodeWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL);

procedure Walk1(T: in out TREE) is separate;

procedure ListWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL;
                   MaxLength : out NATURAL) is separate;

procedure NodeWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL) is separate;

end WALK1;
```

Figure 1-3: Outline of Package Body *WALK1*

output. All formatting decisions are made in *WALK2*.

The structure of *WALK2* is similar to that of *WALK1*, although there are many more functions and routines. It includes the line

```
    with FORMAT; use FORMAT;
```

to provide access to the subprograms in package *FORMAT* that interface to the output. This pass, unlike the previous one, uses the specific attribute accessing operations from package *PP_DIANA* to traverse the tree.

Details about this process are found in Section 4.3 on page 78.

1.2.5. The Package *FORMAT*

The formatting decisions made in *WALK2* are implemented by calling entries in package *FORMAT*. The specification of package *FORMAT* is in Figure 1-4 on page 16. Typical formatting decisions include when to break a line, how many spaces to indent a new line, and so on.

FORMAT maintains a buffer into which to collect together a line of characters for outputting.

Because line indentation is such an important concept in pretty-printing, *FORMAT* provides considerable services for it. It maintains a stack of indentations, and at any moment each line produced is indented by the number of spaces shown at the top of the stack. Entering a program area (such as a procedure body) requiring additional indentation implies pushing a new entry onto the stack, using the procedure *Indent*; leaving such a scope requires calling *Undent* to pop the stack.

The indentation stack is used also to deal with overflow lines. For example, suppose a statement is about to be printed which cannot fit on the line. (Code in *WALK2* knows how much space is required to print the statement from the *pp_chars* attribute on the statement node, and it knows how much space remains on the line by calling function *Remaining* in *FORMAT*.) In such a case the *FORMAT* entry *SetIndent* is called to set indentation as appropriate for the extra characters of the statement.

With this background in mind, here is a brief description of each external entry into package *FORMAT*. The term *indentation constant* refers to the fixed amount each nested region of text is indented beyond the surrounding region.

constant LineLength	This is the number of characters in the output line.
type Column	An object of this type is an integer between zero and *LineLength*.
function Position	This function returns the position on the line of the next character to be stored.
function Remaining	This function returns the number of available characters on the line. (It is always *LineLength* - *Position*.)

```
-- Package that provides operations to format reconstructed Ada source

package FORMAT is

    LineLength :
        constant POSITIVE := 120;      -- length of the output line
    type Column is
        range 0..LineLength;           -- position on the line

    -- There are four procedures to append text to the output buffer

    procedure AddText(Text: in String);
    procedure ResForm                  -- add Ada reserved words
        (Text: in String);
    procedure ComForm                  -- add comments
        (Text: in String);
    procedure IdentForm                -- add program identifiers
        (Text: in String);

    -- There are two function to return status of the output buffer

    function Remaining return Column;  -- unused characters in buffer
    function Position return Column;   -- used characters in buffer

    -- this procedure creates a line break

    procedure NewLine;

    -- this procedure sets the indentation increment

    procedure SetIncrement(Depth: in POSITIVE);

    -- three procedures provide indentation operations

    procedure Indent;                  -- increment from last indentation
    procedure Undent;                  -- revert to previous indentation
    procedure SetIndent(Pos: in Column);
                                       -- set indentation to Pos
end FORMAT;
```

Figure 1-4: Package *FORMAT* Specification

procedure Indent

This procedure pushes onto the indentation stack the next standard indentation. That is, it pushes a number that exceeds the last entry by the indentation constant.

procedure SetIndent

This procedure pushes its argument onto the indentation stack.

procedure Undent

This procedure pops the indentation stack, restoring the indentation to the previous value.

procedure NewLine

This procedure finishes the current line and outputs it, and then stores the indentation for the next line.

procedure AddText

This procedure is used to store text into the output.

procedure ResForm

This procedure is used to store a reserved word into the output. It formats the word as appropriate. For example, reserved words might (as in this document) be printed with boldface type.

procedure ComForm

This procedure is used to store comments into the output, formatting words as appropriate. For example, comments might be printed with italic type.

procedure IdentForm

This procedure is used to store a programmer identifier into the output.

1.3. Observations about DIANA

In this section we record some observations we have made about DIANA during the course of this design. It is these comments that are the principal output from this study. With the exception of the point raised in the next subsection concerning DIANA's handling of ADA comments, we have concluded that DIANA's design, as stressed by the design of PRETTYPRINT, is adequate.

1.3.1. Handling ADA Comments

The design of PRETTYPRINT has revealed a serious problem in DIANA's handling of comments in the ADA text. Obviously this problem is of no concern to writers of compilers and most other tools in APSE, since it affects only tools which are concerned with the exact placement of comments in ADA source text. Nonetheless, it is a problem which requires a solution.

The problem is that there is not an adequate way to determine the correct piece of ADA source text with which to associate a comment. It is instructive to note how this problem arose in our design effort. As outlined in Section 3.1.4 on page 47, PRETTYPRINT's handling of comments is rather poor. At first we felt that we were doing poorly because we had given the problem inadequate attention[5]. However, on further reflection we realized that PRETTYPRINT cannot possibly put the comments 'where they belong' because it has no way of knowing where in fact they do belong.

To a first approximation, the problem is that PRETTYPRINT has no way to know how the creator of the DIANA placed the comments. However, the real problem is that there exist no comment-placement standards to be obeyed by DIANA creators. *Even more seriously*, there are sensible places in ADA text at which to place comments for which there is no DIANA node to which to attach the comment.

To see these problems, consider the following example of ADA code. The ADA code was copied directly from one of the examples in Section 6.1 of the ADA LRM and then reformatted and commented to make several points:

```
procedure                              -- [1] Print a header.
    PRINT_HEADER                       -- [2] It is called whenever ...
     (                                 -- [3] Its parameters are ...
        PAGES  : in NATURAL;           -- [4] number of pages
        HEADER : in LINE               -- [5] the line to print
                 := (1..LINE'LAST => " ");
        CENTER : in BOOLEAN            -- [6] center it?
                 := TRUE
     );
```

(The numbers in [..] serve to identify the comments in the following discussion.) Ideally, it should be possible for the compiler Front End (or other DIANA creator) to leave enough information in the tree so that it would be

[5] It is completely consistent with our design goals as described in Section 1.1 on page 5 to give little attention to such a problem.

possible for a program like PRETTYPRINT to recreate this program as here displayed. For some of the comments, such as [4], [5] and [6], there is no problem in doing so--the comment could be the value of the *lx_comment* attribute on the in node that is the formal parameter. However, it does not appear possible to find two different places in the structure for comments [2] and [3]. Also, PRETTYPRINT could not possibly know how to place comment [1] unless it was aware of the conventions used in creating the DIANA.

Although this example is perhaps slightly contrived, it suggests a real problem. First, there exist programmers who might well write comments in the style suggested here. Such a programmer who went to the trouble to insert these comments would be properly dismayed to discover that they were unsatisfactorily rearranged by tools in an APSE. Second, however, and much more serious, tools are coming into existence which care very much how comments are arranged.

One such tool is ANNA [3], a tool which permits a user to decorate an ADA program with annotations which are recognized by an ANNA processor. Syntactically, all such annotations are ADA comments and would be ignored by any conforming ADA compiler. However, an ANNA processor takes cognizance of comments starting with `--:' or `--|', interpreting them as input for certain kinds of program analysis. Further details are not relevant here.

Although one could design an ANNA processor to take ADA text as its input, such an approach is inconsistent with AJPO's intent for tools in an APSE. It is much more appropriate for the tool to use instead a DIANA representation of an ADA program. However, because the placement of the special comments has semantic implications for ANNA, it could do so only if it were possible to derive from the DIANA the original placement of ADA comments in the source program. For such a tool to be transportable from one APSE to another with a different ADA-to-DIANA transformer, it is necessary that the DRM specify adequately the details of placement of ADA comments in the DIANA tree.

Although our analysis of this problem in connection with building PRETTYPRINT suggests that further DIANA design in connection with ADA comments is *desirable*, the problems faced by the builder of a tool such as ANNA suggest that such redesign is *required*. As ADA matures and sophisticated tools of the type suggested by ANNA become available for inclusion in APSEs, it will become more and more necessary to address and solve this problem.

1.3.2. DIANA Normalizations

The DIANA Reference Manual, following the lead of the ADA Formal Definition, uses the term *normalization* to refer to certain arbitrary decisions made in constructing the DIANA representation that are, in effect, losses of information. For example, one may elect to ignore case distinctions in the spelling of reserved words and program identifiers. Some of these normalizations have an impact on source reconstruction. In the DRM, this topic is introduced in Section 3.1.3 and then discussed at greater length in Appendix III, Section 3. The present discussion is keyed to the latter.

ADA permits an optional identifier following the reserved word end in certain contexts, such as a block body, subprogram body, and so on; if the identifier is present, it must match an identifier at the beginning of the context. As there is no provision in DIANA for recording whether or not this identifier is present, a program such as PRETTYPRINT that does source reconstruction must either always include the labels or always omit them. We have chosen to include them.

In formal parameter declarations for subprograms, the mode in is optional and is not recorded in the DIANA. We have chosen always to include it.

DIANA does not require that extra spaces between lexical tokens be preserved.

Variant spelling of an identifier, as for example "FOO" and "Foo" and "foo", need not be recorded in DIANA.

Alternate writings of numeric constants need not be preserved. For example, in

```
2   002   0_0_2
2#1111_1111#   16#FF#   016#0FF#   255
12e1   1.2e2   0.12e+3   01.2e02
```

all the values on each line are represented identically in the DIANA and so are reconstructed identically. This issue is essentially the same as the variant spelling of identifiers; DIANA does not require that variations be preserved.

One normalization present in ADA-80 is absent in ADA-82, the distinction between an infix and prefix form of an operator. For example, in the earlier version of DIANA based on ADA-80, each of

```
... A + B ...
... "+"(A, B) ...
```

is represented by the same structure. However, the conformance rules of

ADA-82 as expressed in Section 6.3.1 of the ADA LRM require that the distinction be preserved. For this reason, DIANA now has the boolean attribute *sm_prefix* to record which was used. See Section 3.3.4 of the DRM.

1.3.3. Diana Problem Areas

Although DIANA was designed to preserve the structure of the original source program, and indeed the design of PRETTYPRINT shows that the structure is in fact preserved, we discovered that there are some DIANA nodes that present special problems. Generally, these are nodes where the source to be reconstructed cannot be determined without knowing the context in which the node appears.

Attribute	When an attribute[6] appears as the description of the range in a range constraint, the attribute should be preceded by the ADA reserved word "**range**"; in all other cases the attribute appears by itself.
Id_s	When a sequence of identifiers is the list of labels before a statement, each identifier should be surrounded by the special brackets '<< >>' and commas should not appear between the identifiers; in all other cases the identifiers appear without any bracketing and are separated by commas.
Task Type	When a type declaration defines a task type, the ADA reserved word "**type**" should be preceded by the ADA reserved word "**task**"; in all other cases there is no text precoding it.
Header	When constructing the text for nodes in the class *HEADER*, which corresponds to the header part of a function or procedure declaration, it is necessary to have the name of the subprogram, so that it can be printed in the declaration. The identifier must be passed down from the parent node.
Loop and Block	When **loop** or **block** appear as the child of a **named_stm** node (that is, a **loop_identifier** or **block_identifier** has been specified), then the identifier must appear again after the ADA reserved word "**end**" and before the semicolon. The identifier must be passed down from the parent node. In all other cases no identifier appears after "**end**", and there is no identifier to be passed from a parent node.

[6]In this context, the term *attribute* refers to an Ada attribute (such as 'FIRST), and not an attribute of a node in Diana.

Note that in all cases just discussed, information needed for source reconstruction is not found exactly where it is needed. However, in all cases it is easy to code *WALK2* so as to provide the information. Alternatively, additional attributes could have been defined where needed and calculated in *WALK1*. However, the essential adequacy of DIANA is shown in that there are two straightforward ways to deal with the problem.

1.4. Comments about the Design

Here are some observations of interest about the design.

1.4.1. Formatting Decisions

All decisions about how the ADA text is to be formatted are, in effect, "hard-wired" into the code of *WALK2*. Largely, we have followed the lead of the ADA LRM in deciding how to display ADA code.

Were our purpose to design a production pretty-printer, for example for inclusion in an APSE, we would have provided an interface to let the user (or at least the local system maintainer) to parameterize the layout. However, such functionality, though clearly desirable, is not relevant to the goals of the project as described in Section 1.1 *on page 5*.

1.4.2. Assumptions Made in the Design

We have made certain assumptions in the design of PRETTYPRINT, assumptions that the reader should be aware of.

We have consistently assumed that the DIANA structure supplied is correct. An implementation might be made more robust by the addition of suitable checking.

We have assumed that no token is longer than the output line. It is not clear in any case what to do if this assumption fails.

We have assumed that the nesting of the underlying program is not "too deep", in the sense that excessive indentation would leave not enough space on a line for meaningful amounts of text. (Although the program does not fail in any unpleasant way, its treatment of the situation would not be suitable for a production environment. See the discussion of procedure *Indent* in Section 3.3.2 on page 58.)

We have assumed that characters are all of the same width.

We have assumed the existence of a function *Length* that determines of any token (symbol, number or operator) the number of characters required to print it. If the values of attributes *lx_symrep* and *lx_numrep* are strings, of course, then the implementation of *Length* is quite simple.

1.4.3. Missing Parts of the Design

Certain parts of a complete pretty-printer design are missing from this document. These omissions are all consistent with our limited goals as described in Section 1.1 and are recorded here merely for completeness.

A proper pretty-printer should encapsulate all the formatting decisions so as to permit the user (or at least the system maintainer) to change the formatting decisions. In the present design, all such decisions are embedded in the code in *WALK1*. See Section 1.4.1 on page 22.

PRETTYPRINT's handling of comments is quite weak. This fact reveals a problem area in the design of DIANA, discussed in Section 1.3.1 on page 18.

It would be appropriate to recognize certain pragmas that control listing, such as

```
pragma list;     -- turn listing on or off
pragma page;     -- eject paper
```

and perhaps others of our own design.

Page layout is quite weak. Certain structures (such as subprograms) should have surrounding white space, and there should be some control of where page breaks occur.

1.4.4. ADA as a Program Design Language

An interesting recent development in the ADA world has been the use of ADA as a Program Design Language (PDL). We have followed the lead of others in doing so. As the technique is not yet well specified, it seems in order to explain just what we have chosen to do.

ADA by the nature of its design separates the *specification* of a program from its *definition* (which in ADA terms is the body). Taking advantage of this separation, we have provided complete package specifications for the various modules of PRETTYPRINT. In addition, we have sketched the code that would go into the package bodies, with no attempt to provide all details. Our goal has been to provide adequate detail to permit an intelligent implementor to complete the implementation.

1.5. Comments about this Document

The remainder of this document specifies in detail the design (but not the implementation) of PRETTYPRINT. The major thrust of the technical presentation is in the next three chapters. Chapter 2 presents in detail the issues involved in source reconstruction, ignoring temporarily the problem of formatting; and then Chapter 3 addresses the issues involved in implementing pleasing formatting decisions. Finally, Chapter 4 shows how the solutions to these two problems are implemented.

Four appendices present details of the modules that carry out the work. Appendices A and B discuss the two tree traverses, which are carried out in packages *WALK1* and *WALK2*, respectively. Appendix C discusses the changes required in the DIANA structure to accommodate the needs of the pretty-printing process. Appendix D discusses the ADA package *FORMAT* which implements the formatting processes.

To aid the reader, consistent typographic conventions are adhered to in referring to objects and syntactic types in ADA and DIANA. The conventions are as follows:

Entity	Convention	
ADA reserved word	**begin**	**end**
ADA identifier	*FORMAT*	*TREEWALK*
DIANA classes	***OBJECT_DEF***	***TYPE***
DIANA nodes	**constant**	**record**
DIANA attributes	*lx_srcpos*	*as_object_def*

Note the distinction between DIANA, the name of the abstract data type, and *Diana*, the name of the ADA package.

Because we cite frequently certain literature relevant to ADA and DIANA, we use consistently following abbreviations:

DRM	The DIANA Reference Manual, [1].
ADA LRM	The ADA Language Reference Manual, [2].

IDL	The IDL Formal Description. [4].

Note that the version of the DRM cited is the latest revision. There are changes from earlier revisions that are significant to this document.

CHAPTER 2
SOURCE RECONSTRUCTION

In presenting the PRETTYPRINT design, we consider separately two aspects of pretty-printing: *source reconstruction* and *formatting* of the reconstructed source. Although we make a clear-cut distinction in the discussions in this document, the dichotomy is blurred in the completed design, but is nonetheless present. For the purposes of analyzing DIANA in a new application domain, source reconstruction is the *more* important of the two aspects.

In this chapter we consider only the requirements for reproducing *unformatted* ADA source. The formatting issues involved are discussed separately, in Chapter 3 on page 43. The complete PRETTYPRINT design, incorporating formatting into the source reconstruction, is elaborated in Chapter 4 on page 63.

One of the goals of the DIANA design is the ability to reconstruct the ADA source used to create an instance of DIANA. This chapter illustrates that this goal is met, with a small number of exceptions. The exceptions are normalizations that *are necessary during reconstruction. These are described in* Section 1.3.2 on page 20.

We make the following claim: **unformatted ADA source can be reconstructed in one pass over the DIANA tree.** Furthermore, with one exception, the only attributes necessary to reconstruct the source are the lexical (*lx_*) attributes, which describe the representation of identifiers, operators, and numeric literals; and the structural (*as_*) attributes, which describe the structure of the DIANA tree.

The single semantic attribute, *sm_prefix*, is required to differentiate between infix and prefix uses of operators. This distinction is required by the semantics of ADA[1]. Without this requirement, the form of all operators could be normalized and source could be reconstructed without any semantic attributes.

The reconstruction pass can be performed with a simple recursive descent tree walk. In general the text at a node is generated independently of its

[1]The conformance rules for default formal parameters, Ada LRM Section 6.3.1 in particular, require this distinction.

parents, and includes the text for each of its descendants in order (the few exceptions, most notably subprogram_decls and task types are described in Section 2.2 on page 32 below).

2.1. General Treatment of Nodes

In this section we show how pretty-printing might be done for some simpler language. Our purpose is to illustrate the techniques used in PRETTYPRINT to pretty-print ADA. Suppose the internal form of this simple language (analogous to DIANA) has two types of nodes:

inner	nodes with structural attributes: in DIANA the only lexical attributes associated with these nodes are *lx_srcpos*, source position, and *lx_comments*, the comment associated with this node.
leaf	nodes with no structural attributes; these nodes may have a lexical attribute of interest. In DIANA a leaf node may have the attribute *lx_symrep* or *lx_numrep*, or may have no attributes at all.

The canonical inner node has one, two, or three descendants, accessed as structural attributes. The IDL representation of a node with three offspring is

```
inner  =>    child1 : TYPE1,
             child2 : TYPE2,
             child3 : TYPE3;
```

To further the analogy, we assume the existence of a package similar to package *Diana* that defines a type TREE such that leaf and inner nodes belong to the type. In addition, this package provides functions ("operations" on type TREE) that provide access to the structural attributes. For convenience we name the functions in this package CHILD1, CHILD2, and CHILD3. Each of these functions takes one argument, a node of type TREE, and returns a node of type TREE that is the corresponding structural attribute of the node. Using the ADA package *TEXT_IO*, as described in Section 14.3 of the ADA LRM, the canonical procedure for creating the source associated with this node would then be:

```
-- the string "Text/" represents an arbitrary text string
procedure dp_inner(Node: in TREE) is
begin
        PUT("Text0");
        dp_TYPE1(CHILD1(Node));
        PUT("Text1");
        dp_TYPE2(CHILD2(Node));
        PUT("Text2");
        dp_TYPE3(CHILD3(Node));
        PUT("Text3");
end dp_inner;
```

The procedures *dp_TYPE1*, *dp_TYPE2*, and *dp_TYPE3* are procedures that produce the text for nodes of the IDL type TYPE1, TYPE2, and TYPE3 respectively.

In other words, the source text corresponding to any node is merely the correctly ordered concatenation of some text strings inherent to the node ("Text0", "Text1", "Text2", and "Text3") and the source text for its descendants (the text produced by *dp_TYPE1*, *dp_TYPE2*, and *dp_TYPE3*). Concatenation is achieved by correctly ordering the output operations. When we say that procedure *dp_TYPE1* produces text, we mean that the procedure uses the function *PUT* to output text in the same manner that *dp_inner* does.

In practice, some of the text strings inherent to a node are null.

If the IDL type denotes an IDL class, then it is reasonable to expect the procedure that the class to simply dispatch processing depending on the kind of node it receives. For example, if the IDL class *TYPE1* is defined as

```
        TYPE1    ::=      inner | leaf ;
```

and there is a discriminating function, *KIND*, that operates on nodes of the class, then the procedure *dp_TYPE1* can be written.

```
procedure dp_TYPE1(Node: in TREE) is
begin
    case KIND(Node) is
        when  inner => dp_inner(Node);
        when  leaf  => dp_leaf(Node);
    end case;
end dp_TYPE1;
```

The procedure to handle nodes of a class can be optimized to do processing for the nodes in that class if the nodes are sufficiently similar, *i.e.*, they share similar attributes or generate the same text strings. For example, in DIANA the class *ID* contains only nodes that represent identifiers and all identifiers are reconstructed from the *lx_symrep* attribute. We use this to advantage in PrettyPrint by processing all nodes in the class *ID* with a single procedure.

For nodes without descendants, the leaf nodes, a procedure similar to the procedure for inner nodes is used. Although there are no structural attributes, we assume a lexical attribute, *lx_text*, that contains text associated with the leaf. Again, the existence of an accessing function, *LX_TEXT* that returns the value of the attribute is assumed.

```
procedure dp_leaf(Node: in TREE) is
begin
        PUT("Text0");
        PUT(LX_TEXT(Node));
        PUT("Text1");
end dp_leaf;
```

2.1.1. Simple Example

We now apply this technique to the reconstruction of ADA from DIANA. As an example consider the DIANA node **constant**, which represents the declaration of a constant object. The syntax for the declaration of a constant object in ADA follows.

```
constant_object_declaration ::=
  identifier_list : constant subtype_indication := expression ;
| identifier_list : constant constrained_array_definition
        := expression ;
```

The DIANA node, **constant**, represents a constant object declaration.

```
constant => as_id_s         : ID_S,
            as_type_spec    : TYPE_SPEC,
            as_object_def   : OBJECT_DEF;
```

Using the package *Diana* described in Chapter 4 of the DRM, which provides the operations on the DIANA data type, we write the following procedure to reconstruct the source for a constant declaration.

```
procedure dp_constant(Node: in TREE) is
begin
    dp_ID_S(AS_ID_S(Node));                  -- print identifier list
    Put(": constant ");
    dp_TYPE_SPEC(AS_TYPE_SPEC(Node));        -- print subtype
    dp_OBJECT_DEF(AS_OBJECT_DEF(Node));      -- print ':=' expression
    Put(";");
end dp_constant;
```

In other words, reconstructing the text for a constant object declaration requires the following pieces of text to be concatenated:

- the identifier_list; the text for the identifier list is created by a procedure similar to *dp_constant*, one that operates on id_s nodes.
- a colon and the ADA keyword "constant" (note this includes a space after the word "constant").
- the subtype_indication; the text is *created* by *a procedure that* processes the *as_type_spec* child.
- the expression; the procedure processing the *as_object_def* child produces the text for the ":=" as well as the expression, and finally
- a semicolon to end the declaration.

For some level of completeness we consider the procedures to reconstruct the source for two of the children of the node constant: *as_id_s*, the identifier list; and *as_obj_def*, the object definition expression.

The identifier list is a sequence of identifiers.

```
ID_S     ::=     id_s;

id_s     =>      as_list  : Seq Of ID;
```

Sequences are handled using the DIANA operations *LIST*, *HEAD*, and *TAIL*. *LIST* returns a value of type SEQ_TYPE that is a sequence of IDL nodes. *HEAD* returns the first node in the sequence and *TAIL* returns the sequence, possibly empty, that remains *after the first node* is *removed*. *IS_EMPTY* returns a boolean value True when the sequence is empty. (The procedure does not begin with a check for an empty sequence as the ADA syntax does not permit an empty list).

```
procedure dp_id_s(Node: in TREE) is
    L: SEQ_TYPE;                          -- holds list not processed
begin
    L := LIST(Node);
    dp_ID(HEAD(L));                       -- print the first id
    L := TAIL(L);                         -- process the remaining
    while not IS_EMPTY(L)
    loop
        PUT(",");                         -- separate id's with ","
        dp_ID(HEAD(L));                   -- print next id
        L := TAIL(L);                     -- process the remaining
    end loop;
end dp_id_s;
```

The procedure dp_id_s iterates over the sequence of nodes. For each node the procedure dp_ID is called to process the leaf nodes that are the identifiers. Before each node (except the first) a comma is introduced, thereby separating each identifier in the list with commas.

In presenting the procedure for dp_id_s we have simplified the treatment of identifier lists. In actuality the process is complicated by the fact the identifiers could be **label_ids**. Sequences of **label_ids** should not be separated by commas (see Section 2.2.1 on page 33).

The object definition is represented by a node in the class *OBJ_DEF*. The nodes in this class are the nodes of the class *EXP* and the node **void**. **Void** indicates that no object definition appears in the declaration. Although the object definition for a constant declaration may never be void, the node **constant** is also used to represent deferred constants. Deferred constant declarations have no object definition expression and DIANA represents this with a **void** node as the *as_obj_def* child. The procedure to reconstruct the text must account for this fact. In particular, it must assure that the ":=" is not generated when the object declaration is absent.

The procedure to process the class *OBJ_DEF* is an example of processing for an IDL class. The function *KIND* is used to discriminate between nodes within the class.

```
procedure dp_OBJ_DEF(Node: in TREE) is
    case KIND(Node) of
        dn_void   => null;              -- no object definition
        others    => Put(":=");         -- all object defs begin with ":="
                     dp_EXP(Node);
    end case;
end dp_OBJ_DEF;
```

2.2. Anomalies

The procedures needed to describe the remaining DIANA nodes are generally straightforward. The steps required to reconstruct most nodes can be deduced from Chapter 2 of the DRM. However, there are some DIANA nodes for which obvious solutions do not exist. In this section we discuss the implication of these nodes and describe the processing necessary in order to reconstruct the source for these nodes.

That we must treat certain nodes delicately to recreate the source does not imply that DIANA in some way destroys the original source. On the contrary, the source is reconstructable. We are merely pointing out the reconstructions that may not be obvious to the casual observer.

2.2.1. Label Identifiers

The defining occurrence of a label identifier is represented by the DIANA node **label_id**. The defining occurrence of a loop_name or block_name is also represented by **label_id**. Not only is there a semantic difference between the two, there is a textual difference in how the identifiers are represented. The label identifier should be enclosed in double brackets ("<< >>") while the block_*name* should appear without the brackets. To solve this requires upmerging the processing of the **label_id** into the nodes that can be its parent. As there are only two such nodes, there is no great difficulty in doing this.

The node **named_stm** is used to represent named blocks and named loops. The *as_id* son of this node will be a **label_id** representing the name of the block or loop. In this situation the processing of **label_id** is no different that any other identifier.

The node **id_s** represents an identifier list. It is also used to represent the sequence of label identifiers preceding a statement. In the majority of contexts the source is the concatenation of all the identifiers in the sequence, separated by commas. However, when **id_s** represents a sequence of label identifiers, the source is the concatenation of the identifiers *without* commas and *with* each identifier bracketed.

Since brackets around the labels occur only in the context of an identifier list, it is logical to place decision logic for handling labels into the routine processing the identifier list. The solution requires that the routine reconstructing the identifier list know which type of identifier appears in the list. The type of an identifier can be determined by using the *Diana* function *KIND*. Figure 2-1 on page 34 presents the two procedures needed to recreate the text for the node **id_s**. The first procedure recreates the text for all identifiers, both used and defining occurrences, and the second handles the sequence itself.

Procedure *dp_id_s* begins by printing the first identifier in the sequence. Because correct DIANA is assumed, there is no test for an empty identifier list. The procedure *dp_ID* is called to print the identifier. If the type of identifier is **label_id** then the identifier is bracketed. After producing the first identifier, *dp_id_s* enters a loop to process the remaining identifiers. A comma is inserted

```
-- procedure for all nodes in class ID

procedure dp_ID(Node: in TREE) is

-- Node should be in the IDL class ID
-- All identifiers treated the same,
-- lx_symrep contains the representation

begin
    Put(LX_SYMREP(Node));
end ID;


procedure dp_Label_id(Node: in TREE) is
begin
    Put("<<");                      -- label ids bracketed by "<<  >>"
    dp_ID(Node);
    Put(">>");
end dp_Label_id;


-- procedure for node id_s

procedure dp_Id_s(Node: in TREE) is

-- Node is an id_s node

    Seq: SEQ_TYPE;                  -- local variable to hold the sequence

begin
    Seq := LIST(Node);              -- get the list of id nodes
    if KIND(HEAD(Seq)) = dn_label_id then
        dp_Label_id(HEAD(Seq));
    else
        dp_ID(HEAD(Seq));
    end if;
    Seq := TAIL(Node);
    while not IS_EMPTY(Seq)
    loop
        if KIND(HEAD(Seq)) = dn_label_id then
            dp_Label_id(HEAD(Seq));
        else
            Put(",");               -- regular ids separated by ","
            dp_ID(HEAD(Seq));
        end if;
        Seq := TAIL(Node);
    end loop;
end dp_Id_s;
```

Figure 2-1: Procedures To Reconstruct id_s Node

before each of the remaining identifiers if the identifiers are not labels. If the identifier is a label no comma is inserted and the identifier is bracketed.

2.2.2. Task Types

An ADA task type specification is represented in DIANA by a **type** node whose *as_type_spec* child is a **task_spec** node. The node **task_spec** is the only node in the class *TYPE_SPEC* requiring text to be generated before the ADA reserved word "type". The text generated for a task type declaration includes the reserved word "**task**".

```
task type <typename> is <task_specification>;
```

Every other type declaration begins with "**type**".

```
type <typename> is <type_specification>;
```

This problem requires that the procedure reconstructing the source for a **type** node must inspect the *as_type_spec*. The procedure *dp_type* is shown in Figure 2-2.

```
procedure dp_Type(Node: in TREE) is
begin
    if KIND(AS_TYPE_SPEC(Node)) = dn_task_spec then
        Put("task ");
    end if;
    Put("type");
    dp_ID(AS_ID(Node));                    — task identifier
    dp_Vars(AS_VAR_S(Node));               — discriminants
    Put(" is ");
    dp_TYPE_SPEC(AS_TYPE_SPEC(Node));
    Put(";");
end dp_Type;
```

Figure 2-2: Procedure To Recreate Type Declarations

2.2.3. Subprogram Declarations

ADA subprogram declarations are represented by the node **subprogram_decl**. This node has three attributes. The first, *as_designator* is the subprogram identifier. The second, *as_header* is the subprogram header. The third is used for renaming and generics, the *as_subprogram_def* attribute.

The second attribute references a node in the IDL class ***HEADER***. The nodes **function** and **procedure** belong to this class. Both nodes have an attribute, *as_param_s* which references the formal part of the subprogram declaration. In addition, **function** has a *as_name_void* attribute which references the return type of the function.

The nature of the **subprogram_decl** and *HEADER* nodes make it difficult to reconstruct the source in one pass. Consider the reconstruction of the following subprogram declaration.

```
procedure ProcID (param1: in TREE);
```

The declaration is represented by a **subprogram_decl** node. Knowledge of the *as_header* child of this node is needed to determine that the subprogram is a procedure. Then the *as_id* child is needed to recreate the subprogram identifier. Following the identifier, the *as_header* child is needed again to recreate the formal part of the declaration.

There are several methods for solving this problem. One is to use *KIND* to determine the type of the *as_header* child. Another method requires the routine reconstructing the declaration to pass the identifier to the routine processing the header. We illustrate the latter technique. Figure 2-3 on page 37 presents the procedures needed to *reconstruct a subprogram* declaration. Procedure *dp_Subprogram_decl* processes the node **subprogram_decl**, procedure *dp_HEADER* dispatches processing for nodes in the class *HEADER*, and *dp_Procedure* processes the node **procedure**.

Note that this treatment applies to subprogram bodies as well. The routine reconstructing the text for the subprogram body must pass the identifier to the procedure processing the *as_header* attribute.

2.2.4. Blocks

The node **block** is used in three different ADA contexts. It represents a block statement. It also represents the block of a named statement. Lastly, **block** represents the body of a subprogram, task, or package.

The **block** node has three structural attributes.

```
block  =>  as_item_s         : ITEM_S,  -- declarations
           as_stm_s          : STM_S,   -- statements
           as_alternative_s  : ALTERNATIVE_S; -- exceptions
```

The most straightforward **block** is a simple block statement. The text

```
-- process the node subprogram_decl

procedure dp_subprogram_decl(Node: in TREE) is
begin
    -- pass the header and id to the routine for the header
    -- LX_SYMREP returns the text for the identifier
    dp_HEADER(AS_HEADER(Node),LX_SYMREP(AS_DESIGNATOR(Node)));
    -- process the subprogram_def child
    dp_SUBPROGRAM_DEF(AS_SUBPROGRAM_DEF(Node));
    Put(";");
end dp_subprogram_decl;




-- process nodes in the class HEADER; it receives as input the
-- node and the text for the identifier

procedure dp_HEADER(Node: in TREE; Ident: in String) is
begin
    case KIND(Node) is
        when dn_function => dp_function(Node,Ident);
        when dn_procedure => dp_procedure(Node,Ident);
        when dn_entry => dp_entry(Node,Ident);
    end case;
end dp_HEADER;




-- process the node procedure; receives the node and the identifier

procedure dp_procedure(Node: in TREE; Ident: in String) is
begin
    Put("procedure ");
    Put(Ident);
    dp_param_s(AS_PARAM_S(Node));       -- formal part
end dp_procedure;
```

Figure 2-3:   Procedures To Recreate A Subprogram Declaration

generated by this node follows.

```
declare
    <as_item_s>
begin
    <as_stm_s>
exception
    <as_alternative_s>
end;
```

Note that "declare" is only recreated when *as_item_s* is not empty and "exception" is only recreated when *as_alternative_s* is not empty.

The situation is complicated slightly when the block statement is *named*[2]. The named block is represented by the node **named_stm** with a child, *as_stm*, that is a **block** node. In this case the text for the block differs by the identifier after the block end.

```
<identifier>:
    declare
        <as_item_s>
    begin
        <as_stm_s>
    exception
        <as_alternative_s>
    end <identifier>;
```

In the final instance, block is used to represent the body of a subprogram, package, or task. In this case the text associated with the block does not include the ADA reserved word "declare" and the identifier following the "end" is optional (we have consistently chosen to include it).

```
<subprogram, package, or task specification>
    <as_item_s>
begin
    <as_stm_s>
exception
    <as_alternative_s>
end <identifier>;
```

We consider the processing for these three cases to be sufficiently different to warrant treatment by three separate procedures.

```
-- this handles block statements; it is passed the node block

procedure dp_block(Node: in TREE);

-- this handles the named blocks; it is passed block and an
-- the identifier that should follow the end
```

[2] A similar problem exists with named loops. The solution is analogous to the solution used for named blocks

```
procedure dp_block(Node: in TREE; Ident: in String);
-- this handles the subprogram, package, and task bodies.
procedure dp_block_stub(Node: in TREE);
```

The first two procedures are overloaded with the name *dp_block*. The first procedure has as its argument the **block** node. This creates the text for a block statement. It is typically called by the routine that processes the ***STM*** class.

The second procedure receives two arguments: the **block** node and the identifier that names the statement. This procedure is called only from the routine that processes the node **named_stm**. It will generate the text for the block and place the identifier after the reserved word "**end**".

The last procedure actually handles the class *BLOCK_STUB*. There are only two nodes in the class: **block** and **stub**. When the node is stub the procedure will generate the text "**is separate**". When the node is **block** it will generate the text for the block without generating the reserved word "**declare**".

The bodies of the three procedures are shown in Figure 2-4 on page 40.

2.3. Comments

PRETTYPRINT uses the simplest of commenting strategies. The algorithm used prints the comment associated with a node after the text for the node has been reconstructed. When creating comments in an unformatted text string the only constraints are that the comment be preceded by "--" and followed by an end of line. If the comment extends over one line, each additional line must begin with "--". This issue is addressed in Section 3.2.4 on page 52 along with a discussion of other constraints for producing comments in formatted source text.

All DIANA nodes related to the ADA source have the attribute *lx_comments*. This attribute records a comment. The type of the attribute is the IDL private type "comments". We assume this type is implemented such that the function *Void* will return the boolean True if the comment is empty. The treatment of comments for *all* nodes is the same. We create a single procedure to process all comments.

```
procedure dp_block(Node: in TREE) is
begin
    if not IS_EMPTY(LIST(AS_ITEM_S(Node))) then
        Put("declare");
        dp_Item_s(AS_ITEMS_S(Node));
    end if;
    Put("begin");
    dp_stm_s(AS_STM_S);
    if not IS_EMPTY(LIST(AS_ALTERNATIVE_S(Node))) then
        Put("exception");
        dp_Alternative_s(AS_ALTERNATIVE_S(Node));
    end if;
    Put("end;");
end dp_block;


procedure dp_block(Node: in TREE; Ident: in String) is
begin
    if not IS_EMPTY(LIST(AS_ITEM_S(Node))) then
        Put("declare");
        dp_Item_s(AS_ITEMS_S(Node));
    end if;
    Put("begin");
    dp_stm_s(AS_STM_S);
    if not IS_EMPTY(LIST(AS_ALTERNATIVE_S(Node))) then
        Put("exception");
        dp_Alternative_s(AS_ALTERNATIVE_S(Node));
    end if;
    Put("end");                         -- named block: identifier follows end
    Put(Ident);
    Put(";");
end dp_block;

procedure dp_block_stub(Node: in TREE) is
begin
    if KIND(Node) = dn_stub then
        put ("is separate");
    else
                                        -- "declare" not printed
        dp_Item_s(AS_ITEMS_S(Node));
        Put("begin");
        dp_stm_s(AS_STM_S);
        if not IS_EMPTY(LIST(AS_ALTERNATIVE_S(Node))) then
            Put("exception");
            dp_Alternative_s(AS_ALTERNATIVE_S(Node));
        end if;
        Put("end;");
    end if;
end dp_block;
```

Figure 2-4: Procedures for Reconstructing Blocks

```
procedure dp_comments(Node: in TREE) is

begin
    if not Void(LX_COMMENTS(Node)) then
        Put("--");
        Put(LX_COMMENTS(Node));
        Put(cr & lf);                   -- end of line terminators
    end if;
end dp_comments;
```

To include comments in the source reconstructed from a constant node, the example from Section 2.1.1 on page 30, we simply add a statement to call *dp_Comments* at the end of the procedure.

```
procedure dp_constant(Node: in TREE) is

begin
    dp_ID_S(AS_ID_S(Node));
    Put(": constant ");
    dp_TYPE_SPEC(AS_TYPE_SPEC(Node));
    dp_OBJECT_DEF(AS_OBJECT_DEF(Node));
    Put(";");
    dp_comments(Node);                  -- print comment if present
end dp_constant;
```

The source reconstruction algorithm appends the comment to the text for the node it is attached to. Thus, the effect on the reconstruction of a constant object declaration varies. Consider the ADA statement

```
ID1, ID2: constant SUBTYPE := EXP;      -- comment text
```

If the comment text is attached to the id_s node the statement is reconstructed as

```
ID1, ID2                                -- comment text
: constant SUBTYPE := EXP;
```

If the comment is attached to the OBJ_TYPE_SPEC the statement is reconstructed as

```
ID1, ID2: constant SUBTYPE              -- comment text
:= EXP;
```

Of course, a responsible pretty-printer will indent the continuation of the statement on the next line. In this chapter we have only presented the design for a pretty-printer to produce unformatted text. Formatting issues are discussed in Chapter 3.

CHAPTER 3
FORMATTING

In this chapter we discuss the formatting of source text and the formatting of ADA source text in particular. PRETTYPRINT formats reconstructed source text by calling entries in *FORMAT*, an ADA package which provides an interface to the output medium, assistance in placement of line breaks, and support for indentation.

This chapter begins with a discussion of the classical formatting problems, followed by a general discussion of our design for solving the set of classical problems. The chapter closes with the presentation of the package *FORMAT*.

3.1. Classical Formatting Problems

A proper understanding of the problems associated with pretty-printing requires an appreciation of the goal of a pretty-printer. In a word, the goal of every pretty-printer is to produce "*readable*" source. Readability is that quality that makes a program easy to understand, debug, modify, test, and maintain. Insofar as programming is an art, what makes a program readable is a question of aesthetics and is often debated. Indeed, some aspects of pretty-printing that we present as facts are actually our own opinions.

There are some generally accepted tenets of pretty-printing: the formatting of the source should help the reader visualize the syntax; the nesting of the program should apparent at a glance; and individual statements, declarations, and expressions should be discernible.

Consistency is another benefit accrued by pretty-printing. Programs formatted by the same pretty-printer are consistently arranged. A pride of programmers that are sharing code can use the output of a pretty-printer as the *de facto* standard of readability. The pretty-printed programs thus share a homogeneous style and another programmer's code does not look foreign.

The ADA LRM uses an implicit formatting style for its programming examples. The pretty-printer we have designed formats in this style. The examples presented in this section also are formatted in this style.

3.1.1. Indentation

Indentation is the single most important aspect of program formatting. . Proper indentation can be used to indicate program nesting, and to diagram control constructs. In the idyllic situation where all programs are terse and compact enough that each statement can exist on its own line indentation is easy. One example of the use of indentation for ADA is to show nested scopes.

```
procedure Nest is
    Number : constant := 42;          — declarations indented
    Object : INTEGER;
begin
    Object := Number;                 — statements  indented
end Nest;
```

The declarations and statements within the procedure body are distinguishable from the syntax defining the body by their indentation. The "**begin**" and "**end**" are not indented and clearly demarcate the bounds of the body.

Similarly, indentation can make the conditional clauses of an ADA "if" statement more visible.

```
procedure Nest is
    Number : constant := 42;          — declarations indented
    Object : INTEGER;
begin
    Object := Number;                 — statements indented
    if Object = Number then
        Object := Number;             — each conditional clause indented
    else
        Object := Number;
    end if;
end Nest;
```

The effect of the indentation is additive. Each construct that uses indentation indents in from the current level. The increasing indentation is the first problem a pretty-printer has to solve. The number of spaces for each increment of indentation must be small enough that at the maximum excursion the space remaining on the line is usable. The obvious solution is to base the indentation increment on the maximum indentation depth. Of course, the quantum nature of the output medium (*i.e.*, the minimum indentation is at least one space) may force the maximum excursion to exceed a reasonable bound for even the smallest increment. Indentation beyond this point should be prevented.

It should be noted that a program with control structure nesting so deep that further indentation is prevented is probably too complex to be readable anyway. The program should be analyzed to see where complexity can be removed, most likely by dividing it into less complex modules, thereby making each module more readable and the whole more understandable.

3.1.2. Line Breaks

The programs that a pretty-printer has to format are not ideal. Simply choosing to indent based on control structures will not gain readability by itself if statements extend beyond the length of the line. Consider the formatting of an if statement.

```
if condition then
    object1 := expression1 + expression2;
    procedure_call;
else
    object2 := expression1 + expression2;
end if;
```

As long as the line width is *large enough*, this formatting is very readable. The situation is less sanguine when fewer columns are available. The following example is unreadable because the indentation is lost when the statement is continued on the next line.

```
if condition then
    object1 := expression1 +
    expression2;
    procedure_call;
else
    object2 := expression1 +
expression2;
    end if;
```

Using the current level of indentation for the continuation of the statements increases the readability but still leaves room for improvement.

```
if condition then
    object1 := expression1 +
    expression2;
    procedure_call;
else
    object2 := expression1 +
    expression2;
end if;
```

At first glance, the continuation of the broken line, the text "expression2", looks like a separate statement. It has the same visual importance as the call to procedure "procedure_call". Indenting the continuation of lines will avoid this confusion. The following is much more readable.

```
if condition then
    object1 := expression1 +
        expression2;
    procedure_call;
else
    object2 := expression1 +
        expression2;
```

```
     end if;
```

Choosing where to break a line can be as important as choosing which column to start the continuation of a broken line. Consider the following statement.

```
  object1 := expression1 + expression2 + function_call(param1,param2);
```

On a shorter line this statement would have to be broken and continued on the next line. *A truly awful break would be within the function call.*

```
  object1 := expression1 + expression2 + function_call(param1,
     param2);
```

An improvement can be made by not breaking up syntactic elements. The function call is an expression that should not be broken up unless absolutely necessary. Further improvement can be made by moving the operator to the next line. This move makes it obvious that the continuation is indeed part of the expression on the previous line.

```
  object1 := expression1 + expression2
     + function_call(param1,param2);
```

A final improvement can be made by indenting the continuation of the broken statement to illustrate some of the semantics. In this case the function call is part of the expression on the right hand side of the *assignment statement*. Beginning the continuation so it lines up to the right of the assignment operator can help to show this relationship and aid in the reader's understanding.

```
  object1 := expression1 + expression2
             + function_call(param1,param2);
```

Choosing not to break a line can be as important as choosing where to break a line. For example, an if statement that will fit on one line should most likely be placed on one line. The terseness of

```
  if condition then statement; end if;
```

recommends it above the sprawling

```
  if condition then
     statement;
  end if;
```

It is also advantageous to include more that one statement on a line when the statements are brief.

```
  x := y;   z := x;
```

The statements above are sufficiently terse to be included on the same line without loss of readability.

3.1.3. Lists and Sequences

Certain ADA syntactic constructs are lists of items. When breaking a construct over several lines lists should receive special consideration. If a list must be broken up to fit on a line, then placing each item in the list on a separate line is often the most readable. A procedure specification is the most illustrative example of this situation.

```
procedure Proc(X: in T1; Y: in T2; Z: in T3);
```

Suppose only the first two parameter specifications fit on the line. Simply breaking the line so that parameter specifications are unbroken is reasonable. The continuation should be indented to show it is part of the list of actuals.

```
procedure Proc(X: in T1; Y: in T2;
               Z: in T3);
```

One possible improvement is to treat each parameter specification with equal importance. Thus, if one parameter specification has to appear on a separate line, all specifications should be on separate lines. The example is reformatted to show this.

```
procedure Proc(X: in T1;
               Y: in T2;
               Z: in T3);
```

In declaring the last example superior to the one immediately preceding it we are treading lightly. Like any aesthetic pronouncement it has a subjective basis. Nonetheless, this formatting style is used throughout the PRETTYPRINT design[1].

3.1.4. Comments

The consideration of comments during text formatting is a poorly understood issue. In many cases the introduction of comments into the source text, especially comments which, like ADA, are terminated by a line break, will force formatting decisions. For example, an if statement that can be placed on one line,

```
if condition then statement; end if;
```

may be forced by comments to exist on three lines.

```
if condition then                    -- the condition checks Foo
   statement;                        -- the statement sets Foo
end if;                              -- Foo now usable
```

[1]It doesn't weaken our case to add that this is the way long procedure specifications are formatted in the ADA LRM.

One of the biggest issues is how to decide where comments should be broken if there is insufficient room on a line. Unlike ADA source, where the syntax is defined, there is no way to determine the context of the comment. One such dilemma exists when considering the formatting of a statement with a comment when the statement does not fit on the line. Consider the following statement.

```
object := expression1 + expression2 + func(x,y);  -- expression2 is real
```

If the statement is printed on a line that is narrower, then the question of how to break the line is unsolvable. It may be possible to break the comment over several lines.

```
object := expression1 + expression2 + func(x,y);  -- expression2
                                                  -- is real
```

The comment may be more readable as a complete line and it may be possible to fit the statement on one line, and the comment on the next.

```
object := expression1 + expression2 + func(x,y);
                                      -- expression2 is real
```

It can be argued that the statement should be broken, even if it fits, so the comment has some of the proper context.

```
object := expression1 + expression2
          + func(x,y);  -- expression2 is real
```

Of course, in our example the context is lost because the comment refers to expression2.

An omniscient pretty-printer would recognize when a comment is best represented by breaking the statement it describes.

```
object := expression1 + expression2  -- expression2 is real
          + func(x,y);
```

The problems related to comments are compounded when recreating comments from an internal representation such as DIANA. In this case the comments are in some manner attached to the nodes of a parse tree. The pretty-printer operating from this tree must then recreate the source and intelligently re-insert the comments into the source. To be effective, the comments must be intelligently associated with the nodes in the internal representation, and the pretty-printer must understand this association. To a first approximation this association can be done with a simple strategy. In the design of PRETTYPRINT we have assumed that a comment associated with a node should be printed after the text for the node is printed. However, it is easy to imagine comments that may be ruined by anything less than an omniscient pretty-printer. The most insidious, though unlikely, example is the following piece of ADA text:

```
    object := expression1 + expression2;
--     |           |                |
--     |           |                ---- expression2 is column number
--     |           |
--     |           ---------------------- expression1 is the line number
--     |
--     ---------------------------------- the object is returned to caller
```

3.1.5. Whitespace

One way to make a program more readable is to make its components easily distinguishable. Towards this end whitespace, blank lines on the page, can be used as a visual separator. Controlling whitespace is something a programmer can do better than a pretty-printer because the programmer has an understanding of the logical mapping of the program to the problem the programmer is solving. However, because ADA provides mechanisms for dividing up a program (packages, subprograms, tasks, *etc.*), most logical divisions will also be syntactic divisions. PRETTYPRINT does add whitespace before packages, subprograms, and tasks.

3.1.6. Page Layout

Equally as important as the *consideration* of the placement of source text on individual lines is the layout of the lines on the page. It is desirable to place single ideas on a single page. This allows the reader of the program to focus on a single concept at a time. Just as it is undesirable for a syntactic element to be broken over a line, it is undesirable for a program component to be broken over a page. Although the problem is not considered in the design of PRETTYPRINT, the principles used in formatting statements and declarations on individual lines can be applied to the layout of subprograms and packages on the page.

3.1.7. Use of Fonts

Using fonts to distinguish different lexical entities can be an effective way to improve the readability of a program. In this document we have consistently used a bold typeface when writing ADA reserved words. The reserved words are then set apart from the identifiers of the program, and the syntactic structure can be easily recognized. Another possible use of fonts is an italic font for comments. A pretty-printer that is reconstructing the source text can use fonts effectively in this way.

Another way to improve readability is to use consistent representations for identifiers. ADA allows several representations of an identifier to be used. Case consistency can aid in recognizing user defined identifiers. A pretty-printer can normalize all occurrences of identifiers so that all representations are identical.

3.2. Solutions

In this section we discuss the ways PRETTYPRINT solves the classical formatting problems. This discussion serves as an introduction to the operations of *FORMAT*. Section 3.3 provides a more complete description of the formatting operations and Chapter 4 contains the discussion of all the issues related to recreation of source.

Before considering cases, we review the basic operation of PRETTYPRINT. PRETTYPRINT makes two passes (tree walks) over the DIANA structure. In the first pass it computes the number of characters needed to print each node, the largest element in each sequence, and the maximum nesting depth of the program. In the second pass the source is reconstructed and the values computed during the first pass are used to make decisions concerning line breaks and indentation.

3.2.1. Solving Indentation

Section 3.1.1 on page 44 describes the classic use of indentation to represent program block nesting, and syntax structure. PRETTYPRINT uses indentation in the classical way. Support for indentation is provided by the package *FORMAT* through the two entries *Indent* and *Undent*. *Indent* causes the next line to be indented by an increment from the previous line. *Undent* reverts the indentation to that in force before the current indentation. The complete functionality of these procedures is provided in Section 3.3 on page 54.

The increment used for indentation is based on the depth of control structure nesting. In the first pass over the DIANA structure the maximum nesting level for the program is computed. This value is used to choose the increment for nesting. The indentation increment must balance the need for distinguishable indentation against the need for usable space after indentation. The *FORMAT* function *SetIncrement* computes the indentation increment based on the nesting depth of the program.

3.2.2. Solving Line breaks

The procedure *NewLine* can force the insertion of a line break in the recreated source. Forcing a line break can be used to shape the source to show the syntactic structure. For example, *NewLine* is called after the keywords "**then**", "**else**", "**elsif**", and "**end if**" when displaying an if statement. *NewLine* will automatically indent the next line to the current indentation level.

PRETTYPRINT also needs to be able to intelligently decide where to break long source lines. For this reason the number of characters needed to print the text for a node is recorded with the node. For example consider the following statement.

```
object1 := expression1 + expression2 + function_call(param1,param2);
```

Recorded with the node **function_call** is the number of characters needed to print the function call. Before printing the function call the number of characters required can be compared with the number of characters remaining on the line. Insufficient space can be detected and a line break can be forced before the printing of the function call. In fact, this inquiry can be made prior to the printing of the "+" operator.

```
object1 := expression1 + expression2
    + function_call(param1,param2);
```

The function *Remaining* returns the number of spaces remaining on the current line.

FORMAT provides support for controlling at which column the continuation of a line begins. The function *Position* returns the current position on the line, and the procedure *SetIndent* will set the current indentation to a specific column. It is possible in the example above to force the continuation of the assignment statement to be to the right of the assignment operator. After printing the assignment operator *Position* can be called to return the current position and this value can be used as the argument to *SetIndent*.

```
object1 := expression1 + expression2
           + function_call(param1,param2);
```

The indentation caused by *SetIndent* is canceled with a call to *Undent*.

3.2.3. Solving Lists and Sequences

The support provided by the subprograms *Position* and *SetIndent* aid in the processing of lists of syntactic elements. For example, if the parameter specifications of a subprogram specification will not fit on one line, the indentation can be set so that all of the parameter specifications line up underneath the first one. Again, the most illustrative example is the procedure specification.

```
procedure Proc(X: in T1;
               Y: in T2;
               Z: in T3);
```

Recorded with the nodes representing lists of items is the number of characters needed to print the entire list, and the size of the largest element in the list. Using the former, PRETTYPRINT can determine if a list will fit on the current line. Using the latter, it can choose an indentation such that all elements in the list can appear on a single line. By comparing the space remaining with the size of the largest element in the list, the following situation can be detected.

```
procedure Proc(X: in T1;
               Y_is_a_long_name: in
                                 T2;
               Z: in T3);
```

The indentation can be selected so that the largest parameter specification will fit on one line. The resulting format is clearer.

```
procedure Proc
              (X: in T1;
               Y_is_a_long_name: in T2;
               Z: in T3);
```

3.2.4. Solving Comments

There is little support for comments in PRETTYPRINT. One of the inherent problems with formatting comments from an internal representation is a lack of understanding of how comments are associated with the nodes. In particular DIANA does not specify this association. For this design we have assumed that the comment attached to the node is the comment that appeared after the node in the original source. When recreating the source, any comments are placed in the recreated source after the text for the node has been created.

PRETTYPRINT does not account for the length of comments in determining whether text will fit on the line. This is a conscious decision on the part of the designers. The reason for this decision is that it is impossible to distinguish between the following cases.

```
objectl := expressionl + expression2; -- first example

objectl := expressionl +
              expression2;  -- a second example with a longer comment

objectl := expressionl + expression2;
-- The third example has a comment the length of three lines.
-- The entire comment is associated with the  statement above,
-- although the content may indeed refer to a statement below.
```

Without a way to interpret the meaning of a comment, which is beyond the scope of a pretty-printer, there is no way to intelligently associate comments. Instead of basing formatting decisions on questionable input, we have chosen to ignore comments while making formatting decisions.

3.2.5. Solving WhiteSpace and Page Layout

PRETTYPRINT adds whitespace to the reconstructed source by calling *NewLine* multiple times. Whitespace is produced as a buffer for subprogram and package specifications and bodies. It is also produced between every compilation unit.

PRETTYPRINT does not address the question of page layout. The solution to the problem is not very difficult. The second pass over the DIANA structure can be used to produce a tree attributed with the number of lines needed to print each node. Then a third pass can pass over the tree creating formatted source while determining page breaks. The types of decisions necessary are analogous to the decisions needed for line breaks. If a program body contains more lines than remain on a page, a page break can be inserted before the body so it remains intact.

3.2.6. Solving Use of Fonts

FORMAT provides a mechanism to differentiate between different lexical items in ADA. *ReservedForm*, *IdentForm*, and *ComForm* are used to append reserved words, identifiers, and comments respectively. No commitment is made as to how they are represented.

3.3. Package "FORMAT"

The formatting operations supplied by *FORMAT* were casually introduced in the previous section. In this section we describe the complete functionality of the ADA package *FORMAT*. Figure 3-1 shows the *FORMAT* package specification. The package body is not specified in this document.

The package *FORMAT* provides functional support in two important ways. Firstly, *FORMAT* handles all of the output of the recreated source, and secondly, *FORMAT* provides support for proper indentation.

The output of source is handled by operations that append text to the previously generated source text. The package hides any device dependency (particularly if boldface and italics are to be used). If buffering of output is required, the buffering will be transparent to the the programs that call the *FORMAT* routines.

The principle reason that buffering of output may be desired is for creating special print effects. When text is appended, the text is qualified as either an ADA reserved word, identifier, comment, or other lexeme. *FORMAT* can use output device characteristics such as boldface type, italic fonts, and underlining to visually distinguish these classes of text. Depending on the output device, underlining and boldface may require printing two output lines without a linefeed to achieve the desired effect. In such an instance buffering the line before printing is absolutely necessary.

Buffering also eases some formatting problems. Consider the problem of printing a lexeme larger than the space remaining after indentation. In such a case the indentation should be reduced to allow enough space for the lexeme to be printed If the spacing for indentation is output directly to an output device it is impossible to reclaim that space. A new line would have to be started with less indentation to accommodate the lexeme, leaving a blank line in the output. However, if the output is buffered, the space used for the indentation can be reclaimed from the buffer.

Whether the output is buffered or not is transparent to the subprograms traversing the DIANA tree constructing the source. The reconstruction routines are only concerned with the current output line. The operations provided append text to the current line and query *FORMAT* regarding the status of the line (such as the number of characters remaining, or the current position on the line). The traversal routines also may force a line break into the output either ex-

```
-- Package that provides operations to format reconstructed Ada source

package FORMAT is

    LineLength :
        constant POSITIVE := 120;          -- length of the output line
    type Column is
        range 0..LineLength;               -- position on the line

    -- There are four procedures to append text to the output buffer

    procedure AddText(Text: in String);
    procedure ResForm                      -- add Ada reserved words
        (Text: in String);
    procedure ComForm                      -- add comments
        (Text: in String);
    procedure IdentForm                    -- add program identifiers
        (Text: in String);

    -- There are two function to return status of the output buffer

    function Remaining return Column;      -- unused characters in buffer
    function Position return Column;       -- used characters in buffer

    -- this procedure creates a line break

    procedure NewLine;

    -- this procedure sets the indentation increment

    procedure SetIncrement(Depth: in POSITIVE);

    -- three procedures provide indentation operations

    procedure Indent;                      -- increment from last indentation
    procedure Undent;                      -- revert to previous indentation
    procedure SetIndent(Pos: in Column);
                                           -- set indentation to Pos
end FORMAT;
```

Figure 3-1: FORMAT Package Specification

plicitly, by invoking the function NewLine, or implicitly by appending text beyond the end of the line. At each line break a new line is started by moving the position to the current indentation level (whether this is done by actually producing the required number of spaces or by using tabs is device dependent and not considered).

The indentation support automatically indents each new line of output. The interface provided by *FORMAT* allows the indentation to be incremented, for typical nesting level indentation, or to be set to a specific column. The model we use to describe the operations is a LIFO stack. The actual implementation of the operations is hidden.

3.3.1. Output Support Operations

The ADA program source program has four classes of lexical items. The first class consists of operators and delimiters. Lexical items in this class, such as semicolons, are appended to the buffer using the procedure *Addtext*.

The other classes are ADA reserved words, ADA identifiers, and comments. *FORMAT* accounts for these by providing three additional procedures for appending text to the output stream. These procedures receive the text as input and append a formatted form of text to the created source (the implementation of how the text is formatted is device dependent and not considered here). The three procedures are *ResForm* (to format reserved words), *ComForm* (to format comments), and *IdentForm* (to format identifiers). By using a separate procedure for each class, the representation of the class in the output text can be hidden from the routines recreating the source.

Two functions, *Remaining* and *Position*, are provided to allow inquiry into the status of the current output line.

3.3.1.1. Procedure AddText

The simplest formatting operations are the procedures that append text to the output stream. The most straightforward of these procedures is *AddText*. This procedure appends the text it receives to the output stream and updates the status of the current line.

There are two cases to consider when the text to be appended is larger than the space remaining on the current line. If this is the first lexeme after the indentation then the indentation is reduced to accommodate the lexeme. (The lexeme will always be less than the line width so this is possible - see the assumptions listed in Section 1.4.2 on page 22). Otherwise, a line break is inserted into the output stream and a new line is begun. Inserting the line break and beginning the new line is achieved by calling the procedure *NewLine*. The text is then appended to the new line. (*NewLine* causes the line to be indented properly).

After the text has been appended to the line the line status (current position on the line and number of spaces remaining) will be updated.

3.3.1.2. Procedure ResForm

Procedure *ResForm* appends an ADA reserved word to the output stream. Functionally it is identical to *AddText*. It calls *NewLine* if the current line is exceeded and will update the status of the current line after the text is appended.

The representation of reserved words is not specified. The way in which *FORMAT* records the formatting is also not specified. If the output is buffered, it is possible to modify the buffer so that each character can be given an attribute which indicates if the character is to be printed as bold, italics, or underlined.

3.3.1.3. Procedure IdentForm

Procedure *IdentForm* appends an ADA identifier to the output stream. Functionally it is identical to *AddText* and *ResForm*. It calls *NewLine* if the current line is exceeded and will update the status of the current line after the text is appended.

The representation of identifiers is not specified. The way in which *FORMAT* records the formatting is also not specified. If the output is buffered, it is possible to modify the buffer so that each character can be given an attribute that indicates if the character is to be printed as bold, italics, or underlined.

IdentForm can be used to represent identifiers consistently. For example, all identifiers can be normalized such that the first letter is in upper case and the remaining characters are in lower case. The representation of identifiers in DIANA is not specified. The DIANA *producer*[2] that creates the DIANA structure is

[2] see the discussion of Diana users in Section 1.1.3 of the DRM

not required to preserve the case of identifiers.

3.3.1.4. Procedure ComForm

Procedure *ComForm* appends a comment to the output stream. Functionally it is identical to *AddText* and *ResForm*. However, since a comment is terminated by the end of the line, *ComForm* calls *NewLine* to insert a line break after the comment has been appended. Of course, the status of the current line is updated after the comment is appended.

The procedure *ComForm* receives as input an ADA comment. The comment is simply an ADA text string. *ComForm* appends the characters "--" to the output buffer followed by the comment. Some care is needed when adding comments. If an insufficient amount of space remains on the line a new line must be started. If the comment extends over several lines the comment is broken at the space nearest the end of the line and is continued on the next line, again beginning the line with the comment delimiter "--".

3.3.1.5. Procedure NewLine

The procedure *NewLine* inserts a line break into the output stream and creates the proper indentation on the next line. The indentation is determined by the indentation operations listed below in Section 3.3.2. Using the stack model, each time a new line is created the indentation at the top of the stack is read. This value is the number of blank spaces needed at the beginning of the new line. The way *NewLine* creates the indentation is possibly device-dependent and thus not specified here (*e.g.*, a device that supports tabs may use tabs).

3.3.1.6. Function Remaining and Function Position

The functions *Remaining* and *Position* provide a means of inquiry as to the status of the current line. *Remaining* returns the number of unused characters at the end of the current line and *Position* returns the current position on the line. The sum of the two values will add up to the length of the output line, *LineLength*.

3.3.2. Indentation Stack Operations

The model we use to describe the operation of the indentation support is a LIFO stack. All indentation of the source program is properly nested. When there is a new indentation, the new value replaces the old value. When the scope of the indentation is ended the text is "undented"[3], that is the indentation reverts back to the indentation in effect before the current indentation. This proper nesting is well modeled by a stack. Although we are not specifying the implementation of the indentation, we will refer to the operations in terms of a stack.

The *FORMAT* entries related to indentation are *SetIncrement*, *Indent*, *Undent*, and *SetIndent*.

3.3.2.1. Procedure SetIncrement

SetIncrement receives as its argument the maximum nesting depth of the program. It then chooses an indentation increment based on this number. It tries to maximize the increment, to make each indentation more distinctive, while keeping the maximum excursion small (the choice of how far across the line the maximum excursion should go is not specified).

3.3.2.2. Procedure Indent

Indent increases the current indentation level by a computed increment (up to a predefined maximum indentation), and pushes that value onto the stack. For example, if the current line is indented ten spaces and the indentation increment is five spaces, then the value fifteen would be saved on the stack as the value for the next indentation.

It is possible that the program is so deeply nested that even with a minimum increment the indentation becomes too large. A maximum indentation is enforced to ensure that there is reasonable space available after indentation. When the maximum is reached, an additional call to *Indent* does not increment the indentation; it pushes another copy of the current value onto the stack.

3.3.2.3. Procedure Undent

Undent pops the last indentation value off of the stack. This operation reverts the indentation to the value previously in effect.

Note that setting a new indentation level (or removing it through *Undent*) has no immediate effect on the output. Indentation occurs at the next line break. When the line break occurs the value at the top of the stack is used to

[3] we use the neologism undent for convenience

determine the indentation for the next line. Line breaks are inserted when appending text that exceeds the remaining space on a line, or when *NewLine* is called. Thus an *Indent* operation followed by *Undent* before a line break can be inserted has no effect on the reconstructed source.

3.3.2.4. Procedure SetIndent

SetIndent is another operation on the indentation stack. It allows the indentation to be set to a specified column. Unlike *Indent* which increments the indentation by a fixed increment, *SetIndent* pushes the column it receives as its actual parameter onto the stack. The procedure *Undent* is used to remove this indentation and revert to the previous indentation.

SetIndent is useful for aligning lexical items. This has been illustrated in Section 3.2.3 on page 52

3.4. Use of Format Operations

To illustrate the use of the operations of the package *FORMAT*, we consider how the following piece of ADA source could be formatted.

```
procedure foobar(param1: in type1; param2: in
type2) is
begin statement1; -- comment1
statement2; end foobar;
```

The procedure *ResForm* appends reserved words to the output text. In this example the reserved words are "procedure", "in", "is", "begin", and "end". The identifiers ("foobar", "param1", "param2", "type1", and "type2") are appended using the procedure *IdentForm*. *ComForm* is used for adding the comment. We assume the two statements are short enough to be appended using *AddText*. The following lists in order the successive calls to entries in *FORMAT* that are needed to format the source code in the example.

```
(...)
ResForm("procedure ");          -- includes space after "procedure"
IdentForm("foobar");
AddText("(");
SetIndent(Position);            -- set indent to line up parameters
IdentForm("param1");
(...)                           -- here ": in type1;" is output
NewLine;                        -- new line gets indentation
IdentForm("param2");
(...)                           -- here ": in type2" is output
Addtext(")");
Undent;                         -- remove indentation
ResForm("is");
NewLine;                        -- new line gets no indentation
ResForm("begin");
Indent;                         -- indent by increment for block
NewLine;                        -- new line gets indentation
AddText("statement1;");
ComForm("comment1");            -- "--" is added by ComForm
AddText("statement2");
Undent;                         -- undent - end of block
NewLine;                        -- new line gets no indentation
ResForm("end ");
IdentForm("foobar");
AddText(";");
(...)
```

The resultant formatted program is more readable.

```
procedure foobar(param1: in type1;
                 param2: in type2) is
begin
    statement1;                 -- comment1
    statement2;
end foobar;
```

CHAPTER 4
RECREATING FORMATTED SOURCE

This chapter presents the complete design of the DIANA to ADA pretty-printer, PRETTYPRINT. It is a blend of the simple source reconstruction described in Chapter 2 with the formatting operations outlined in Chapter 3.

The formatted source is constructed in two passes over the DIANA-like structure (it is in fact PP_DIANA, a refinement of DIANA). The first pass, *WALK1*, computes the maximum nesting depth of the program and the number of characters in the recreated source disregarding indentation and comments. The second pass, *WALK2*, reconstructs the source using the formatting operations of package *FORMAT*. The second pass uses the character count information computed in the first pass to make formatting decisions about indentation and where to break lines.

This chapter begins with the definition of the refinement of DIANA that defines PP_Diana, a structure with attributes to record the results of the first pass. Subsequent sections describe the two passes in detail.

4.1. IDL Refinement of DIANA

The IDL design provides two methods for defining an IDL structure in terms of a previously defined IDL structure. *Derivation* is one method: Appendix II of the DRM describes the ADA abstract parse tree as a derivation of the DIANA definition. Derivation allows the deletion and addition of IDL type, node, and class definitions.

Refinement is the other means of defining a new IDL structure in terms of an existing structure. In refinement, only IDL additions are permitted; deletions are not. DIANA_Concrete, defined at the end of Chapter 2 of the DRM, is a refinement of DIANA. For a complete discussion of the semantics of these features of IDL, refer to the "IDL - Interface Description Language Formal Description" [4].

We use IDL refinement to define a structure that is DIANA augmented with attributes for pretty-printing. We add three attributes to the structure.

pp_chars All nodes have this attribute. It represents the number of characters needed to print the text for the node, ignoring indentation and comments.

pp_max_chars	This is an attribute of only nodes with the *as_list* attribute. The value of *pp_max_chars* is the maximum of *pp_chars* for each node in the sequence that is the *as_list* attribute.
pp_indent	This is an attribute of the root node, **compilation**. The value of this attribute is the maximum nesting depth below this node. It measures indentation only for block and control structure nesting.

The refinement of a structure is specified with the following IDL syntax.

```
Structure AnotherName Refines SomeName Is
    — Additional IDL statements to define further the
    — structure SomeName, such as a specification  of the
    — internal and external representations for private
    — types in the abstract structure Somename.
    — New nodes may be defined.
    — New attributes may be defined.
    End
```

Consider the definition of the IDL structure SomeName described in Section Section 1.4 of the DRM and repeated in Figure 4-1 on page 65. Following the definition of SomeName in the same figure is an IDL definition of the IDL structure, RefinedName. RefinedName is a refinement of the IDL structure SomeName that adds the attributes *pp_chars* to the nodes **tree** and **leaf**.

The effect of the refinement is that in the IDL structure RefinedName, the node **tree** now has the attributes *op* and *src* defined in the IDL specification of SomeName, and the attribute *pp_chars* as defined in the refinement. The effect is as if the node had been defined with three attributes originally.

We define a refinement of DIANA, PP_DIANA, that is the structure necessary for PRETTYPRINT. The entire IDL refinement is included as Appendix C.

The input to PRETTYPRINT is DIANA. The structure that *WALK1* modifies and *WALK2* uses is PP_DIANA. The process by which DIANA is modified into PP_DIANA is not specified in this design. It is an assumption of the design that the operation that reads the DIANA, the procedure *Get_PP_Tree* of package *MAIN*, returns to *MAIN* a PP_DIANA tree.

An IDL processor can be used to create the interface programs for PrettyPrint. An IDL processor can generate both a reader, to read an ASCII representation of DIANA and return a PP_DIANA tree, and an interface program

```
Structure ExpressionTree Root EXP Is

      -- First we define a private type.

      Type Source_Position;


      -- Next we define the notion of an expression, EXP.

      EXP  ::=  leaf | tree ;


      -- Next we define the nodes and their attributes.

      tree  =>  op: OPERATOR,  left: EXP,  right: EXP ;
      tree  =>  src: Source_Position ;
      leaf  =>  name: String ;
      leaf  =>  src: Source_Position ;


      -- Finally we define the  notion of an OPERATOR as the
      -- union of a collection of nodes; the null => productions
      -- are needed to define the node types since
      -- node type names are never implicitly defined.

      OPERATOR  ::=  plus | minus | times | divide ;

      plus => ;   minus => ;   times => ; divide => ;

End


-- Define a new structure by refinement of the old

Structure RefinedName Refines SomeName Is

       -- add the attribute pp_chars to
       -- leaf and tree

       tree    =>    pp_chars: Integer;
       leaf    =>    pp_chars: Integer;
End
```

Figure 4-1: Example of IDL Refinement

that receives as input the internal form of DIANA and returns a PP_DIANA tree. To automatically generate the interface, the IDL processor needs a specification of the process.

An IDL process specification describes the structure of the input data, output data, and the internally used data structure. Process specifications are described in the IDL manual. A simple process specification is shown below. In the example, the process SomeProcess is defined as a process. The input port declaration (begun with the IDL keyword "Pre") names the input port, "Inport", and states the data will be the IDL structure "SomeName". Likewise the output declaration names the port "Outport" and defines the output as a "RefinodName" structure.

```
Process SomeProcess Is

    -- define the input structure

    Pre Inport : SomeName;

    -- define the output structure

    Post Outport : RefinedName;

End
```

Many processes, including PRETTYPRINT, use a different data structure internally. The IDL definition of the process PRETTYPRINT, Figure 4-2, has an invariant clause ("Inv PP_Diana") to show that PRETTYPRINT uses PP_DIANA internally. The process specification for PRETTYPRINT does not specify any output. The output of PRETTYPRINT is a text file which is not an IDL structure and is not considered in the IDL process specification.

The process specification for PRETTYPRINT, shown in Figure 4-2, defines the necessary interface for PRETTYPRINT. An IDL processor can take this specification, along with the definition of DIANA and PP_DIANA, and create the interface programs for PRETTYPRINT.

4.1.1. Operations on PP_Diana

PP_DIANA, like DIANA, is an abstract data type. In Chapter 4 of the DRM the DIANA operations are defined. In this chapter we define additional operations that access the attributes defined in the refinement.

Chapter 4 of the DRM provides the specification of the ADA package *Diana*.

```
Process PrettyPrint Inv PP_Diana Is

    — PrettyPrint uses PP_Diana internally

    — the only input is Diana

    Pre Inport: Diana;

End
```

Figure 4-2: IDL Process Specification of PrettyPrint

The package is also discussed in Section 1.2.2.1 on page 9 of this document. The package *Diana* provides operations on the DIANA data type. We here define the package *PP_Diana* that provides the operations on the data type PP_DIANA.

Because PP_DIANA is a refinement of DIANA, the packaga must contain all of the operations in the package *Diana*. In addition, the package must contain operations on the three *pp_* attributes that have been added. Figure 4-3 shows the package that defines the operations. This package adds six new operations. For each attribute there are two new subprograms: a procedure used to set the value of the attribute and a function used to read the value of the attribute.

```
with USERPK; use USERPK;
package PP_Diana is

(...)

— the package contains every operation in package DIANA

procedure PP_CHARS(t: in out TREE; v: in INTEGER);
function PP_CHARS(t: in TREE) return INTEGER;

procedure PP_MAXCHARS(t: in out TREE; v: in INTEGER);
function PP_MAXCHARS(t: in TREE) return INTEGER;

procedure PP_NEST(t: in out TREE; v: in INTEGER);
function PP_NEST(t: in TREE) return INTEGER;

private

(...)            — not considered here

end PP_Diana;
```

Figure 4-3: PP_DIANA Operations

4.2. The First Tree Walk — WALK1

In the first walk over the PP_DIANA tree the values of the *pp_* attributes are computed. The tree walk is designed to use the general tree traversal operations of the package *Diana* described in Chapter 4 of the DRM.

In essence, during the first pass over the tree the number of characters required to print each node, and the level of nesting at the root are passed up the tree. The number of characters needed to print a node is related (through the addition of a constant) to the number of characters needed to print the nodes that are its structural descendants. The number of characters needed to print a leaf of the tree is related to the length of its lexical (*lx_symrep* or *lx_numrep*) attribute.

Similarly, the nesting depth below a node is related to the nesting depth of its structural offspring. In particular each offspring is indented some amount (possibly zero) from the parent node. This amount is added to the nesting depth of the child and compared to the values for the other offspring. The maximum determines the nesting depth at the node. The nesting depth for each leaf is zero.

The number of characters required to print a node is needed during the second walk over the tree and is recorded at each node as the *pp_chars* attribute. The nesting depth is only needed at the root so compilation is the only node with the attribute *pp_nest*. (In the IDL refinement that defines PP_Diana, in Appendix C, *pp_nest* appears in only one place).

Nodes with *as_list* attributes have the attribute *pp_maxchars*. This attribute stores the maximum of the *pp_chars* attributes of the nodes in the the *as_list* sequence. This value is recorded for use in the second pass also.

The package *WALK1* specification and body is shown in Figure 4-4 on page 69. The package specification shows that one procedure is exported, also named *WALK1*, and this procedure operates on an object of type *PP_Diana.TREE*. *TREE* is a type defined in the package *PP_Diana*. All nodes in PP_Diana are of type *TREE*. When *WALK1* is called it expects to be passed the root node of a PP_DIANA structure.

The package body shows that, in addition to *WALK1*, there are two mutually recursive subprograms, *Listwalk* and *Nodewalk*, needed to traverse the tree. In addition there are two constant arrays available: *Nest* and *Char*. The following

```
-- Package to perform the first walk over the tree.

with PP_Diana; use PP_Diana;           -- DIANA package, for pretty printing

package WALK1 is
    procedure WALK1(T: in out TREE);
end WALK1;




-- Package to perform the first walk over the tree.

package body WALK1 is

    Son_Count : ARITIES range unary .. ternary;

    Nest: constant array (NODE_NAME,Son_Count) of NATURAL
          := ( ... );
    Char : constant array (NODE_NAME) of NATURAL
          := ( .. );

function Max(X: in NATURAL; Y: in NATURAL)
        returns NATURAL is separate;

procedure ListWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL;
                   MaxLength : out NATURAL);

procedure NodeWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL);

procedure Walk1(T: in out TREE) is separate;

procedure ListWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL;
                   MaxLength : out NATURAL) is separate;

procedure NodeWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL) is separate;

end WALK1;
```

Figure 4-4:   Package Walk1 Specification And Body

sections define *Nest*, *Char*, *Listwalk* and *Nodewalk* in detail.

4.2.1. Subunit Walk1

The subunit that defines procedure *WALK1* is shown in Figure 4-5 on page 70.

```
separate (WALK1)
procedure Walk1(T: in out TREE) is

    Depth:     NATURAL;
    Length:    NATURAL;
    Maxlength: NATURAL;

begin

    -- the root is a compilation node, to be treated as a list
    -- ListWalk returns the nesting depth => Depth
    -- the number of chars => Length
    -- and the size of largest comp_unit => MaxLength

    ListWalk (T,Depth,Length,MaxLength);

    PP_CHARS(T,Length);
    PP_MAXCHARS(T,MaxLength);
    PP_INDENT(T,Depth);

end WALK1;
```

Figure 4-5: Walk1 Subunit

WALK1 receives as input the node that is the root of the source program, a compilation node. The node has one structural attribute, *as_list* that is a sequence of comp_unit nodes. The function *LIST* returns the sequence. *WALK1* calls the procedure Listwalk to walk down the sequence computing the values for the number of characters (Length), the maximum number of characters in any compilation unit (MaxLength), and the maximum nesting depth in any compilation unit (Depth). *WALK1* then uses the *PP_Diana* operations to set the value of the *pp_chars*, *pp_maxchars* and *pp_nest*. Compilation is the only node with all three *pp_* attributes.

4.2.2. Nest: Nesting Constant Array

The constant array *Nest* is a doubly-subscripted array. The first subscript is for indexing by node type, the second subscript for indexing by the structural offspring of a node. For example, the entry *Nest*(dn_block,2) returns the amount the second child of the **block** node is indented. Nodes with less than three offspring have the value zero in entries for non-existent children. Nodes with no offspring, the leafs, have the value zero for all entries. Nodes with the attribute *as_list* only are considered to have one child, the child being the sequence of nodes.

As an example we consider the node **accept**, used to denote an accept statement. The ADA syntax for the accept statement 's

```
accept_statement ::=
    accept entry_name(formal_part) do
        sequence_of_statements
    end entry_name;
```

The IDL description of the DIANA node **accept** is

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

The first and second children of the node, *as_name* and *as_param_assoc_s* do not generate text that is indented. (The attributes represent the entry_name and the formal_part respectively). The third child, *as_stm_s* does represent text that is indented, the sequence of statements. The value of *Nest* for this node conveys this information.

```
Nest: constant array (NODE_NAME,Son_Count) of NATURAL
        := (dn_accept => (0,0,1),      -- only indent the third son
            ... );
```

Nest is used to compute the nesting depth. For each node **foo**, the nesting depth for the *i*th child is the nesting depth for the node that is the child, plus the value of *Nest*(dn_Foo,*i*). The nesting depth for the node is the maximum of the nesting depths for each child.

Note that the nesting depth represented in *Nest* is only an approximation of the indentation. It is possible for the indentation to be affected by other factors. For instance, a statement that continues beyond the end of a line may be indented on the next line.

4.2.2.1. Practical Considerations in Nest

The value of *Nest* for the node **record** accounts for the nesting of its children *and* the fact it is nested within its parent. The ADA syntax for a record type specification is

```
type_declaration ::=
    type identifier is type_definition;

record_type_definition ::=
        record
            component_list
        end record
```

In general the type definition of a type declaration is not indented. However, when the type definition is a record type definition the keyword "**record**" should be indented. This information cannot be stored with the node **type** so it is added to the ***record***. *The value of Nest for* ***record*** would ordinarily represent the fact the component list is indented once from the record definition. To compensate this value is changed to two to account for the indentation of the record definition.

```
Nest: constant array (NODE_NAME,Son_Count) of Integer
      := (dn_record => (2,0,0), ... );
```

This entry indicates that the text for the first child, the component list of the record declaration, should be indented two levels from the rest of the text of the node. In reality, the record node is indented once and the component list is indented once again.

4.2.3. Char: the Character Count Constant Array

The constant array *Char* defines the number of characters necessary, in addition to the structural offspring, the print the node. We use the example of a *record* type specification.

The number of characters to print the record definition is the number of characters for the component list plus the number of characters needed to print "***record*** ", *and* "**end record**" (note the spaces after "record" needed to separate the text from the component_list). The entry in *Char* for the node **record** reflects this knowledge.

```
Char : constant array (NODE_NAME) of Integer
      := (dn_record => 17, ...);
```

The value seventeen indicates that printing a record type specification requires seventeen characters in addition to what is required to print the component_list. Specifically, the seventeen characters are "**record** " (seven characters -- including the space) and "**end record**" (ten characters).

Note that *Char* does not account for indentation. In the case of the record definition the character count assumes that all of the text appears on the same line.

4.2.3.1. Practical Considerations in Char

There are some character counts that *Char* cannot represent. The most notable is the number of characters for variable object declarations (and constant and in parameter declarations). If the variable object declaration has a defining expression the characters ":=" must be accounted for. If the expression if absent that characters will not appear. The value in *Char* includes the count for the ":=". Although this affects the value of the *pp_chars* attribute for the node, the effect can be accounted for in *WALK2*. When processing a **var** node with a **void** *as_object_def*, *WALK2* can subtract two characters from the total represented in *pp_chars*.

There are other instances of small inaccuracies in the generated character count. None are serious. Although the global total, *pp_chars* at the **compilation** node, is only approximate, the local totals needed for formatting decisions are accurate. For example in **block** nodes the ADA keyword "**declare**" appears only when the node represents a block statement with a non-empty list of declarations. This situation is not serious because *WALK2* uses the character counts to determine line breaks. In a block statement "**declare**" is preceded and followed by line breaks and does not influence the line break decisions for the list of declarations.

The computation of the length of lists is tricky. Consider the formal part of a subprogram declaration. It is a list of parameter specifications, separated by semicolons, enclosed by parentheses. The number of semicolons is dependent on the size of the list: when there are three parameters there are two semi-colons. *WALK1* does not count the number of items in any lists. *Char* accounts for this by adding one to the character counts of each parameter specification (they only occur in this context), and accounts for the extra semicolon in the list by subtracting one from the character count for the formal part. This method is also used for identifier lists.

4.2.4. Nodewalk

The subunit for the procedure *Nodewalk* is shown in Figure 4-6 on page 75. The procedure *NodeWalk* is used to walk down the structural children of nodes. For a node it produces two out parameters:

Depth	the nesting depth bel⁻w this node, and
Length	the number of characters needed to print the node.

The procedure *Nodewalk* traverses the tree using the general tree traversal operations supplied by the package *Diana*. The function *ARITY* returns a value of type *ARITIES* that indicates the structure of the node, *i.e.*, the number of offspring the node has:

nullary	indicates no offspring (a leaf).
unary indicates one offspring.	
binary	indicates two offspring.
ternary	indicates three, and
arbitrary	indicates the node has as its descendant a sequence of nodes (*i.e.*, has an *as_list* attribute).

The functions *SON1*, *SON2*, and *SON3* are used to access the structural attributes. The functior *SON1* returns the node that is the first child. The subprograms *SON2* and *SON3* similarly return the second an third offspring.

The function *KIND* returns a value indicating the type of node. This value can be used to index into *Char* and *Nest*.

Nodewalk processes the node based on its structure, using the value returned by *ARITY* to discriminate between nodes. When the node is unary, binary, or ternary, *Nodewalk* is recursively called to compute the values of nesting depth and character count for the descendants. It returns the nesting depth for the node, and the character count (the sum of the character counts of the children and the value of *Char* for the node).

When the node is nullary, a leaf, a further discrimination must be done. The node may have no attributes of interest (such as **null_statement**), only the *lx_symrep* attribute, or only the *lx_numrep* attribute. IDL private types symbol_rep and number_rep are implemented so the function *Length* returns the number of

```
separate (WALK1)
procedure NodeWalk
        (Node : in out TREE;
         Depth : out NATURAL;
         Length : out NATURAL) is

    LocalDepth,ReturnedDepth: NATURAL := 0;
    LocalLength,ReturnedLength: NATURAL := 0;
    LocalMaxlength : NATURAL := 0;
    WhichSon : TREE;

begin

    case ARITY(node) is

        when nullary =>
            Depth := 0;                -- leaf nodes have zero nesting
            case KIND(node) is
                when dn_and_then | (...) =>
                    LocalLength := Char(KIND(Node));
                when dn_numeric_literal =>
                    LocalLength := Length(LX_NUMREP(node)) +
                              Char(KIND(Node));
                when others =>
                    LocalLength := Length(LX_SYMREP(Node)) +
                              Char(KIND(Node));
            end case;
        when unary | binary | ternary =>
            for Son in unary .. ARITY(Node)
            loop
                case Son is
                    when unary =>
                        WhichSon := SON1(node);
                    when binary =>
                        WhichSon := SON2(node);
                    when ternary =>
                        WhichSon := SON3(node);
                end case;
                NodeWalk(WhichSon,ReturnedDepth,Returnedlength);
                Depth := Max(LocalDepth, ReturnedDepth +Nest(KIND(node),Son));
                LocalLength := LocalLength + ReturnedLength;
            end loop;
        when arbitrary =>
            Listwalk(Node,ReturnedDepth,ReturnedLength,Localmaxlength);
            Depth := ReturnedDepth + Nest(KIND(Node),1);
            Length := ReturnedLength + Char(KIND(Node));
                                       -- set value of pp_maxchars
            PP_MAXCHARS(Node,LocalMaxlength);
    end case;
    PP_CHARS(Node,Length);             -- set value of pp_chars attribute
    Length := LocalLength;
end NodeWalk;
```

Figure 4-6: Procedure Nodewalk

characters in their representation.

When the node is arbitrary (its descendant is a sequence) the procedure *Listwalk* is called to process the node. The value of the *pp_maxchars* attribute is set to the value returned by MaxChar of *Listwalk*.

The last statement of the procedure sets the value of the attribute *pp_chars*. This attribute is recorded with all nodes in the tree.

Note that when *Nodewalk* computes the number of characters needed to print a node it considers neither the comment for the node nor the indentation of the node. The value of *pp_chars* represents the number of characters needed to print the source, without comments, on an arbitrarily long line.

4.2.5. Listwalk

The subunit for the procedure *Listwalk* is shown in Figure 4-7 on page 77. The procedure walks down a sequence, of type SEQ_TYPE, and produces values for three out parameters:

Depth	the maximum nesting depth of all nodes in the sequence;
Length	the number of characters needed to print the sequence -- this is the sum of the number of characters needed to print each node in the sequence; and
Maxlength	the maximum number of characters needed to print any one node from the list.

Listwalk receives as input a node of type 'arbitrary', *i.e.*, a node with an *as_list* attribute. The *PP_Diana* function *LIST* returns the sequence for the node. *HEAD* returns the node at the head of the list; *TAIL* returns the sequence that remains after removing the head. The function *IS_EMPTY* returns true if the sequence has no nodes. Thus the inner loop is executed once for every node in the sequence. During each iteration the depth is computed to be the maximum of the previously computed depth and the nesting depth for the current node; maxlength is computed analogously. The length of the sequence is computed by as an accumulated sum.

Listwalk does not set the value of any attributes directly -- it returns values through its out parameters. The attributes are set in *Nodewalk* or *WALK1*.

```
separate (WALK1)
procedure ListWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL;
                   MaxLength : out NATURAL) is
    LocalLength,ReturnedLength : NATURAL := 0;
    LocalMaxlength : NATURAL := 0;
    LocalDepth,ReturnedDepth : NATURAL := 0;
    Seq: SEQ_TYPE;
    Hd: TREE;

begin
    Seq:= LIST(Node);
    LocalDepth := 0; LocalLength := 0; LocalMaxLength :=0;
    while not IS_EMPTY(Seq)
    loop
        Hd := HEAD(Seq);
        NodeWalk(Hd,ReturnedDepth,ReturnedLength);
        LocalMaxlength := Maximum(LocalMaxlength,ReturnedLength);
        LocalLength := LocalLength + ReturnedLength;
        LocalDepth := Maximum(LocalDepth,ReturnedDepth);
        Seq := TAIL(Seq);
    end loop;

    Depth := LocalDepth;
    Length := LocalLength;
    MaxLength := LocalMaxlength;
end ListWalk;
```

Figure 4-7: Procedure Listwalk

4.3. Second Tree Walk to Generate Formatted Text

WALK2 walks over the PP_DIANA structure producing formatted source. This pass is similar in structure to the tree walk described in Chapter 2 in that it creates the source as it walks the PP_DIANA structure. It is different in that it uses the operations supplied in *FORMAT* to make formatting decisions as it proceeds.

The package specification for the second traversal is repeated below.

```
-- Package to perform the second walk over the tree.

with PP_Diana; use PP_Diana;          -- DIANA package, for pretty printing

package WALK2 is
    procedure WALK2(T: in TREE);
end WALK2;
```

The package body for this package is very large. There are approximately 160 mutually recursive procedures used to traverse the tree. The package body is provided in Appendix B.

There is a procedure for nearly every DIANA node, and procedures for many of the DIANA classes. In particular, not every DIANA class is represented by a separate procedure. Classes consisting of one node do not appear; instead the procedure for the node is used. (These cases are easily recognized in DIANA -- the class that contains only the node "**foo**" is named "***FOO***"). Further, the nodes in the classes ***ID*** and ***OP*** do not have separate procedures. These nodes can be processed by the procedure for the class ***DESIGNATOR*** since all nodes in this class have the *lx_symrep* attribute and the only text associated with the node is the lexeme contained in *lx_symrep*.

Appendix B contains the package body for *WALK2*. Also, a few of the subunits for the stubs in the body are included in this appendix.

The naming of the procedures in the package is straightforward. The procedure that produces formatted source for the node **foo** is preceded by the prefix "dp_" to produce the name of the procedure: "dp_foo". Procedures for IDL classes derive their name similarly from the class name. Class ***FOO*** is processed by procedure "dk_FOO".

4.3.1. Use of DIANA Operators

The general node operations (SON1, SON2, *etc.*) are used for the first pass over the tree. During the second pass the tree walking procedures are more specific and consequently the specific DIANA node operations, operations that address attributes by name (*e.g.*, AS_EXP), are used. (The use of both traversal methods is driven by the design goal of stressing the package *Diana* as stated in Section 1.1 on page 5.)

In general, there are two operations for each attribute, that is two subprograms defined in the package *PP_Diana*. The operations are named as the attribute, so there are two subprograms associated with attribute *foo*.

```
procedure FOO (t: in out TREE; v: in TREE);
    — This procedure  sets the foo attribute in the node "t"
    — with the value "v"
function FOO (t: in TREE) return TREE;
    — This function returns the value of the foo attribute of
    — node "t"
```

The attribute *as_list* is a special case. The function *LIST* returns the value, a sequence, of type *(SEQ_TYPE*. Sequence types are handled using the operations HEAD, TAIL, and IS_EMPTY as previously described in Section 4.2.

The *pp__* attributes are accessed using similar operations, described in Section 4.1.

4.3.2. Example WALK2 Subunits

In this section we present two sample subunits from *WALK2*. For purposes of comparison we reconsider the reconstruction of label identifiers and task types that were first introduced in Chapter 2

4.3.2.1. Label Identifiers

All identifiers are processed by a common routine.

```
procedure dk_ID(Node: in TREE) is

— dk_ID is used for all identifier nodes, including DEF_ID, and USED_ID
— since all elements in the class have only one attribute of interest,
— they are all processed by a single procedure

begin
    IdentForm( LX_SYMREP(Node ));
end dk_ID;
```

In the final design, the treatment of the labels is merged into the procedure that handles sequences of identifiers. Here a check for the type of the node determines if the identifier is to be bracketed. In addition, some formatting decisions are made. The length of the sequence is compared with the remaining space on the line. If there isn't enough space for the entire sequence, then each identifier is placed on a separate line.

```
procedure dp_Id_s(Node: in TREE) is
    Toolong : Boolean;
    Seq : SEQ_TYPE;
-- when TooLong is true put each identifier on a separate line.
-- Checks for labels, and brackets labels.
begin
    TooLong := Max_Id_Width < PP_CHARS(Node);
    Seq := LIST(node);
    if KIND(HEAD(Seq)) = dn_label_id then
        AddText("<<");
        dk_ID(HEAD(Seq));
        AddText(">>");
    else
        dk_ID(HEAD(Seq));
    end if;
    Seq := TAIL(Seq);
    while not IS_EMPTY(Seq)
    loop
        if KIND(HEAD(Seq)) = dn_label_id then
            if TooLong then NewLine end if;
            AddText("<<");
            dk_ID(HEAD(Seq));
            AddText(">>");
        else
            AddText(",");
            if TooLong then NewLine end if;
            dk_ID(HEAD(Seq));
        end if;
    end loop;
end dp_ID_S;
```

4.3.2.2. Task Types

The procedure that prints all type declaration must determine if the type specification is a task declaration. If it is, then the keyword "task" must be printed. The procedure below is very similar to the procedure in Section 2.2.2 on page 35. The difference is that the specialized output routines ResForm and AddText are used to create the output line.

```
procedure dp_Type(Node: in TREE) is

begin
     if KIND(AS_TYPE_SPEC(Node)) = dn_task_spec then
          ResForm("task ");
     end if;
     ResForm("type ");
     dk_ID(AS_ID(Node));
     dp_Var_s(AS_VAR_S(Node));
     ResForm(" is ");
     dk_TYPE_SPEC(AS_TYPE_SPEC(Node));
     AddText(";");
end dp_Type;
```

APPENDIX A
FIRST TRAVERSAL

This appendix lists the ADA package *WALK1* that contains the procedures to perform the first tree-walk of the DIANA tree. The discussion of this package is in Section 4.2.

A.1. Package Specification

```
-- Package to perform the first walk over the tree.

with PP_Diana; use PP_Diana;            -- DIANA package, for pretty printing

package WALK1 is
    procedure WALK1(T: in out TREE);
end WALK1;
```

A.2. Package Body

```
-- Package to perform the first walk over the tree.

package body WALK1 is

    Son_Count : range 1..3;
    Nest: constant array (NODE_NAME,Son_Count) of Natural
          := ( dn_record              => (1,0,0),
               dn_variant_part        => (0,1,0),
               dn_cond_clause         => (0,1,0),
               dn_alternative_s       => (1,0,0),
               dn_alternative         => (0,1,0),
               dn_loop                => (0,1,0),
               dn_block               => (1,1,1),
               dn_package_spec        => (1,1,0),
               dn_task_spec           => (1,0,0),
               dn_accept              => (0,0,1),
               dn_select              => (0,1,0),
               dn_select_clause       => (0,1,0),
               dn_cond_entry          => (1,1,0),
               dn_timed_entry         => (1,1,0),
               others                 => (0,0,0));

    Char : constant array (NODE_NAME) of Natural
          := ( dn_pragma              => 8,
               dn_param_assoc_s       => 1,
               dn_constant            => 14,
               dn_var                 => 4,
               ...
               dn_code                => 0 );

procedure ListWalk(Node : in out SEQ_TYPE;
                   Depth : out Natural;
                   Length : out Natural;
                   MaxLength : out Natural);

procedure NodeWalk(Node : in out TREE;
                   Depth : out Natural;
                   Length : out Natural);

procedure Walk1(T: in out TREE) is separate;

procedure ListWalk(Node : in out SEQ_TYPE;
                   Depth : out Natural;
                   Length : out Natural;
                   MaxLength : out Natural) is separate;

procedure NodeWalk(Node : in out TREE;
                   Depth : out Natural;
                   Length : out Natural) is separate;

end WALK1;
```

A.3. Subunits

```
separate (WALK1)
procedure Walk1(T: in out TREE) is

    Depth:      NATURAL;
    Length:     NATURAL;
    Maxlength: NATURAL;

begin

    -- the root is a compilation node, to be treated as a list
    -- ListWalk returns the nesting depth => Depth
    -- the number of chars => Length
    -- and the size of largest comp_unit => MaxLength

    ListWalk (T,Depth,Length,MaxLength);

    PP_CHARS(T,Length);
    PP_MAXCHARS(T,MaxLength);
    PP_INDENT(T,Depth);

end WALK1;


separate (WALK1)
procedure ListWalk(Node : in out TREE;
                   Depth : out NATURAL;
                   Length : out NATURAL;
                   MaxLength : out NATURAL) is
    LocalLength,ReturnedLength : NATURAL := 0;
    LocalMaxlength : NATURAL := 0;
    LocalDepth,ReturnedDepth : NATURAL := 0;
    Seq: SEQ_TYPE;
    Hd: TREE;

begin
    Seq:= LIST(Node);
    LocalDepth := 0; LocalLength := 0; LocalMaxLength :=0;
    while not IS_EMPTY(Seq)
    loop
        Hd := HEAD(Seq);
        NodeWalk(Hd,ReturnedDepth,ReturnedLength);
        LocalMaxlength := Maximum(LocalMaxlength,ReturnedLength);
        LocalLength := LocalLength + ReturnedLength;
        LocalDepth := Maximum(LocalDepth,ReturnedDepth);
        Seq := TAIL(Seq);
    end loop;

    Depth := LocalDepth;
    Length := LocalLength;
    MaxLength := LocalMaxlength;
end ListWalk;
```

```
separate (WALK1)
procedure NodeWalk
        (Node : in out TREE;
         Depth : out NATURAL;
         Length : out NATURAL) is

    LocalDepth,ReturnedDepth: NATURAL := 0;
    LocalLength,ReturnedLength: NATURAL := 0;
    LocalMaxlength : NATURAL := 0;
    WhichSon : TREE;

begin

    case ARITY(node) is

        when nullary =>
            Depth := 0;                    -- leaf nodes have zero nesting
            case KIND(node) is
                when dn_and_then | (...) =>
                    LocalLength := Char(KIND(Node));
                when dn_numeric_literal =>
                    LocalLength := Length(LX_NUMREP(node)) +
                              Char(KIND(Node));
                when others =>
                    LocalLength := Length(LX_SYMREP(Node)) +
                              Char(KIND(Node));
            end case;
        when unary | binary | ternary =>
            for Son in unary .. ARITY(Node)
            loop
                case Son is
                    when unary =>
                        WhichSon := SON1(node);
                    when binary =>
                        WhichSon := SON2(node);
                    when ternary =>
                        WhichSon := SON3(node);
                end case;
                NodeWalk(WhichSon,ReturnedDepth,Returnedlength);
                Depth := Max(LocalDepth, ReturnedDepth +Nest(KIND(node),Son));
                LocalLength := LocalLength + ReturnedLength;
            end loop;
        when arbitrary =>
            Listwalk(Node,ReturnedDepth,ReturnedLength,Localmaxlength);
            Depth := ReturnedDepth + Nest(KIND(Node),1);
            Length := ReturnedLength + Char(KIND(Node));
                -- set value of pp_maxchars
            PP_MAXCHARS(Node,LocalMaxlength);
    end case;
    PP_CHARS(Node,Length);              -- set value of pp_chars attribute
    Length := LocalLength;
end NodeWalk;
```

APPENDIX B
SECOND TRAVERSAL

This appendix lists the ADA package *WALK2* that contains the procedures to perform the second tree-walk of the DIANA tree. The discussion of this package is in Section 4.3.

B.1. Package Specification

```
-- Package to perform the second walk over the tree.

with PP_Diana; use PP_Diana;           -- DIANA package, for pretty printing

package WALK2 is
   procedure WALK2(T: in TREE);
end WALK2;
```

B.2. Package Body

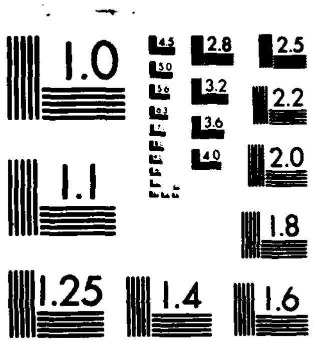
```
-- Package to perform the second walk over the tree.

package body WALK2 is

procedure Walk2(T: in TREE) is separate;

-- procedure stubs for the second traversal
-- organized by Ada LRM chapter

-- 2.  Lexical Elements
-- ======================

procedure dp_void (Node: in TREE) is separate;

 -- 2.3 Identifiers, 2.4 Numeric Literals, 2.6 String Literals

procedure dk_DESIGNATOR (Node: in TREE);

-- 2.8  Pragmas

procedure dp_pragma (Node: in TREE) is separate;

procedure dp_param_assoc_s (Node: in TREE) is separate;

-- 3.  Declarations and Types
-- ==========================
-- 3.1  Declarations

procedure dk_DECL (Node: in TREE);

-- 3.2  Objects and Named Numbers
```

```
procedure dk_OBJECT_DEF (Node: in TREE);

procedure dk_EXP_VOID (Node: in TREE);

procedure dp_constant (Node: in TREE) is separate;

procedure dp_var (Node: in TREE) is separate;

procedure dp_number (Node: in TREE) is separate;

procedure dp_id_s (Node: in TREE) is separate;

-- 3.3  Types and Subtypes
-- 3.3.1 Type Declarations

procedure dp_type (Node: in TREE) is separate;

procedure dk_TYPE_SPEC (Node: in TREE);

-- 3.3.2 Subtype Declarations

procedure dp_subtype (Node: in TREE) is separate;

procedure dk_subtype_indication (Node: in TREE);

procedure dp_constrained (Node: in TREE) is separate;

procedure dk_constraint (Node: in TREE);

-- 3.4  Derived Type Definitions

procedure dp_derived (Node: in TREE) is separate;

-- 3.5  Scalar Types

procedure dk_RANGE (Node: in TREE);

procedure dp_range (Node: in TREE) is separate;

-- 3.5.1  Enumeration Types

procedure dp_enum_literal_s (Node: in TREE) is separate;

procedure dk_ENUM_LITERAL (Node: in TREE);

procedure dp_def_char (Node: in TREE) is separate;

-- 3.5.4  Integer Types

procedure dp_integer (Node: in TREE) is separate;

-- 3.5.6  Real Types

-- 3.5.7  Floating Point Types

procedure dk_RANGE_VOID (Node: in TREE);

procedure dp_float (Node: in TREE) is separate;
```

AD-A128 857   A DIANA-DRIVEN PRETTY-PRINTER FOR ADA(U) TARTAN LABS   2/2
INC PITTSBURGH PA K J BUTLER ET AL. 22 FEB 83 TL-83-3
MDA903-82-C-0148

UNCLASSIFIED   F/G 9/2   NL

END
DATE
FILMED
6 83
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

```
— 3.5.9  Fixed Point Types

procedure dp_fixed (Node: in TREE) is separate;

— 3.6  Array Types

procedure dp_array (Node: in TREE) is separate;

procedure dp_dscrt_range_s (Node: in TREE) is separate;

procedure dk_DSCRT_RANGE (Node: in TREE);

procedure dp_index (Node: in TREE) is separate;

— 3.7  Record Types

procedure dp_record (Node: in TREE) is separate;

procedure dk_COMP (Node: in TREE);

procedure dp_null_comp (Node: in TREE) is separate;

— 3.7.1  Discriminants

procedure dk_VAR_S (Node: in TREE);

procedure dp_var_s (Node: in TREE) is separate;

— 3.7.2  Discriminant Constraints

procedure dp_dscrmt_aggregate (Node: in TREE) is separate;

— 3.7.3  Variant Parts

procedure dp_variant_part (Node: in TREE) is separate;

procedure dp_variant_s (Node: in TREE) is separate;

procedure dp_choice_s (Node: in TREE) is separate;

procedure dp_variant (Node: in TREE) is separate;

procedure dp_inner_record (Node: in TREE) is separate;

procedure dk_CHOICE (Node: in TREE);

procedure dp_others (Node: in TREE) is separate;

— 3.8  Access Types

procedure dp_access (Node: in TREE) is separate;

— 3.8.1 Incomplete Type Declarations

procedure dk_TYPE_SPEC (Node: in TREE);

— 3.9  Declarative Parts

procedure dk_ITEM (Node: in TREE);
procedure dp_item_s (Node: in TREE) is separate;

-- 4.  Names and Expressions
-- ------------------------------
-- 4.1  Names

procedure dk_NAME (Node: in TREE);

procedure dp_used_char (Node: in TREE) is separate;

-- 4.1.1  Indexed Components

procedure dk_EXP_S (Node: in TREE);

procedure dp_exp_s (Node: in TREE) is separate;

procedure dp_indexed (Node: in TREE) is separate;

-- 4.1.2  Slices

procedure dp_slice (Node: in TREE) is separate;

-- 4.1.3  Selected Components

procedure dp_selected (Node: in TREE) is separate;

procedure dp_all (Node: in TREE) is separate;

-- 4.1.4  Attributes

procedure dp_attribute (Node: in TREE) is separate;

procedure dp_attribute_call (Node: in TREE) is separate;

-- 4.2  Literals

-- Refer to 4.4.C for numeric_literal, string_literal,
-- and null_access.
-- Refer to 4.1 for character_literal


-- 4.3  Aggregates

procedure dk_EXP (Node: in TREE);

procedure dp_aggregate (Node: in TREE) is separate;

procedure dk_COMP_ASSOC (Node: in TREE);

procedure dp_named (Node: in TREE) is separate;

-- 4.4  Expressions

procedure dp_binary (Node: in TREE) is separate;

procedure dk_BINARY_OP (Node: in TREE);

procedure dp_and_then (Node: in TREE) is separate;

procedure dp_or_else  (Node: in TREE) is separate;
procedure dk_TYPE_RANGE (Node: in TREE);
procedure dp_membership (Node: in TREE) is separate;
procedure dk_MEMBERSHIP_OP (Node: in TREE);
procedure dp_in_op (Node: in TREE) is separate;
procedure dp_not_in (Node: in TREE) is separate;
procedure dp_parenthesized (Node: in TREE) is separate;
procedure dp_numeric_literal (Node: in TREE) is separate;
procedure dp_string_literal (Node: in TREE) is separate;
procedure dp_null_access (Node: in TREE) is separate;
— 4.5  Operators and Expression Evaluation

— 4.6  Type Conversions
procedure dp_conversion (Node: in TREE) is separate;
— 4.7  Qualified Expressions
procedure dp_qualified (Node: in TREE) is separate;
— 4.8  Allocators
procedure dp_allocator (Node: in TREE) is separate;
— 5.  Statements
— ____________
— 5.1  Simple and Compound Statements - Sequences of Statements
procedure dp_stm_s (Node: in TREE) is separate;
procedure dk_STM (Node: in TREE);
procedure dp_labeled (Node: in TREE) is separate;
procedure dp_null_stm (Node: in TREE) is separate;
— 5.2  Assignment Statement
procedure dp_assign (Node: in TREE) is separate;
— 5.3  If Statements
procedure dp_if (Node: in TREE) is separate;
procedure dp_cond_clause (Node: in TREE) is separate;
— 5.4  Case Statements
procedure dp_case (Node: in TREE) is separate;
procedure dp_alternative_s (Node: in TREE) is separate;

procedure dp_alternative (Node: in TREE) is separate;

-- 5.5  Loop Statements

procedure dp_named_stm (Node: in TREE) is separate;

procedure dk_ITERATION (Node: in TREE);

procedure dp_loop (Node: in TREE) is separate;

procedure dp_for (Node: in TREE) is separate;

procedure dp_reverse (Node: in TREE) is separate;

procedure dp_while (Node: in TREE) is separate;

-- 5.6  Block Statements

procedure dp_block (Node: in TREE) is separate;

-- 5.7  Exit Statements

procedure dk_NAME_VOID (Node: in TREE);

procedure dp_exit (Node: in TREE) is separate;        -- is no name given

-- 5.8  Return Statements

procedure dp_return (Node: in TREE) is separate;

-- 5.9  Goto Statements

procedure dp_goto (Node: in TREE) is separate;

-- 6.  Subprograms
-- ---------------
-- 6.1  Subprogram Declarations

procedure dk_SUBPROGRAM_DEF (Node: in TREE);


procedure dp_subprogram_decl (Node: in TREE) is separate;

procedure dk_SUBP_BODY_DESC (Node: in TREE);

-- 'pragma_id' and 'argument_id' only occur in the predefined environment

procedure dk_HEADER (Node: in TREE);

procedure dp_procedure (Node: in TREE) is separate;

procedure dp_function (Node: in TREE) is separate;

procedure dp_param_s (Node: in TREE) is separate;

procedure dk_PARAM (Node: in TREE);

procedure dp_in (Node: in TREE) is separate;
procedure dp_in_out (Node: in TREE) is separate;

procedure dp_out (Node: in TREE) is separate;

-- 6.3  Subprogram Bodies

procedure dk_BLOCK_STUB (Node: in TREE);

procedure dp_subprogram_body (Node: in TREE) is separate;

-- 6.4  Subprogram Calls

procedure dp_procedure_call (Node: in TREE) is separate;

procedure dp_function_call (Node: in TREE) is separate;

procedure dk_PARAM_ASSOC (Node: in TREE);

procedure dp_assoc (Node: in TREE) is separate;

procedure dk_ACTUAL (Node: in TREE);

-- 7.  Packages
-- ___________
-- 7.1  Package Structure

procedure dp_package_decl (Node: in TREE) is separate;

procedure dk_PACKAGE_DEF (Node: in TREE);

procedure dp_package_spec (Node: in TREE) is separate;

procedure dp_decl_s (Node: in TREE) is separate;

procedure dp_package_body (Node: in TREE) is separate;

-- 7.4  Private Type and Deferred Constant Declarations

procedure dp_private (Node: in TREE) is separate;

procedure dp_l_private (Node: in TREE) is separate;


-- 8.  Visibility Rules
-- ____________________
-- 8.4  Use Clauses

procedure dp_name_s (Node: in TREE) is separate;

procedure dp_use (Node: in TREE) is separate;

-- 8.5  Renaming Declarations

procedure dp_rename (Node: in TREE) is separate;

-- 9.  Tasks
-- ________
-- 9.1  Task Specifications and Task Bodies
```

```
procedure dk_TASK_DEF (Node: in TREE);
procedure dp_task_decl (Node: in TREE) is separate;
procedure dp_task_spec (Node: in TREE) is separate;
procedure dk_BLOCK_STUB_VOID (Node: in TREE);
procedure dp_task_body (Node: in TREE) is separate;
-- 9.5  Entries, Entry Calls and Accept Statements
procedure dk_DSCRT_RANGE_VOID (Node: in TREE);
procedure dp_entry (Node: in TREE) is separate;
procedure dp_entry_call (Node: in TREE) is separate;
procedure dp_accept (Node: in TREE) is separate;
-- 9.6  Delay Statements, Duration and Time
procedure dp_delay (Node: in TREE) is separate;
-- 9.7  Select Statements
-- 9.7.1  Selective Waits
procedure dp_select (Node: in TREE) is separate;
procedure dp_select_clause_s (Node: in TREE) is separate;
procedure dp_select_clause (Node: in TREE) is separate;
procedure dk_STM (Node: in TREE);
procedure dp_terminate (Node: in TREE) is separate;
-- 9.7.2  Conditional Entry Calls
procedure dp_cond_entry (Node: in TREE) is separate;
-- 9.7.3  Timed Entry Calls
procedure dp_timed_entry (Node: in TREE) is separate;
-- 9.10  Abort Statements
procedure dp_abort (Node: in TREE) is separate;
-- 10.  Program Structure and Compilation Issues
-- ==============================================
-- 10.1  Compilation Units - Library Units
procedure dp_compilation (Node: in TREE) is separate;
procedure dk_UNIT_BODY (Node: in TREE);
procedure dp_pragma_s (Node: in TREE) is separate;
procedure dp_comp_unit (Node: in TREE) is separate;

-- Context Clauses - With Clauses

procedure dk_CONTEXT_ELEM (Node: in TREE);

procedure dp_context (Node: in TREE) is separate;

procedure dp_with (Node: in TREE) is separate;

-- 10.2  Subunits of Compilation Units

procedure dp_subunit (Node: in TREE) is separate;

procedure dk_SUBUNIT_BODY (Node: in TREE);

procedure dp_stub (Node: in TREE) is separate;

-- 11.  Exceptions
-- ---------------
-- 11.1  Exception Declarations

procedure dk_EXCEPTION_DEF (Node: in TREE);

procedure dp_exception (Node: in TREE) is separate;

-- 11.2  Exception Handlers

-- 11.3  Raise Statements

procedure dp_raise (Node: in TREE) is separate;

-- 12.  Generic Program Units
-- --------------------------
-- 12.1  Generic Declarations

procedure dk_GENERIC_HEADER (Node: in TREE);

procedure dp_generic (Node: in TREE) is separate;

procedure dp_generic_param_s (Node: in TREE) is separate;

procedure dk_GENERIC_PARAM (Node: in TREE);

procedure dk_FORMAL_SUBPROG_DEF (Node: in TREE);

procedure dp_box (Node: in TREE) is separate;

procedure dp_no_default (Node: in TREE) is separate;

procedure dk_FORMAL_TYPE_SPEC (Node: in TREE);

procedure dp_formal_dscrt (Node: in TREE) is separate;

procedure dp_formal_fixed (Node: in TREE) is separate;

procedure dp_formal_float (Node: in TREE) is separate;

procedure dp_formal_integer (Node: in TREE) is separate;
```

```
-- 12.3  Generic Instantiation

procedure dp_generic_assoc_s (Node: in TREE) is separate;

procedure dp_instantiation (Node: in TREE) is separate;

procedure dk_GENERIC_ASSOC (Node: in TREE);

-- 13. Representation Clauses and
-- ==================================
-- Implementation Dependent Features
-- ==============================
-- 13.1  Representation Clauses

procedure dk_REP (Node: in TREE);

-- 13.2 Length Clause
-- 13.3 Enumeration Representation Clauses

procedure dp_simple_rep (Node: in TREE) is separate;

-- 13.4  Record Representation Clauses

procedure dp_alignment (Node: in TREE) is separate;

procedure dp_comp_rep_s (Node: in TREE) is separate;

procedure dp_comp_rep (Node: in TREE) is separate;

-- 13.5  Address Clauses

procedure dp_address (Node: in TREE) is separate;

-- 13.8  Machine Code Insertions

procedure dp_code (Node: in TREE) is separate;


end WALK2;
```

B.3. Subunits

```
separate (WALK2)
procedure Walk2(Node: in out TREE) is
    --
    -- The root node must be a compilation node
    --
begin
    dp_compilation(Node);
end Walk2;
```

APPENDIX C
IDL REFINEMENT OF DIANA FOR PRETTY PRINTING

This appendix lists the refinement of DIANA that is used for the pretty printer. The refinement adds three attributes to DIANA that are useful for formatting the recreated ADA source. The attributes are discussed in Section 4.1 on page 63.

```
 Structure PP_Diana Refines Diana Is
--
-- Pretty Printer Refinement
--
-- Version of 1983 February 22

-- 2.  Lexical Elements
-- ====================

-- 2.3 Identifiers, 2.4 Numeric Literals, 2.6 String Literals

-- 2.8  Pragmas

 pragma =>                    pp_chars: Integer;

 param_assoc_s =>             pp_chars: Integer,
                              pp_maxchars: Integer;

-- 3.  Declarations and Types
-- ==========================
-- 3.1  Declarations

-- 3.2  Objects and Named Numbers

 constant =>                  pp_chars: Integer;

 var =>                       pp_chars: Integer;

 var_id =>                    pp_chars: Integer;

 const_id =>                  pp_chars: Integer;

 number =>                    pp_chars: Integer;

 number_id =>                 pp_chars: Integer;

 id_s =>                      pp_chars: Integer,
                              pp_maxchars: Integer;

-- 3.3  Types and Subtypes
-- 3.3.1 Type Declarations

 type =>                      pp_chars: Integer;
```

```
    type_id =>                  pp_chars: Integer;

-- 3.3.2 Subtype Declarations

    subtype =>                  pp_chars: Integer;

    subtype_id =>               pp_chars: Integer;

    constrained =>              pp_chars: Integer;

-- 3.4  Derived Type Definitions

    derived =>                  pp_chars: Integer;

-- 3.5  Scalar Types

    range =>                    pp_chars: Integer;

-- 3.5.1  Enumeration Types

    enum_literal_s =>           pp_chars: Integer,
                                pp_maxchars: Integer;

    enum_id =>                  pp_chars: Integer;

    def_char =>                 pp_chars: Integer;

-- 3.5.4  Integer Types

    integer =>                  pp_chars: Integer;

-- 3.5.6  Real Types

-- 3.5.7  Floating Point Types

    float =>                    pp_chars: Integer;

-- 3.5.9  Fixed Point Types

    fixed =>                    pp_chars: Integer;

-- 3.6  Array Types

    array =>                    pp_chars: Integer;

    dscrt_range_s =>            pp_chars: Integer,
                                pp_maxchars: Integer;

    index =>                    pp_chars: Integer;

-- 3.7  Record Types

    record =>                   pp_chars: Integer,
                                pp_maxchars: Integer;

    null_comp =>                pp_chars: Integer;

    comp_id =>                  pp_chars: Integer;
```

```
-- 3.7.1  Discriminants

   var_s =>                     pp_chars: Integer,
                                pp_maxchars: Integer;

   dscrmt_id =>                 pp_chars: Integer;

-- 3.7.2  Discriminant Constraints

   dscrmt_aggregate =>          pp_chars: Integer,
                                pp_maxchars: Integer;

-- 3.7.3  Variant Parts

   variant_part =>              pp_chars: Integer;

   variant_s =>                 pp_chars: Integer,
                                pp_maxchars: Integer;

   choice_s =>                  pp_chars: Integer,
                                pp_maxchars: Integer;

   variant =>                   pp_chars: Integer;

   inner_record =>              pp_chars: Integer,
                                pp_maxchars: Integer;

   others =>                    pp_chars: Integer;

-- 3.8  Access Types

   access =>                    pp_chars: Integer;

-- 3.8.1 Incomplete Type Declarations

-- 3.9  Declarative Parts

   item_s =>                    pp_chars: Integer,
                                pp_maxchars: Integer;

-- 4.  Names and Expressions
-- ==========================
-- 4.1  Names

   used_object_id =>            pp_chars: Integer;

   used_name_id =>              pp_chars: Integer;

   used_bltn_id =>              pp_chars: Integer;

   used_op =>                   pp_chars: Integer;

   used_bltn_op =>              pp_chars: Integer;

   used_char =>                 pp_chars: Integer;

-- 4.1.1  Indexed Components
```

```
    exp_s =>                    pp_chars: Integer,
                                pp_maxchars: Integer;

    indexed =>                  pp_chars: Integer;

-- 4.1.2  Slices

    slice =>                    pp_chars: Integer;

-- 4.1.3  Selected Components

    selected =>                 pp_chars: Integer;

    all =>                      pp_chars: Integer;

-- 4.1.4  Attributes

    attribute =>                pp_chars: Integer;

    attribute_call =>           pp_chars: Integer;

-- 4.2  Literals

-- Refer to 4.4.C for numeric_literal, string_literal,
-- and null_access.
-- Refer to 4.1 for character_literal

-- 4.3  Aggregates

    aggregate =>                pp_chars: Integer,
                                pp_maxchars: Integer;

    named =>                    pp_chars: Integer;

-- 4.4  Expressions

    binary =>                   pp_chars: Integer;

    and_then =>                 pp_chars: Integer;

    membership =>               pp_chars: Integer;

    in_op =>                    pp_chars: Integer;

    parenthesized =>            pp_chars: Integer;

    numeric_literal =>          pp_chars: Integer;

    string_literal =>           pp_chars: Integer;

    null_access =>              pp_chars: Integer;

-- 4.5  Operators and Expression Evaluation

-- 4.6  Type Conversions

    conversion =>               pp_chars: Integer;
```

```
-- 4.7  Qualified Expressions

   qualified =>                  pp_chars: Integer;

-- 4.8  Allocators

   allocator =>                  pp_chars: Integer;

-- 5.  Statements
-- ______________
-- 5.1  Simple and Compound Statements - Sequences of Statements

   stm_s =>                      pp_chars: Integer,
                                 pp_maxchars: Integer;

   labeled =>                    pp_chars: Integer;

   label_id =>                   pp_chars: Integer;

-- 5.2  Assignment Statement

   assign =>                     pp_chars: Integer;

-- 5.3  If Statements

   if =>                         pp_chars: Integer,
                                 pp_maxchars: Integer;

   cond_clause =>                pp_chars: Integer;

-- 5.4  Case Statements

   case =>                       pp_chars: Integer;

   alternative_s =>              pp_chars: Integer,
                                 pp_maxchars: Integer;

   alternative =>                pp_chars: Integer;

-- 5.5  Loop Statements

   named_stm =>                  pp_chars: Integer;

   loop =>                       pp_chars: Integer;

   for =>                        pp_chars: Integer;

   reverse =>                    pp_chars: Integer;

   iteration_id =>               pp_chars: Integer;

   while =>                      pp_chars: Integer;

-- 5.6  Block Statements

   block =>                      pp_chars: Integer;
```

```
-- 5.7  Exit Statements

    exit =>                     pp_chars: Integer;

-- 5.8  Return Statements

    return =>                   pp_chars: Integer;

-- 5.9  Goto Statements

    goto =>                     pp_chars: Integer;

-- 6.  Subprograms
-- ==============
-- 6.1  Subprogram Declarations

    subprogram_decl =>          pp_chars: Integer;

    proc_id =>                  pp_chars: Integer;

    function_id =>              pp_chars: Integer;

    def_op =>                   pp_chars: Integer;

    procedure =>                pp_chars: Integer;

    function =>                 pp_chars: Integer;

    param_s =>                  pp_chars: Integer,
                                pp_maxchars: Integer;

    in =>                       pp_chars: Integer;

    in_out =>                   pp_chars: Integer;

    out =>                      pp_chars: Integer;

    in_id =>                    pp_chars: Integer;

    in_out_id =>                pp_chars: Integer;

    out_id =>                   pp_chars: Integer;

-- 6.3  Subprogram Bodies

    subprogram_body =>          pp_chars: Integer;

-- 6.4  Subprogram Calls

    procedure_call =>           pp_chars: Integer;

    function_call =>            pp_chars: Integer;

    assoc =>                    pp_chars: Integer;

-- 7.  Packages
-- ============
-- 7.1  Package Structure
```

```
    package_decl =>              pp_chars: Integer;

    package_id =>                pp_chars: Integer;

    package_spec =>              pp_chars: Integer;

    decl_s =>                    pp_chars: Integer,
                                 pp_maxchars: Integer;

    package_body =>              pp_chars: Integer;

-- 7.4  Private Type and Deferred Constant Declarations

    private =>                   pp_chars: Integer;
    l_private =>                 pp_chars: Integer;

    private_type_id =>           pp_chars: Integer;

    l_private_type_id =>         pp_chars: Integer;

-- 8.  Visibility Rules
-- ======================
-- 8.4  Use Clauses

    name_s =>                    pp_chars: Integer,
                                 pp_maxchars: Integer;

    use =>                       pp_chars: Integer,
                                 pp_maxchars: Integer;

-- 8.5  Renaming Declarations

    rename =>                    pp_chars: Integer;

-- 9.  Tasks
-- ==========
-- 9.1  Task Specifications and Task Bodies

    task_decl =>                 pp_chars: Integer;

    task_spec =>                 pp_chars: Integer;

    task_body =>                 pp_chars: Integer;

    task_body_id =>              pp_chars: Integer;

-- 9.5  Entries, Entry Calls and Accept Statements

    entry =>                     pp_chars: Integer;

    entry_id =>                  pp_chars: Integer;

    entry_call =>                pp_chars: Integer;

    accept =>                    pp_chars: Integer;

-- 9.6  Delay Statements, Duration and Time
```

```
    delay =>                    pp_chars: Integer;

-- 9.7  Select Statements

-- 9.7.1  Selective Waits

    select =>                   pp_chars: Integer;

    select_clause_s =>          pp_chars: Integer,
                                pp_maxchars: Integer;

    select_clause =>            pp_chars: Integer;

    terminate =>                pp_chars: Integer;

-- 9.7.2  Conditional Entry Calls

    cond_entry =>               pp_chars: Integer;

-- 9.7.3  Timed Entry Calls

    timed_entry =>              pp_chars: Integer;

-- 9.10  Abort Statements

    abort =>                    pp_chars: Integer;

-- 10.  Program Structure and Compilation Issues
-- ==============================================
-- 10.1  Compilation Units - Library Units

    compilation =>              pp_nest: Integer;   -- maximum nesting

    compilation =>              pp_chars: Integer,
                                pp_maxchars: Integer;

  pragma_s =>                   pp_chars: Integer,
                                pp_maxchars: Integer;

    comp_unit =>                pp_chars: Integer;

-- Context Clauses - With Clauses

    context =>                  pp_chars: Integer,
                                pp_maxchars: Integer;

    with =>                     pp_chars: Integer,
                                pp_maxchars: Integer;

-- 10.2  Subunits of Compilation Units

    subunit =>                  pp_chars: Integer;

    stub =>                     pp_chars: Integer;

-- 11.  Exceptions
-- ===============
-- 11.1  Exception Declarations

    exception =>                pp_chars: Integer;

    exception_id =>             pp_chars: Integer;

-- 11.2  Exception Handlers

-- 11.3  Raise Statements

    raise =>                    pp_chars: Integer;

-- 12.  Generic Program Units
-- ========================
-- 12.1  Generic Declarations

    generic =>                  pp_chars: Integer;

    generic_id =>               pp_chars: Integer;

    generic_param_s =>          pp_chars: Integer,
                                pp_maxchars: Integer;

    box =>                      pp_chars: Integer;

    formal_dscrt =>             pp_chars: Integer;
    formal_fixed =>             pp_chars: Integer;
    formal_float =>             pp_chars: Integer;
    formal_integer =>           pp_chars: Integer;

-- 12.3  Generic Instantiation

    generic_assoc_s =>          pp_chars: Integer,
                                pp_maxchars: Integer;

    instantiation =>            pp_chars: Integer;

-- 13. Representation Clauses and
-- ==================================
-- Implementation Dependent Features
-- ==============================
-- 13.1  Representation Clauses

-- 13.2 Length Clause
-- 13.3 Enumeration Representation Clauses

    simple_rep =>               pp_chars: Integer;

-- 13.4  Record Representation Clauses

    alignment =>                pp_chars: Integer;

    record_rep =>               pp_chars: Integer;

    comp_rep_s =>               pp_chars: Integer,
                                pp_maxchars: Integer;

    comp_rep =>                 pp_chars: Integer;
```

```
-- 13.5  Address Clauses

   address =>                   pp_chars: Integer;

-- 13.8  Machine Code Insertions

   code =>                      pp_chars: Integer;

-- 14.0 Input-Output
-- ----------------
-- I/O procedure calls are not specially handled. They are
-- represented by procedure or function calls (see 6.4).

-- Predefined Diana Environment
-- ----------------------------
--
--

   attr_id =>                   pp_chars: Integer;

   pragma_id =>                 pp_chars: Integer,
                                pp_maxchars: Integer;

End
```

APPENDIX D
FORMAT CONTROL

In this appendix we present the ADA package that contains the subprograms to do the formatting of the reconstructed source.

D.1. Package Specification

```
-- Package that provides operations to format reconstructed Ada source

package FORMAT is
    LineLength:                         -- length of the output line
        constant POSITIVE := 120;
    type Column is                      -- position on the line
        range 0..LineLength;
    procedure                           -- store Text into the output
        AddText(Text: in String);
    procedure                           -- store an Ada reserved word
        ResForm(Text: in String);
    procedure                           -- store a comment
        ComForm(Text: in String);
    procedure                           -- store a program identifier
        IdentForm(Text: in String);
    function Remaining return Column;   -- unused characters in buffer
    function Position return Column;    -- used characters in buffer
    procedure NewLine;                  -- output new line with indentation
    procedure Indent;                   -- increment from last indentation
    procedure Undent;                   -- revert to previous indentation
    procedure SetIndent(Pos: in Column);
                                        -- set indentation to Pos
end FORMAT;
```

REFERENCES

[1] A. Evans, K. Butler.
Diana Reference Manual.
Technical Report TL-83-4, Tartan Laboratories Inc., February, 1983.
Revision 3.

[2] J.D. Ichbiah, B. Krieg-Brueckner, B.A. Wichmann, H.F. Ledgard, J.C. Heliard, J.R. Abrial, J.G.P. Barnes, M. Woodger, O. Roubine, P.N. Hilfinger, R. Firth.
Reference Manual for the Ada Programming Language
Draft revised MIL-STD 1815, July 1982 edition, Honeywell, Inc., and Cii-Honeywell Bull, 1982.

[3] B. Krieg-Brueckner, D.C. Luckham, F.W. von Henke, O. Owe.
ANNA: A Language for Annotating Ada Programs.
Technical Report, Computer Systems Laboratory, Stanford University, 1982.

[4] J.R. Nestor, W.A. Wulf, D.A. Lamb.
IDL - Interface Description Language: Formal Description.
Technical Report CMU-CS-81-139, Carnegie-Mellon University, Computer Science Department, June, 1982.
Revision 2.0.

LMEI

-8