AD-A272 792

(2)

DIANA REFERENCE MANUAL

Draft Revision 4
5 May 1986

DTIC
S ELECTE
NOV 16 1993
A

Kathryn L. McKinley
Carl F. Schaefer

Intermetrics, Inc.
IR-MD-078

Prepared For:
Naval Research Laboratory
Washington, D.C. 20375

Contract N00014-84-C-2445

Prepared By:
Intermetrics, Inc.
4733 Bethesda Ave.
Bethesda, MD 20814

93-28047

327 pgs

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

93 11 15 076

Best
Available
Copy

TABLE OF CONTENTS

Acknowledgements for the First Edition

DIANA is based on two earlier proposals for intermediate forms for Ada programs: TCOL and AIDA. It could not have been designed without the efforts of the two groups that designed these previous schemes. Thus we are deeply grateful to:

- AIDA: Manfred Dausmann, Guido Persch, Sophia Drossopoulou, Gerhard Goos, and Georg Winterstein -- all from the University of Karlsruhe.

- TCOL: Benjamin Brosgol (Intermetrics), Joseph Newcomer (Carnegie-Mellon University), David Lamb (CMU), David Levine (Intermetrics), Mary Van Deusen (Prime), and Wm. Wulf (CMU).

The actual design of DIANA was conducted by teams from Karlsruhe, Carnegie-Mellon, Intermetrics and Softech. Those involved were Benjamin Brosgol, Manfred Dausmann, Gerhard Goos, David Lamb, John Nestor, Richard Simpson, Michael Tighe, Larry Weissman, Georg Winterstein, and Wm. Wulf. Assistance in creation of the document was provided by Jeff Baird, Dan Johnston, Paul Knueven, Glenn Marcy, and Aaron Wohl -- all from CMU.

We are grateful for the encouragement and support provided for this effort by Horst Clausen (IABG), Larry Druffel (DARPA), and Marty Wolfe (CENTACS) as well as our various funding agencies.

Finally, the design of DIANA was conducted at Eglin Air Force Base with substantial support from Lt. Col. W. Whitaker. We could not have done it without his aid.

DIANA's original design was funded by Defense Advanced Research Projects Agency (DARPA), the Air Force Avionics Laboratory, the Department of the Army, Communication Research and Development Command, and the Bundesamt fuer Wehrtechnik und Beschaffung.

Gerhard Goos
Wm. A. Wulf
Editors, First Edition

Acknowledgements For The Second Edition

Subsequent to DIANA's original design, the Ada Joint Program Office of the United States Department of Defense has supported at Tartan Laboratories, Incorporated a continuing effort at revision. This revision has been performed by Arthur Evans, Jr., and Kenneth J. Butler, with considerable assistance from John R. Nestor and Wm. A. Wulf, all of Tartan.

We are grateful to the following for their many useful comments and suggestions.

- o Georg Winterstein, Manfred Dausmann, Sophia Droussopoulou, Guido Persch, and Jergen Uhl, all of the Karlsruhe Ada Implementation Group;

- o Julie Sussman and Rich Shapiro of Bolt Beranek and Newman, Inc.; and to

- o Charles Wetherell and Peggy Quinn of Bell Telephone Laboratories.

Additional comments and suggestions have been offered by Grady Booch, Benjamin Brosgol, Gil Hanson, Jeremy Holden, Bernd Krieg-Brueckner, David Lamb, H.-H. Nageli, Teri Payton, and Richard Simpson.

We thank the Ada Joint Program Office (AJPO) for supporting DIANA's revision, and 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.

DIANA is being maintained and revised 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 of DIANA Maintenance for Tartan is Arthur Evans, Jr.

Acknowledgements For This Edition

The United States Department of Defense has continued to support DIANA through a DIANA maintenance program and revision effort at Intermetrics, Inc. This draft revision of the Diana Reference Manual has been prepared by Carl Schaefer and Kathryn McKinley.

The authors are grateful to David Lamb (Queen's University) for his excellent recommendations on restructuring the classes of DIANA, to Rudy Krutar (Naval Research Laboratory) for his valuable assistance as Contracting Officer's Technical Representative, and to Ben Hyde and Gary Bray (Intermetrics) for their valuable suggestions. Kellye Sheehan (MCC) contributed to the early stages of the maintenance effort.

DIANA is being maintained and revised by Intermetrics, Inc. for the Ada Joint Program Office of the Department of Defense under contract number N00014-84-C-2455, administered by the Naval Research Laboratory. The contract expires 30 September 1986. The Project Director of DIANA Maintenance for Intermetrics is Carl Schaefer.

DTIC QUALITY INSPECTED 5

A-1

Preface to the First Edition

This document defines DIANA, an intermediate form of Ada [7] programs that is especially suitable for communication between the Front and Back Ends of Ada compilers. It is based on the Formal Definition of Ada [6] and resulted from the merger of the best aspects of two previous proposals: AIDA [4, 10] and TCOL [2]. Although DIANA is primarily intended as an interface between the parts of a compiler, it is also suitable for other programming support tools and carefully retains the structure of the original source program.

The definition of DIANA given here is expressed in another notation, IDL, that is formally defined in a separate document [9]. The present document is, however, completely self-contained; those aspects of IDL that are needed for the DIANA definition are informally described before they are used. Interested readers should consult the IDL formal description either if they are concerned with either a more precise definition of the notation or if they need to define other data structures in an Ada support environment. In particular, implementors may need to extend DIANA in various ways for use with the tools in a specific environment and the IDL document provides information on how this may be done.

This version of DIANA has been "frozen" to meet the needs of several groups who require a stable definition in a very short timeframe. We invite comments and criticisms for a longer-term review. We expect to re-evaluate DIANA after some practical experience with using it has been accumulated.

Preface To The Second Edition

Since first publication of the DIANA Reference Manual in March, 1981, further developments in connection with Ada and DIANA have required revision of DIANA. These developments include the following:

- o The original DIANA design was based on Ada as defined in the July 1980 Ada Language Reference Manual [7], referred to hereafter as Ada-80; the present revision is based on Ada as defined in the July 1982 Ada LRM [8], referred to hereafter as Ada-82.
- o Experience with use of DIANA has revealed errors and flaws in the original design; these have been corrected.

This publication reflects our best efforts to cope with the conflicting pressures on us both to impact minimally on existing implementations and to create a logically defensible design.

Tartan Laboratories Inc. invites any further comments and criticisms on DIANA in general, and this version of the reference manual in particular. Any correspondence may be sent via ARPANet mail to Diana-Query@USC-ECLB. Paper mail may be sent to

DIANA Manual
Tartan Laboratories Inc.
477 Melwood Avenue
Pittsburgh PA 15213

We believe the changes made to DIANA make no undue constraint on any DIANA users or potential DIANA users, and we wish to hear from those who perceive any of these changes to be a problem.

Preface To This Edition

This is a draft revision of the DIANA Reference Manual.

Experience with DIANA has revealed weaknesses both in the definition of DIANA and in the DIANA Reference Manual. This draft revision incorporates changes in both areas.

Changes to the definition of DIANA include:

- o Overhauling the representation of types and subtypes to accord better with the definition of subtypes in Ada.
- o "Partitioning" the DIANA so that any node or class (except the node **void**) is directly a member of no more than one class.
- o "Hoisting" attributes to the highest appropriate class.
- o Otherwise regularizing the nomenclature of classes, nodes, and attributes.

Changes to the DIANA Reference Manual include:

- o Separation of semantic specification from rationale.
- o Systematic coverage of static semantics of DIANA.
- o Inclusion of hierarchical diagrams providing a pictorial representation of class-membership relations.
- o Inclusion of several substantial examples.
- o Inclusion of a cross-reference index of nodes and attributes.

Chapter 7, External Representation of DIANA, and Chapter 8, The DIANA Package in Ada, are incomplete in this draft.

Intermetrics, Inc. invites any further comments and criticisms on DIANA in general, and this draft version of the reference manual in particular. Any correspondence may be sent via ARPANet mail to DIANA-QUERY@USC-ISIF. Paper mail may be sent to

DIANA Maintenance
Intermetrics, Inc.
4733 Bethesda Ave.
Bethesda, MD 20814

CHAPTER 1

INTRODUCTION

> The purpose of standardization is
> to aid the creative craftsman, not to
> enforce the common mediocrity [11].

1.1 THE DESIGN OF DIANA

In a programming environment such as that envisioned for Ada(1), there will be a number of tools -- formatters (pretty printers), language-oriented editors, cross-reference generators, test-case generators, etc. In general, the input and output of these tools is NOT the source text of the program being developed; instead it is some intermediate form that has been produced by another tool in the environment. This document defines DIANA, **Descriptive Intermediate Attributed Notation for Ada.** DIANA is an intermediate form of Ada programs which has been designed to be especially suitable for communication between two essential tools -- the Front and Back Ends of a compiler -- but also to be suitable for use by other tools in an Ada support environment. DIANA encodes the results of lexical, syntactic and STATIC semantic analysis, but it does NOT include the results of DYNAMIC semantic analysis, of optimization, or of code generation.

DIANA is an abstract data type. The DIANA representation of a particular Ada program is an instance of this abstract type. As with all abstract types, DIANA defines a set of operations that provide the only way in which instances of the type can be examined or modified. The actual data or file structures used to represent the type are hidden by these operations, in the sense that the implementation of a private type in Ada is hidden.

References may be made to a DIANA "tree", "abstract syntax tree", or "attributed parse tree"; similarly, references may be made to "nodes" in these trees. In the context of DIANA as an abstract data type, it is important to appreciate the implications of such terms. This terminology does NOT imply that the data structure used to implement DIANA is necessarily a tree using pointers, etc. Rather, the notion of attributed trees serves as the abstract model for the definition of DIANA.

The following principles governed the original design of DIANA:

- **DIANA should be representation-independent.** An effort was made to avoid implying any particular implementation for the DIANA abstract type. Implementation-specific information (such as a value on the target machine) is represented in DIANA by other abstract types which must be supplied by each implementation. In addition, DIANA may be extended or contracted to cater to implementation-specific purposes.

- **DIANA should be suitable for various kinds of processing.** Although the primary purpose of DIANA is communication between the Front and Back Ends of compilers, other environment tools should be able to use it as well. The needs of such programs were considered carefully.

- **DIANA should be efficiently implementable.** DIANA is intended to be used; hence it was necessary to consider issues such as size and processing speed.

- **The DIANA description and notation should be regular.** Consistency in these areas is essential to both understanding and processing.

---

(1) Ada is a registered trademark of the U.S. Department of Defense.

Although DIANA is representation-independent, there must be at least one form of the DIANA representation that can be communicated between computing systems. Chapter 6 defines an externally visible ASCII form of the DIANA representation of an Ada program. In this form, the DIANA representation can be communicated between arbitrary environment tools and even between arbitrary computing systems. The form may also be useful during the development of the tools themselves.

1.2 THE DEFINITION OF THE DIANA OPERATIONS

Every object of type DIANA is the representation of some specific Ada program (or portion of an Ada program). A minimum set of operations on the DIANA type must provide the ability to:

- o determine the type of a given object (in DIANA terms, the object's node type).
- o obtain the value of a specific attribute of a node.
- o build a node from its constituent parts.
- o determine whether or not a given pair of instances of a DIANA type are in fact the same instance, as opposed to equivalent ones. For the scalar types (Integer and Boolean), no distinction is drawn between equality and equivalence.
- o assign a specific node to a variable, or a specific scalar value to a scalar variable.
- o set the value of an attribute of a given node.

The sequence type **Seq Of** can be considered as a built-in type that has a few special operators. The operators defined for a sequence type allow an implementation to:

- o create a sequence of a given type
- o determine whether or not a sequence is empty
- o select an element of a sequence
- o add an element to a sequence
- o remove an element from a sequence
- o compare two sequences to see if they are the same sequence
- o assign a sequence to a variable of a sequence type

1.3 THE DEFINITION OF A DIANA USER

Inasmuch as DIANA is an abstract data type, there is no need that it be implemented in any particular way. Additionally, because DIANA is extendable, a particular implementation may choose to use a superset of the DIANA defined in this reference manual. In the face of innumerable variations on the same theme, it is appropriate to offer a definition of what it means to "use" DIANA. Since it makes sense to consider DIANA only at the interfaces, two types of DIANA users are considered: those which "produce" DIANA, and those that "consume" it. These aspects are considered in turn:

- **producer**

  In order for a program to be considered a DIANA producer, it must produce as output a structure that includes all of the information contained in DIANA as defined in this document. Every attribute defined herein must be present, and each attribute must have the value defined for correct DIANA and may not have any other value. This requirement means, for example, that additional values, such as the evaluation of non-static expressions, may not be represented using the DIANA-defined attributes. An implementation is not prevented from defining additional attributes, and in fact it is expected that most DIANA producers will also produce additional attributes.

  There is an additional requirement on a DIANA producer: The DIANA structure must have the property that it could have been produced from a legal Ada program. This requirement is likely to impinge most strongly on a tool other than a compiler Front End that produces DIANA. As an example of this requirement, in an arithmetic expression, an offspring of a mutiplication could not be an addition but would instead have to be a parenthesized node whose offspring was the addition, since Ada's parsing rules require the parentheses. The motivation for this requirement is to ease the construction of a DIANA consumer, since the task of designing a consumer is completely open-ended unless it can make some reasonable assumptions about its input.

- **consumer**

  In order for a program to be considered a DIANA consumer, it must depend on no more than DIANA as defined herein for the representation of an Ada program. This definition does not prevent a consumer from requiring other kinds of input (such as information about the library, which is not represented in DIANA); however, the DIANA structure must be the only form of representation for an Ada program. This restriction does not prevent a consumer from being able to take advantage of additional attributes that may be defined in an implementation; however, the consumer must also be able to accept input that does not have these additional attributes. It is also incorrect for a program to expect attributes defined herein to have values that are not here specified. For example, it is wrong for a program to expect the attribute sm_value to contain values of expressions that are not static.

There are twc attributes that are defined herein that are NOT required to be supported by a DIANA user: lx_comments and lx_srcpos. These attributes are too implementation-specific to be required for all DIANA users.

It should be noted that the definition of a producer and that of a consumer are not mutually exclusive; for example, a compiler Front End that produces DIANA may also read DIANA for separate compilation purposes.

Having defined a DIANA producer and a DIANA consumer, it is now possible to specify the requirements for a DIANA user. It is not proper to claim that a given implementation uses DIANA unless EITHER it meets the following two criteria:

- o It must be able to read and/or write (as appropriate) the external form of DIANA defined in Chapter 6 of this document.
- o The DIANA that is read/written must be either the output of a DIANA producer or suitable input for a DIANA consumer, as specified in this section.

OR it meets this criterion:

- o The implementation provides a package equivalent to that described in Chapter 7.

1.4 THE STRUCTURE OF THIS DOCUMENT

As previously stated, DIANA is an abstract data type that can be modeled as an attributed tree. This document defines both the domain and the operations of this abstract type. The domain of the DIANA type is a subset of the (mathematical) domain known as attributed trees. In order to specify this subset precisely a subset of a notation called IDL [9] is used. A knowledge of IDL is necessary to read or understand this document. Chapter 2 consists of the IDL description of the DIANA domain, organized in the same manner as the Ada Reference Manual. The DIANA operations are described in section 1.2.

Though the IDL description of DIANA may suffice to describe the structure of DIANA, it does not convey the full semantics of that structure. For example, in certain cases the set of allowed values of an attribute may be a subset of the values belonging to the type of the attribute (although the IDL language would permit the definition of a subclass in such cases, to do so would undoubtedly disrupt the hierarchy and cause such a proliferation of subclasses that DIANA would be almost impossible to understand). In addition, the IDL does not specify the instances in which two attributes must denote the same node. Restrictions such as those described above are given in the semantic specification of DIANA, the third chapter of this document.

Chapter 4 is a rationale for the design of DIANA. While the semantic specification is organized according to the class structure of DIANA, the rationale is composed of sections dealing with different semantic concepts which are not necessarily applicable to any one DIANA class.

Chapter 5 contains examples of various kinds of DIANA structures. Each example contains a segment of Ada code and an illustration of the resulting DIANA structure.

Chapter 6 describes the external form of DIANA, an ASCII representation suitable for communication between different computing systems.

Chapter 7 consists of a package specification for the DIANA interface, written in Ada.

Appendix A is a cross-reference guide for the nodes, classes and attributes of DIANA.

Appendix B is a list of references.

1.4.1 NOTATION

To assist the reader in understanding this material, certain typographic and notational conventions are followed consistently throughout this document, as illustrated in Figure 1-1.

DECL	IDL class name
constant_id	IDL node name
<u>sm_exp_type</u>	IDL attribute
Type	IDL reserved word
"is"	Ada reserved word

Figure 1-1. Typographic and Notational Conventions Used in this Document.

These conventions include:

- The appearance of class names, node names, and attribute names IN THE TEXT are distinguished by the following typographic conventions: class and node names are bold-faced, and attribute names are underlined. These conventions are not followed in the IDL specification, the diagrams, or the cross-reference guide.

- Ada reserved words appear in quotes.

- IDL reserved words appear in lower-case letters, except for the first letter, which is capitalized.

- Class names appear in all upper-case letters.

- node names appear in all lower-case letters.

- attribute names appear in all lower case letters, with one of the prefixes defined below.

- There are four kinds of attributes defined in DIANA: structural, lexical, semantic, and code. The names of these attributes are lexically distinguished in the definition by the following prefixes:

  + **as_**
    Structural attributes define the abstract syntax tree of an Ada program.

  + **lx_**
    Lexical attributes provide information about the source form of the program, such as the spelling of identifiers, or position in the source file.

  + **sm_**
    Semantic attributes encode the results of semantic analysis -- type and overload resolution, for example.

  + **cd_**
    Code attributes provide information from representation specifications that must be observed by the Back End.

- A class name or node name ending in '_s' is always a sequence of what comes before the '_' (if the prefix is extremely long it may be slightly shortened in the sequence name). Thus the reader can be sure on seeing exp_s that the definition

```
exp_s => as_list: Seq Of EXP;
```

appears somewhere.

- A class name ending in '_ELEM' contains both the node or class denoted by the prefix of the class name and a node representing a pragma. The name of the node representing the pragma consists of the prefix of the class name and the suffix '_pragma'. Hence the reader knows that for the class name STM_ELEM the following definition exists

```
STM_ELEM ::= STM | stm_pragma;
```

Throughout the remainder of this document all references to the Ada Reference Manual (ANSI/MIL-STD-1815A-1983) will have the following form: [ARM, section number].

CHAPTER 2

IDL SPECIFICATION

This chapter contains the IDL description of DIANA. It is organized in a manner that parallels the Ada Reference Manual -- each section contains the corresponding segment of Ada syntax along with the related IDL definitions. In some cases a section does not contain any IDL definitions because that particular construct is represented by a node or class which also represents another construct, and the IDL definitions were included in the section pertaining to the other construct. For example, the section covering operators (section 4.5) does not contain IDL definitions because operators in DIANA are represented as function calls, and the related IDL definitions are included in the section on subprogram calls (section 6.4).

```
Structure Diana
Root compilation Is

-- Private Type Definitions

    Type source_position;
    Type comments;
    Type symbol_rep;
    Type value;
    Type operator;
    Type number_rep;


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

-- Syntax 2.0
--     has no equivalent in concrete syntax

    void => ;


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

-- Syntax 2.3
--     not of interest for Diana


    DEF_NAME ::=            SOURCE_NAME | PREDEF_NAME;
    DEF_NAME =>             lx_symrep : symbol_rep;

    SOURCE_NAME ::=         OBJECT_NAME | TYPE_NAME | UNIT_NAME | LABEL_NAME;

    OBJECT_NAME =>          sm_obj_type : TYPE_SPEC;

    UNIT_NAME =>            sm_first : DEF_NAME;


-- 2.8  Pragmas

-- Syntax 2.8.A
-- pragma ::=
--     pragma identifier [(argument_association {, argument_association})];

    pragma =>               as_used_name_id : used_name_id,
                            as_general_assoc_s : general_assoc_s;
                            -- seq of EXP and/or assoc

    general_assoc_s =>      as_list : Seq Of GENERAL_ASSOC;


-- Syntax 2.8.B
--  argument_association ::=
--       [argument_identifier =>] name
```

```
--      | [argument_identifier =>] expression


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

-- Syntax 3.1
--  declaration ::=
--        object_declaration         | number_declaration
--      | type_declaration           | subtype_declaration
--      | subprogram_declaration     | package_declaration
--      | task_declaration           | generic_declaration
--      | exception_declaration      | generic_instantiation
--      | renaming_declaration       | deferred_constant_declaration


    DECL ::=                ID_S_DECL | ID_DECL;

    ID_DECL ::=             type_decl
                          | subtype_decl
                          | task_decl
                          | UNIT_DECL;

    ID_DECL =>              as_source_name : SOURCE_NAME;

    ID_S_DECL ::=           EXP_DECL
                          | exception_decl
                          | deferred_constant_decl;

    ID_S_DECL =>            as_source_name_s : source_name_s;

    EXP_DECL ::=            OBJECT_DECL
                          | number_decl;

    EXP_DECL =>             as_exp : EXP;

-- 3.2  Objects and Named Numbers

-- Syntax 3.2.A
--  object declaration ::=
--     identifier_list : [constant] subtype_indication [:= expression];
--   | identifier_list : [constant] constrained_array_definition [:= expression


    EXP ::=                 void;

    CONSTRAINED_DEF ::=     subtype_indication;

    OBJECT_DECL ::=         constant_decl | variable_decl;
    OBJECT_DECL =>          as_type_def : TYPE_DEF;

    constant_decl => ;
    variable_decl => ;
```

```
    OBJECT_NAME ::=          INIT_OBJECT_NAME;

    INIT_OBJECT_NAME ::=     VC_NAME;
    INIT_OBJECT_NAME =>      sm_init_exp : EXP;

    VC_NAME ::=              variable_id | constant_id;
    VC_NAME =>               sm_renames_obj : Boolean,
                             sm_address : EXP; -- EXP or void

    variable_id =>           sm_is_shared : Boolean;

    constant_id =>           sm_first : DEF_NAME;


-- Syntax 3.2.B
--  number_declaration ::=
--     identifier_list : constant := universal_static_expression;

    number_decl  => ;

    INIT_OBJECT_NAME ::=     number_id;

    number_id => ;


-- Syntax 3.2.C
--  identifier_list ::= identifier {, identifier}

    source_name_s =>         as_list : Seq Of SOURCE_NAME;


-- 3.3  Types and Subtypes
-- 3.3.1 Type Declarations

-- Syntax 3.3.1.A
--  type_declaration ::= full_type_declaration
--     | incomplete_type_declaration | private_type_declaration
--
--  full_type_declaration ::=
--     type identifier [discriminant_part] is type_definition;
--

    type_decl =>             as_dscrmt_decl_s : dscrmt_decl_s,
                             as_type_def : TYPE_DEF;

    TYPE_NAME ::=            type_id;
    TYPE_NAME =>             sm_type_spec : TYPE_SPEC;

    type_id =>               sm_first : DEF_NAME;


-- Syntax 3.3.1.B
--  type_definition ::=
```

```
--      enumeration_type_definition  | integer_type_definition
--    | real_type_definition         | array_type_definition
--    | record_type_definition       | access_type_definition
--    | derived_type_definition


    TYPE_DEF ::=              enumeration_def
                            | CONSTRAINED_DEF
                            | ARR_ACC_DER_DEF
                            | record_def;

    CONSTRAINED_DEF ::=       integer_def
                            | float_def
                            | fixed_def;

    ARR_ACC_DER_DEF ::=       constrained_array_def
                            | unconstrained_array_def
                            | access_def
                            | derived_def;
    ARR_ACC_DER_DEF =>        as_subtype_indication : subtype_indication;

    TYPE_SPEC ::=             DERIVABLE_SPEC;

    DERIVABLE_SPEC ::=        FULL_TYPE_SPEC | PRIVATE_SPEC;
    DERIVABLE_SPEC =>         sm_derived : TYPE_SPEC,
                              sm_is_anonymous : Boolean;

    FULL_TYPE_SPEC ::=        task_spec | NON_TASK;

    NON_TASK ::=              SCALAR | UNCONSTRAINED | CONSTRAINED;
    NON_TASK =>               sm_base_type : TYPE_SPEC;

    SCALAR ::=                enumeration | integer | REAL;
    SCALAR =>                 sm_range : RANGE;
    SCALAR =>                 cd_impl_size : Integer;

    REAL ::=                  float | fixed;
    REAL =>                   sm_accuracy : value;

    UNCONSTRAINED ::=         UNCONSTRAINED_COMPOSITE | access;
    UNCONSTRAINED =>          sm_size : EXP; -- EXP or void

    UNCONSTRAINED_COMPOSITE ::= array | record;
    UNCONSTRAINED_COMPOSITE =>  sm_is_limited : Boolean,
                                sm_is_packed : Boolean;

    CONSTRAINED ::=           constrained_array
                            | constrained_record
                            | constrained_access;
    CONSTRAINED =>            sm_depends_on_dscrmt : Boolean;

-- 3.3.2 Subtype Declarations
```

```
-- Syntax 3.3.2.A
--  subtype_declaration ::= subtype identifier is subtype_indication;

    subtype_decl =>        as_subtype_indication : subtype_indication;

    TYPE_NAME ::=          subtype_id;

    subtype_id => ;


-- Syntax 3.3.2.B

--  subtype_indication ::= type_mark [constraint]
--
--  type_mark ::= type_name | subtype_name


    CONSTRAINT ::=         void;

    CONSTRAINED_DEF =>     as_constraint : CONSTRAINT;

    subtype_indication =>  as_name : NAME;


-- Syntax 3.3.2.C

--  constraint ::=
--       range_constraint | floating_point_constraint | fixed_point_constraint
--     | index_constraint | discriminant_constraint

    CONSTRAINT ::=          DISCRETE_RANGE
                          | REAL_CONSTRAINT
                          | index_constraint
                          | dscrmt_constraint;


-- 3.4  Derived Type Definitions

-- Syntax 3.4
--  derived_type_definition ::= new subtype_indication

    derived_def => ;


-- 3.5  Scalar Types

-- Syntax 3.5
--  range_constraint ::= range range

--  range ::= range_attribute
--     | simple_expression .. simple_expression


    DISCRETE_RANGE ::=     RANGE
```

```
                            | discrete_subtype;

    RANGE ::=               range | range_attribute | void;
    RANGE =>                sm_type_spec : TYPE_SPEC;

    range =>                as_exp1 : EXP,
                            as_exp2 : EXP;

    range_attribute =>      as_name : NAME,
                            as_used_name_id : used_name_id,
                            as_exp : EXP; -- EXP or void

-- 3.5.1  Enumeration Types

-- Syntax 3.5.1.A
--  enumeration_type_definition ::=
--     (enumeration_literal_specification {, enumeration_literal_specification}

    enumeration_def =>      as_enum_literal_s : enum_literal_s;

    enum_literal_s =>       as_list : Seq Of ENUM_LITERAL;

    enumeration =>          sm_literal_s : enum_literal_s;

-- Syntax 3.5.1.B
--  enumeration_literal_specification ::= enumeration_literal

--  enumeration_literal ::= identifier | character_literal

    OBJECT_NAME ::=         ENUM_LITERAL;

    ENUM_LITERAL ::=        enumeration_id | character_id;

    ENUM_LITERAL =>         sm_pos : Integer,
                            sm_rep : Integer;

    enumeration_id => ;
    character_id => ;

-- 3.5.4  Integer Types

-- Syntax 3.5.4
--  integer_type_definition ::= range_constraint

    integer_def => ;

    integer => ;

-- 3.5.6  Real Types
```

```
-- Syntax 3.5.6
--  real_type_definition ::=
--     floating_point_constraint | fixed_point_constraint


    REAL_CONSTRAINT ::=      float_constraint
                           | fixed_constraint;

    REAL_CONSTRAINT =>       sm_type_spec : TYPE_SPEC;


-- 3.5.7  Floating Point Types

-- Syntax 3.5.7
--  floating_point_constraint ::=
--     floating_accuracy_definition [range_constraint]

--  floating_accuracy_definition ::= digits static_simple_expression


    float_def => ;

    REAL_CONSTRAINT =>       as_exp : EXP,
                             as_range : RANGE;

    float_constraint => ;

    float => ;


-- 3.5.9  Fixed Point Types

-- Syntax 3.5.9
--  fixed_point_constraint  ::=
--     fixed_accuracy_definition [range_constraint]

--  fixed_accuracy_definition ::= delta static_simple_expression

    fixed_def => ;

    fixed_constraint => ;

    fixed =>                 cd_impl_small : value;


-- 3.6  Array Types

-- Syntax 3.6.A
--  array_type_definition ::=
--     unconstrained_array_definition | constrained_array_definition

--  unconstrained_array_definition ::=
--     array(index_subtype_definition {, index_subtype_definition}) of
--                 component_subtype_indication
```

```
--  constrained_array_definition ::=
--     array index_constraint of component_subtype_indication

    constrained_array_def => as_constraint : CONSTRAINT;

    index_constraint =>      as_discrete_range_s : discrete_range_s;

    discrete_range_s =>      as_list : Seq Of DISCRETE_RANGE;

    unconstrained_array_def => as_index_s : index_s;

    scalar_s =>              as_list : Seq Of SCALAR;

    array =>                 sm_index_s : index_s,
                             sm_comp_type : TYPE_SPEC;

    constrained_array =>     sm_index_subtype_s : scalar_s;


-- Syntax 3.6.B
--  index_subtype_definition ::= type_mark range <>

    index =>                 as_name   : NAME,
                             sm_type_spec : TYPE_SPEC;

    index_s =>               as_list : Seq Of index;


-- Syntax 3.6.C
--  index_constraint ::= (discrete_range {, discrete_range})

--  discrete_range ::= discrete_subtype_indication | range

    discrete_subtype =>      as_subtype_indication : subtype_indication;


-- 3.7  Record Types

-- Syntax 3.7.A
--  record_type_definition ::=
--     record
--          component_list
--     end record

    REP ::=                void;

    record_def =>          as_comp_list : comp_list;

    record =>              sm_discriminant_s : dscrmt_decl_s,
                           sm_comp_list : comp_list,
```

```
                        sm_representation : REP; -- REP or void

    constrained_record =>  sm_normalized_dscrmt_s : exp_s;


-- Syntax 3.7.B
--  component_list ::=
--       component_declaration {component_declaration}
--     | {component_declaration} variant_part
--     | null;

--  component_declaration ::=
--     identifier_list : component_subtype_definition [:= expression];

--  component_subtype_definition ::= subtype_indication


    DECL ::=               null_comp_decl;


    INIT_OBJECT_NAME ::=   COMP_NAME;
    COMP_NAME ::=          component_id | discriminant_id;

    COMP_NAME =>           sm_comp_rep : COMP_REP_ELEM;

    component_id => ;


-- 3.7.1  Discriminants

-- Syntax 3.7.1
--  discriminant_part ::=
--     (discriminant_specification {; discriminant_specification})

--  discriminant_specification ::=
--     identifier_list : type_mark [:= expression]


    ITEM ::=               DSCRMT_PARAM_DECL;

    DSCRMT_PARAM_DECL ::=  dscrmt_decl;

    DSCRMT_PARAM_DECL =>   as_source_name_s : source_name_s,
                           as_name : NAME,
                           as_exp : EXP;

    dscrmt_decl_s =>       as_list : Seq Of dscrmt_decl;

    dscrmt_decl => ;

    discriminant_id =>     sm_first : DEF_NAME;


-- 3.7.2  Discriminant Constraints
```

```
-- Syntax 3.7.2
--  discriminant_constraint ::=
--     (discriminant_association {, discriminant_association})
--  discriminant_association ::=
--     [discriminant_simple_name {|discriminant_simple_name} =>] expression

    dscrmt_constraint =>   as_general_assoc_s : general_assoc_s;

-- 3.7.3  Variant Parts

-- Syntax 3.7.3.A
--  variant_part ::=
--     case discriminant_simple_name is
--         variant
--        {variant}
--     end case;

--  variant  ::=
--     when choice {| choice} =>
--         component_list

    VARIANT_PART ::=       variant_part | void;

    variant_part =>        as_name : NAME,
                           as_variant_s : variant_s;

    variant_s =>           as_list : Seq Of VARIANT_ELEM;

    VARIANT_ELEM ::=       variant | variant_pragma;

    variant =>             as_choice_s : choice_s,
                           as_comp_list : comp_list;

    choice_s =>            as_list : Seq Of CHOICE;

    comp_list =>           as_decl_s : decl_s,
                           as_variant_part : VARIANT_PART,
                           as_pragma_s : pragma_s;

    variant_pragma =>      as_pragma : pragma;

-- Syntax 3.7.3.B
--  choice ::= simple_expression
--     | discrete_range | others | component_simple_name

    CHOICE ::=             choice_exp | choice_range | choice_others;

    choice_exp =>          as_exp : EXP;
```

```
    choice_range =>         as_discrete_range : DISCRETE_RANGE;

    choice_others => ;


-- 3.8  Access Types

-- Syntax 3.8
--  access_type_definition ::= access subtype_indication

    access_def => ;

    access =>               sm_storage_size : EXP, -- EXP or void
                            sm_is_controlled : Boolean,
                            sm_desig_type : TYPE_SPEC,
                            sm_master : ALL_DECL;

    constrained_access =>   sm_desig_type : TYPE_SPEC;


-- 3.8.1 Incomplete Type Declarations

-- Syntax 3.8.1
--  incomplete_type_declaration ::= type identifier [discriminant_part];

    TYPE_DEF ::=            void;

    TYPE_SPEC ::=           incomplete;

    incomplete =>           sm_discriminant_s : dscrmt_decl_s;

    TYPE_SPEC ::=           void;


-- 3.9  Declarative Parts

-- Syntax 3.9.A
--  declarative_part ::=
--     {basic_declarative_item} {later_declarative_item}

--  basic_declarative_item ::= basic_declaration
--     | representation_clause | use_clause

    DECL ::=                REP;

    DECL ::=                USE_PRAGMA;

    USE_PRAGMA ::=          use | pragma;


-- Syntax 3.9.B
--  later_declarative_item ::= body
--     | subprogram_declaration | package_declaration
```

```
--     | task_declaration        | generic_declaration
--     | use_clause              | generic_instantiation

--  body  ::=  proper_body | stub

--  proper_body ::= subprogram_body | package_body | task_body


    ITEM ::=                DECL | SUBUNIT_BODY;

    item_s =>               as_list : Seq Of ITEM;

    UNIT_DECL ::=           generic_decl
                          | NON_GENERIC_DECL;

    UNIT_DECL =>            as_header : HEADER;

    NON_GENERIC_DECL ::=    subprog_entry_decl
                          | package_decl;

    NON_GENERIC_DECL =>     as_unit_kind : UNIT_KIND;


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

-- Syntax 4.1.A
--  name ::= simple_name
--     | character_literal     | operator_symbol
--     | indexed_component     | slice
--     | selected_component    | attribute

--  simple_name ::= identifier

    NAME ::=                DESIGNATOR
                          | NAME_EXP;

    NAME_EXP ::=            NAME_VAL
                          | indexed
                          | slice
                          | all;
    NAME_EXP =>             as_name : NAME;
    NAME_EXP =>             sm_exp_type : TYPE_SPEC;

    NAME_VAL ::=            attribute
                          | selected;
    NAME_VAL =>             sm_value : value;

    DESIGNATOR ::=          USED_OBJECT | USED_NAME;
    DESIGNATOR =>           sm_defn : DEF_NAME,
                            lx_symrep : symbol_rep;

    USED_NAME ::=           used_op | used_name_id;
```

```
    used_op => ;
    used_name_id => ;

    USED_OBJECT ::=        used_char | used_object_id;
    USED_OBJECT =>         sm_exp_type : TYPE_SPEC,
                           sm_value : value;

    used_char => ;
    used_object_id =>;


-- Syntax 4.1.B
--  prefix ::= name | function_call

    NAME_VAL ::=           function_call;

-- 4.1.1  Indexed Components

-- Syntax 4.1.1
--  indexed_component ::= prefix(expression {, expression})

    exp_s =>               as_list : Seq Of EXP;

    indexed =>             as_exp_s : exp_s;


-- 4.1.2  Slices

-- Syntax 4.1.2
--  slice ::= prefix(discrete_range)

    slice =>               as_discrete_range : DISCRETE_RANGE;


-- 4.1.3  Selected Components

-- Syntax 4.1.3
--  selected_component ::=  prefix.selector

--  selector ::= simple_name
--     | character_literal | operator_symbol | all

    selected =>            as_designator : DESIGNATOR;

    all => ;


-- 4.1.4  Attributes

-- Syntax 4.1.4
--  attribute ::= prefix'attribute_designator

--  attribute_designator ::= simple_name [(universal_static_expression)]
```

```
    attribute =>            as_used_name_id : used_name_id,
                            as_exp : EXP;


-- 4.2  Literals


-- 4.3  Aggregates

-- Syntax 4.3.A
--  aggregate ::=
--     (component_association {, component_association})

    aggregate =>            as_general_assoc_s : general_assoc_s;
    aggregate =>            sm_normalized_comp_s : general_assoc_s;


-- Syntax 4.3.B
--  component_association ::=
--     [choice {| choice} => ] expression


    GENERAL_ASSOC ::=       NAMED_ASSOC | EXP;

    NAMED_ASSOC ::=         named;
    NAMED_ASSOC =>          as_exp : EXP;

    named =>                as_choice_s : choice_s;


-- 4.4  Expressions

-- Syntax 4.4.A
--  expression ::=
--       relation {and relation}  | relation {and then relation}
--     | relation {or relation}   | relation {or else relation}
--     | relation {xor relation}


    EXP_VAL ::=             short_circuit;

    short_circuit =>        as_exp1 : EXP,
                            as_short_circuit_op : SHORT_CIRCUIT_OP,
                            as_exp2 : EXP;

    SHORT_CIRCUIT_OP ::=    and_then | or_else;

    and_then => ;
    or_else  => ;


-- Syntax 4.4.B
--  relation ::=
--       simple_expression [relational_operator simple_expression]
```

```
--     | simple_expression [not] in range
--     | simple_expression [not] in type_mark


    EXP_VAL_EXP ::=         MEMBERSHIP;

    MEMBERSHIP ::=          range_membership | type_membership;
    MEMBERSHIP =>           as_membership_op : MEMBERSHIP_OP;

    range_membership =>     as_range : RANGE;

    type_membership =>      as_name : NAME;

    MEMBERSHIP_OP ::=       in_op | not_in;

    in_op => ;
    not_in => ;


-- Syntax 4.4.C
--   simple_expression ::=
--      [unary_operator] term {binary_adding_operator term}

--   term ::= factor {multiplying_operator factor}

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


-- Syntax 4.4.D
--   primary ::=
--       numeric_literal | null | aggregate | string_literal | name | allocator
--      | function_call | type_conversion | qualified_expression | (expression)


    EXP ::=                   NAME
                            | EXP_EXP;

    EXP_EXP ::=               EXP_VAL
                            | AGG_EXP
                            | qualified_allocator
                            | subtype_allocator;
    EXP_EXP =>                sm_exp_type : TYPE_SPEC;

    EXP_VAL ::=               numeric_literal
                            | null_access
                            | EXP_VAL_EXP;
    EXP_VAL =>                sm_value : value;

    EXP_VAL_EXP ::=           QUAL_CONV
                            | parenthesized;
    EXP_VAL_EXP =>            as_exp : EXP;

    AGG_EXP ::=               aggregate
                            | string_literal;
```

```
    AGG_EXP =>               sm_discrete_range : DISCRETE_RANGE;

    parenthesized => ;

    numeric_literal =>       lx_numrep : number_rep;

    string_literal =>        lx_symrep : symbol_rep;

    null_access => ;

-- 4.5  Operators and Expression Evaluation

-- Syntax 4.5
--  logical_operator ::= and | or | xor

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

--  adding_operator ::= + | - | &

--  unary_operator ::= + | -

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

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


-- 4.6  Type Conversions

-- Syntax 4.6
--  type_conversion ::= type_mark(expression)

    QUAL_CONV ::=            conversion
                           | qualified;
    QUAL_CONV =>             as_name : NAME;

    conversion =>;

-- 4.7  Qualified Expressions

-- Syntax 4.7
--  qualified_expression ::=
--     type_mark'(expression) | type_mark'aggregate

    qualified => ;


-- 4.8  Allocators

-- Syntax 4.8
--  allocator ::=
--     new subtype_indication | new qualified_expression


    qualified_allocator => as_qualified : qualified;
```

```
    subtype_allocator =>    as_subtype_indication : subtype_indication,
                            sm_desig_type : TYPE_SPEC;


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

-- Syntax 5.1.A
--  sequence_of_statements ::= statement {statement}


    STM_ELEM ::=            STM | stm_pragma;

    stm_s =>                as_list : Seq Of STM_ELEM;

    stm_pragma =>           as_pragma : pragma;


-- Syntax 5.1.B
--  statement ::=
--     {label} simple_statement | {label} compound_statement


    STM ::=                 labeled;

    labeled =>              as_source_name_s : source_name_s,
                            as_pragma_s : pragma_s,
                            as_stm : STM;


-- Syntax 5.1.C
--  simple_statement ::= null_statement
--     | assignment_statement | procedure_call_statement
--     | exit_statement       | return_statement
--     | goto_statement       | entry_call_statement
--     | delay_statement      | abort_statement
--     | raise_statement      | code_statement


    STM ::=                 null_stm
                          | abort;

    STM ::=                 STM_WITH_EXP;

    STM_WITH_EXP ::=        return
                          | delay;

    STM_WITH_EXP ::=        STM_WITH_EXP_NAME;

    STM_WITH_EXP =>         as_exp : EXP;

    STM_WITH_EXP_NAME ::=   assign
```

```
                          | exit
                          | code;

STM_WITH_EXP_NAME =>        as_name : NAME;

STM ::=                     STM_WITH_NAME;

STM_WITH_NAME ::=           goto
                          | raise;

STM_WITH_NAME ::=           CALL_STM;
CALL_STM ::=                entry_call
                          | procedure_call;

STM_WITH_NAME =>            as_name : NAME;


-- Syntax 5.1.D
--  compound_statement ::=
--       if_statement           | case_statement
--     | loop_statement         | block_statement
--     | accept_statement       | select_statement

STM ::=                     accept
                          | BLOCK_LOOP
                          | ENTRY_STM;

STM_WITH_EXP ::=            case;

STM ::=                     CLAUSES_STM;

CLAUSES_STM ::=             if
                          | selective_wait;

CLAUSES_STM =>              as_test_clause_elem_s : test_clause_elem_s,
                            as_stm_s : stm_s;


-- Syntax 5.1.E
--  label ::= <<label_simple_name>>

LABEL_NAME ::=              label_id;
LABEL_NAME =>               sm_stm : STM;

label_id => ;


-- Syntax 5.1.F
--  null_statement ::= null ;

null_stm => ;


-- 5.2  Assignment Statement
```

```
-- Syntax 5.2
--  assignment_statement ::=
--     variable_name := expression;

    assign => ;


-- 5.3  If Statements

-- Syntax 5.3.A
--  if_statement ::=
--      if condition then
--          sequence_of_statements
--     {elsif condition then
--          sequence_of_statements}
--     [else
--          sequence_of_statements]
--      end if;

    if => ;

    TEST_CLAUSE ::=        cond_clause;
    TEST_CLAUSE =>         as_exp : EXP,
                           as_stm_s : stm_s;

    cond_clause => ;


-- Syntax 5.3.B
--  condition ::= boolean_expression


-- 5.4  Case Statements

-- Syntax 5.4
--  case_statement ::=
--     case expression is
--          case_statement_alternative
--         {case_statement_alternative}
--     end case;

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

    ALTERNATIVE_ELEM ::=   alternative | alternative_pragma;

    case =>                as_alternative_s : alternative_s;

    alternative_s =>       as_list : Seq Of ALTERNATIVE_ELEM;

    alternative =>         as_choice_s : choice_s,
                           as_stm_s : stm_s;
```

```
    alternative_pragma =>  as_pragma : pragma;


-- 5.5  Loop Statements

-- Syntax 5.5.A
--  loop_statement ::=
--     [loop_simple_name:]
--          [iteration_scheme] loop
--               sequence_of_statements
--          end loop [loop_simple_name];

    BLOCK_LOOP ::=          loop;

    BLOCK_LOOP =>           as_source_name : SOURCE_NAME;

    SOURCE_NAME ::=         void;

    LABEL_NAME ::=          block_loop_id;

    block_loop_id => ;

    ITERATION ::=           void;

    loop =>                 as_iteration : ITERATION,
                            as_stm_s : stm_s;


-- Syntax 5.5.B
--  iteration_scheme ::=  while condition
--     | for loop_parameter_specification
--
--  loop_parameter_specification ::=
--     identifier in [reverse] discrete_range


    ITERATION ::=           FOR_REV;
    FOR_REV ::=             for | reverse;
    FOR_REV =>              as_source_name : SOURCE_NAME,
                            as_discrete_range : DISCRETE_RANGE;

    for => ;
    reverse => ;

    OBJECT_NAME ::=         iteration_id;
    iteration_id => ;

    ITERATION ::=           while;

    while =>                as_exp  : EXP;

-- 5.6  Block Statements
```

```
-- Syntax 5.6
--  block_statement ::=
--     [block_simple_name:]
--          [declare
--               declarative_part]
--          begin
--               sequence_of_statements
--          [exception
--                exception_handler
--               {exception_handler}]
--          end [block_simple_name];

    BLOCK_LOOP ::=          block;

    block =>                as_block_body : block_body;

    block_body =>           as_item_s : item_s,
                            as_stm_s : stm_s,
                            as_alternative_s : alternative_s;


-- 5.7  Exit Statements

-- Syntax 5.7
--  exit_statement ::=
--     exit [loop_name] [when condition];


    NAME ::=                void;

    exit =>                 sm_stm : STM;


-- 5.8  Return Statements

-- Syntax 5.8
--  return_statement ::= return [expression];

    return => ;


-- 5.9  Goto Statements

-- Syntax 5.9
--  goto_statement ::= goto label_name;

    goto => ;


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

```
-- Syntax 6.1.A
--  subprogram_declaration ::= subprogram_specification;


    subprog_entry_decl =>  ;

    UNIT_NAME ::=          NON_TASK_NAME;

    NON_TASK_NAME ::=      SUBPROG_PACK_NAME;
    NON_TASK_NAME =>       sm_spec : HEADER;

    SUBPROG_PACK_NAME ::=  SUBPROG_NAME;
    SUBPROG_PACK_NAME =>   sm_unit_desc : UNIT_DESC,
                           sm_address : EXP;

    SUBPROG_NAME ::=       procedure_id | function_id | operator_id;
    SUBPROG_NAME =>        sm_is_inline : Boolean,
                           sm_interface : PREDEF_NAME;

    UNIT_DESC ::=          UNIT_KIND | BODY
                         | implicit_not_eq | derived_subprog;

    UNIT_KIND ::=          void;

    derived_subprog =>     sm_derivable : SOURCE_NAME;

    implicit_not_eq =>     sm_equal : SOURCE_NAME;

    procedure_id => ;
    function_id => ;
    operator_id => ;


-- Syntax 6.1.B
--  subprogram_specification ::=
--       procedure identifier [formal_part]
--     | function designator [formal_part] return type_mark

--  designator ::= identifier | operator_symbol

--  operator_symbol ::= string_literal


    HEADER ::=             SUBP_ENTRY_HEADER;

    SUBP_ENTRY_HEADER ::=  procedure_spec | function_spec;

    SUBP_ENTRY_HEADER =>   as_param_s : param_s;

    procedure_spec => ;

    function_spec =>       as_name : NAME;
```

```
-- Syntax 6.1.C
--  formal_part ::=
--     (parameter_specification {; parameter_specification})

--  parameter_specification ::=
--     identifier_list : mode type_mark [:= expression]

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


    param_s =>              as_list : Seq Of PARAM;

    DSCRMT_PARAM_DECL ::=   PARAM;

    PARAM ::=               in | out | in_out;

    in =>                   lx_default : Boolean;
    in_out => ;
    out =>    ;

    INIT_OBJECT_NAME ::=    PARAM_NAME;

    PARAM_NAME ::=          in_id | in_out_id | out_id;
    PARAM_NAME =>           sm_first : DEF_NAME;

    in_id => ;
    in_out_id => ;
    out_id => ;


-- 6.3  Subprogram Bodies

-- Syntax 6.3
--  subprogram_body ::=
--      subprogram_specification is
--          [declarative_part]
--      begin
--          sequence_of_statements
--     [exception
--           exception_handler
--          {exception_handler}]
--      end [designator];


    BODY ::=                block_body | stub | void;

    subprogram_body =>      as_header : HEADER;


-- 6.4  Subprogram Calls

-- Syntax 6.4
--  procedure_call_statement ::=
```

```
--      procedure_name [actual_parameter_part];

--   function_call ::=
--      function_name [actual_parameter_part]

--   actual_parameter_part ::=
--      (parameter_association {, parameter_association})

--   parameter_association ::=
--      [formal_parameter =>] actual_parameter

--   formal_parameter ::= parameter_simple_name

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


    CALL_STM =>              as_general_assoc_s : general_assoc_s;
    CALL_STM =>              sm_normalized_param_s :exp_s;

    procedure_call => ;

    function_call =>         as_general_assoc_s : general_assoc_s;
    function_call =>         sm_normalized_param_s : exp_s;
    function_call =>         lx_prefix : Boolean;

    NAMED_ASSOC ::=          assoc;

    assoc =>                 as_used_name : USED_NAME;


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

-- Syntax 7.1.A
--  package_declaration ::= package_specification;


    package_decl => ;

    SUBPROG_PACK_NAME ::=  package_id;

    package_id => ;


-- Syntax 7.1.B
--  package_specification ::=
--     package identifier is
--          {basic_declarative_item}
--     [private
--          {basic_declarative_item}]
--     end [package_simple_name]
```

```
    package_spec =>          as_decl_s1 : decl_s,
                             as_decl_s2 : decl_s;

    decl_s =>                as_list : Seq Of DECL;


-- Syntax 7.1.C
--  package_body ::=
--      package body package_simple_name is
--          [declarative_part]
--     [begin
--            sequence_of_statements
--     [exception
--           exception_handler
--          {exception_handler}]]
--      end [package_simple_name];


    package_body =>          ;


-- 7.4  Private Type and Deferred Constant Declarations

-- Syntax 7.4.A
--  private_type_declaration ::=
--     type identifier [discriminant_part] is [limited] private;


    TYPE_DEF ::=             private_def | l_private_def;

    private_def => ;
    l_private_def => ;

    TYPE_NAME ::=            private_type_id | l_private_type_id;

    private_type_id => ;
    l_private_type_id => ;

    PRIVATE_SPEC ::=         private | l_private;
    PRIVATE_SPEC =>          sm_discriminant_s : dscrmt_decl_s,
                             sm_type_spec : TYPE_SPEC;

    private => ;

    l_private => ;


-- Syntax 7.4.B
--  deferred_constant_declaration ::=
--     identifier_list : constant type_mark;

    deferred_constant_decl => as_name : NAME;
```

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

-- Syntax 8.4
--  use_clause ::= use package_name {, package_name};

    use =>                  as_name_s : name_s;


-- 8.5  Renaming Declarations

-- Syntax 8.5
--  renaming_declaration ::=
--       identifier : type_mark    renames object_name;
--     | identifier : exception    renames exception_name;
--     | package identifier        renames package_name;
--     | subprogram_specification  renames subprogram_or_entry_name;


    ID_DECL ::=             SIMPLE_RENAME_DECL;

    SIMPLE_RENAME_DECL ::= renames_obj_decl
                         | renames_exc_decl;

    SIMPLE_RENAME_DECL =>  as_name : NAME;

    renames_obj_decl =>    as_type_mark_name : NAME;

    renames_exc_decl => ;

    UNIT_KIND ::=          RENAME_INSTANT;

    RENAME_INSTANT ::=     renames_unit;
    RENAME_INSTANT =>      as_name : NAME;

    renames_unit => ;


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

-- Syntax 9.1.A
--  task_declaration ::= task_specification;
--
--  task_specification ::=
--     task [type] identifier [is
--          {entry_declaration}
--          {representation_clause}
--     end [task_simple_name]]
```

```
    task_decl =>            as_decl_s : decl_s;

    task_spec =>            sm_decl_s : decl_s,
                            sm_body : BODY,
                            sm_address : EXP,
                            sm_size : EXP,
                            sm_storage_size : EXP;


-- Syntax 9.1.B
--  task_body ::=
--      task body task_simple_name is
--          [declarative_part]
--      begin
--          sequence_of_statements
--     [exception
--           exception_handler
--          {exception_handler}]
--      end [task_simple_name];


    task_body =>            ;

    UNIT_NAME ::=           task_body_id;

    task_body_id =>         sm_type_spec : TYPE_SPEC,
                            sm_body : BODY;


-- 9.4  Task Dependence - Termination of Tasks

    ALL_DECL ::=            block_master;

    block_master =>         sm_stm : STM;


-- 9.5  Entries, Entry Calls and Accept Statements

-- Syntax 9.5.A
--  entry_declaration ::=
--     entry identifier [(discrete_range)] [formal_part];
--

    SUBP_ENTRY_HEADER ::=  entry;

    entry =>                as_discrete_range : DISCRETE_RANGE;

    SOURCE_NAME ::=         entry_id;

    entry_id =>             sm_spec : HEADER,
                            sm_address : EXP;


-- Syntax 9.5.B
```

```
--  entry_call_statement ::= entry_name [actual_parameter_part];

    entry_call => ;


-- Syntax 9.5.C
--  accept_statement ::=
--     accept entry_simple_name [(entry_index)] [formal_part] [do
--          sequence_of_statements
--     end [entry_simple_name]];
--
--  entry_index ::= expression

    accept =>             as_name : NAME,
                          as_param_s : param_s,
                          as_stm_s : stm_s;


-- 9.6  Delay Statements, Duration and Time

-- Syntax 9.6
--  delay_statement ::= delay simple_expression;

    delay => ;


-- 9.7  Select Statements

-- Syntax 9.7
--  select_statement ::= selective_wait
--     | conditional_entry_call | timed_entry_call


-- 9.7.1  Selective Waits

-- Syntax 9.7.1.A
--  selective_wait ::=
--      select
--          select_alternative
--     {or
--          select_alternative}
--     [else
--          sequence_of_statements]
--      end select;

    selective_wait => ;


-- Syntax 9.7.1.B
--  selective_alternative ::=
--     [when condition =>]
--          selective_wait_alternative
```

```
--  selective_wait_alternative ::= accept_alternative
--     | delay_alternative | terminate_alternative

--  accept_alternative ::= accept_statement [sequence_of_statements]

--  delay_alternative ::= delay_statement  [sequence_of_statements]

--  terminate_alternative ::= terminate;


    TEST_CLAUSE_ELEM ::=   TEST_CLAUSE |  select_alt_pragma;

    TEST_CLAUSE ::=        select_alternative;

    test_clause_elem_s =>  as_list : Seq Of TEST_CLAUSE_ELEM;

    select_alternative => ;

    select_alt_pragma =>   as_pragma : pragma;

    STM ::=                terminate;

    terminate => ;


-- 9.7.2  Conditional Entry Calls

-- Syntax 9.7.2
--  conditional_entry_call ::=
--     select
--           entry_call_statement
--          [sequence_of_statements]
--     else
--          sequence_of_statements
--     end select;


    ENTRY_STM ::=          cond_entry | timed_entry;
    ENTRY_STM =>           as_stm_s1 : stm_s,
                           as_stm_s2 : stm_s;

    cond_entry => ;


-- 9.7.3  Timed Entry Calls

-- Syntax 9.7.3
--  timed_entry_call ::=
--     select
--           entry_call_statement
--          [sequence_of_s*-tements]
--     or
--           delay_alternative
```

```
--     end select;

   timed_entry => ;

-- 9.10  Abort Statements

-- Syntax 9.10
--  abort_statement ::= abort task_name {, task_name};

   name_s =>              as_list : Seq Of NAME;

   abort =>               as_name_s : name_s;

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

-- Syntax 10.1.A
--  compilation ::= {compilation_unit}

   compilation =>         as_compltn_unit_s : compltn_unit_s;

   compltn_unit_s =>      as_list : Seq Of compilation_unit;

-- Syntax 10.1.B
--  compilation_unit ::=
--     context_clause library_unit | context_clause secondary_unit

--  library_unit ::=
--       subprogram_declaration| package_declaration
--     | generic_declaration   | generic_instantiation
--     | subprogram_body

--  secondary_unit ::= library_unit_body | subunit

--  library_unit_body ::= subprogram_body | package_body

   ALL_DECL ::=           void;

   pragma_s =>            as_list : Seq Of pragma;

   compilation_unit =>    as_context_elem_s : context_elem_s,
                          as_all_decl : ALL_DECL,
                          as_pragma_s : pragma_s;

   CONTEXT_ELEM ::=       context_pragma;
```

```
    context_pragma =>       as_pragma : pragma;


-- Context Clauses - With Clauses

-- Syntax 10.1.1.A
--  context_clause ::= {with_clause {use_clause}}

    context_elem_s =>       as_list : Seq Of CONTEXT_ELEM;


-- Syntax 10.1.1.B
--  with_clause ::= with unit_simple_name {, unit_simple_name};


    CONTEXT_ELEM ::=        with;

    with =>                 as_name_s : name_s,
                            as_use_pragma_s : use_pragma_s;

    use_pragma_s =>         as_list : Seq Of USE_PRAGMA;


-- 10.2  Subunits of Compilation Units

-- Syntax 10.2.A
--  subunit ::=
--     separate (parent_unit_name) proper_body


    subunit =>              as_name : NAME,
                            as_subunit_body : SUBUNIT_BODY;

    SUBUNIT_BODY ::=        subprogram_body | package_body | task_body;
    SUBUNIT_BODY =>         as_source_name : SOURCE_NAME,
                            as_body : BODY;


-- Syntax 10.2.B
--  body_stub ::=
--       subprogram_specification is separate;
--     | package body package_simple_name is separate;
--     | task body task_simple_name is separate;

    stub => ;


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

-- Syntax 11.1
--  exception_declaration ::= identifier_list : exception;
```

```
    exception_decl => ;

    SOURCE_NAME ::=         exception_id;

    exception_id =>         sm_renames_exc : NAME;


-- 11.2  Exception Handlers

-- Syntax 11.2
--  exception_handler ::=
--     when exception_choice {| exception_choice} =>
--          sequence_of_statements
--
--  exception_choice ::= exception_name | others


-- 11.3  Raise Statements

-- Syntax 11.3
--  raise_statement ::= raise [exception_name];

    raise => ;


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

-- Syntax 12.1.A
--  generic_declaration ::= generic_specification;

--  generic_specification ::=
--       generic_formal_part subprogram_specification
--     | generic_formal_part package_specification

    HEADER ::=              package_spec;

    generic_decl =>         as_item_s : item_s;

    NON_TASK_NAME ::=       generic_id;

    generic_id =>           sm_generic_param_s : item_s,
                            sm_body : BODY,
                            sm_is_inline : Boolean;


-- Syntax 12.1.B
--  generic_formal_part ::= generic {generic_parameter_declaration}
```

```
-- Syntax 12.1.C
--  generic_parameter_declaration ::=
--       identifier_list : [in [out]] type_mark [:= expression];
--     | type identifier is generic_type_definition;
--     | private_type_declaration
--     | with subprogram_specification [is name];
--     | with subprogram_specification [is <>];


    UNIT_KIND ::=           GENERIC_PARAM;

    GENERIC_PARAM ::=       name_default
                          | box_default
                          | no_default;

    name_default =>         as_name : NAME;

    box_default => ;

    no_default => ;


-- Syntax 12.1.D
--  generic_type_definition ::=
--       (<>) | range <> | digits <> | delta <>
--     | array_type_definition | access_type_definition


    TYPE_DEF ::=            formal_dscrt_def
                          | formal_integer_def
                          | formal_fixed_def
                          | formal_float_def;

    formal_dscrt_def => ;
    formal_fixed_def => ;
    formal_float_def => ;
    formal_integer_def => ;


-- 12.3  Generic Instantiation

-- Syntax 12.3.A
--  generic_instantiation ::=
--       package identifier is
--            new generic_package_name [generic_actual_part];
--     | procedure identifier is
--            new generic_procedure_name [generic_actual_part];
--     | function identifier is
--            new generic_function_name [generic_actual_part];

--  generic_actual_part ::=
--     (generic_association {, generic_association})
```

```
    RENAME_INSTANT ::=      instantiation;

    instantiation =>        as_general_assoc_s : general_assoc_s;
    instantiation =>        sm_decl_s : decl_s;


-- Syntax 12.3.B
--  generic_association ::=
--     [generic_formal_parameter =>] generic_actual_parameter

--  generic_formal_parameter ::= parameter_simple_name | operator_symbol


-- Syntax 12.3.C
--  generic_actual_parameter ::= expression | variable_name
--     | subprogram_name | entry_name | type_mark


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

-- Syntax 13.1
--  representation_clause ::=
--       type_representation_clause | address_clause

--  type_representation_clause ::= length_clause
--     | enumeration_representation_clause | record_representation_clause


    REP ::=                 NAMED_REP | record_rep;
    REP =>                  as_name : NAME;

    NAMED_REP =>            as_exp : EXP;


-- 13.2 Length Clause
-- 13.3 Enumeration Representation Clauses

-- Syntax 13.2
--  length_clause ::= for attribute use simple_expression;

-- Syntax 13.3
--  enumeration_representation_clause ::=
--       for type_simple_name use aggregate;


    NAMED_REP ::=           length_enum_rep;
    length_enum_rep => ;


-- 13.4  Record Representation Clauses
```

```
-- Syntax 13.4.A
--  record_representation_clause ::=
--     for type_simple_name use
--          record [alignment_clause]
--               {component_clause}
--          end record;

--  alignment_clause ::= at mod static_simple_expression;

    ALIGNMENT_CLAUSE ::=   alignment | void;

    alignment =>           as_pragma_s : pragma_s,
                           as_exp : EXP;

    record_rep =>          as_alignment_clause : ALIGNMENT_CLAUSE,
                           as_comp_rep_s : comp_rep_s;


-- Syntax 13.4.B
--  component_clause ::=
--     component_simple_name at static_simple_expression range static_range;

    COMP_REP_ELEM ::=      comp_rep | void;
    COMP_REP_ELEM ::=      comp_rep_pragma;

    comp_rep_s =>          as_list : Seq Of COMP_REP_ELEM;

    comp_rep =>            as_name : NAME,
                           as_exp : EXP,
                           as_range : RANGE;

    comp_rep_pragma =>     as_pragma : pragma;

-- 13.5  Address Clauses

-- Syntax 13.5
--  address_clause ::= for simple_name use at simple_expression;

    NAMED_REP ::=          address;
    address => ;

-- 13.8  Machine Code Insertions

-- Syntax 13.8
--  code_statement ::= type_mark'record_aggregate;

    code => ;
```

```
-- 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
-- ============================

    PREDEF_NAME ::=         attribute_id
                          | pragma_id
                          | argument_id
                          | bltn_operator_id
                          | void;

    attribute_id => ;

    TYPE_SPEC ::=           universal_integer | universal_fixed | universal_real;

    universal_integer => ;
    universal_fixed => ;
    universal_real => ;

    argument_id => ;

    bltn_operator_id =>     sm_operator : operator;

    pragma_id =>            sm_argument_id_s : argument_id_s;

    argument_id_s =>        as_list : Seq Of argument_id;

    ALL_SOURCE ::=          DEF_NAME | ALL_DECL | TYPE_DEF | SEQUENCES
                          | STM_ELEM | GENERAL_ASSOC | CONSTRAINT | CHOICE
                          | HEADER | UNIT_DESC | TEST_CLAUSE_ELEM
                          | MEMBERSHIP_OP | SHORT_CIRCUIT_OP | ITERATION
                          | ALTERNATIVE_ELEM  | COMP_REP_ELEM | CONTEXT_ELEM
                          | VARIANT_ELEM | ALIGNMENT_CLAUSE | VARIANT_PART
                          | comp_list | compilation | compilation_unit | index;


    SEQUENCES ::=           alternative_s | argument_id_s | choice_s
                          | comp_rep_s  | compltn_unit_s | context_elem_s
                          | decl_s | dscrmt_decl_s | general_assoc_s
                          | discrete_range_s | enum_literal_s | exp_s | item_s
                          | index_s | name_s | param_s | pragma_s | scalar_s
                          | source_name_s | stm_s | test_clause_elem_s
                          | use_pragma_s | variant_s;

    ALL_SOURCE =>           lx_srcpos : source_position,
                            lx_comments : comments;

    ALL_DECL ::=            ITEM | subunit;

End
```

CHAPTER 3

SEMANTIC SPECIFICATION

This chapter describes the semantics of DIANA. The structure of this chapter parallels the DIANA class hierarchy. Each section corresponds to a class in the DIANA class hierarchy, and each subsection corresponds to a subclass of that class in the hierarchy. Each node is discussed in the section corresponding to the class which directly contains it.

Since the class structure of DIANA is a hierarchy, it was possible to construct hierarchy diagrams to illustrate pictorially the relationships between the various nodes and classes. At the end of each major section is a hierarchy diagram which depicts the nodes and classes discussed in that section, along with the attributes which are defined for those nodes and classes. Beneath each class or node name is a list of attribute names corresponding to the attributes which are defined at that level. Hence an attribute appearing immediately below a class name is defined for all classes and nodes which are below that class in the diagram.

It should be noted that the classes **ALL_SOURCE** and **SEQUENCES** have been omitted from this chapter due to the simple nature of their structure and the fact that they represent optional features of DIANA. All nodes which may represent source text have the attributes lx_srcpos and lx_comments; however, DIANA does not require that these attributes be represented in a DIANA structure. The sole purpose of class **ALL_SOURCE** is to define these two attributes for its constituents; the only nodes which do not inherit these attributes are those belonging to class **TYPE_SPEC**. In order to be consistent, the IDL specification of DIANA defines a sequence node (or header) for each sequence; however, an implementation is not required to represent the sequence node itself. Class **SEQUENCES** is a set of sequence nodes, all of which have a single attribute (other than lx_srcpos and lx_comments) called as_list which denotes the actual sequence.

The following conventions are observed throughout this chapter:

(a) All attributes which are inherited by the node **void** are undefined. In addition, no operations are defined for the attributes inherited by the node **void**.

(b) Although a class may contain the node **void**, an attribute which has that class as its type cannot be **void** unless the semantic specification explicitly states that the attribute may be **void**.

(c) The attributes lx_srcpos and lx_comments are undefined for any nodes which do not represent source code. For certain nodes, such as those in class **PREDEF_NAME**, these attributes will never be defined.

(d) Unless otherwise specified, all nodes represent source code.

(e) A sequence cannot be empty unless the semantic specification explicitly allows it.

(f) If the manual specifies that the copying of a node is optional, and an implementation chooses to copy that node, then the copying of any nodes denoted by structural attributes of the copied node is also optional.

Section 3.1

ALL_DECL

3.1 ALL_DECL

The four immediate offspring of class **ALL_DECL** are **void**, **subunit**, **block_master**, and **ITEM**.

The **subunit** node represents a subunit, and has two non-lexical attributes -- as_name and as_subunit_body. The attribute as_name denotes the name of the parent unit (a **selected**, **used_name_id**, or **used_op** node); as_subunit_body designates the node corresponding to the proper body.

The **block_master** node represents a block statement that may be a master because it contains immediately within its declarative part the definition of an access type which designates a task. Its only non-lexical attribute, sm_stm, denotes a **block_body** node. The **block_master** node can only be referenced by the sm_master attribute of the access node, thereby serving as an intermediate node between the access type definition and the block statement. The **block_master** node does not represent source code.

3.1.1 ITEM

In general, the nodes in class **ITEM** correspond to explicit declarations; i.e. declarative items that can be found in formal parts, declarative parts, component lists, and program unit specifications. Certain declarative nodes (**subtype_decl**, **constant_decl**, **renames_obj_decl**, and **subprog_entry_decl**) may also appear in another context -- as a part of a sequence of declarations constructed for the instantiation of a generic unit. When used in this special context these declarative nodes are not accessible through structural attributes, and do not correspond to source code.

Certain implicit declarations described in the Ada manual are not represented in DIANA: the predefined operations associated with type definitions; label, loop, and block names; anonymous base types created by a constrained array or scalar type definition; anonymous task types; derived subprograms. Although the entities themselves have explicit representations (i.e. defining occurrences), their declarations do not.

With the exception of the node **null_comp_decl** and the nodes in classes **REP** and **USE_PRAGMA**, all of the nodes in class **ITEM** have a child representing the identifier(s) or symbol(s) used to name the newly defined entity (or entities). These nodes, members of class **SOURCE_NAME**, are termed the "defining occurrence" of their respective identifiers; they carry all of the information that describes the associated entity.

The classes **DSCRMT_PARAM_DECL**, **SUBUNIT_BODY**, and **DECL** comprise **ITEM**.

3.1.1.1 DSCRMT_PARAM_DECL

The **DSCRMT_PARAM_DECL** class is composed of nodes representing either a discriminant specification or a formal parameter specification. The as_name attribute defined on this class denotes a **selected** or **used_name_id** node corresponding to the type mark given in the specification.

The **dscrmt_dec** node represents a discriminant specification. The as_source_name_s attribute denotes a sequence of **dscrmt_id** nodes, and the as_exp attribute references a node corresponding to the default initial value; if there is no initial value given, as_exp is **void**.

3.1.1.1.1 PARAM

A node in class **PARAM** may represent either a formal parameter specification contained in a formal part, or a generic formal object declaration. The as_source_name_s attribute denotes a sequence of **in_id**, **in_out_id**, and **out_id** nodes, unless the **PARAM** node corresponds to a generic formal object declaration, in which case only **in_id** and **in_out_id** nodes are permitted.

The **in** node represents a formal parameter declaration of mode in. Its as_exp attribute denotes the default value of the parameter, and is **void** if none is given. The **in** node also has an lx_default attribute which indicates whether or not the mode is specified explicitly.

The **in_out** and **out** nodes represent formal parameter declarations of mode in out and out, respectively. The as_exp attribute of these nodes is always **void**.

3.1.1.2 SUBUNIT_BODY

The class **SUBUNIT_BODY** is composed of nodes representing declarations of subunit bodies. The as_body attribute defined on this class may denote either a **block_body** or a **stub** node, depending on whether the declaration corresponds to a proper body or a body stub.

The **subprogram_body** node represents the declaration of a subprogram body. The as_source_name attribute denotes a node in class **SUBPROG_NAME**, and the as_header attribute references a **procedure_spec** or a **function_spec** node.

The **package_body** node represents the declaration of a package body; its as_source_name attribute refers to a **package_id node**. If the package body is empty (i.e. it contains no declarative part, no sequence of statements, and no exception handlers) then as_body still denotes a **block_body** node; however, all of the sequences in the **block_body** node are empty.

The **task_body** node represents the declaration of a task body; its as_source_name attribute denotes a **task_body_id**.

3.1.1.3 DECL

The class DECL contains the nodes associated with basic declarative items, record component declarations, and entry declarations.

The node **null_comp_decl** represents a record component list defined by the word "null". It has no attributes other than lexical ones, and appears only as the first member of a sequence denoted by the as_decl_s attribute of a **comp_list** node; the only kind of node which can succeed it in the sequence is a **pragma** node.

3.1.1.3.1 USE_PRAGMA

The class USE_PRAGMA contains the nodes **pragma** and **use**.

The **pragma** node represents a pragma. The as_used_name_id attribute denotes the name of the pragma, and the as_general_assoc_s attribute references a possibly empty sequence of argument associations (the sequence may contain a mixture of **assoc** and **EXP** nodes).

The **use** node represents a use clause. The as_name_s attribute represents the list of package names given in the use clause. If the use clause appears as a basic declarative item, the sequence can contain both **used_name_id** and **selected** nodes; if it is a part of a context clause, it will contain **used_name_id** nodes only.

3.1.1.3.2 REP

The nodes in class REP correspond to representation clauses which may appear as declarative items (i.e. address clauses, length clauses, record representation clauses, and enumeration representation clauses).

The node **record_rep** represents a record representation clause. The attribute as_name references a **used_name_id** corresponding to the record type name; as_alignment_clause and as_comp_rep_s denote the alignment clause and component clauses, respectively. The attribute as_alignment_clause is **void** if the representation clause does not contain an alignment clause, and as_comp_rep_s may be empty if no component clauses or pragmas are present.

3.1.1.3.2.1 NAMED_REP

The nodes **length_enum_rep** and **address** comprise the class NAMED_REP, a group of representation clauses which consist of a name and an expression.

The **length_enum_rep** node may represent either a length clause or an enumeration representation clause. In the former case the as_name attribute denotes an **attribute** node and the as_exp attribute corresponds to the simple expression. In the case of an enumeration representation clause the as_name attribute denotes a **used_name_id** corresponding to the enumeration type, and as_exp references an **aggregate** node.

The **address** node represents an address clause. Its as_name attribute references a node from the class **USED_SOURCE_NAME** corresponding to the name of the entity for which the address is being specified. The as_exp attribute records the address expression.

3.1.1.3.3 ID_DECL

The **ID_DECL** class represents those declarations which define a single entity rather than a sequence of entities (i.e. declarations defining an identifier, not an identifier list). Included in this class are the **type_decl**, **subtype_decl**, and **task_decl** nodes, as well as the **UNIT_DECL** and **SIMPLE_RENAME_DECL** classes, representing unit declarations and renaming declarations, respectively.

The **type_decl** node represents a type declaration -- incomplete, private, generic, derived, or full. The only type declaration that is not represented by this node is that of a task type, which is denoted by a **task_decl** node instead. The **type_decl** node has three non-lexical attributes: as_source_name, as_dscrmt_decl_s, and as_type_def.

The as_source_name attribute of a **type_decl** node denotes a node representing a new defining occurrence of the type name, the kind of node depending on the kind of type declaration. Certain type names will have more than one declaration point -- those corresponding to incomplete types or (limited) private types. The as_source_name attribute of the **type_decl** node associated with a (limited) private type declaration references a **private_type_id** or **l_private_type_id** node; for all other type declarations as_source_name will designate a **type_id** node. The subsequent full type declaration for an incomplete or (limited) private type is treated as an ordinary full type declaration; hence the as_source_name attribute of the full type declaration corresponding to a (limited) private type will denote a **type_id** rather than a **private_type_id** or **l_private_type_id**.

The as_dscrmt_decl_s attribute of a **type_decl** node is a possibly empty sequence containing the discriminant declarations which appear in the type declaration; for declarations of derived types and generic formal types which are not private this sequence is always empty.

The as_type_def attribute associated with a **type_decl** node designates a node representing the portion of source code following the reserved word "is"; hence the as_type_def attribute for an incomplete type definition is **void**, and may not be **void** for any other kind of type declaration. The permitted values of the as_type_def attribute for the remainder of the type declarations are as follows: for a (limited) private type declaration -- a **private_def** or **l_private_def** node; for a generic type declaration -- a **TYPE_DEF** node having the prefix "formal_", an **unconstrained_array_def** node, a **constrained_array_def** node, or an **access_def** node; for a derived type declaration -- a **derived** node; and finally, for a full type declaration -- an **enumeration_def**, **integer_def**, **float_def**, **fixed_def**, **unconstrained_array_def**, **constrained_array_def**, **record_def**, or **access_def** node.

The **subtype_decl** node represents a subtype declaration; it defines two attributes: as_source_name and as_subtype_indication. The former denotes a **subtype_id** node, and the latter a **subtype_indication** node. The **subtype_id** represents the defining occurrence of the subtype name, and the **subtype_indication** node records the type mark and constraint appearing in the subtype declaration.

The second context in which a **subtype_decl** node may appear is as a part of a normalized parameter list for a generic instantiation, in which case the **subtype_decl** node does not represent actual source code. This case is discussed in more detail in section 3.6.1.1.

The **task_decl** node represents the declaration of either a task type or a single task object with an anonymous type, depending on whether or not the reserved word "type" is included in the specification. The difference is indicated by the value of the as_source_name attribute -- a **type_id** node in the former case, a **variable_id** node in the latter. The as_decl_s attribute is a possibly empty sequence of nodes representing the entry declarations and representation clauses given in the task specification (**subprog_entry_decl** and **REP** nodes). The declaration of a task object (or objects) of a named type is represented by a **variable_decl** node rather than a **task_decl** node.

3.1.1.3.3.1 SIMPLE_RENAME_DECL

The class SIMPLE_RENAME_DECL contains nodes representing the renaming of an object or an exception. The renaming of an entity as a subprogram or a package is represented by a **subprog_entry_decl** node or a **package_decl** node, respectively.

A renaming declaration for an object is represented by a **renames_obj_decl** node. The as_source_name attribute denotes a **variable_id** or a **constant_id**, depending on the kind of object renamed. A constant object is represented by a **constant_id**; constant objects include constants, discriminants, parameters of mode **in**, loop parameters, and components of constant objects. An object that does not belong to any of the previous categories is represented by a **variable_id** (this includes objects of a limited type). The as_name attribute of a **renames_obj_decl** node denotes a node of type **NAME** which represents the object being renamed. The as_type_mark_name attribute references a selected or **used_name_id** node corresponding to the type mark appearing in the renaming declaration.

The **renames_obj_decl** node may also appear in a normalized parameter list for a generic instantiation. This case does not correspond to source code, and is discussed in detail in section 3.6.1.1.

The renaming of an exception is represented by a **renames_exc_decl** node, for which the as_source_name attribute always designates an **exception_id**. The as_name attribute can be either a **selected** node or a **used_name_id** node corresponding to the exception being renamed.

3.1.1.3.3.2 UNIT_DECL

The class **UNIT_DECL** represents the declaration of a subprogram, package, generic unit, or entry. The as_header attribute which is defined on the class references a **HEADER** node, the type of which is determined by the reserved word appearing in the declaration (i.e. "procedure", "function", "package", or "entry").

The **generic_decl** node corresponds to the declaration of a generic unit. The as_source_name attribute references a **generic_id** representing the name of the generic unit. The as_header attribute may denote a **procedure_spec**, a **function_spec**, or a **package_spec**. The attribute as_item_s is a possibly empty sequence of generic formal parameter declarations -- a list of nodes of type **in**, **in_out**, **type_decl**, or **subprog_entry_decl**.

3.1.1.3.3.2.1 NON_GENERIC_DECL

The class **NON_GENERIC_DECL** encompasses subprogram, package, and entry declarations. The as_unit_kind attribute that is defined on the class determines the kind of declaration the **subprog_entry_decl** or **package_decl** node represents: a renaming declaration, an instantiation, a generic formal parameter declaration, or an "ordinary" declaration.

An entry (family) declaration is represented by a **subprog_entry_decl** node for which the as_source_name attribute is an **entry_id**, the as_header attribute is an **entry** node, and the as_unit_kind attribute is **void**. The renaming of an entry as a procedure is treated as a procedure declaration (i.e. the as_source_name attribute is a **procedure_id**, not an **entry_id**).

The as_source_name attribute of a **package_decl** node will always designate a **package_id**, and the sm_header attribute -- a package_spec. However, the as_unit_kind attribute may have one of three values: **renames_unit** (representing the name of the unit being renamed), **instantiation** (representing the name of the generic unit and the generic actual part), or **void** (if the declaration is an "ordinary" one).

The declaration of a procedure, a function, or an operator is represented by a **subprog_entry_decl** node, for which the as_header attribute can be either a **procedure_spec** or a **function_spec**. In addition to the three values of as_unit_kind described in the previous paragraph, the as_unit_kind attribute of a **subprog_entry_decl** node may designate a node from class **GENERIC_PARAM** if the subprogram in the declaration is a generic formal parameter.

The as_source_name attribute for a subprogram declaration is a node from class **SUBPROG_NAME**, with one exception. A declaration renaming an enumeration literal as a function will have an **ENUM_LITERAL** node as its as_source_name attribute (the **function_spec** node denoted by the as_header attribute will contain an empty parameter list). For all other declarations the type of node designated by the as_source_name attribute is determined by the kind of declaration introducing the new name (i.e. a declaration renaming an attribute as a function will have a **function_id** as its as_source_name attribute).

A **subprog_entry_decl** node may also appear in a normalized parameter list for a generic instantiation. In this case the declaration will always be a renaming declaration which does not correspond to source code (see section 3.6.1.1 for details).

3.1.1.3.4 ID_S_DECL

The **ID_S_DECL** class contains nodes corresponding to declarations which may define more than one entity -- variable declarations, (deferred) constant declarations, record component declarations, number declarations, and exception declarations. Although any of these declarations may introduce a single identifier, a node from class **ID_S_DECL** will always be used to represent the declaration, never a node from class **ID_DECL**.

An **exception_decl** node represents an exception declaration; the as_source_name_s attribute designates a sequence of **exception_id** nodes.

A **deferred_constant_decl** node denotes a deferred constant declaration. The as_source_name_s attribute refers to a sequence of **constant_id** nodes; each **constant_id** node represents the first defining occurrence of the associated identifier. The as_name attribute of the **deferred_constant_decl** node is a **used_name_id** or **selected** node representing the type mark given in the declaration. The subsequent full declaration of the deferred constant(s) will be represented by a **constant_decl** node.

3.1.1.3.4.1 EXP_DECL

The **EXP_DECL** class represents multiple object declarations that can include an initial value -- number declarations, variable declarations, and constant declarations.

A number declaration is denoted by a **number_decl** node for which the as_source_name_s attribute is a sequence of **number_id** nodes, and the as_exp attribute references a node corresponding to the static expression given in the declaration.

3.1.1.3.4.1.1 OBJECT_DECL

Class **OBJECT_DECL** represents variable, constant, and component declarations.

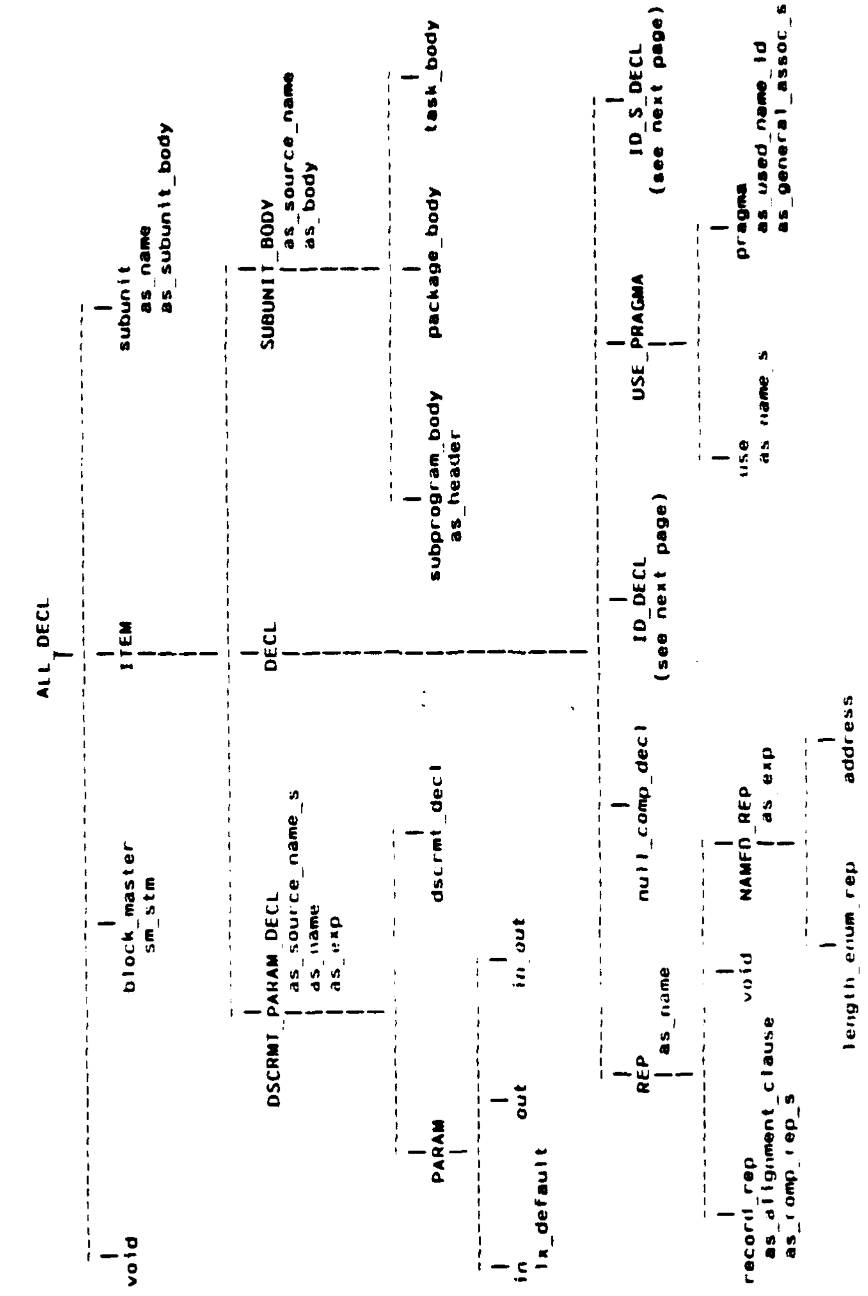
A **variable_decl** node represents either a variable declaration in a declarative part or a component declaration in a record type definition; **as_source_name_s** is a sequence of **variable_id** nodes or **component_id** nodes, respectively. The as_exp attribute denotes the (default) initial value, and is **void** if none is given. For a variable declaration, as_type_def may denote either a **subtype_indication** node or a **constrained_array_def** node; for a component declaration as_type_def refers to a **subtype_indication** node.

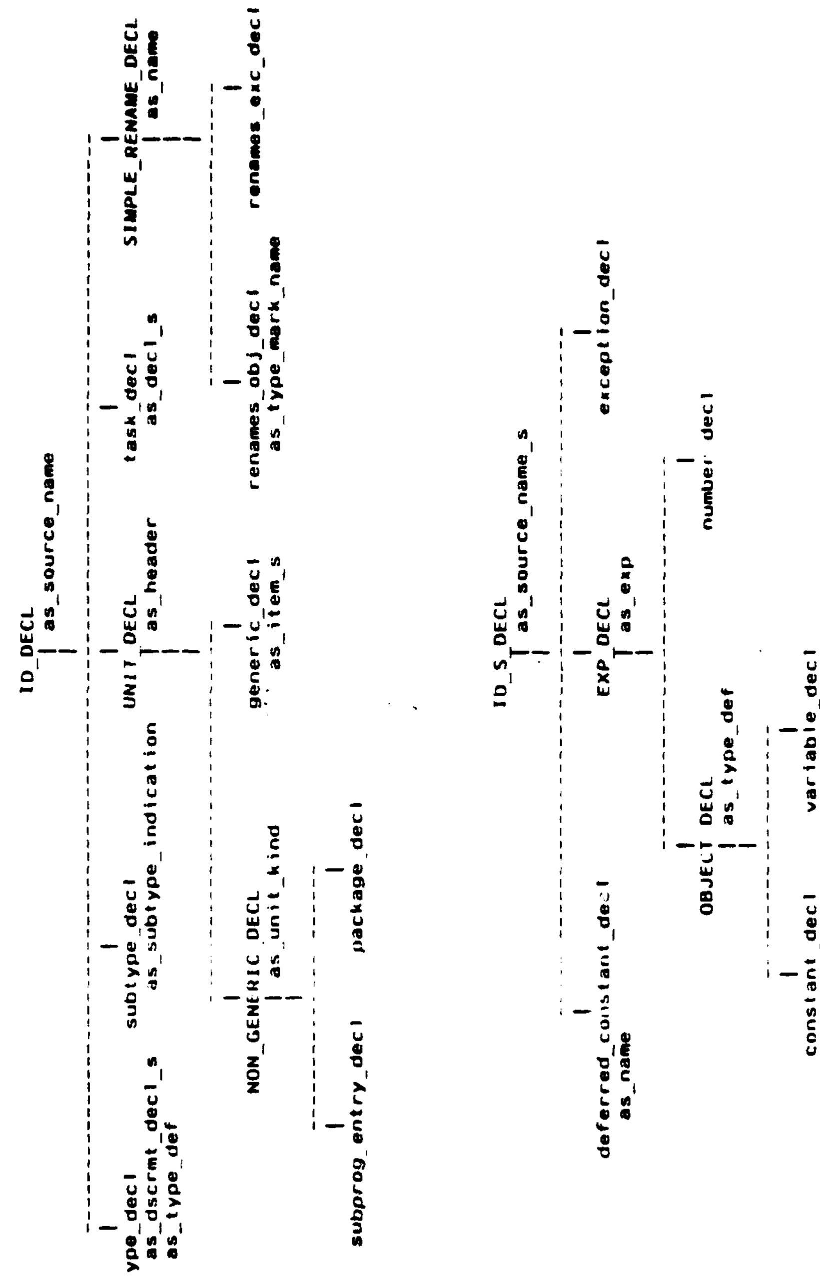
A **constant_decl** node represents a full constant declaration. The attribute as_source_name_s is a sequence of **constant_id** nodes; as_exp represents the initial value. The as_type_def attribute may denote either a **subtype_indication** node or a **constrained_array_def** node.

A **constant_decl** node may also appear in a special normalized parameter sequence for a generic instantiation, in which case it does not represent source code (see section 3.6.1.1 for details).

Unlike other object declarations, which contain named types only, the declarations in class **OBJECT_DECL** may introduce anonymous subtypes via a constrained array definition or the inclusion of a constraint in the subtype indication. If the object(s) being declared are of a named type, then the sm_obj_type attribute of each defining occurrence node in the **as_source_name_s** sequence denotes the same entity -- the **TYPE_SPEC** node referenced in the defining occurrence node corresponding to the type mark.

If the object declaration contains an anonymous subtype (i.e. as_type_def denotes a **constrained_array_def** node or a **subtype_indication** node with a non-void as_constraint attribute) then a different **TYPE_SPEC** node will be created for the sm_obj_type attribute of each defining occurrence node in the **as_source_name_s** sequence. The sm_is_anonymous attribute of each will have the value TRUE. If the constraint is non-static, then each **TYPE_SPEC** node references its own copy of the **CONSTRAINT** node corresponding to the new constraint; if the constraint is static then each **TYPE_SPEC** may or may not reference its own copy. DIANA does not require that the node referenced by the **TYPE_DEF** attribute of the **OBJECT_DECL** node have a unique node representing the constraint, even if the constraint is non-static; the **OBJECT_DECL** node is allowed to share the **CONSTRAINT** node with one of the **TYPE_SPEC** nodes.


ALL_DECL
void
block_master
sm_stm
ITEM
subunit
as_name
as_subunit_body
DSCRMT_PARAM_DECL
as_source_name_s
as_name
as_exp
DECL
SUBUNIT_BODY
as_source_name
as_body
PARAM
dscrmt_decl
subprogram_body
as_header
package_body
task_body
in
lx_default
out
in_out
REP
as_name
null_comp_decl
ID_DECL
(see next page)
USE_PRAGMA
ID_S_DECL
(see next page)
record_rep
as_alignment_clause
as_comp_rep_s
void
NAMED_REP
as_exp
length_enum_rep
address
use
as_name_s
pragma
as_used_name_id
as_general_assoc_s


ID_DECL
as_source_name
type_decl
as_dscrmt_decl_s
as_type_def
subtype_decl
as_subtype_indication
UNIT_DECL
as_header
task_decl
as_decl_s
SIMPLE_RENAME_DECL
as_name
NON_GENERIC_DECL
as_unit_kind
generic_decl
as_item_s
renames_obj_decl
as_type_mark_name
renames_exc_decl
subprog_entry_decl
package_decl
ID_S_DECL
as_source_name_s
deferred_constant_decl
as_name
EXP_DECL
as_exp
exception_decl
OBJECT_DECL
as_type_def
number_decl
constant_decl
variable_decl

Section 3.2

DEF_NAME

3.2 DEF_NAME

The appearances of identifiers, operators, and enumeration characters in a *DIANA tree are divided into defining and used occurrences*; the class **DEF_NAME** contains all of the nodes representing defining occurrences. Each entity of an Ada program has a defining occurrence; uses of the name or symbol denoting the entity always refer to this definition. The defining occurrence contains the semantic information pertaining to the associated entity; none of the nodes in class **DEF_NAME** have any structural attributes.

The names represented by this class fall into two principal categories: predefined names and user-defined names. Defining occurrences corresponding to user-defined entities are introduced by the as_source_name or as_source_name_s attribute of nodes in class **ITEM**, **BLOCK_LOOP**, **LABELED**, and **FOR_REV**. Defining occurrences associated with predefined entities are not accessable via structural attributes since they do not have a declaration point.

Each node in class **DEF_NAME** has an lx_symrep attribute to retain the source representation of the identifier or character literal associated with the defining occurrence. Those nodes in class **SOURCE_NAME** generally have lx_srcpos and lx_comments attributes for which the values are defined; the values of these attributes are undefined for nodes in class **PREDEF_NAME**. Certain nodes in class **SOURCE_NAME** may be used to represent both predefined and user-defined names (nodes such as exception_id); however, lx_srcpos and lx_comments for these nodes are undefined when representing a predefined name.

The names associated with certain entities may have more than one point of definition; in particular, those corresponding to:

(a) deferred constants

(b) incomplete types

(c) non-generic (limited) private types

(d) discriminants

(e) non-generic formal parameters

(f) program units

For these names, the first defining occurrence (which is indicated by the sm_first attribute) is treated as THE definition. In general, all references to the entity refer to the first defining occurrence, and the multiple defining occurrences of an entity all have the same attribute values. Types and deferred constants present special cases which are discussed in subsequent sections.

3.2.1 PREDEF_NAME

The nodes in class **PREDEF_NAME** correspond to the names of entities for which the Ada language does not provide a means of declaration; consequently a node from class **PREDEF_NAME** will NEVER be designated by a structural attribute.

The nodes **attribute_id**, **argument_id**, **pragma_id**, **bltn_operator_id**, and **void** comprise this class. The nodes **argument_id** (the name of a pragma argument or argument value) and **attribute_id** (the name of an Ada attribute) have no attributes other than lx_symrep. The **pragma_id** represents the name of a pragma. The sm_argument_id_s attribute denotes a sequence of argument identifiers associated with the pragma (i.e. the sequence for pragma LIST contains nodes denoting the argument identifiers ON and OFF); if a particular pragma has no argument identifiers the sequence is empty. The node **bltn_operator_id** corresponds to a predefined operator; the different operators are distinguished by the sm_operator attribute.

3.2.2 SOURCE_NAME

The **SOURCE_NAME** class is composed of those nodes corresponding to defining occurrences of entities which may be declared by the user.

The **exception_id** node represents an exception name. If the **exception_id** is a renaming then the sm_renames attribute is a **used_name_id** or a **selected** node denoting the original exception name (the node which is designated by the as_name attribute of the **renames_exc_decl** node). If the exception name is not introduced by a renaming declaration then sm_renames is **void**.

An entry (family) name is denoted by an **entry_id** node which has two non-lexical attributes : sm_spec and sm_address. The sm_spec attribute references the **entry** node (which contains the discrete range and formal part) designated by the as_header attribute of the **subprog_entry_decl** node. The sm_address attribute denotes the expression given in an address clause; if no address clause is applicable this attribute is **void**.

3.2.2.1 LABEL_NAME

The class **LABEL_NAME** represents those identifiers associated with statements; the sm_stm attribute defined on this class denotes the statement to which the name corresponds. A label_id node represents the name of a statement label and is introduced by a **labeled** node; sm_stm can reference any node in class **STM**. A **block_loop_id** represents the name of a block or a loop; sm_stm denotes the block or loop node which introduces the **block_loop_id**.

3.2.2.2 TYPE_NAME

The class **TYPE_NAME** contains nodes associated with the names of types or subtypes; it has an sm_type_spec attribute attribute defined on it. Certain type names may have more than one defining occurrence; in particular, those corresponding to private and limited private types which are not generic formal types, and those associated with incomplete types.

A **private_type_id** or **l_private_type_id** node represents the defining occurrence of a type name introduced by a (limited) private type declaration; the type may or may not be a generic formal type. A **private_type_id** or **l_private_type_id** node has an sm_first attribute that references itself, and an sm_type_spec attribute denoting a **private** or **l_private** node.

If the (limited) private type is not a generic formal type then its name has a second defining occurrence corresponding to the subsequent full type declaration. The second defining occurrence is represented by a **type_id** node; the sm_first attribute references the **private_type_id** or **l_private_type_id** node of the corresponding (limited) private type declaration, and the sm_type_spec attribute denotes the full type specification, a node belonging to class **FULL_TYPE_SPEC**.

Used occurrences of a (limited) private type name will reference the **private_type_id** or **l_private_type_id** as the definition.

Each defining occurrence of the name of an incomplete type is represented by a **type_id** node, the sm_first attribute of which denotes the **type_id** node corresponding to the incomplete type declaration. Ordinarily, the sm_type_spec attribute of the **type_id** nodes for both the incomplete and the full type declaration refer to the full type specification -- a node from class **FULL_TYPE_SPEC**. The single exception occurs when the incomplete type is declared "immediately within the private part of a package" [ARM, section 3.8.1] and the package body containing the full type declaration is a separate compilation unit, in which case the sm_type_spec attribute of the **type_id** corresponding to the incomplete type declaration denotes an **incomplete** node.

The defining occurrences of all other kinds of type names are represented by **type_id** nodes. The sm_first attribute references the node which contains it, and the sm_type_spec attribute denotes a node belonging to the class **FULL_TYPE_SPEC**.

A new **TYPE_SPEC** node is created for the sm_type_spec attribute of a **type_id** node unless the **type_id** corresponds to an incomplete type declaration and the full type declaration is in the same compilation unit. A new **private** or **l_private** node is always created for the sm_type_spec attribute of a **private_type_id** or **l_private_type_id** node.

A **subtype_id** node represents the defining occurrence of a subtype name; its only non-lexical attribute is sm_type_spec, which references the appropriate subtype specification. If the **subtype_id** is introduced by a subtype declaration in which the subtype indication contains a constraint then a new **TYPE_SPEC** node is created to represent the subtype specification. If the subtype declaration does not impose a new constraint then the sm_type_spec attribute references the **TYPE_SPEC** node associated with the type mark appearing in the declaration.

A **subtype_id** may also be introduced by a declarative node in a normalized parameter list for a generic instantiation, in which case the **subtype_id** does not correspond to source code. The correct values for its attributes in this instance are defined in section 3.6.1.1.

3.2.2.3 OBJECT_NAME

The class **OBJECT_NAME** contains nodes representing defining occurrences of entities having a value and a type; it is composed of **iteration_id**, **ENUM_LITERAL**, and **INIT_OBJECT_NAME**. The sm_obj_type attribute which is defined on the class denotes the subtype of the object or literal.

An **iteration_id** represents the defining occurrence of a loop parameter, and is introduced by an **iteration** node. The sm_obj_type attribute references the **enumeration** or **integer** node denoted by the sm_base_type attribute of the **DISCRETE_RANGE** node associated with the iteration scheme.

3.2.2.3.1 ENUM_LITERAL

The class **ENUM_LITERAL** is composed of nodes representing the defining occurrences of literals associated with an enumeration type. The nodes **enumeration_id** and **character_id** comprise this class -- **enumeration_id** corresponds to an identifier, **character_id** to a character literal.

ENUM_LITERAL defines the attributes sm_pos and sm_rep, both of which are of type Integer. The attribute sm_pos contains the value of the predefined Ada attribute POS, i.e. the universal integer corresponding to the actual position number of the enumeration literal. The sm_rep attribute contains the value of the predefined Ada attribute VAL; the user may set this value with an enumeration representation clause. If no such clause is in effect, the value of sm_rep will be the same as that of sm_pos. The sm_obj_type attribute references the enumeration node corresponding to the enumeration type to which the literal belongs.

An **ENUM_LITERAL** node may be introduced by either an **enumeration_def** node or a **subprog_entry_decl** node. The latter corresponds to the renaming of an enumeration literal as a function, in which case the semantic attributes of the **ENUM_LITERAL** node will have the same values as those of the node corresponding to the original literal.

An **ENUM_LITERAL** node may be introduced by a declarative node in a special normalized parameter list for a generic instantiation; in this instance the **ENUM_LITERAL** node does not correspond to source code. This case is discussed in detail in section 3.6.1.1.

3.2.2.3.2 INIT_OBJECT_NAME

The class **INIT_OBJECT_NAME** contains nodes corresponding to defining occurrences of objects which may have an initial value; it defines an attribute sm_init_exp to record this value. This attribute represents those (default) initial values which are explicitly given; i.e. the default value NULL for an access object is not represented by sm_init_exp unless it is explicitly specified in the source code. The objects denoted by the nodes of this class include named numbers, variables, constants, record components, and formal parameters.

The node **number_id** represents the definition of a named number. The sm_obj_type attribute denotes a **universal_integer** or **universal_real** node, and the sm_init_exp attribute references the node denoted by the as_exp attribute of the corresponding **number_decl** node.

3.2.2.3.2.1 VC_NAME

The class **VC_NAME** is composed of the nodes **variable_id** and **constant_id**, denoting the names of variables and constants, respectively. The attributes sm_renames_obj and sm_address are defined for the nodes in this class.

The sm_renames_obj attribute is of type Boolean, and indicates whether or not the name of the object is a renaming; the value of this attribute determines the meaning of the sm_init_exp attribute for nodes in this class. If the name is introduced by a renaming declaration then sm_init_exp denotes the **NAME** node referenced by the as_name attribute of the **renames_obj_decl** node. Otherwise, sm_init_exp is the **EXP** node designated by the as_exp attribute of the associated **OBJECT_DECL** node, and consequently may be **void**.

The sm_address attribute denotes the expression for the address of the object as given in an address clause; if no such clause is applicable sm_address is **void**. In the case of a renaming, the value of the sm_address attribute is determined by the original object; if the original object cannot be named in an address clause then sm_address is **void**.

For a **VC_NAME** node corresponding to an ordinary object declaration the sm_obj_type attribute denotes either the **TYPE_SPEC** node corresponding to the type mark in the declaration, or an anonymous **TYPE_SPEC** node if the declaration contains a constrained array definition or a constraint in the subtype indication. If the **variable_id** or **constant_id** is introduced by a renames_obj_decl node, then sm_obj_type is the **TYPE_SPEC** node corresponding to the subtype of the original object (hence this **TYPE_SPEC** node does not necessarily correspond to the type mark in the renaming declaration, although it will have the same base type).

A **constant_id** represents the name of a constant object. A constant object may be either a (deferred) constant or the renaming of one of the following: a (deferred) constant, a discriminant, a loop parameter, a (generic) formal parameter of mode in, or a component of a constant object. The sm_first attribute references the **constant_id** node corresponding to the first defining occurrence of the associated name. For a **constant_id** node associated with the full declaration of a deferred constant this attribute will reference the **constant_id** corresponding to the deferred declaration; for all other **constant_id** nodes the sm_first attribute will contain a self-reference.

The attributes of the **constant_id** nodes representing the defining occurrences of a deferred constant have the same values. The sm_obj_type attribute designates a **private** or **l_private** node, and sm_init_exp denotes the initialization expression given in the full constant declaration. Used occurrences of a deferred constant *name* reference the **constant_id** of the deferred declaration.

The **variable_id** node represents the name of an object which is declared in an object declaration or a renaming declaration but is not a constant object. The sm_is_shared attribute has a Boolean value indicating whether or not a SHARED pragma has been applied to the variable. If the **variable_id** represents a renaming then sm_is_shared indicates whether or not the original object is shared.

Both the **constant_id** and the **variable_id** nodes may be introduced by declarative nodes in a normalized parameter list for a generic instantiation, in which case they do not represent source code. The appropriate values for the attributes of each are discussed in section 3.6.1.1.

3.2.2.3.2.2 COMP_NAME

The nodes **component_id** and **discriminant_id** comprise the class **COMP_NAME**, which represents the defining occurrences of identifiers associated with record components and record discriminants. The attribute sm_comp_rep is defined for the nodes in this class; it references the node corresponding to the applicable component representation clause, and is **void** if no such clause exists. The attribute sm_comp_rep can never denote a **comp_rep_pragma** node.

The sm_init_exp attribute represents the default initial value, referencing the **EXP** node designated by the as_exp attribute of the **variable_decl** or **dscrmt_decl** node (hence sm_init_exp can be **void**).

Unlike component names, discriminant names may have multiple defining occurrences, therefore an sm_first attribute is defined for the **discriminant_id** node (the instance of a component name in a component representation clause is considered to be a used occurrence rather than a defining occurrence). If an incomplete or non-generic (limited) private declaration contains a discriminant part, the discriminants will have a second definition point at the full type declaration; the sm_first attribute of both **discriminant_id** nodes will reference the **discriminant_id** node corresponding to the earlier incomplete or (limited) private declaration.

3.2.2.3.2.3 PARAM_NAME

The class **PARAM_NAME** contains nodes corresponding to the names of formal parameters declared in the formal parts of subprograms, entries, accept statements, and generic units. The nodes **in_id**, **in_out_id**, and **out_id** comprise **PARAM_NAME**, representing parameters of mode in, in out, and out, respectively (an **out_id** node can never be used to represent a generic formal object).

The attribute sm_init_exp records the initial value; it denotes the EXP node referenced by the as_exp attribute of the corresponding **in**, **in_out**, or **out** node. The attribute sm_init_exp is **void** for **in_out_id** and **out_id** nodes.

Formal parameters associated with subprogram declarations, entry declarations, and accept statements may have more than one defining occurrence. The sm_first attribute for a **PARAM_NAME** node belonging to an entry declaration or an accept statement will always reference the **PARAM_NAME** node of the entry declaration. The sm_first attribute of a **PARAM_NAME** node corresponding to a subprogram name denotes the **PARAM_NAME** node of the subprogram declaration, body declaration, or stub declaration which first introduces the identifier.

3.2.2.4 UNIT_NAME

The class **UNIT_NAME** represents the defining occurrences of those identifiers and symbols associated with program units; it contains the nodes **task_body_id**, **generic_id**, and **package_id**, as well as the class **SUBPROG_NAME**.

The **task_body_id** node denotes a task unit name introduced by the declaration of a body or a stub. The sm_first attribute references the **type_id** or **variable_id** node (depending on whether or not the task type is anonymous) of the task specification. The sm_type_spec attribute denotes the **task_spec** node denoted by either the sm_type_spec attribute of the **type_id** node or the sm_obj_type attribute of the **variable_id** node.

If the body of the task is in the same compilation unit then the sm_body attribute of the **task_body_id** references the body (a **block_body** node). If the body is in another compilation unit, but the stub is not, then sm_body denotes the stub (a **stub** node). Otherwise sm_body is **void**.

3.2.2.4.1 NON_TASK_NAME

The nodes in class **NON_TASK_NAME** correspond to the names of program units which are not tasks. The node **generic_id** and the class **SUBPROG_PACK_NAME** comprise this class.

The **generic_id** node corresponds to the defining occurrence of the name of a generic unit (the name of an instantiated unit is represented by a member of class **SUBPROG_PACK_NAME**). The sm_first attribute of a **generic_id** always references the **generic_id** of the generic specification. The sm_spec attribute denotes the **procedure_spec**, **function_spec**, or **package_spec** associated with the subprogram or package specification. The attribute sm_generic_param_s represents the formal part of the generic specification, and references the same sequence as the as_item_s attribute of the corresponding generic_decl node. The sm_is_inline attribute indicates whether or not an INLINE pragma has been given for the generic unit. The value of the sm_body attribute is determined in the same manner as the sm_body attribute of the **task_body_id** node (discussed in the previous section).

3.2.2.4.1.1 SUBPROG_PACK_NAME

Defining occurrences of packages and subprograms are represented by members of class **SUBPROG_PACK_NAME**. The attributes sm_address and sm_unit_desc are defined on this class. The sm_address attribute records the expression given in an address clause for the unit, if such a clause does not exist then sm_address is **void**. The sm_unit_desc attribute is a multi-purpose attribute; in some cases it is used to indicate that a particular unit is a special case (such as a renaming), in others it is used as a "shortcut" to another node (such as the unit body).

The node **package_id** represents the defining occurrence of a package; its sm_spec attribute denotes a **package_spec** node. If the **package_id** does not correspond to a renaming or an instantiation then sm_spec references the **package_spec** designated by the as_header attribute of the **package_decl** node, sm_first references the **package_id** of the package specification, and sm_unit_desc denotes a node from class **BODY** (the value of this attribute is determined in the same manner as the value of the sm_body attribute of the **task_body_id**, which is discussed in section 3.2.2.4).

If the **package_id** corresponds to a renaming then the sm_unit_desc attribute references a **renames_unit** node which provides access to the original unit. The sm_first attribute of the **package_id** contains a self-reference, while the sm_spec and sm_address attributes have the same values as those of the original package.

If the **package_id** is introduced by an instantiation then sm_unit_desc designates an **instantiation** node containing the generic actual part as well as a normalized parameter list. The sm_first attribute of the **package_id** contains a self-reference; the value of sm_address is determined by the existence of an address clause for the instantiated package, consequently it may be **void**. The sm_spec attribute references a new **package_spec** node that is created by copying the specification of the generic unit and replacing every occurrence of a formal parameter by a reference to a an entity in the normalized parameter list. The construction of the new specification is discussed in further detail in section 3.6.1.1.

3.2.2.4.1.1.1 SUBPROG_NAME

The class **SUBPROG_NAME** represents defining occurrences of subprograms; it comprises the nodes **procedure_id**, **function_id**, and **operator_id**. The attributes sm_is_inline and sm_interface are defined for the nodes in this class. The sm_is_inline attribute has a boolean value which indicates whether or not an INLINE pragma has been given for the subprogram. If an INTERFACE pragma is given for the subprogram then sm_interface denotes the pragma, otherwise it is **void**.

The **procedure_id** node corresponds to a defining occurrence of a procedure or an entry renamed as a procedure; its sm_spec attribute references a **procedure_spec** node. In addition to representing a function, a **function_id** may represent an attribute or operator renamed as a function. An **operator_id** may denote an operator or a function renamed as as operator. The sm_spec attribute of a **function_id** or ar **operator_id** designates a **function_spec** node.

If the **SUBPROG_NAME** node is introduced by an "ordinary" declaration then sm_unit_desc denotes a member of class **BODY**, and sm_spec references the **HEADER** node denoted by the as_header attribute of the associated **subprog_entry_decl** node. The appropriate **BODY** node is selected in the manner described for the sm_body attribute of the **task_body_id** node in section 3.2.2.4.

Like a **package_id**, a **SUBPROG_NAME** node may be introduced by a renaming declaration, in which case sm_unit_desc denotes a **renames_unit** node and sm_first contains a self-reference. However, the sm_spec attribute of the **SUBPROG_NAME** node does not denote the specification of the original unit, but that of the renaming declaration. The values of the attributes sm_address, sm_is_inline, and sm_interface are the same as those of the original unit.

The instantiation of a subprogram is treated in the same manner as the instantiation of a package. The sm_unit_desc attribute denotes an **instantiation** node, sm_first contains a self-reference, and a new **procedure_spec** or **function_spec** is constructed in the manner described in section 3.6.1.1. The values of sm_address and sm_interface are determined by the presence of an associated address clause or INTERFACE pragma; either may be **void**. The sm_is_inline attribute is true if an INLINE pragma is given for the generic unit OR the instantiated unit.

A **SUBPROG_NAME** node may also represent a generic formal parameter, in which case sm_unit_desc denotes a node belonging to class **GENERIC_PARAM**, the sm_first attribute contains a self-reference, and sm_spec denotes the **HEADER** node introduced by the generic parameter declaration. The attributes sm_address and sm_interface are **void**, and sm_is_inline is false.

The sm_unit_desc attribute of an **operator_id** may reference an **implicit_not_eq** node, which indicates that the inequality operator has been declared implicitly by the user through the declaration of an equality operator. The inequality operator is not the predefined operator, but because it cannot be explicitly declared it has no corresponding body, hence that of the corresponding equality operator must be utilized. Access to the equality operator is provided by the **implicit_not_eq** node. An **operator_id** which contains a reference to an **implicit_not_eq** node can be denoted only by semantic attributes.

The sm_spec attribute of an **operator_id** representing an implicitly declared inequality operator may reference either the **function_spec** of the corresponding equality operator or a copy of it. The attribute sm_address is **void**, sm_interface has the same value as the sm_interface attribute of the corresponding equality operator, and the value of sm_is_inline is determined by whether or not an INLINE pragma is given for the implicitly declared inequality operator.

The sm_unit_desc attribute of a **SUBPROG_NAME** node may reference a **derived_subprog** node, in which case the procedure or function is a derived subprogram. The specification of the derived subprogram is obtained by copying that of the corresponding subprogram of the parent type, and making the following substitutions:

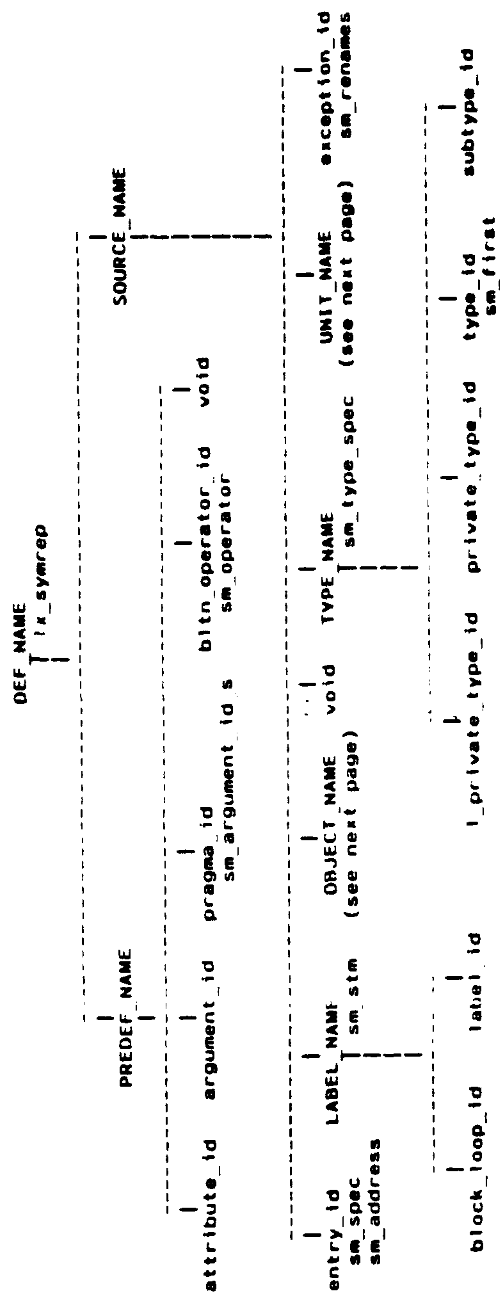
(a) each reference to the parent type is replaced by a reference to the derived type

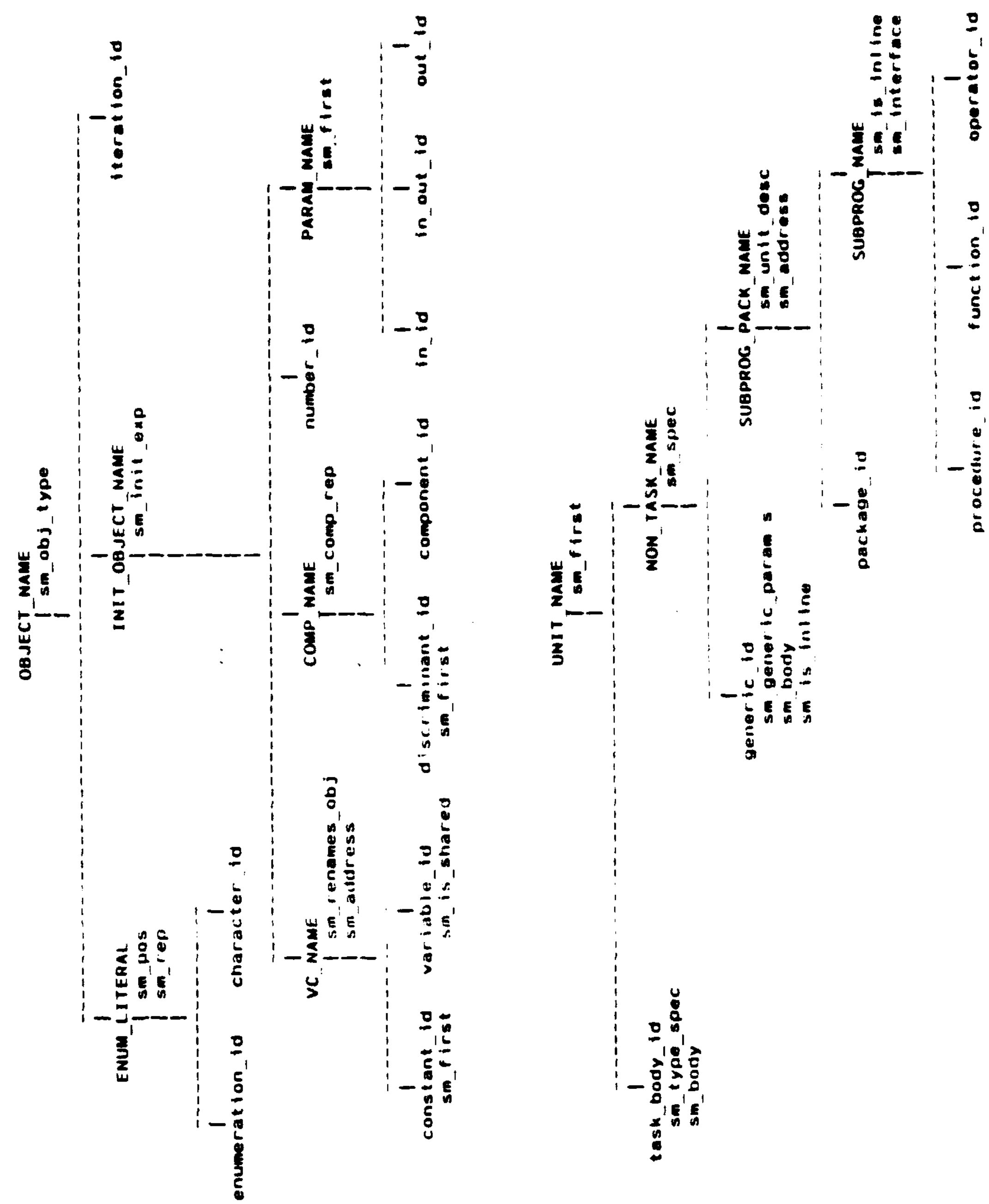
(b) each reference to a subtype of the parent type is replaced by a reference to the derived type

(c) each expression of the parent type is replaced by a type conversion which has the expression as the operand and the derived type as the target type.

The remaining attributes have the following values: sm_address is **void**, sm_interface has the same value as the sm_interface attribute of the corresponding derivable subprogram, and the value of sm_is_inline is determined by the existence of an INLINE pragma for the derived subprogram.

The nodes in class **SUBPROG_NAME** may be introduced by declarative nodes in a normalized parameter list constructed for a generic instantiation, in which case they do not correspond to source code. A more detailed discussion may be found in section 3.6.1.1.


DEF_NAME
lx_symrep
PREDEF_NAME
SOURCE_NAME
attribute_id
argument_id
pragma_id
sm_argument_id_s
bltn_operator_id
sm_operator
void
entry_id
sm_spec
sm_address
LABEL_NAME
sm_stm
OBJECT_NAME
(see next page)
void
TYPE_NAME
sm_type_spec
UNIT_NAME
(see next page)
exception_id
sm_renames
block_loop_id
label_id
l_private_type_id
private_type_id
type_id
sm_first
subtype_id


OBJECT_NAME
sm_obj_type
ENUM_LITERAL
sm_pos
sm_rep
enumeration_id
character_id
INIT_OBJECT_NAME
sm_init_exp
iteration_id
VC_NAME
sm_renames_obj
sm_address
constant_id
sm_first
variable_id
sm_is_shared
COMP_NAME
sm_comp_rep
discriminant_id
sm_first
component_id
number_id
PARAM_NAME
sm_first
in_id
in_out_id
out_id
UNIT_NAME
sm_first
task_body_id
sm_type_spec
sm_body
NON_TASK_NAME
sm_spec
generic_id
sm_generic_param_s
sm_body
sm_is_inline
SUBPROG_PACK_NAME
sm_unit_desc
sm_address
package_id
SUBPROG_NAME
sm_is_inline
sm_interface
procedure_id
function_id
operator_id

3.3 TYPE_SPEC

The classes **TYPE_SPEC** and **TYPE_DEF** are complementary -- the former represents the semantic concept of an Ada type or subtype, the latter represents the syntax of the declaration of an Ada type or subtype. A **TYPE_SPEC** node does not represent source code; it has no lexical or structural attributes, only semantic attributes and code attributes. A **TYPE_DEF** node has no other purpose than to record source code, containing only lexical and structural attributes. A node from class **TYPE_SPEC** will NEVER be designated by a structural attribute, a node from class **TYPE_DEF** node will NEVER be designated by a semantic attribute.

Each distinct type or subtype is represented by a distinct node from class **TYPE_SPEC**; furthermore, there are never two **TYPE_SPEC** nodes for the same entity. Although anonymous type and subtype declarations are not represented in DIANA, the anonymous types and subtypes are themselves represented by nodes from class **TYPE_SPEC**.

The nodes **universal_integer**, **universal_real**, and **universal_float** represent the universal types; they have no attributes.

The **incomplete** node represents a special kind of incomplete type. Ordinarily incomplete types are not represented by **TYPE_SPEC** nodes, all references are to the full type specification. The sole exception occurs when an incomplete type declaration is given in the private part of the package, and the subsequent full type declaration appears in the package body, which is in a separate compilation unit. In this case the incomplete type is represented by an **incomplete** node, and all references denote this node rather than the full type specification. The **incomplete** node defines an sm_discriminant_s attribute which denotes the sequence of discriminant declarations designated by the as_dscrmt_decl_s attribute of the **type_decl** node introducing the incomplete type. This sequence may be empty.

3.3.1 DERIVABLE_SPEC

The class **DERIVABLE_SPEC** consists of nodes representing types which may be derived. The attribute sm_derived which is defined on this class refers to the parent type if the type is derived; otherwise it is **void**. The sm_derived attribute is **void** for a subtype of a derived type. The nodes in this class also have a boolean attribute, sm_is_anonymous, which indicates whether or not the type or subtype has a name.

A derived type is always represented by a new node corresponding to a new base type (the node is the same kind as that of the parent type). If the constraints on the parent type are not identical to those on the parent subtype then the base type is anonymous, and a new node corresponding to a subtype of that type is also created. The node associated with the subtype records the additional constraint (the constraint may be given explicitly in the derived type definition or implicitly by the type mark).

In addition to being created by a derived type definition, a derived type may be introduced by a numeric type definition. The base type of a user-defined numeric type is an anonymous derived type represented by the appropriate **integer**, **float**, or **fixed** node. The sm_derived attribute of the node for the anonymous base type refers to the node corresponding to the appropriate predefined type.

If a derived type is an enumeration type then a sequence of new enumeration literals is created for the derived type, unless the parent type is a generic formal type. The value of sm_pos in each new ENUM_LITERAL node is the same as that in the corresponding node from the parent type; however, the sm_obj_type attribute denotes the **enumeration** node for the derived type. The value of sm_rep depends on whether or not a representation clause is given for the derived type; if not, the value is taken from the corresponding node from the parent type.

If a derived type is a record type and a representation clause is given for that derived type, then a sequence of new discriminants and a sequence of new record components are created for the derived type. If a representation clause is not given for the derived record type then construction of the new sequences is optional. Excluding sm_comp_rep, the values of all of the attributes in each new **COMP_NAME** node will be the same as those in the corresponding node from the parent type; sm_comp_rep will be the same only if no representation clause is given for the derived type.

Certain members of class DERIVABLE_SPEC may represent generic formal types. The attributes of these nodes reflect the properties of the generic formal types, not those of the corresponding actual subtypes. For instance, the sm_size attribute of an **array** node corresponding to a generic formal array type is always **void**, reflecting the fact that a representation clause cannot be given for a generic type. The value of this attribute implies nothing about the value of this attribute in the **array** node of a corresponding actual subtype. The values of attributes having a uniform value when corresponding to generic formal types are discussed in the appropriate sections.

The sm_derived attribute of a **DERIVABLE_SPEC** node representing a generic formal type is always **void**, and sm_is_anonymous is always false.

3.3.1.1 PRIVATE_SPEC

The nodes in class **PRIVATE_SPEC** -- **private** and **l_private** -- represent private and limited private types, respectively. The attributes sm_discriminant_s and sm_type_spec are defined for these nodes. The sm_discriminant_s attribute references the sequence of discriminant declarations introduced by the (limited) private type declaration; hence it may be empty. The sm_type_spec attribute designates the full type specification (a node from class **FULL_TYPE_SPEC**) unless the node corresponds to a generic formal private type, in which case the value of sm_type_spec is undefined.

A subtype or derived type declaration which imposes a new constraint on a (limited) private type results in the creation of a **constrained_record** node if the declaration occurs in the visible part or outside of the package; if the declaration occurs in the private part or the package body then a node from class **CONSTRAINED** or class **SCALAR** is created. The sm_base_type attribute of the new node references the **private** or **l_private** node associated with the type mark.

An attribute of type **TYPE_SPEC** that denotes a (limited) private type always references the **private** or **l_private** node. Access to the associated full type specification is provided by the sm_type_spec attribute of the **PRIVATE_SPEC** node.

3.3.1.2 FULL_TYPE_SPEC

The class **FULL_TYPE_SPEC** represents types which are fully specified. The node task_spec and the class **NON_TASK** comprise **FULL_TYPE_SPEC**.

The **task_spec** node represents a task type. A task type may be anonymous if the reserved word "type" is omitted from the task specification, in which case the **task_spec** node will be introduced by the sm_obj_type attribute of a variable_id rather than the sm_type_spec attribute of a **type_id**.

The **task_spec** node defines five additional semantic attributes: sm_decl_s, sm_body, sm_address, sm_size, and sm_storage_size. The sm_decl_s attribute denotes the sequence of entry declarations and representation clauses designated by the as_decl_s attribute of the associated **task_decl** node. The attribute sm_body denotes the **block_body** node corresponding to the task body if it is in the same compilation unit; if not, sm_body refers to the **stub** node if the stub is in the same compilation unit; if neither the body nor the stub is in the same compilation unit, then sm_body is **void**. Each of the remaining semantic attributes (sm_address, sm_size, and sm_storage_size) denotes the **EXP** node of the corresponding representation clause, if one exists; otherwise it is **void**.

3.3.1.2.1 NON_TASK

Class **NON_TASK** represents fully specified types which are not tasks. The nodes in this class are used to denote both types and subtypes. The attribute sm_base_type which is defined on this class references the base type -- a node containing all of the representation information. The sm_base_type attribute of a **NON_TASK** node representing a generic formal type always contains a self-reference. The classes **SCALAR**, **UNCONSTRAINED**, and **CONSTRAINED** comprise **NON_TASK**.

3.3.1.2.1.1 SCALAR

The nodes in class **SCALAR** represent scalar types and subtypes. A scalar subtype is denoted by the same kind of node as the type from which it is constructed (unless it is constructed from a private type); however, a type may always be distinguished from a subtype by the fact that the sm_base_type attribute of a node corresponding to a type references itself.

The **SCALAR** class has an sm_range attribute which references a node corresponding to the applicable range constraint. In most cases this node already exists (the source code has supplied a constraint, or the range from the appropriate predefined type is applicable); however, in certain instances a new **range** node must be constructed.

A new **range** node is created for an **enumeration** node introduced by either an enumeration type definition or a derived type definition which does not impose a constraint. The as_exp1 and as_exp2 attributes of the **range** node denote **USED_OBJECT** nodes corresponding to the first and last values of the enumeration type. A new **RANGE** node is also created when more than one object is declared in an object declaration containing an anonymous subtype with a non-static range constraint. The subtypes of the objects do not share the same **RANGE** node in this case; a new copy of the **RANGE** node is made for the new subtype of each additional object in the declaration (if the constraint is static, the copy is optional).

The attribute cd_impl_size which is defined on this class contains the universal integer value of the Ada attribute SIZE; it may be less than a user-defined size.

The nodes in class **SCALAR** may also represent generic formal scalar types. The **enumeration** node represents a formal discrete type; the **integer** node a formal integer type; the **float** node a formal floating point type; and the **fixed** node a formal fixed point type. The sm_range attribute for a generic formal scalar type is undefined.

The node **enumeration** represents an enumeration type. If the type is not a generic formal type then the sm_literal_s attribute references the sequence of enumeration literals -- either the sequence denoted by the as_enum_literal_s attribute of the **enumeration_def** node or a new sequence of literals created for a derived type. If the **enumeration** node represents a generic formal type then sm_literal_s denotes an empty sequence.

The **integer** node represents an integer type; it defines no attributes of its own.

3.3.1.2.1.1.1 REAL

The nodes in class REAL -- **float** and **fixed** -- represent floating point types and fixed point types, respectively. If the type is not a generic formal type the sm_accuracy attribute contains the value of the accuracy definition: digits for the **float** node, and delta for the **fixed** node. The value of sm_accuracy for a generic formal type is undefined. The **fixed** node defines an additional attribute, cd_impl_small, which has the value of the Ada attribute SMALL.

3.3.1.2.1.2 UNCONSTRAINED

An unconstrained array, record, or access type is represented by a node from class **UNCONSTRAINED**. The sm_base_type attribute of an **array**, **record**, or **access** node always contains a self-reference. The sm_size attribute which is defined for this class references the **EXP** node given in a length clause for that type; if no such clause is given then sm_size is **void**.

The **access** node represents an unconstrained access type. An access type is unconstrained if its designated type is an unconstrained array type, an unconstrained record type, a discriminated private type, or an access type having a designated type which is one of the above; otherwise, it is constrained. A derived access type is unconstrained if its parent subtype is unconstrained and the derived type definition does not contain an explicit constraint.

The sm_desig_type attribute denotes the **TYPE_SPEC** node corresponding to the designated type -- an **incomplete** node, or a node from class **UNCONSTRAINED** or class **PRIVATE_SPEC** (if sm_desig_type denotes an **access** node, then the sm_desig_type attribute of that **access** node cannot refer to another **access** node). The **TYPE_SPEC** node referenced by the sm_desig_type attribute of an **access** node is never anonymous.

The **access** node also defines the attributes sm_storage_size, sm_is_controlled, and sm_master. The sm_storage_size attribute denotes the **EXP** node given in a length clause if one is applicable, otherwise it is void. The attribute sm_is_controlled is of type Boolean, and indicates whether or not a CONTROLLED pragma is in effect for that type.

The attribute sm_master is defined only for those access types having a task as a designated subtype. In those cases it references the master which contains the corresponding access type definition. If the master is a program unit then sm_master denotes the declaration of the unit -- a **task_decl**, **subprog_entry_decl**, or **package_decl** node. If the master is a block then sm_master denotes a **block_master** node, which contains a reference to the block statement containing the access type definition.

The **array** and **access** nodes may represent generic formal types, in which case the sm_size attribute is **void**, sm_storage_size is **void**, sm_is_controlled is false, and sm_is_packed is false.

3.3.1.2.1.2.1 UNCONSTRAINED_COMPOSITE

The class **UNCONSTRAINED_COMPOSITE** represents unconstrained composite types; it is composed of the nodes **array** and **record**. Two Boolean attributes are defined on this class: sm_is_limited and sm_is_packed. The attribute sm_is_limited indicates whether or not the type has any subcomponents which are of a limited type; sm_is_packed records the presence or absence of a PACK pragma for that type.

The **array** node defines two attributes of its own: sm_index_s and sm_comp_type. The sm_index_s sequence represents the index subtypes (undefined ranges) of the array. The attribute sm_comp_type references a **TYPE_SPEC** node corresponding to the component subtype; if the subtype indication representing the component subtype imposes a new constraint then this **TYPE_SPEC** node is an anonymous subtype.

The node **record** defines the attributes sm_discriminant_s, sm_comp_list, and sm_representation. The sm_discriminant_s attribute denotes the sequence of discriminant declarations referenced by the as_dscrmt_decl_s attribute of the **type_decl** node introducing the record type; this sequence may be empty. The sm_comp_list attribute represents the component list, and the attribute sm_representation designates the representation clause for that record type; if none is applicable then sm_representation is **void**.

3.3.1.2.1.3 CONSTRAINED

A constrained array, record, or access type is represented by a node from class **CONSTRAINED**. The class **CONSTRAINED** defines the boolean attribute sm_depends_on_dscrmt, which is true for a record component subtype which depends on a discriminant, and false in all other cases. The sm_derived attribute for a **constrained_array** or **constrained_record** node is always **void**.

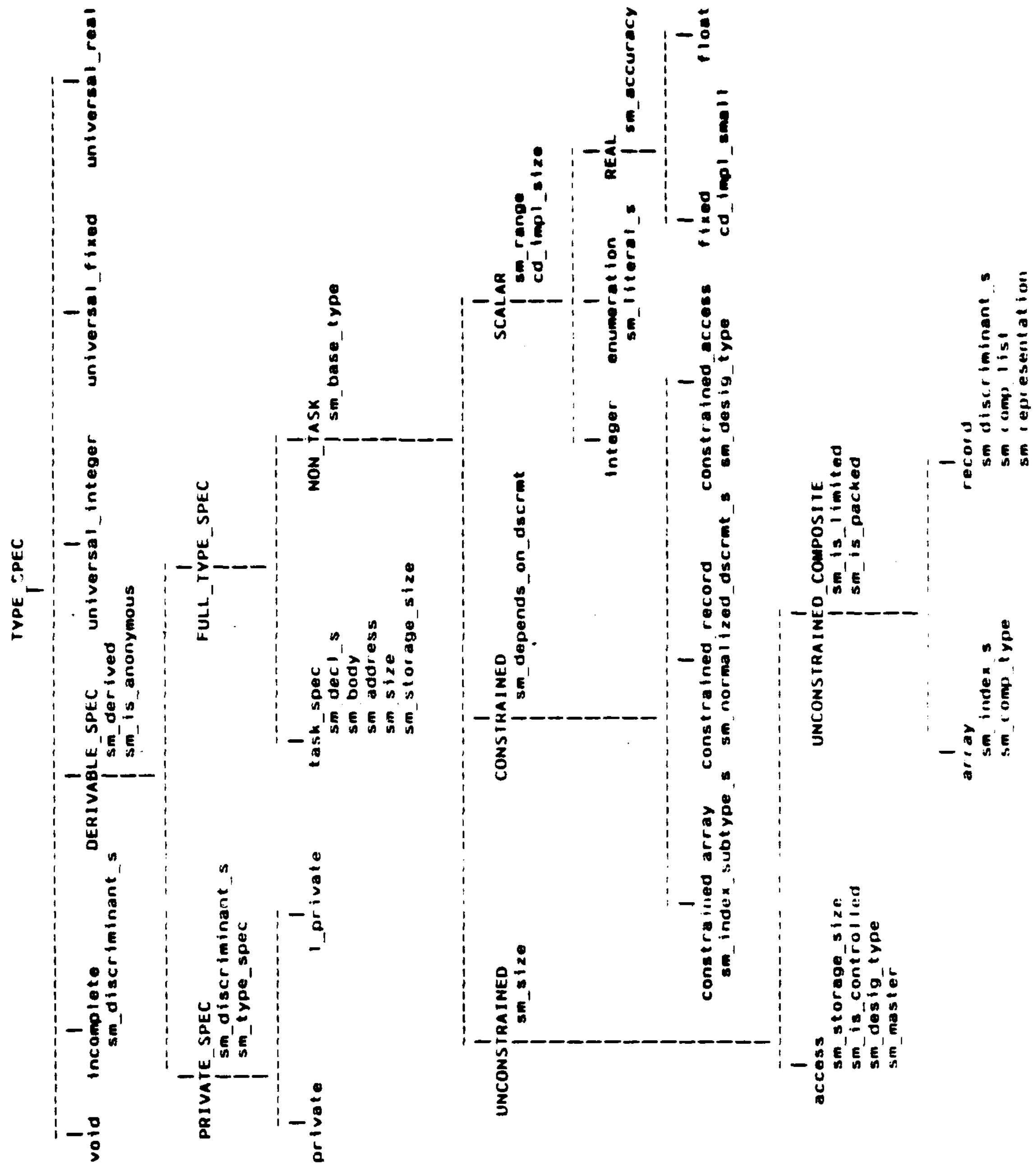
The **constrained_array** node defines an sm_index_subtype_s attribute which denotes a sequence that does not correspond to source code. This sequence is a semantic representation of the index constraint, and is derived from the as_discrete_range_s sequence of the **index_constraint** node. The sm_index_subtype_s sequence consists of **integer** and/or **enumeration** nodes, some of which may be created solely for this sequence. If a particular discrete range is given by a type mark then a new node is not created to represent that discrete range, the **enumeration** or **integer** node associated with the type mark is used. Otherwise, a new **enumeration** or **integer** node is created to represent the new anonymous index subtype.

The sm_base_type attribute of a **constrained_array** node always denotes an **array** node. If the type is introduced by a constrained array definition then an anoymous base type is created; i.e. the sm_type_spec attribute of the **type_id** node or the sm_obj_type attribute of the **VC_NAME** node denotes a **constrained_array** node which has an anonymous **array** node as its base type. If the constrained array definition is part of an object declaration then the **constrained_array** node will be anonymous as well. The **array** node representing the base type does not correspond to source code; its sm_index_s attribute is a sequence of undefined ranges which also are not derived from source code. The **array** node incorporates the information in the constrained array type definition pertaining to the component subtype, and the **constrained_array** node retains the constraint information.

A **constrained_record** node has an sm_normalized_dscrmt_s attribute which is a normalized sequence of the expressions given in the discriminant constraint. No new nodes must be created in order to construct this sequence. The sm_base_type attribute of a **constrained_record** node may denote a node of type **record**, **private**, of **l_private**.

The **constrained_access** node represents a constrained access type or subtype. Its sm_desig_type attribute denotes the designated subtype. If the **constrained_access** node is introduced by either a type declaration in which the subtype indication contains an *explicit constraint*, or a *subtype declaration* that imposes a new constraint, then the designated subtype is a new anonymous subtype. The sm_base_type attribute of a **constrained_access** node references an **access**, **private**, or **l_private** node.

The **constrained_array** and **constrained_access** nodes may represent generic formal types, in which case the sm_depends_on_dscrmt attribute is false.


TYPE_SPEC
void
incomplete
sm_discriminant_s
DERIVABLE_SPEC
sm_derived
sm_is_anonymous
universal_integer
universal_fixed
universal_real
PRIVATE_SPEC
sm_discriminant_s
sm_type_spec
FULL_TYPE_SPEC
private
l_private
task_spec
sm_decl_s
sm_body
sm_address
sm_size
sm_storage_size
NON_TASK
sm_base_type
UNCONSTRAINED
sm_size
CONSTRAINED
sm_depends_on_dscrmt
SCALAR
sm_range
cd_impl_size
integer
enumeration
sm_literal_s
REAL
sm_accuracy
fixed
cd_impl_small
float
constrained_array
sm_index_subtype_s
constrained_record
sm_normalized_dscrmt_s
constrained_access
sm_desig_type
access
sm_storage_size
sm_is_controlled
sm_desig_type
sm_master
UNCONSTRAINED_COMPOSITE
sm_is_limited
sm_is_packed
array
sm_index_s
sm_comp_type
record
sm_discriminant_s
sm_comp_list
sm_representation

Section 3.4

TYPE_DEF

3.4 TYPE_DEF

The nodes in class **TYPE_DEF** represent the following constructs in the source code:

(a) a subtype indication

(b) the portion of a type declaration following the reserved word "is"

(c) the subtype indication or constrained array definition in an object declaration

With the exception of the nodes **constrained_array_def** and **subtype_indication**, the nodes in this class may be designated only by the as_type_def attribute of the **type_decl** node.

This class contains numerous nodes which do not define attributes of their own, their purpose being to differentiate the various kinds of type definitions. The nodes **private_def** and **l_private_def** correspond to private and limited private type definitions, respectively. The nodes **formal_dscrt_def**, **formal_integer_def**, **formal_float_def**, and **formal_fixed_def** correspond to generic formal scalar type definitions.

The node **enumeration_def** corresponds to an enumeration type definition; the attribute as_enum_literal_s denotes a sequence corresponding to the enumeration literals given in the definition.

The node **record_def** corresponds to a record type definition; as_comp_list is the component list given in the definition.

3.4.1 CONSTRAINED_DEF

The class **CONSTRAINED_DEF** consists of nodes representing source code containing a constraint, hence the attribute as_constraint is defined on this class.

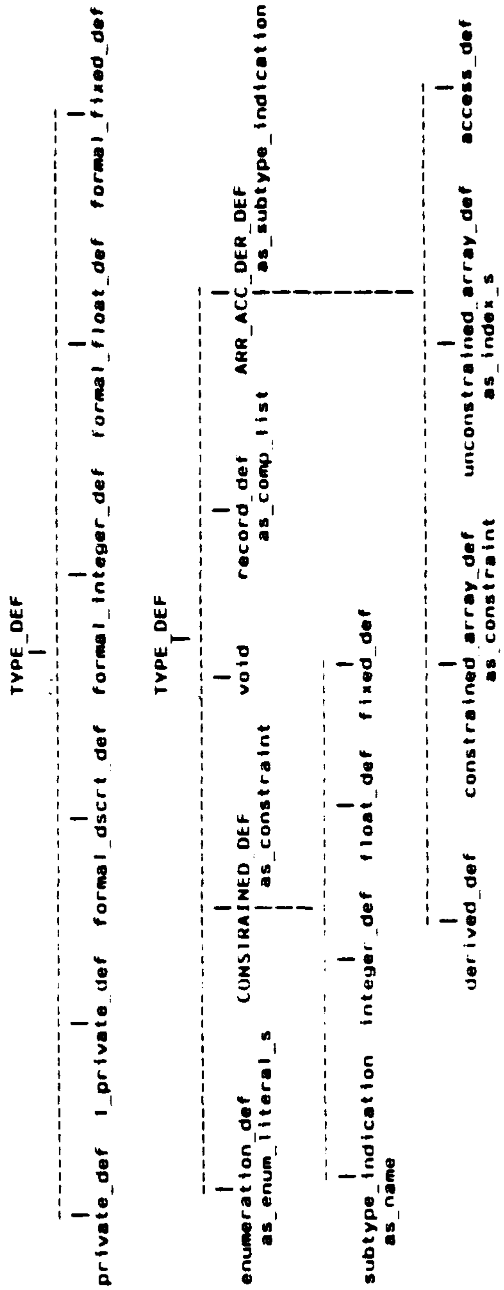
The nodes **integer_def**, **float_def**, and **fixed_def** correspond to numeric type definitions; the as_constraint attribute references a node representing the range constraint, floating point constraint, or fixed point constraint given in the definition.

The **subtype_indication** node records the occurrence of a subtype indication in the source code. It is never designated by the as_type_def attribute of a **type_decl** node; however, it may be referenced by the the as_type_def attribute of an **OBJECT_DECL** node; or by the as_subtype_indication attribute of a **subtype_decl**, **discrete_subtype**, **subtype_allocator**, or **ARR_ACC_DER_DEF** node. The as_constraint attribute denotes the constraint given in the subtype indication (if there is no constraint then this attribute is **void**), and as_name represents the type mark.

3.4.2 ARR_ACC_DER_DEF

The class **ARR_ACC_DER_DEF** is composed of those nodes associated with type definitions containing a subtype indication; in particular, array type definitions, access type definitions, and derived type definitions. For an array definition the as_subtype_indication attribute denotes a node corresponding to the component subtype; for an access type definition as_subtype_indication is the designated subtype; for a derived type definition it is the parent subtype.

The nodes corresponding to array definitions each have an additional attribute. The **unconstrained_array_def** node has an as_index_s attribute which denotes a sequence representing the undefined ranges given in the unconstrained array definition. A constrained array definition is represented by the **constrained_array_def** node, which has an as_constraint attribute corresponding to the sequence of discrete ranges given in the definition (an **index_constraint** node).


TYPE_DEF
private_def
l_private_def
formal_dscrt_def
formal_integer_def
formal_float_def
formal_fixed_def
TYPE_DEF
enumeration_def
as_enum_literal_s
CONSTRAINED_DEF
as_constraint
void
record_def
as_comp_list
ARR_ACC_DER_DEF
as_subtype_indication
subtype_indication
as_name
integer_def
float_def
fixed_def
derived_def
constrained_array_def
as_constraint
unconstrained_array_def
as_index_s
access_def

Section 3.5

CONSTRAINT

3.5 CONSTRAINT

The members of class **CONSTRAINT** represent discrete ranges and the various kinds of constraints defined by the Ada programming language (this class is the union of the Ada syntactic categories "discrete_range" and "constraint"). This class consists of the nodes **index_constraint** and **dscrmt_constraint**, as well as the classes **DISCRETE_RANGE** and **REAL_CONSTRAINT**.

The node **index_constraint** represents an array index constraint. The attribute as_discrete_range_s denotes a sequence of nodes representing the discrete ranges.

A discriminant constraint is represented by a **dscrmt_constraint** node. The as_general_assoc_s attribute corresponds to the sequence of discriminant associations (a sequence of nodes of type **named** and/or **EXP**).

3.5.1 DISCRETE_RANGE

The class **DISCRETE_RANGE** contains the node **discrete_subtype** and the class **RANGE**.

A discrete subtype indication is represented by a **discrete_subtype** node. The as_subtype_indication attribute references a node representing the subtype indication itself.

3.5.1.1 RANGE

The nodes which comprise class **RANGE** -- **range**, **range_attribute**, and **void** -- represent ranges and range constraints. The context determines whether a node belonging to class **RANGE** represents a range or a range constraint. If the node is introduced by an as_constraint attribute then it represents a range constraint; otherwise it is simply a range.

The context also determines the value of the sm_type_spec attribute. For a **RANGE** node introduced by a subtype indication sm_type_spec refers to the **SCALAR** node associated with the type mark. If the **RANGE** node is introduced by a type definition or a derived type definition creating a new scalar type then sm_type_spec denotes the specification of the new base type. Otherwise sm_type_spec designates the node corresponding to the appropriate base type, as specified by the Ada Reference Manual. For instance, sm_base_type of a **RANGE** node corresponding to a slice denotes the specification of the index type.

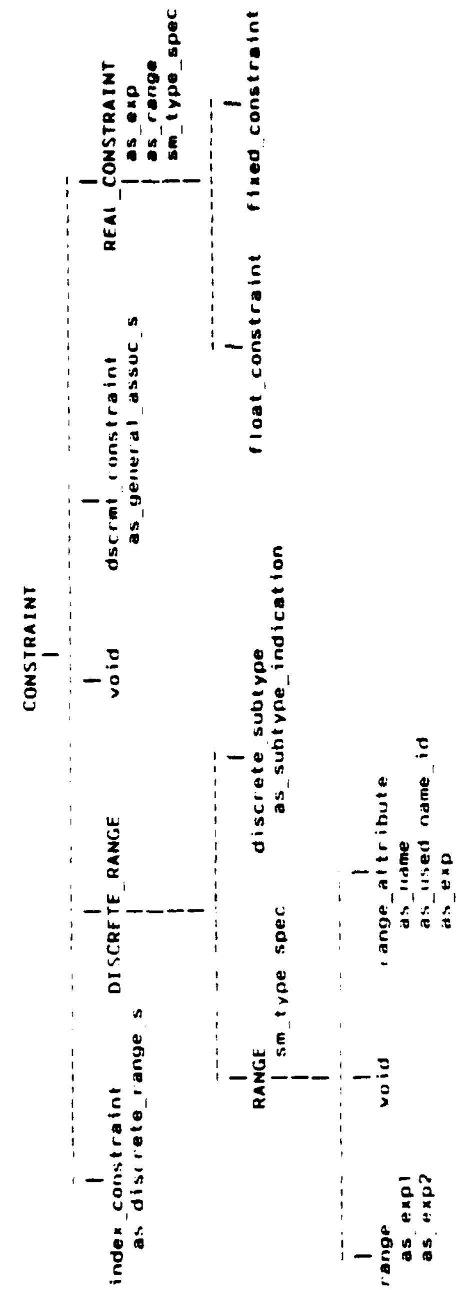
The **range** node corresponds to a range given by two simple expressions, which are denoted by the attributes as_exp1 (the lower bound) and as_exp2 (the upper bound).

The **range_attribute** node represents a range attribute. The as_name attribute references the **NAME** node corresponding to the prefix, the attribute as_used_name_id designates the **attribute_id** node for **RANGE**, and as_exp denotes the argument specifying the desired dimension (if no argument is given then as_exp is **void**).

3.5.2 REAL_CONSTRAINT

The class **REAL_CONSTRAINT** contains the nodes **float_constraint** and **fixed_constraint**, representing floating point constraints and fixed point constraints, respectively. This class defines two structural attributes: as_exp and as_range. The as_exp attribute references the node representing the simple static expression for digits or delta. The attribute as_range denotes the range given in the constraint; it may be **void** for floating point constraints and for fixed point constraints which do not correspond to fixed point type definitions.

The nodes belonging to **REAL_CONSTRAINT** also have an sm_type_spec attribute. If the **REAL_CONSTRAINT** node corresponds to a subtype indication then sm_type_spec of the **REAL_CONSTRAINT** node and the corresponding **RANGE** node (if there is one) denotes the type specification associated with the type mark. If the constraint is introduced by a real type definition or a derived type definition then sm_type_spec of the **REAL_CONSTRAINT** node and the **RANGE** node (if there is one) references the type specification of the new base type.


CONSTRAINT
index_constraint
as_discrete_range_s
DISCRETE_RANGE
void
dscrmt_constraint
as_general_assoc_s
REAL_CONSTRAINT
as_exp
as_range
sm_type_spec
float_constraint
fixed_constraint
RANGE
sm_type_spec
discrete_subtype
as_subtype_indication
range
as_exp1
as_exp2
void
range_attribute
as_name
as_used_name_id
as_exp

3.6.1.1 RENAME_INSTANT

The nodes in class **RENAME_INSTANT** indicate that a subprogram or a package has been renamed or instantiated. The meaning of the as_name attribute which is defined on this class depends on whether the node is a **renames_unit** node or an **instantiation** node.

The node **renames_unit** represents the renaming of an entity as a subprogram or a package. The attribute as_name denotes the name of the original entity as given in the renaming declaration. The valid values of as_name are determined by the kind of entity being renamed; they are as follows:

(a) package - selected or **used_name_id**

(b) procedure - selected or **used_name_id**

(c) function - selected or **used_name_id**

(d) operator - selected or **used_op**

(e) entry - selected or **used_name_id** or **indexed**

(f) enumeration literal - selected or **used_char** or **used_object_id**

(g) attribute - **attribute**

The **instantiation** node signifies the instantiation of a generic subprogram or package. The as_name attribute designates a **used_name_id** or **selected** node corresponding to the name of the generic unit, and the as_general_assoc_s attribute denotes a possibly empty sequence of parameter associations (nodes of type EXP and assoc). The sm_decl_s attribute of the **instantiation** node is a normalized list of the generic parameters, including entries for all default parameters.

Declarative nodes are used to represent the actual parameters in the sm_decl_s sequence. Each parameter has its own declarative node, and each declarative node introduces a new SOURCE_NAME node. The lx_symrep attribute of each SOURCE_NAME node contains the symbol representation of the generic formal parameter; however, the values of the semantic attributes are determined by the actual parameter. None of the new nodes created during the process of constructing the sm_decl_s sequence represent source code.

The declarations are constructed as follows:

(a) For every generic formal in parameter, a constant declaration is created. The as_source_name_s sequence of the constant_decl node contains a single **constant_id** node. The as_type_def attribute is undefined, and the as_exp attribute designates either the actual expression or the default expression of the generic parameter declaration.

The sm_first attribute of the **constant_id** node contains a self-reference (it does not refer to the **in_id** of the generic formal object declaration), sm_renames_obj is false, and sm_obj_type denotes the **TYPE_SPEC** node of the actual parameter (or default expression). The attribute sm_init_exp designates the same node as the as_exp attribute of the **constant_decl** node.

(b) For every generic formal in out parameter, a renaming declaration is created. The as_source_name attribute of the **renames_obj_decl** node denotes a new **variable_id** node, and the as_type_mark_name attribute is undefined. The as_name attribute designates the name of the actual parameter as given in the generic actual part.

The attribute values of the **variable_id** are determined exactly as if the declaration were a genuine renaming of the actual parameter as the formal parameter (see section 3.2.2.3.2.1).

(c) For every generic formal type a subtype declaration is created. The as_source_name attribute of the **subtype_decl** node designates a new **subtype_id** node which has an sm_type_spec attribute denoting the **TYPE_SPEC** node associated with the actual subtype. The **subtype_indication** node designated by the as_subtype_indication attribute has a **void** as_constraint attribute and an as_name attribute which represents the type mark of the actual subtype.

(d) For every generic formal subprogram, a new subprogram declaration is created. The **subprog_entry_decl** node is a renaming declaration, therefore the as_unit_kind attribute denotes a **renames_unit** node which references either the actual parameter or the appropriate default. The as_header attribute denotes the **HEADER** node of the generic actual parameter.

The as_source_name attribute designates a new **SUBPROG_NAME** or **ENUM_LITERAL** node, depending on the actual (or default) parameter. The kind of node and the values of its attributes (except for sm_spec) are determined precisely as if the declaration were an explicit renaming of the actual entity as the formal subprogram (see sections 3.1.1.3.3.2.1 and 3.2.2.4.1.1.1). The sm_spec attribute denotes the header of the actual parameter rather than that of the generic formal parameter declaration.

Once the normalized declaration list is constructed the specification part of the generic unit is copied; however, every reference to a formal parameter in the original generic specification is changed to a reference to the corresponding newly created declaration. In addition, all references to the discriminants of a formal type are changed to denote the corresponding discriminants of the newly created subtype (i.e. the discriminants of the actual type). All references to the formal parameters of a formal subprogram are changed to denote the corresponding parameters of the newly created subprogram (i.e. the formal parameters of the actual subprogram). The value of the as_name attribute of a **DSCRMT_PARAM_DECL** node is undefined in this copy of the specification, as is the value of the as_type_def attribute of an **OBJECT_DECL** node.

The sm_spec attribute of the **procedure_id**, **function_id**, or **package_id** corresponding to the instantiated unit designates this new specification.

3.6.1.2 GENERIC_PARAM

The nodes in class **GENERIC_PARAM** are used to indicate that a subprogram is a generic formal parameter. The nodes **name_default**, **box_default**, and **no_default** comprise **GENERIC_PARAM**.

The **name_default** node signifies that a generic formal subprogram has an explicitly given default. The as_name attribute represents the name of the default as given -- a node from class **DESIGNATOR** or an **indexed** node.

The node **box_default** indicates that a box rather than a name is given for the default; it defines no attributes of its own.

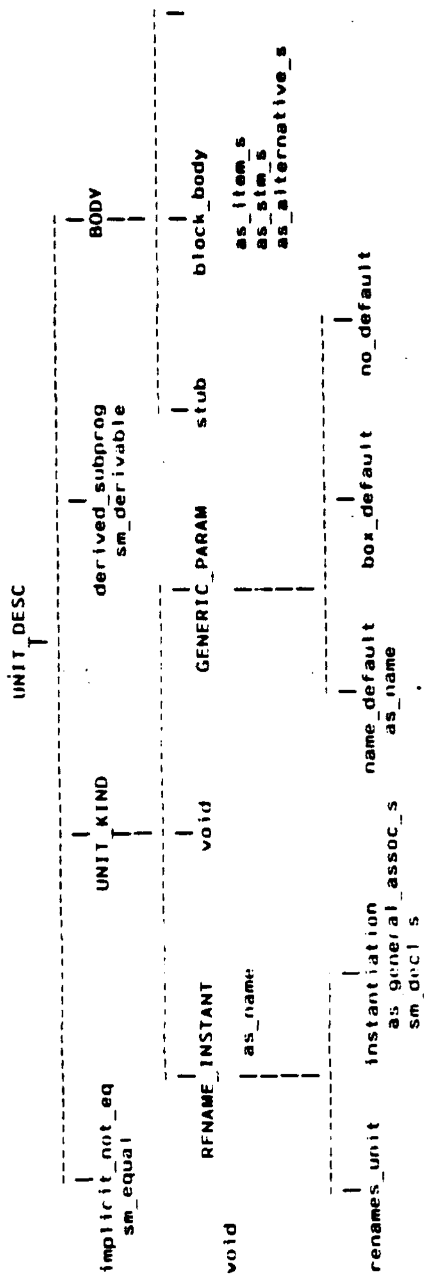
The **no_default** node records the fact that no default is specified; it defines no attributes of its own.

3.6.2 BODY

The class **BODY** represent unit bodies; it contains the nodes **stub**, **block_body**, and **void**.

The **stub** node corresponds to a body stub; it defines no attributes of its own.

The **block_body** node represents the contents of either a proper body or a block statement. It has three structural attributes -- as_item_s, as_stm_s, and as_alternative_s -- corresponding to the declarative part, the sequence of statements, and the exception handlers, respectively.


UNIT_DESC
implicit_not_eq
sm_equal
UNIT_KIND
derived_subprog
sm_derivable
BODY
void
RENAME_INSTANT
as_name
void
GENERIC_PARAM
stub
block_body
as_item_s
as_stm_s
as_alternative_s
renames_unit
instantiation
as_general_assoc_s
sm_decl_s
name_default
as_name
box_default
no_default

Section 3.7

HEADER

3.7 HEADER

The nodes in class **HEADER** contain all of the information given in the specification of a subprogram, entry, or package except for the name of the entity. **HEADER** contains the node **package_spec** and the class **SUBP_ENTRY_HEADER**.

A **HEADER** node corresponding to either the renaming of a package or an instantiation will contain no information; i.e. any sequence attributes will denote empty sequences and any class-valued attributes will be **void**.

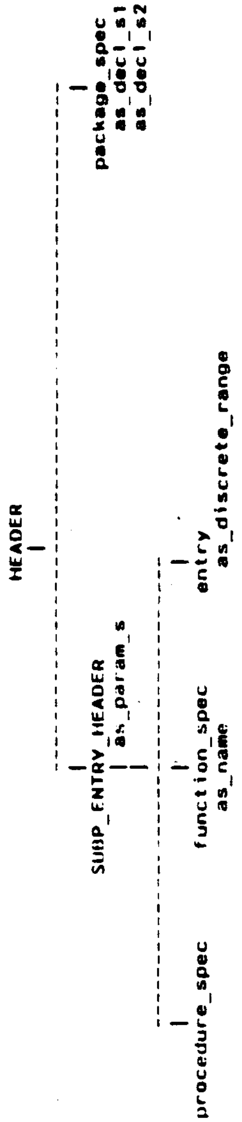
The node **package_spec** represents the declarative parts of a package specification. It has two semantic attributes -- as_decl_s1 and as_decl_s2 -- corresponding to the visible and private parts of the specification, respectively. Either or both of these sequences may be empty.

3.7.1 SUBP_ENTRY_HEADER

The nodes in class **SUBP_ENTRY_HEADER** record the information given in the formal part of a subprogram or entry declaration. This class defines an attribute as_param_s which denotes a possibly empty sequence of parameter specifications. The nodes **procedure_spec**, **function_spec**, and **entry** comprise **SUBP_ENTRY_HEADER**.

The node **function_spec** has an additional attribute, as_name, representing the type mark given in the function specification. If the **function_spec** corresponds to the instantiation of a generic function then as_name is **void**; otherwise it designates a **used_name_id** or a **selected** node.

The **entry** node has the attribute as_discrete_range, denoting the discrete range given in the entry declaration. If the declaration introduces a single entry rather than an entry family then as_discrete_range is **void**.


HEADER
package_spec
as_decl_s1
as_decl_s2
SUBP_ENTRY_HEADER
as_param_s
procedure_spec
function_spec
as_name
entry
as_discrete_range

3.8 GENERAL_ASSOC

The class **GENERAL_ASSOC** represents the following kinds of associations:

(a) parameter

(b) argument

(c) generic

(d) component

(e) discriminant

The classes **NAMED_ASSOC** and **EXP** comprise **GENERAL_ASSOC**. If the association is given in named form then it is represented by a node from class **NAMED_ASSOC**; otherwise it is denoted by a node from class **EXP**.

3.8.1 NAMED_ASSOC

The **NAMED_ASSOC** class contains two nodes -- **named** and **assoc**. It defines an attribute as_exp which records the expression given in the association.

The **assoc** node corresponds to associations which contain a single name; i.e. parameter, argument, and generic associations. The as_used_name attribute represents the argument identifier or (generic) formal parameter given in the association.

The node **named** represents associations that may contain more than one choice -- component associations (of an aggregate) and discriminant associations (of a discriminant constraint). It defines an as_choice_s attribute which references a sequence of nodes representing the choices or discriminant names given in the association. The simple names of components or discriminants that occur within associations are represented by **used_name_id** nodes rather than **used_object_id** nodes.

3.8.2 EXP

The **EXP** class represents names and expressions; its three components are **NAME**, **EXP**, and **void**.

Certain names and expressions may introduce anonymous subtypes; i.e. slices, aggregates, string literals, and allocators. The anonymous subtype is represented by a **constrained_array** or a **constrained_record** node, and is designated by the sm_exp_type attribute of the expression introducing it. Anonymous index subtypes (for an anonymous array subtype) are introduced by discrete ranges which are not given by type marks. Subsequent sections will discuss in further detail the circumstances which produce an anonymous subtype, as well as the representation of the subtype.

3.8.2.1 NAME

The class **NAME** represents used occurrences of names; it contains the classes **DESIGNATOR** and **NAME_EXP**, and the node **void**.

3.8.2.1.1 DESIGNATOR

The nodes in class **DESIGNATOR** correspond to used occurrences of simple names, character literals, and operator symbols. DIANA does not require that each used occurrence of an identifier or symbol be represented by a distinct node (although it does allow such a representation); hence it is possible for a single instance of a node corresponding to a used occurrence to represent all of the logical occurrences of the associated identifier. Used occurrences of named numbers which occur in certain contexts are an exception to the previous statement; see section 3.8.2.1.1.1.

DESIGNATOR consists of the classes **USED_OBJECT** and **USED_NAME**, and defines the attributes sm_defn and lx_symrep. The sm_defn attribute references the **DEF_NAME** node corresponding to the defining occurrence of the entity (if the entity is predefined the **DEF_NAME** node is not accessable through structural attributes). The lx_symrep attribute is the string representation of the name of the entity.

3.8.2.1.1.1 USED_OBJECT

The class **USED_OBJECT** represents appearances of enumeration literals, objects, and named numbers. The sm_defn attribute of a node from this class denotes a node from class **OBJECT_NAME**.

USED_OBJECT defines the attributes sm_exp_type and sm_value. The sm_exp_type attribute denotes the subtype of the entity; i.e. the node designated by the sm_obj_type attribute of the defining occurrence of the entity. The sm_value attribute records the static value of a constant scalar object; if the entity does not satisfy these conditions then sm_value has a distinguished value indicating that it is not evaluated.

The nodes **used_char** and **used_object_id** constitute this class; together they represent the used occurrences of all the entities having defining occurrences belonging to class **OBJECT_NAME**. The **used_char** node represents a used occurrence of a character literal; a **used_object_id** node represents the use of an object, an enumeration literal denoted by an identifier, or a named number. The sm_defn attribute of a **used_char** node references a **character_id**, the sm_defn attribute of a **used_object_id** may designate any node from class **OBJECT_NAME** except for a **character_id**.

Although the names of objects most often occur in expressions, the names of certain objects -- those of record components (including discriminants) and parameters -- may also occur on the left-hand side of named associations; these instances are represented by **used_name_id** nodes rather than **used_object_id** nodes.

The use of the new name of an enumeration literal renamed as a function is represented by a **used_char** or **used_object_id** node rather than a **function_call** node.

If a **used_object_id** corresponds to a named number, and the use represented by the **used_object_id** occurs in a context requiring an implicit type conversion of the named number, then the sm_exp_type attribute of the **used_object_id** denotes the target type rather than a universal type. This means that it is not always possible for a single used occurrence of a named number to represent all used occurrences of that named number; however, a single used occurrence having a particular target type CAN represent all used occurrences of that named number requiring that particular target type.

3.8.2.1.1.2 USED_NAME

The class **USED_NAME** represents used occurrences of identifiers or symbols corresponding to entities which do not have a value and a type. It contains the node **used_op** and **used_name_id**.

The node **used_op** represents the use of an operator symbol, hence its sm_defn attribute denotes either an operator_id or a **bltn_operator_id**.

A **used_name_id** node represents a use of the name of any of the remaining kinds of entities. It may also record the occurrence of the simple name of a discriminant, a component, or a parameter on the left-hand side of a named association (however, it does not denote a used occurrence of such an object in any other context). Excluding this special case, sm_defn may reference any member of class **DEF_NAME** except for an **operator_id, a bltn_operator_id**, or a member of class **OBJECT_NAME**.

3.8.2.1.2 NAME_EXP

The nodes in class **NAME_EXP** represent names which are not simple identifiers or character symbols; i.e. function calls and names having a prefix. The attributes as_name and sm_exp_type are defined for the nodes in this class. The as_name attribute represents either the name of the function or the prefix.

If the **NAME_EXP** node corresponds to an expression then sm_exp_type corresponds to the subtype of the entity, otherwise it is **void**. The only **NAME_EXP** nodes which can possibly have a **void** sm_exp_type attribute are the **indexed, attribute,** and **selected** nodes.

The node **all** represents a dereferencing; i.e. a selected component formed with the selector "all". The as_name attribute corresponds to the access object, and sm_exp_type is the designated subtype.

The **indexed** node represents either an indexed component or a reference to a member of an entry family. For an indexed component the as_exp_s attribute denotes a sequence of index expressions, as_name is the array prefix, and sm_exp_type is the component subtype. The as_exp_s attribute of an entry family member is a one-element sequence containing the entry index; as_name is the entry name, and sm_exp_type is **void**.

A slice is represented by a **slice** node. The as_name attribute denotes the array prefix and the as_discrete_range attribute is the discrete range.

The sm_exp_type attribute denotes the subtype of the slice. The subtype of a slice is anonymous unless it can be determined statically that the bounds of the slice are identical to the bounds of the array prefix, in which case the sm_exp_type attribute of the **slice** node is permitted to reference the **constrained_array** node associated with the array prefix. Otherwise, an anonymous subtype is created for the **slice** node. The anonymous subtype is represented by a **constrained_array** node having the same base type as that of the array prefix; however, the constraint is taken from the discrete range given in the slice.

3.8.2.1.2.1 NAME_VAL

The class **NAME_VAL** contains **NAME_EXP** nodes which may have a static value, consequently the sm_value attribute is defined for the nodes in this class. If the value is not static, sm_value has a distinguished value indicating that the expression is not evaluated. **NAME_VAL** comprises the nodes **attribute**, **selected**, and **function_call**.

The node **attribute** corresponds to an Ada attribute other than a RANGE attribute (which is represented by a **range_attribute** node). The DIANA attribute as_name denotes the prefix, as_used_name references the attribute_id corresponding to the given attribute name, and as_exp is the universal static expression. If no universal expression is present then as_exp is void.

The value of the sm_exp_type attribute of an **attribute** node depends on the kind of Ada attribute it represents, as well as the context in which it occurs. If the **attribute** node represents the BASE attribute then sm_exp_type is **void**. If the Ada attribute returns a value of a universal type, and that value is the object of an implicit type conversion (determined by the context), then sm_exp_type references the target type. Otherwise sm_exp_type denotes the TYPE_SPEC node corresponding to the type of the attribute as specified in the Ada Reference Manual.

The node **selected** represents a selected component formed with any selector other than the reserved word "all" (this includes an expanded name). The as_name attribute denotes the prefix, and as_designator corresponds to the selector. If the **selected** node represents an object (i.e. an entity having a value and a type, for instance a record component) then sm_exp_type is the subtype of the object; otherwise it is **void**.

All function calls and operators are represented by **function_call** nodes, with the exception of the short circuit operators and the membership operators. The as_name attribute denotes the name of the function or operator -- a **used_name_id**, **used_op**, or **selected** node. The lx_prefix attribute records whether the function call is given using infix or prefix notation. The as_general_assoc_s attribute is a possibly empty sequence of parameter associations (nodes of type EXP and **assoc**); sm_normalized_param_s is a normalized list of actual parameters, including any expressions for default parameters. The sm_exp_type attribute denotes the return type. If the function call corresponds to a predefined operator then sm_exp_type references the appropriate base type, as specified in section 4.5 of the Ada Reference Manual.

Although the use of an enumeration literal is considered to be equivalent to a parameterless function call, it is represented by a **used_char** or **used_object_id** node rather than a **function_call** node (this includes the use of an enumeration literal renamed as a function). However, the use of an attribute renamed as a function is represented by a **function_call** node, not an **attribute** node.

3.8.2.2 EXP_EXP

The class EXP_EXP represents expressions which are not names. The attribute sm_exp_type which is defined on this class denotes the TYPE_SPEC node corresponding to the subtype of the expression. EXP_EXP contains the nodes **qualified_allocator** and **subtype_allocator** as well as the classes AGG_EXP and EXP_VAL.

The nodes **qualified_allocator** and **subtype_allocator** represent the two forms of allocators. Each node has the appropriate structural attribute -- as_qualified or as_subtype_indication -- to retain the information given in the allocator. The sm_exp_type attribute denotes the TYPE_SPEC node corresponding to the subtype of the access value to be returned, as determined from the context. The **subtype_allocator** defines an additional attribute, sm_desig_type, which denotes a TYPE_SPEC node corresponding to the subtype of the object created by the allocator. If the subtype indication contains an explicit constraint then sm_desig_type denotes a new TYPE_SPEC node corresponding to the anonymous subtype of the object created by the allocator.

3.8.2.2.1 AGG_EXP

The **AGG_EXP** class represents aggregates and string literals; it is composed of the nodes **aggregate** and **string_literal**. The **aggregate** node may represent an aggregate or a subaggregate. The **string_literal** node represents a string literal (which may also be a subaggregate if it corresponds to the last dimension of an aggregate corresponding to a multidimensional array of characters).

The class **AGG_EXP** defines an sm_discrete_range attribute to represent the bounds of a subaggregate; sm_discrete_range is **void** for a node representing an aggregate. The sm_exp_type attribute of a node corresponding to an aggregate denotes the subtype of the aggregate; it is **void** for a subaggregate. This implies that in an **aggregate** or **string_literal** node exactly one of these two attributes is **void**.

If sm_exp_type is not **void**, it designates a **constrained_array** or **constrained_record** node corresponding to the subtype. An aggregate or a string literal has an anonymous subtype unless it can be determined statically that the constraints on the aggregate are identical to those of the subtype obtained from the context, in which case sm_exp_type may (but does not have to) reference the node associated with that subtype.

If the aggregate has an anonymous subtype it is constructed from the base type of the context type and the bounds as determined by the rules in the Ada Reference Manual. If the bounds on the subaggregates for a particular dimension of a multidimensional aggregate are not the same (a situation which will result in a CONSTRAINT_ERROR during execution) DIANA does not specify the subaggregate from which the bounds for the index constraint are taken.

The **string_literal** node defines only one additional attribute, lx_symrep, which contains the string itself.

The **aggregate** node has two different representations of the sequence of component associations; both may contain nodes of type **named** and EXP. The as_general_assoc_s attribute denotes the sequence of component associations as given; sm_normalized_comp_s is a sequence of normalized component associations which are not necessarily in the same form as given, for the following reasons:

(a) Each named association having multiple choices is decomposed into separate associations for the sm_normalized_comp_s sequence, one for each choice in the given association; hence the as_choice_s sequence of a **named** node in the normalized list contains only one element. The manner in which this decomposition is done is not specified, the only requirements being that the resulting associations be equivalent, and that each association be either the component expression itself or a named association with only one choice. Consider the array aggregate

(1 | 2 | 3 => 10)

The named association could be broken down in such a way that the sm_normalized_comp_s sequence appeared as if it came from any of the following aggregates:

```
( 1 => 10, 2 => 10, 3 => 10 )
        ( 1..3 => 10 )
        ( 10, 10, 10 )
```

In the process of normalizing the component associations new **named** nodes may be created, and duplication of the component expressions is optional. For the remainder of this section all **named** component associations will be treated as if they had only one choice.

(b) For a record aggregate, if a choice is given by a component name then the component expression rather than the **named** node is inserted in the proper place in the sequence, hence the normalized sequence for a record aggregate is actually a sequence of **EXP** nodes.

(c) In an array aggregate an association containing a choice which is a simple expression may be replaced by the component expression if it can be determined statically that the choice belongs to the appropriate index subtype (this substitution is optional).

(d) A named association with an "others" choice is not allowed in the sm_normalized_comp_s sequence. For each component or range of components denoted by the "others" either a component expression is inserted in the proper spot in the sequence, or a new **named** node is created containing the appropriate range.

Due to some of the changes mentioned above it is possible for the sm_normalized_comp_s sequence of an array aggregate to contain a mixture of **EXP** and **named** nodes.

3.8.2.2.2 EXP_VAL

The **EXP_VAL** class contains nodes representing expressions which may have static values, hence the sm_value attribute is defined for the nodes in this class. If the value is not static then sm_value has a distinguished value which indicates that the expression is not evaluated.

A numeric literal is represented by a **numeric_literal** node. It has an attribute lx_numrep containing the numeric representation of the literal. If the literal is the object of an implicit conversion then sm_exp_type denotes the target type rather than a universal type.

The **null_access** node corresponds to the access value NULL; it defines no attributes of its own. Although a distinct **null_access** node may be created for each occurrence of the access value NULL, DIANA also permits a single **null_access** node to represent all occurrences of the literal NULL for that particular access type.

The node **short_circuit** represents the use of a short circuit operator. The as_short_circuit_op attribute denotes the operator (**and_then** or **or_else**); as_exp1 and as_exp2 represent the expressions to the left and right of the operator, respectively.

3.8.2.2.2.1 EXP_VAL_EXP

The class **EXP_VAL_EXP** defines an as_exp attribute; it comprises the node **parenthesized** and the classes **MEMBERSHIP** and **QUAL_CONV**.

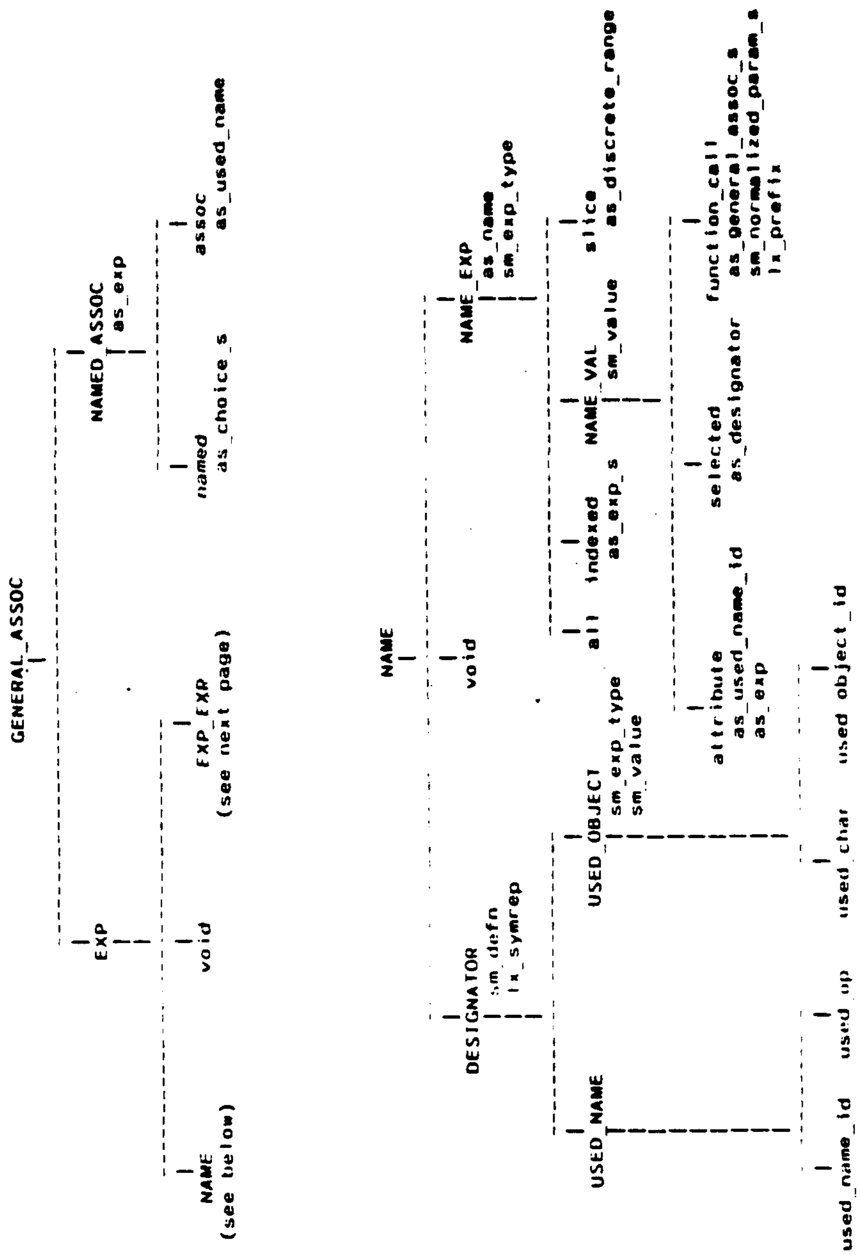
The **parenthesized** node represents a pair of parentheses enclosing an expression. The as_exp attribute denotes the enclosed expression, sm_value is the value of the expression if it is static, and sm_exp_type is the subtype of the expression. A **parenthesized** node can NEVER be denoted by a semantic attribute, nor can it be included directly in a sequence that is constructed exclusively for a semantic attribute (such as a normalized sequence); the node representing the actual expression is referenced instead.

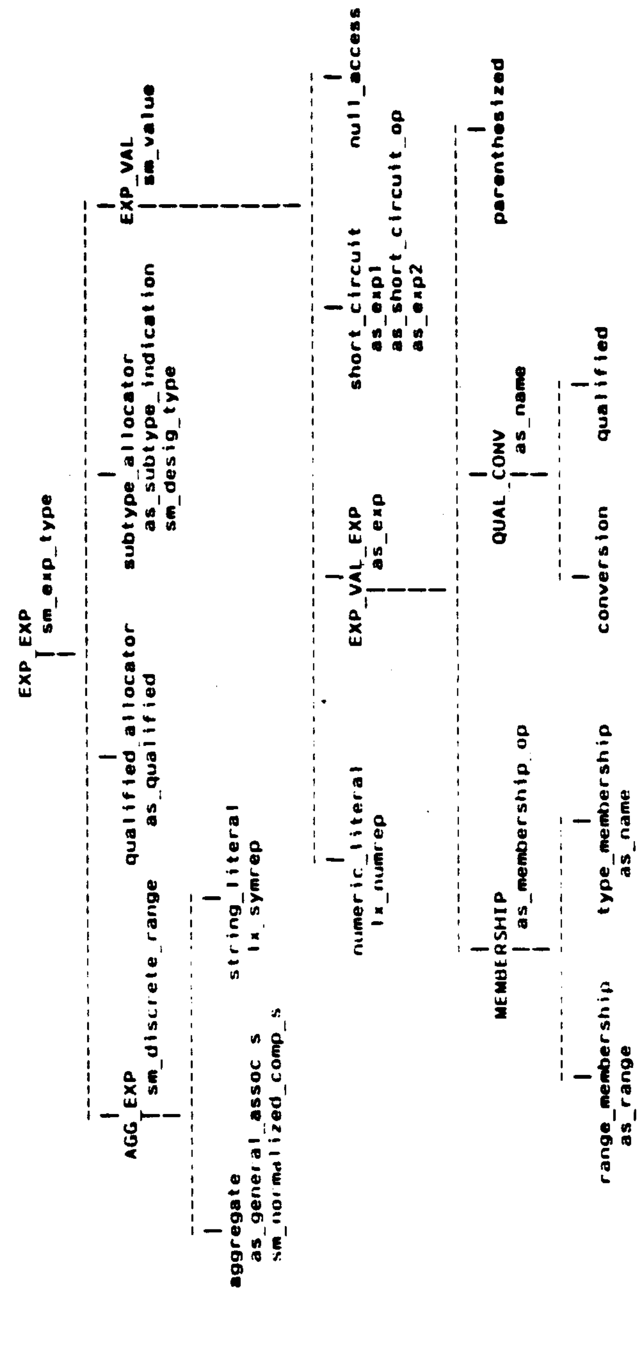
3.8.2.2.2.1.1 MEMBERSHIP

The class **MEMBERSHIP** represents the use of a membership operator. The attribute as_exp records the simple expression, and the as_membership attribute denotes the applicable membership operator (**in_op** or **not_in**). **MEMBERSHIP** contains two nodes: **range_membership** and **type_membership**. Each contains the appropriate structural attribute to retain the type or range given in the expression.

3.8.2.2.2.1.2 QUAL_CONV

The nodes in class **QUAL_CONV** -- **qualified** and **conversion** -- correspond to qualified expressions and explicit conversions, respectively. The as_exp attribute denotes the given expression or aggregate, and as_name references the node associated with the type mark. The sm_exp_type attribute denotes the TYPE_SPEC node corresponding to the type mark.


GENERAL_ASSOC
NAMED_ASSOC
as_exp
named
as_choice_s
assoc
as_used_name
EXP
void
EXP_EXP
(see next page)
NAME
(see below)
NAME
void
NAME_EXP
as_name
sm_exp_type
DESIGNATOR
sm_defn
lx_symrep
USED_OBJECT
sm_exp_type
sm_value
all
indexed
as_exp_s
NAME_VAL
sm_value
slice
as_discrete_range
attribute
as_used_name_id
as_exp
selected
as_designator
function_call
as_general_assoc_s
sm_normalized_param_s
lx_prefix
USED_NAME
used_name_id
used_op
used_char
used_object_id


EXP_EXP
sm_exp_type
AGG_EXP
sm_discrete_range
aggregate
as_general_assoc_s
sm_normalized_comp_s
string_literal
lx_symrep
qualified_allocator
as_qualified
subtype_allocator
as_subtype_indication
sm_desig_type
EXP_VAL
sm_value
numeric_literal
lx_numrep
EXP_VAL_EXP
as_exp
short_circuit
as_exp1
as_short_circuit_op
as_exp2
null_access
MEMBERSHIP
as_membership_op
range_membership
as_range
type_membership
as_name
QUAL_CONV
as_name
conversion
qualified
parenthesized

Section 3.9

STM_ELEM

3.9 STM_ELEM

The class **STM_ELEM** contains nodes representing items which may appear in a sequence of statements; i.e. nodes corresponding to statements or pragmas.

The node **stm_pragma** represents a pragma which appears in a sequence of statements. Its only non-lexical attribute, as_pragma, designates a **pragma** node.

3.9.1 STM

A node from class **STM** represents an Ada statement. Some of the **STM** nodes are grouped together because they are similar in their structure to other nodes in the class; the manner in which these nodes are classified does not imply any semantic similarity.

The node **null_stm** represents a NULL statement; it defines no attributes of its own.

The node **labeled** represents a labeled statement. The as_source_name_s attribute denotes a sequence of label names (**label_id** nodes). These label names are defining occurrences, hence the **labeled** node serves as a "declaration" for the associated labels. The as_pragma_s attribute represents the pragmas occurring between the label(s) and the statement itself; it designates a possibly empty sequence of **pragma** nodes. The as_stm attribute denotes the actual statement, it may reference any type of **STM** node other than another **labeled** node.

The **accept** node represents an accept statement. The as_name attribute records the entry simple name; it may denote either a **used_name_id** or an **indexed** node, depending on whether or not the entry is a member of an entry family. The attribute as_param_s denotes a possibly empty sequence of nodes from class **PARAM** corresponding to the formal part. The as_stm_s attribute is a possibly empty sequence representing the statements to be executed during a rendezvous.

The **abort** node represents an abort statement. The as_name_s attribute is a sequence of nodes corresponding to the task names given in the abort statement.

The node **terminate** corresponds to a terminate statement; it defines no attributes of its own.

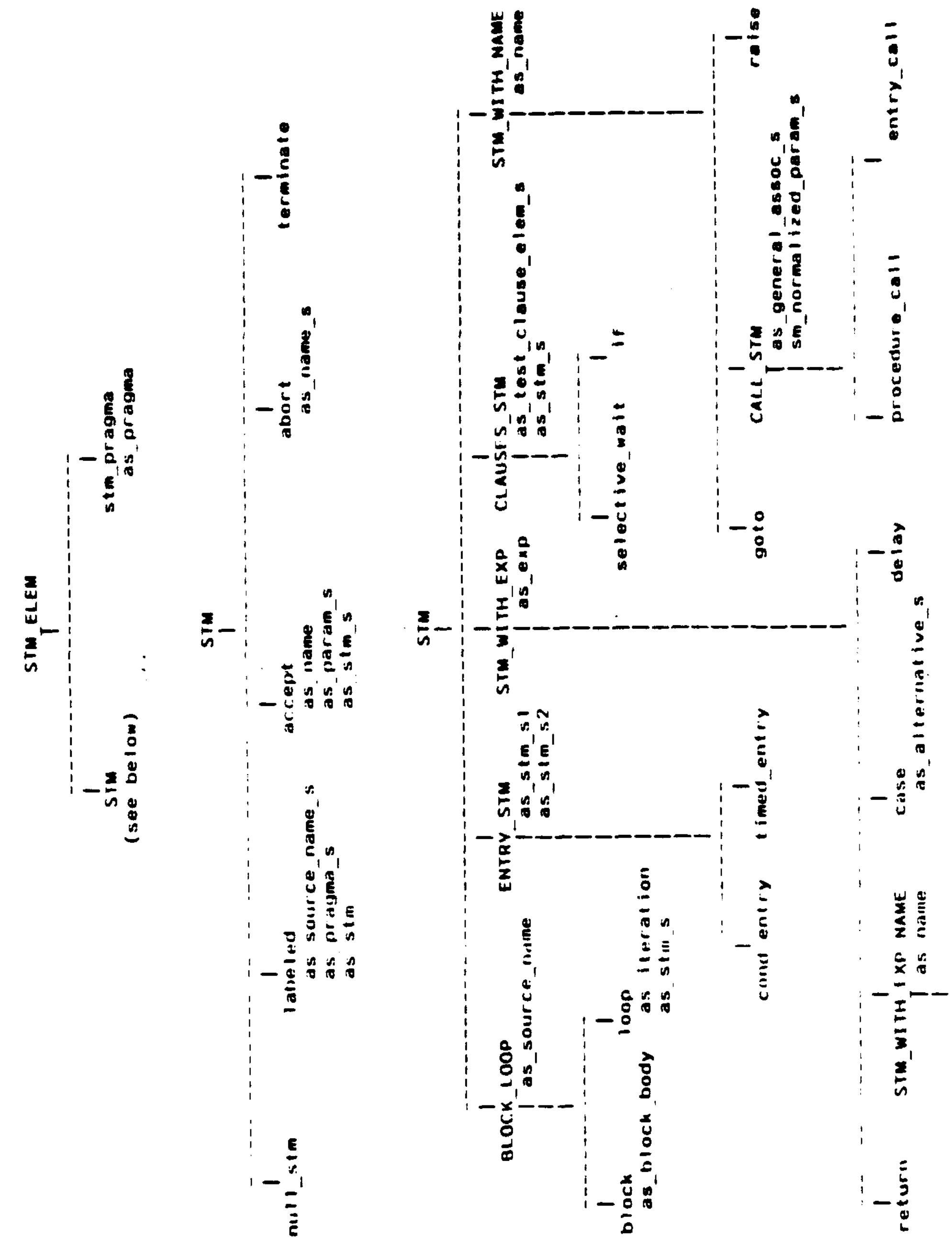
The node **goto** represents a goto statement. The as_name attribute corresponds to the label name given in the statement.

The **raise** node represents a raise statement. The attribute as_name denotes the exception name, if specified; otherwise it is **void**.

3.9.1.5.1 CALL_STM

The class **CALL_STM** represents procedure calls and entry calls; it comprises the nodes **procedure_call** and **entry_call**. **CALL_STM** defines two attributes: as_general_assoc_s and sm_normalized_param_s. The attribute as_general_assoc_s denotes a possibly empty sequence containing a mixture of **assoc** and **EXP** nodes representing the parameter associations. The sm_normalized_param_s attribute designates a possibly empty sequence corresponding to a normalized list of actual parameters.

A call to an entry that has been renamed as a procedure is represented by a **procedure_call** node rather than an **entry_call** node.


STM_ELEM
STM (see below)
stm_pragma
as_pragma
STM
null_stm
labeled
as_source_name_s
as_pragma_s
as_stm
accept
as_name
as_param_s
as_stm_s
abort
as_name_s
terminate
STM
BLOCK_LOOP
as_source_name
ENTRY_STM
as_stm_s1
as_stm_s2
STM_WITH_EXP
as_exp
CLAUSES_STM
as_test_clause_elem_s
as_stm_s
STM_WITH_NAME
as_name
block
as_block_body
loop
as_iteration
as_stm_s
cond_entry
timed_entry
selective_wait
if
goto
CALL_STM
as_general_assoc_s
sm_normalized_param_s
raise
return
STM_WITH_EXP_NAME
as_name
case
as_alternative_s
delay
procedure_call
entry_call
assign
code
exit
sm_stm

3.10 MISCELLANEOUS NODES AND CLASSES

3.10.1 CHOICE

A node from class **CHOICE** represents either the use of a discriminant simple name in a discriminant association, or a choice contained in one of the following:

(a) a record variant

(b) a component association of an aggregate

(c) a case statement alternative

(d) an exception handler

Nodes in this class may appear only as a part of a sequence of choice nodes. **CHOICE** comprises the nodes choice_exp, choice_range, and choice_others.

The node **choice_exp** represents a choice that is a simple name or an expression; it has a single structural attribute -- as_exp. If the **choice_exp** node corresponds to a simple name (that of a discriminant, a component, or an exception) then as_exp references a **used_name_id** node. Otherwise, **choice_exp** must represent a choice consisting of a simple expression, which is represented by a node from class **EXP**.

A choice which is a discrete range is represented by a **choice_range** node. The as_discrete_range attribute references the discrete range.

A **choice_others** node corresponds to the choice "others"; it defines no attributes of its own.

3.10.2 ITERATION

The members of class **ITERATION** -- **while**, **FOR_REV**, and **void** -- represent the iteration schemes of a loop (**void** corresponds to the absence of an iteration scheme). These nodes are introduced by the as_iteration attribute of a **loop** node.

The **while** node represents a "while" iteration scheme. The as_exp attribute denotes a node representing the given condition.

3.10.2.1 FOR_REV

The **FOR_REV** class represents a "for" iteration scheme. If the reserved word "reverse" appears in the loop parameter specification then the iteration is represented by a **reverse** node; otherwise it is denoted by a **for** node. The as_source_name attribute designates an **iteration_id** corresponding to the defining occurrence of the loop parameter. The as_discrete_range attribute represents the discrete range.

3.10.3 MEMBERSHIP_OP

The class **MEMBERSHIP_OP** consists of the nodes **in_op** and **not_in.** These nodes are introduced by the as_membership_op attribute of a **MEMBERSHIP** node, their function being to indicate which operator is applicable.

3.10.4 SHORT_CIRCUIT_OP

The nodes in class **SHORT_CIRCUIT_OP** -- **and_then** and **or_else** -- serve to distinguish the two types of short-circuit expressions. They are introduced by the as_short_circuit_op attribute of the **short_circuit** node.

3.10.5 ALIGNMENT_CLAUSE

The class **ALIGNMENT_CLAUSE** represents the alignment clause portion of a record representation clause. It is composed of the nodes **alignment** and **void** (**void** corresponds to the absence of an alignment clause).

The **alignment** node contains the attributes as_pragma_s and as_exp. The former is a possibly empty sequence of **pragma** nodes corresponding to the pragmas occurring between the reserved word "record" and the alignment clause. The as_exp attribute refers to the node associated with the static simple expression.

3.10.6 VARIANT_PART

The **VARIANT_PART** class represents the variant part of a record type definition; it contains the nodes **variant_part** and **void** (**void** corresponds to the absence of a variant part).

The **variant_part** node defines the attributes as_name and as_variant_s. The as_name attribute references a **used_object_id** corresponding to the discriminant simple name; as_variant_s is a sequence containing at least one **variant** node and possibly **variant_pragma** nodes.

3.10.7 TEST_CLAUSE_ELEM

The class TEST_CLAUSE_ELEM represents alternatives for an if statement or a selective wait statement. It contains the node **select_alt_pragma** and the class TEST_CLAUSE. These nodes may appear only in a **test_clause_elem_s** sequence.

The node **select_alt_pragma** represents a pragma which occurs at a place where a select alternative is allowed. It may appear only in a **test_clause_elem_s** sequence of a **selective_wait** node. The as_pragma attribute denotes the pragma itself.

3.10.7.1 TEST_CLAUSE

A TEST_CLAUSE node (**cond_clause** or **select_alternative**) represents a condition and sequence of statements occurring in an if statement or a selective wait statement. The as_exp attribute corresponds to the condition, and the as_stm_s attribute to the sequence of statements. The **cond_clause** node may appear only in a **test_clause_elem_s** sequence of an **if** node, and the **select_alternative** node may occur only in a **test_clause_elem_s** sequence of a **selective_wait** node.

3.10.8 ALTERNATIVE_ELEM

The class ALTERNATIVE_ELEM represents case statement alternatives, exception handlers, and pragmas which occur at a place where either of the previous items are allowed. The nodes **alternative** and **alternative_pragma** constitute ALTERNATIVE_ELEM; they may occur only as members of **alternative_s** sequences.

The **alternative_pragma** node has a single structural attribute, as_pragma, which denotes the pragma.

The **alternative** node contains two non-lexical attributes: as_choice_s and as_stm_s. For a case statement alternative the as_choice_s sequence may contain any of the nodes belonging to class CHOICE: however, for an exception handler the sequence is restricted to containing **choice_exp** and **choice_others** nodes. The as_stm_s attribute represents the sequence of statements given in the alternative or handler.

3.10.9 COMP_REP_ELEM

The class **COMP_REP_ELEM** consists of the nodes **comp_rep** and **comp_rep_pragma**, which may appear only in the as_comp_rep_s sequence of a **record_rep** node.

The **comp_rep** node represents a component representation clause. The as_name attribute references a **used_object_id** corresponding to the component simple name, as_exp represents the static simple expression, and as_range denotes the static range.

A pragma that occurs at the place of a component clause is represented by a **comp_rep_pragma** node; as_pragma denotes the pragma.

3.10.10 CONTEXT_ELEM

The nodes in class **CONTEXT_ELEM** represent items which may appear at a place where a context clause is allowed. They may occur only as members of the **context_elem_s** sequence of a **compilation_unit** node.

The **with** node represents a with clause and any subsequent use clauses and pragmas. The as_name_s attribute is a sequence of **used_name_id** nodes corresponding to the library unit names given in the with clause. The as_use_pragma_s attribute is a possibly empty sequence which can contain nodes of type **use** and **pragma**.

The **context_pragma** node has a single non-lexical attribute, as_pragma, which denotes the pragma.

3.10.11 VARIANT_ELEM

The nodes in class **VARIANT_ELEM** correspond to items which may appear at a spot where a variant is allowed. These nodes are contained in the sequence denoted by the as_variant_s attribute of the **variant_part** node.

The **variant** node has two structural attributes: as_choice_s and as_comp_list. The as_choice_s attribute is a sequence representing the choices applicable to that particular variant; as_comp_list corresponds to the component list.

The sole non-lexical attribute of the **variant_pragma** node is as_pragma, denoting the pragma.

3.10.12 compilation

The node **compilation** corresponds to a compilation; it defines the attribute as_compltn_unit_s, a possibly empty sequence of **compilation_unit** nodes.

3.10.13 compilation_unit

A **compilation_unit** node represents an item or items which may appear at a place where a compilation unit is allowed; i.e. it may represent a compilation or a sequence of pragmas.

A **compilation_unit** node represents a sequence of pragmas only when a compilation consists of pragmas alone. In this case the as_context_elem_s sequence is empty, the as_all_decl attribute is **void**, and as_pragma_s denotes the sequence of pragmas which constitute the compilation.

For a **compilation_unit** node corresponding to a compilation unit the as_context_elem_s attribute is a possibly empty sequence representing the context clause and any pragmas preceding the compilation unit. The as_all_decl attribute denotes the library unit or the secondary unit, which may be represented by one of the following: a node belonging to class **UNIT_DECL**, a **subunit**, a **subprogram_body**, or a **package_body**. The as_pragma_s attribute denotes the pragmas which follow the compilation unit and do not belong to a subsequent compilation unit. The pragmas allowed to appear in this sequence include INLINE, INTERFACE, LIST, and PAGE. LIST and PAGE pragmas which occur between compilation units but after any INLINE or INTERFACE pragmas may appear in either the as_pragma_s sequence of the preceding compilation unit or the as_context_elem_s sequence of the succeeding compilation unit.

3.10.14 comp_list

A record component list is represented by a **comp_list** node, which contains three structural attributes: as_decl_s, as_variant_part, and as_pragma_s. The as_decl_s attribute designates a sequence corresponding to either a series of component declarations or the reserved word "null". The attribute as_variant_part denotes the variant part of the record, if one exists. The as_pragma_s attribute records the occurrence of pragmas between the variant part and the end of the record declaration (i.e. pragmas appearing between "end case" and "end record").

If the record is a null record then as_variant_part is **void**, and the sequence denoted by as_pragma_s is empty. The as_decl_s attribute is a sequence having a **null_comp_decl** node as its first element, and any number of **pragma** nodes after it.

If the record is not a null record then as_decl_s is a possibly empty sequence which can contain nodes of type **variable_decl** and **pragma**. If the record type does not have a variant part then as_variant_part is **void** and as_pragma_s is empty. It is not possible for as_decl_s to be empty and as_variant_part to be **void** in the same **comp_list** node.

3.10.15 index

The **index** node represents an undefined range, and appears only in sequences associated with unconstrained array types and unconstrained array definitions (such sequences are denoted by the as_index_s attribute of the the **array** and the **unconstrained_array_def** nodes). The as_name attribute refers to the **used_name_id** or **selected** node corresponding to the type mark given in the index subtype definition. The sm_type_spec attribute references the TYPE_SPEC node associated with the type mark.

```
                              CHOICE
                                |
         ------------------------------------------------
         |                      |                       |
    choice_exp             choice_range           choice_others
      as_exp                 as_discrete_range


                             ITERATION
                                |
         ------------------------------------------------
         |                      |                       |
       while                 FOR_REV                   void
         as_exp                 | as_source_name
                                | as_discrete_range
                                |
                         ------------------
                         |                |
                        for            reverse


                           MEMBERSHIP_OP
                                |
                          -------------
                          |           |
                        in_op       not_in


                          SHORT_CIRCUIT_OP
                                |
                         ----------------
                         |              |
                      and_then       or_else


                          ALIGNMENT_CLAUSE
                                |
                       --------------------
                       |                  |
                   alignment             void
                     as_pragma_s
                     as_exp


                            VARIANT_PART
                                |
                       --------------------
                       |                  |
                  variant_part           void
                    as_name
                    as_variant_s
```

```
                    TEST_CLAUSE_ELEM
                           |
                 ---------------------------------
                 |                               |
            TEST_CLAUSE                  select_alt_pragma
                 | as_exp                   as_pragma
                 | as_stm_s
                 |
       -----------------------------
       |                           |
  cond_clause              select_alternative
```

```
                  ALTERNATIVE_ELEM
                         |
     ---------------------------------------
     |                                     |
  alternative                    alternative_pragma
     as_choice_s                    as_pragma
     as_stm_s
```

```
                  COMP_REP_ELEM
                        |
       ----------------------------------
       |                                |
   comp_rep                     comp_rep_pragma
      as_name                      as_pragma
      as_range
      as_exp
```

```
                  CONTEXT_ELEM
                       |
   ------------------------------------------
   |                                        |
  with                              context_pragma
     as_name_s                         as_pragma
     as_use_pragma_s
```

```
                  VARIANT_ELEM
                       |
       -------------------------------
       |                             |
   variant                    variant_pragma
      as_choice_s                as_pragma
      as_comp_list
```

```
compilation
   as_compltn_unit_s

compilation_unit
   as_context_elem_s
   as_all_decl
   as_pragma_s

comp_list
   as_decl_s
   as_variant_part
   as_pragma_s

index
   as_name
   sm_type_spec
```

CHAPTER 4

RATIONALE

4.1 DESIGN DECISIONS

During the course of designing DIANA many design decisions were affected by the need to adhere to the design principles set forth in the first chapter of this document. This section discusses some of these decisions and the reasons that they were made. Each subsection explains the design decisions pertaining to a particular design principle.

4.1.1 INDEPENDENCE OF REPRESENTATION

One of the major design principles of DIANA requires that the definition of DIANA be representation independent. Unfortunately, some of the information which should be included in a DIANA structure is implementation-dependent. For example, DIANA defines a source position attribute for each node which represents source code. This attribute is useful for reconstructing the source program, for reporting errors, for source-level debuggers, and so on. It is not, however, a type that should be defined as part of this standard since each computer system has idiosyncratic notions of how a position in the source program is encoded. For that matter, the concept of source position may not be meaningful if the DIANA arises from a syntax editor. For these reasons, attributes such as source position are merely defined to be private types.

A private type names an implementation-specific data structure that is inappropriate to specify at the abstract structure level. DIANA defines six private types. Each of these corresponds to one of the kinds of information which may be installation or target machine specific. They include types for the source position of a node, the representation of identifiers, the representation of various values on the target system, and the representation of comments from the source program. The DIANA user must supply an implementation for each of these types.

As is explained in the Ada reference manual, a program is assumed to be compiled in a 'standard environment'. An Ada program may explicitly or implicitly reference entities defined in this environment, and the DIANA representation of the program must reflect this. The entities that may be referenced include the predefined attributes and types. The DIANA definition of these entities is not given in this document but is assumed to be available.

4.1.1.1 SEPARATE COMPILATION

It would not be appropriate for DIANA to provide the library management upon which separate compilation of Ada program units is based. Nonetheless, the possibility of separate compilation affects the design of DIANA. The intermediate representation of a previously compiled unit may need to be used again. Furthermore, all of the information about a program unit may not be known when it is first compiled.

The design of DIANA carefully avoids constraints on a separate compilation system, aside from those implied directly by the Ada language. The design can be extended to cover the full APSE requirements[3]. Special care has been taken that several versions of a unit body can exist corresponding to a single specification, that simultaneous compilation within the same project is possible, and that units of other libraries can be used effectively [5]. The basic decision which makes these facilities implementable is to forbid forward references.

Certain entities may have more than one definition point in Ada. In such cases, DIANA restricts the attribute values of all of the defining occurrences to be identical. In the presence of separate compilation the requirement that the values of the attributes at all defining occurrences are the same cannot always be met. The forward references assumed by DIANA are **void** in these cases. The reasons for this approach are:

- A unit can be used even when the corresponding body is not yet compiled. In this case, the forward reference must have the value **void** since the entity does not exist.
- Updating a DIANA representation would require write access to a file which may cause synchronization problems (see [5]).
- A library system may allow for several versions of bodies for the same specification. If an attribute were updated its previous value would be overwritten. Moreover, the maintenance of different versions should be part of the library system and should not influence the intermediate representation.

4.1.2 EFFICIENT IMPLEMENTATION AND SUITABLITY FOR VARIOUS KINDS OF PROCESSING

The design goals of efficient realization and suitability for many kinds of processing are interrelated. It was necessary to define a structure containing the information needed to perform different kinds of processing without overburdening any one kind of processing with the task of computing and retaining a great deal of extraneous information.

Since many tools will be manipulating the source text in some way, it was decided that DIANA should retain the structure of the original source program. In order to do this, structural attributes were defined. These attributes define a tree representing the original source. It is always possible to regenerate the source text from its DIANA form (except for purely lexical issues, such as the placement of comments) by merely traversing the nodes denoted by structural attributes.

Unfortunately, the structure of the original source program is not always suitable for semantic processing. Hence, semantic attributes were added to augment the structural ones. These attributes transform the DIANA structure from a tree to a network. In some cases these attributes are merely "shortcuts" to nodes which are already in the DIANA structure, but in other cases semantic attributes denote nodes which do not correspond to source text at all.

In the process of adding semantic attributes to the definition of DIANA, it was necessary to decide which information should be represented explicitly and which should be recomputed from stored information. Obviously, storing as little information as possible makes the DIANA representation smaller; however, such an approach also increases the time required for semantic processing. Storing all of the information required would improve processing speed at the expense of storage requirements. The attribution principles of DIANA are a compromise between these extremes.

In order to decide whether or not to include a particular attribute the following criteria were considered:

- o DIANA should contain only such information as would be typically discovered via static (as opposed to dynamic) semantic analysis of the original program.

- o If information can be easily recomputed, it should be omitted.

These two points are discussed at length in the following two subsections.

4.1.2.1 STATIC SEMANTIC INFORMATION

DIANA includes only the information that is determined from STATIC semantic analysis, and excludes information whose determination requires DYNAMIC semantic analysis.

This decision affects the evaluation of non-static expressions and evaluation of exceptions. For example, the attribute sm_value should not be used to hold the value of an expression that is not static, even if an implementation's semantic analyzer is capable of evaluating some such expressions. Similarly, exceptions are part of the execution (i.e. dynamic) semantics of Ada and should not be represented in DIANA. Thus the attribute sm_value is not used to represent an exception to be raised.

Of course, an implementation that does compute these additional values may record the information by defining additional attributes. However, any DIANA consumer that relies on these attributes cannot be considered a correct DIANA "user", as defined in this document.

4.1.2.2 WHAT IS 'EASY TO RECOMPUTE'?

Part of the criteria for including an attribute in DIANA is that it should be omitted if it is easy to recompute from the stored information. It is important to avoid such redundant encodings if DIANA is to remain an implementable internal representation. Of course, this guideline requires a definition of this phrase.

An attribute is easily computed if:

- o it requires visits to no more than three to four nodes; or
- o it can be computed in one pass through the DIANA tree, and all nodes with this attribute can be computed in the same pass.

The first criterion is clear; the second requires discussion.

Consider first an attribute that is needed to perform semantic analysis. As an implementation is building a DIANA structure, it is free to create extra (non-DIANA) attributes for its purposes. Thus the desired attributes can be created by those implementations that need them. To require these attributes to be represented by all DIANA users is an imposition on implementations which use algorithms that do not require these particular attributes.

Consider now an attribute needed to perform code generation. As long as the attribute can be determined in a single pass, the routine that reads in the DIANA can readily add it as it reads in the DIANA. Again, some implementors may not need the attribute, and it is inappropriate to burden all users with it.

It is for these reasons that suggestions for pointers to the enclosing compilation unit, pointers to the enclosing namescope, and back pointers in general have been rejected. These are attributes that are easily computed in one pass through the DIANA tree and indeed may not be needed by all implementations.

Of course, a DIANA producer can create a structure with extra attributes beyond those specified for DIANA. Nevertheless, any DIANA consumer that relies on these additional attributes is not a DIANA "user", as defined in this document.

4.1.3 REGULARITY OF DESCRIPTION

In order to increase the clarity of the DIANA description, it was decided that the class structure of DIANA should be a hierarchy. Unfortunately, some nodes should belong in more than one class. To circumvent this problem, several intermediate nodes were defined, nodes for which the only non-lexical attribute denotes a node that already belongs to another class. These intermediate nodes do not convey any structural or semantic information beyond the value of the non-lexical attribute. DIANA contains the following intermediate nodes:

block_master

discrete_subtype
integer_def
float_def
fixed_def
choice_exp
choice_range
stm_pragma
select_alt_pragma
alternative_pragma
comp_rep_pragma
context_pragma
variant_pragma

It should be noted that not all nodes containing a single non-lexical attribute are intermediate nodes. For instance, the **renames_unit** node has a single non-lexical attribute as_name; however, the **renames_unit** node is not an intermediate node because it is used to convey the fact that a unit has been renamed, in addition to recording the name of the original unit via the as_name attribute. On the other hand, the **choice_exp** node was introduced merely because the class **EXP** could not be included in both **ASSOC** and **CHOICE**. It contains no more information than the EXP node denoted by its as_exp attribute.

It is not the intention of DIANA to require that intermediate nodes be represented as such; they are included in the definition of DIANA only to maintain a hierarchy. This is a natural place for an implementation to optimize its internal representation by excluding the intermediate nodes, and directly referencing each node denoted by the non-lexical attribute of an intermediate node.

In the DIANA description, attributes are defined at the highest possible level; i.e. if all of the nodes of a class have the same attribute then the attribute is defined on the class rather than on the individual nodes. In this way a node may "inherit" attributes from the class to which it belongs, and from the class to which that class belongs, etc.

The node **void** receives a slightly different treatment than the other DIANA nodes. It is the only node which violates the DIANA hierarchy, and it is the only node which inherits attributes which cannot be used. The node **void** represents "nothing". It may be thought of as a null pointer, although it does not have to be represented as such. Instead of requiring a different kind of **void** node for each class to which **void** belongs, **void** was allowed to belong to more than one class (thus constituting the only exception to the hierarchy). Because **void** is a member of many classes, it inherits numerous attributes. Rather than move the attribute definitions from the classes and put them on all of the constituent nodes except for **void**, it was decided to allow **void** to inherit attributes. Since it is meaningless for "nothing" to have attributes, a restriction was added to the semantic specification of DIANA. The attributes of **void** may not be manipulated in any way; they cannot be set or examined, hence they are in effect not represented in a DIANA structure.

4.2 DECLARATIONS

Explicit declarations are represented in a DIANA structure by declarative nodes. These nodes preserve the source text for source reconstruction and conformance checking purposes. They do not record the results of semantic analysis; that information is contained in the corresponding defining occurrence nodes. All declarative nodes have a child that is a node or sequence of nodes (of type **SOURCE_NAME**) representing the identifier(s) used to name the newly defined entity or entities.

Declarative nodes are members of class **ITEM**. The nodes in class **ITEM** are grouped according to similarities in the syntax of the code that they represent and similarities in the context in which they can appear.

4.2.1 MULTIPLE ENTITY DECLARATIONS

Certain kinds of declarations -- object declarations, number declarations, discriminant declarations, component declarations, parameter declarations, and exception declarations -- can introduce more than one entity. The nodes corresponding to these declarations belong to two different classes, **DSCRMT_PARAM_DECL** and **ID_S_DECL**, both of which define an as_source_name_s attribute. These classes are distinguished from each other because they appear in different contexts. Discriminant declarations can appear only in discriminant parts, and parameter declarations can appear only in formal parts. The remaining multiple entity declarations are basic declarative items, and can appear in declarative parts. In addition, the basic declarative items can appear in sequences containing pragmas, which cannot be given in discriminant parts or formal parts.

ID_S_DECL is further subdivided into classes according to the syntactic similarities of the declarations it represents. For instance, object declarations and number declarations may have (default) initial values, hence **constant_decl**, **variable_decl**, and **number_decl** belong to class **EXP_DECL**, which defines an as_exp attribute to record that value.

4.2.1.1 OBJECT DECLARATIONS AND COMPONENT DECLARATIONS

The type portion of object declarations may be given in two different ways: either by a subtype indication or a constrained array definition. The node

constrained_array_def is already a member of class **TYPE_DEF**, which represents the syntax of type definitions. Rather than include **constrained_array_def** in another class and disrupt the heirarchy, the node **subtype_indication** was added to class **TYPE_DEF**. Thus the class **OBJECT_DECL**, which comprises the nodes **constant_decl** and **variable_decl**, defines the attribute as_type_def to represent the type specification given in the object declaration. Obviously as_type_def cannot denote any kind of **TYPE_DEF** node other than **subtype_indication** and **constrained_array_def** in this particular context.

The only other kind of declaration which introduces objects and does not require the type specification to be a type mark is a component declaration, the type portion of which is given by a subtype indication. Rather than define a node exclusively for component declarations, they are represented by the same kind of node as variable declarations, since the **variable_decl** node allows subtype indications for the as_type_def attribute. This is also convenient because component declarations may be interspersed with pragmas, and both **pragma** and **variable_decl** belong to class **DECL** (a sequence of component declarations is represented by a **decl_s** sequence). Component declarations and variable declarations may be distinguished by the fact that they appear in different contexts, and by the type of nodes in the as_source_name_s sequence. The former contains **component_id** nodes and the latter contains **variable_id** nodes.

A record component list may contain the reserved word "NULL" rather than component declarations or a variant part. This is indicated by the insertion of a **null_comp_decl** node instead of **variable_decl** nodes in the sequence of component declarations. Hence it was necessary to include the **null_comp_decl** node in class **DECL**. It is convenient for the **null_comp_decl** node to be part of a sequence because it may be followed by pragmas (a pragma can appear after a semicolon delimiter). Although **null_comp_decl** belongs to **DECL**, the ONLY place that it can appear in a DIANA structure is as the first node in the as_decl_s sequence of the **comp_list** node (this restriction is given in the semantic specification of DIANA).

4.2.2 SINGLE ENTITY DECLARATIONS

The remaining kinds of declarations introduce single entities. They are represented by the classes SUBUNIT_BODY and ID_DECL. Like the classes **DSCRMT_PARAM_DECL** and ID_S_DECL, SUBUNIT_BODY and ID_DECL are distinguished because they represent declarative items which occur in different contexts. **SUBUNIT_BODY** represents body declarations, both proper body and stub declarations. These declarations are separated from the declarations in **ID_DECL** because body declarations are exclusively later declarative items (the few members of **ID_DECL** that are later declarative items are basic declarative items as well). **ID_DECL** contains basic declarative items and items that can appear in task specifications.

Body declarations include subprogram body declarations, package body declarations, task body declarations, and stub declarations. These declarations are represented by the nodes **subprogram_body**, **package_body**, and **task_body**. The difference between a proper body and a stub is indicated by the value of the as_body attribute, a **block_body** in the former case, and a **stub** in the latter.

4.2.2.1 PROGRAM UNIT DECLARATIONS AND ENTRY DECLARATIONS

Due to syntactic similarities, declarations of entries and program units other than tasks are represented by nodes from class **UNIT_DECL**, which contains only three members: **generic_decl**, **subprog_entry_decl**, and **package_decl**. The combination of the kind of node representing the declaration and the values of the as_header and as_unit_kind attributes uniquely determine the exact form of the declaration. The different kinds of declarations are listed with their appropriate attribute values in Table 4.1.

The **HEADER** and **UNIT_KIND** nodes also record information peculiar to that sort of declaration. For example, the **renames_unit** node not only indicates that a declaration is a renaming declaration, but retains the name of the original unit as well.

A task declaration can introduce either a task type, or a single task object with an anonymous type, depending on whether or not the declaration contains the reserved word "type". The syntax of a task declaration differs from that of an ordinary type or object declaration, hence the **type_decl** and **variable_decl** nodes are not suitable for representing a task declaration. Because the same information is given in the task declaration regardless of the kind of entity it introduces, a **task_decl** node represents both kinds of task declarations. If the defining occurrence associated with the declaration is a **variable_id** then the declaration creates both an object and a type; if the defining occurrence is a **type_id** then the declaration creates a type.

Since a task declaration always defines a new task type, a new task type specification (a **task** node) is created for each declaration. If the type is *anonymous* it is introduced by the sm_obj_type attribute of the **variable_id**; otherwise the **task** node is introduced by the sm_type_spec attribute of the **type_id**.

It should be noted that a task object may also be declared with an ordinary object declaration. Since declarative nodes record the syntax of the declaration, a **variable_decl** node rather than a **task_decl** node denotes the declaration in this case. This kind of declaration does not introduce a new task type, thus a new task type specification is not created for the task object(s).

declaration, at which point the context indicates whether or not the declaration and its defining occurrence(s) are generic.

4.2.5 IMPLICIT DECLARATIONS

The Ada programming language defines different kinds of implicit declarations. Certain operations are implicitly declared after a type definition (including derived subprograms following a derived type definition). Labels, loop names, and block names are implicitly declared at the end of the corresponding declarative part. These declarations are not explicitly represented in DIANA; to do so would interfere with source reconstruction.

Since a label, loop name, or block name can be associated with only one statement, and the label or name precedes that statement in the source text, it is natural for the defining occurrence of a label (a **label_id**) to be its appearance in a labeled statement, and the defining occurrence of a block or loop name (a **block_loop_id**) to be its appearance in a block or loop statement.

Implicitly declared operations and derived subprograms are associated with a single type definition. Unfortunately, the names of these operations and derived subprograms are not used in that type definition. As a result, there is no appropriate structural attribute to introduce the defining occurrences of the operations associated with a type. All appearances of the names of these operations and derived subprograms are represented in the DIANA structure as used occurrences. A defining occurrence still exists for each such operation or derived subprogram; however, it can only be referenced by semantic attributes.

Section 4.3

SIMPLE NAMES

4.3 SIMPLE NAMES

Simple names comprise identifiers, character literals, and operator symbols.

The attributes lx_srcpos and lx_comments are defined for all DIANA nodes that represent source code. An implementation has the option of including these attributes or not; however, if an implementation does choose to include them then it is necessary to have a distinct node for every occurrence of a simple name in the source code. Since it is not desirable to have to copy all of the semantic attributes associated with the name of an entity every time the name is used, the appearances of simple names in a DIANA tree are divided into defining and used occurrences. The former are represented by class **DEF_NAME** and the latter by class **DESIGNATOR**.

In order to avoid constraining an implementation, DIANA does not REQUIRE that a distinct used occurrence node be created for every use of a simple name. A single used occurrence node may be created for a particular name, and all references to that entity in the source code may be represented by references to that single used occurrence node in the DIANA structure.

The defining nodes for entities together with their attributes play the same role as a dictionary or symbol table in conventional compiler strategy. Unless there is interference from separate compilation, it is possible for all information about an entity to be specified by attributes on its defining node. The node for a used occurrence of an entity always refers back to this defining occurrence via the sm_defn attribute.

Defining occurrences are represented by different kinds of nodes rather than a single construct, thereby allowing the appropriate semantic attributes to be attached to each. For instance, the defining occurrence of a discriminant is represented by a **discriminant_id**, which has an attribute to record the applicable component clause (if there is one); the defining occurrence of a constant is represented by a **constant_id**, which has an attribute that references the applicable address clause (if there is one).

DIANA also distinguishes the kinds of usage depending on the properties of the entity that is referenced. For example, a used occurrence of an object name is represented by a **used_object_id**, while that of an operator is represented by a **used_op**.

DIANA has the following set of defining occurrences.

attribute_id
argument_id
block_loop_id
bltn_operator_id
character_id
component_id
constant_id
discriminant_id
entry_id
enumeration_id
exception_id
function_id
generic_id
iteration_id
in_id
in_out_id
label_id
l_private_type_id
number_id
operator_id
out_id
package_id
pragma_id
private_type_id
procedure_id
subtype_id
task_body_id
type_id
variable_id

and the following set of used occurrences:

used_char
used_name_id
used_object_id
used_op

4.3.1 DEFINING OCCURRENCES OF PREDEFINED ENTITIES

The consistency of this scheme requires the provision of a definition point for predefined identifiers as well. Although these nodes will never be introduced by a structural attribute because they do not have an explicit declaration, they can be referenced by the sm_defn attribute of a node corresponding to a used occurrence.

Certain kinds of entities, such as exceptions, may be either predefined or user-defined. Such an entity is represented by the same kind of node in either case -- a node from class SOURCE_NAME, which represents the defining occurrences of all entities which can by declared by the user. If, however, a SOURCE_NAME node corresponds to a predefined entity then the lx_srcpos and lx_comments attributes will be undefined since it does not correspond to source text.

Other entities can never be declared by the user; i.e. pragmas, pragma argument identifiers, and attributes. These entities are represented by nodes from class **PREDEF_NAME**; the lx_srcpos and lx_comments attributes of nodes belonging to this class are never defined. **PREDEF_NAME** also contains nodes corresponding to defining occurrences of the predefined operators (these operators cannot be declared by the user, although they may be overloaded). A user-defined operator is represented by a node from class **SOURCE_NAME**.

4.3.2 MULTIPLE DEFINING OCCURRENCES

In general, every entity has a single defining occurrence. In the instances where multiple defining occurrences can occur, each defining occurrence is represented by a **DEF_NAME** node.

The entities which may have multiple defining occurrences are:

(a) deferred constants

(b) incomplete types

(c) non-generic (limited) private types

(d) discriminants of incomplete or (limited) private types

(e) non-generic formal parameters

(f) program units

With the exception of tasks and (limited) private types, the different defining occurrences of one of these entities are represented by the same kind of node. In addition, the different defining occurrences have the same attribute values (certain incomplete types and program units may have attributes which cannot be set in the first defining occurrence due to interference by separate compilation; however, this is an exception rather than a rule). These defining occurrences have an attribute, sm_first, that refers to the node for the first defining occurrence of the identifier, similar to the sm_defn attribute of used occurrences. The node that is the first defining occurrence has an sm_first attribute that references itself.

All used occurrences must reference the same defining occurrence, the one that occurs first. Nevertheless, the attributes for all defining occurrences of an entity must still be set with the appropriate values.

An entry declaration and its corresponding accept statement are not treated as different definition points of the same entity. Thus the **entry_id** is the unique defining occurrence; a **used_name_id** appears in an accept statement, the sm_defn attribute of which refers to the associated entry_id. However, the formal parts of the entry declaration and the accept statement multiply define the entry formal parameters.

Any names appearing in a record representation clause or an enumeration representation clause are considered used occurrences; this includes the names of record types, record components, and enumeration literals.

4.3.2.1 MULTIPLE DEFINING OCCURRENCES OF TYPE NAMES

There are two forms of type declaration in which information about the type is given at two different places: incomplete and private. In addition to the multiple defining occurrences of the type names there are multiple defining occurrences of the discriminant names if the types include discriminant parts.

The notion of an incomplete type permits the definition of mutually dependent types. Only the new name is introduced at the point of the incomplete type declaration -- the structure of the type is given in a second type declaration which generally must appear in the same declarative part. The defining occurrences of both types are described by **type_id** nodes which have the semantic attribute sm_type_spec. With one exception (which is discussed in the following paragraph) the full type declaration must occur in the same declarative part, hence the sm_type_spec attribute of both defining occurrences can denote the full type specification.

A special case may be introduced when an incomplete type declaration occurs within the private part of a package specification. The full type declaration is not required to appear in the same declarative part; it may be given in the declarative part of the package body, which is not necessarily in the same compilation unit. Since forward references are not allowed in DIANA, if the full type declaration is in a separate compilation unit then the sm_type_spec attribute of the **type_id** corresponding to the incomplete type declaration denotes a special **incomplete** node (which is discussed in detail in the section on types). The sm_type_spec attribute of the node for the full type declaration references the full type specification.

Private types are used to hide information from the user of a package; a private type declaration appears in the visible part of a package without any structural information. The full declaration is given in the private part of the package specification (this restriction ensures that there is no interference from separate compilation). Unfortunately, the solution used for incomplete types cannot be applied to private types -- if both defining occurrences had the same node type and attributes, it could not be determined whether the type is a private one or not. This information is important when the type is used outside of the package.

DIANA views the declarations as though they were declarations of different entities -- one is a private type and the other a normal one. Both denote the same type structure in their sm_type_spec attribute, however. The distinction is achieved by introducing a new kind of a defining occurrence, namely the **private_type_id**. It has the attribute sm_type_spec which provides access to the structural information given in the full type declaration. Limited private types are treated in the same manner, except that their defining occurrence is a **l_private_type_id**. In the case of (limited) private types the sm_first attribute of the **type_id** node refers to the private_type_id or **l_private_type_id**. The **private_type_id** and **l_private_type_id** nodes do not have an sm_first attribute because they always represent the first defining occurrence of the type name.

4.3.2.2 MULTIPLE DEFINING OCCURRENCES OF TASK NAMES

The only other entity to have its different defining occurrences represented by different kinds of nodes is the task. Although a task is a program unit, its defining occurrences cannot be treated like those of other program units. The declaration of a task introduces either a task type or a task object having an anonymous type, hence the first defining occurrence of a task name is represented by a **type_id** or a **variable_id**. Any subsequent defining occurrences of the task name must correspond to either a stub declaration or a proper body declaration; these defining occurrences are represented by **task_body_id** nodes.

All of the information concerning the task is stored in the type specification of the task. Even though used occurrences of the task name do not reference the type specification (they denote the **type_id** or **variable_id**), the type specification may be reached from any of the defining occurrences.

4.3.3 USED OCCURRENCES

The nodes representing used occurrences are included in the class representing expressions because certain names may appear in expressions. Restrictions have been added to the semantic specification to differentiate the used occurrences which can appear in expressions from those which cannot.

The nodes **used_object_id** and **used_char** represent used occurrences of entities having a value and a type; these nodes can appear in the context of an expression. The former denotes objects and enumeration literals, the latter denotes character literals (in this way identifiers consisting of a single character are distinguished from character literals). To allow the nodes representing expressions to be treated in a consistent manner, the attributes sm_value and sm_exp_type were added to the **used_object_id** and **used_char** nodes.

The remaining kinds of used occurrences are represented by the **used_op** and **used_name_id** nodes. The occurrence of an operator is represented by a **used_op**, and that of any other entity by a **used_name_id**.

The names of objects and literals may appear in contexts other than expressions; in particular, in places where the Ada syntax requires a name. Should those used occurrences be represented by **used_object_id** nodes or **used_name_id** nodes? In some instances it might be useful to have ready access to more information than just the name (for example, the subtype of the object denoted by the name might be helpful). Some names (such as the "object_name" in a renaming declaration) must be evaluated just as an expression is evaluated. On the other hand, a name appearing in the left-hand part of a named association is not evaluated, and since the association is not designed for semantic processing (a normalized list of expressions is created for that purpose), it would be wasteful to record additional semantic attributes.

It was decided that the name of an object or literal appearing in the left-hand part of a named association should be represented by a **used_name_id** because the attributes peculiar to a **used_object_id** would not be needed for semantic processing in that context. Since the situation is not as clear in other contexts, all other uses of the name of an object or literal are represented by **used_object_id** nodes.

(b) anonymous base types created by constrained array definitions [ARM, 3.6]

(c) anonymous index subtypes created by constrained array definitions [ARM, 3.6]

(d) anonymous task types introduced by task declarations creating single task objects [ARM, 9.1]

The declarations of these anonymous types are not represented (to do so would interfere with source reconstruction), hence their type specifications are introduced by the appropriate semantic attributes. For instance, the node for an anonymous task type is introduced by the sm_obj_type attribute of a **variable_id** node. At some point it will be necessary to know that such types are anonymous (i.e. that they have not yet been elaborated), consequently the sm_is_anonymous attribute was added to all nodes except for those representing universal types (which are always anonymous).

In order to maintain the consistency of this type representation scheme it was necessary to include some anonymous types and subtypes which are not discussed in the reference manual.

Type definitions containing subtype indications with explicit constraints introduce anonymous subtypes. Hence the component subtype of an array or the designated subtype of an access type may be anonymous. If the constraints on the parent type and the parent subtype of a derived type are not the same then the new base type is anonymous.

Every object and expression in DIANA has an attribute denoting its subtype (sm_obj_type for objects and sm_exp_type for expressions). The subtype specification contains the applicable constraint (necessary for operations such as constraint checking) as well as a path to the base type (which is required for processing such as type resolution). If a new constraint is imposed by an object declaration or an expression then an anonymous type specification must be created in order to record the new constraint. Object or component declarations containing either a constrained array definition or a subtype indication with an explicit constraint introduce anonymous subtypes for each entity in the identifier list. Slices, aggregates, and string literals introduce anonymous subtypes if it cannot by determined statically that the constraints on the expression and those on the array prefix or context subtype are the same.

Unlike class TYPE_DEF, which is subdivided according to syntactic similarities, class TYPE_SPEC is decomposed into subclasses by the semantic characteristics (i.e. attributes) various members have in common. When placing the nodes in the hierarchy, certain compromises were made that cause a few nodes to inherit an attribute that is not really needed. For instance, the **constrained_array** and **constrained_record** nodes inherit the attribute sm_derived, even though they can never represent a derived type (they may, however, represent a subtype of a derived type). It was deemed better to have an occasional unneeded attribute than to cause confusion by defining common attributes in several different places (i.e. moving the **constrained_array** and **constrained_record** nodes outside of class DERIVABLE_SPEC and duplicating the attributes sm_is_anonymous, sm_base_type, and sm_depends_on_dscrmt for them).

4.4.1 CONSTRAINED AND UNCONSTRAINED TYPES AND SUBTYPES

A **TYPE_SPEC** node provides no indication as to whether the entity it represents is a type or a subtype. In the Ada language, the name of a type denotes not only the type, but the corresponding unconstrained subtype as well. An attempt at differentiating types and subtypes would only cause confusion and inconsistencies. A distinction is made, however, between base types and subtypes of base types. The attribute sm_base_type denotes the base type, a type specification where all representation and structural information can be found. Obviously the sm_base_type attribute of a node corresponding to a base type will contain a self-reference.

Certain nodes always represent base types; these are the **task_spec** node, and those in classes **PRIVATE_SPEC** and **UNCONSTRAINED_COMPOSITE**. The **task_spec** and **PRIVATE_SPEC** nodes do not have an sm_base_type attribute at all. As a result of their inclusion in class **NON_TASK** the **UNCONSTRAINED_COMPOSITE** nodes have inherited this attribute; however, it always contains a self-reference.

DIANA also distinguishes between constrained and unconstrained (sub)types for the following classes of types: array, record, and access. The nodes in class **UNCONSTRAINED** represent unconstrained types; those in class **CONSTRAINED** represent constrained types.

This distinction proves to be very useful when performing certain semantic checks involving array, record, or access types. For instance, the types in these classes may have index or discriminant constraints imposed upon them; however, an index or discriminant constraint cannot be imposed on the type if it is already constrained.

The fact that an object is of an unconstrained type rather than a constrained type may also affect certain implementation decisions. For example, in a complete assignment to a record object of an unconstrained type that has default values for its discriminants, the constraints on the object may be changed during execution. Hence an implementation may wish to handle objects of an unconstrained record type in a manner that is different from the way in which objects of a constrained type are treated.

All scalar types are constrained, and may be further constrained any number of times. Hence there is only one kind of node for each kind of scalar type, and each SCALAR node has an sm_range attribute which denotes the applicable range constraint. The nodes for real types have an additional sm_accuracy attribute to record the value of the digits or delta expression. For some types (such as the predefined types and enumeration types) the constraints are implicit, therefore a **range** node which does not correspond to source code must be created. The **range** node that is constructed for an enumeration type will denote the first enumeration literal as the lower bound and the last enumeration literal as the upper bound. The **range** node for a predefined numeric type will have for its bounds expressions (determined by the implementation) which do not correspond to source code.

Constraints cannot be applied to a task type, therefore the question of whether or not it is constrained is irrelevent.

Section 4.4

TYPES AND SUBTYPES

4.4 TYPES AND SUBTYPES

In the Ada language certain types and subtypes may be declared in more than one way. For instance, the following sets of declarations produce equivalent subtypes:

```
type CONSTRAINED_AR is array (INTEGER range 1 .. 10) of BOOLEAN;

type INDEX is INTEGER range 1 .. 10;
type UNCONSTRAINED_AR is array (INDEX range <>) of BOOLEAN;
subtype CONSTRAINED_AR is UNCONSTRAINED_AR (INDEX);
```

the only difference being that the base type and index subtype corresponding to the first declaration are anonymous. In order to reconstruct the source text it is necessary that the syntax of the declarations be recorded; however, semantic processing would be facilitated if the same representation were used for CONSTRAINED_AR regardless of which set of declarations produced it. In order to satisfy both needs DIANA has two classes associated with types and subtypes.

The class TYPE_DEF records syntax. The nodes belonging to this class are not intended to be used for semantic processing, hence they have no semantic attributes, and are never designated by any kind of attribute other than a structural attribute. A **TYPE_DEF** node may correspond to:

(a) a subtype indication

(b) the portion of a type declaration following the reserved word "is"

(c) the type of an object as given in an object declaration

TYPE_DEF is subdivided into classes according to syntactic similarities of the source text which the nodes represent. The class structure of **TYPE_DEF** has no semantic meaning.

TYPE_DEF contains three nodes which are really intermediate nodes: **integer_def**, **float_def**, and **fixed_def**. The syntax of a numeric type definition consists solely of a range, floating point, or fixed point constraint. Unfortunately, the nodes representing these constraints are already members of class **CONSTRAINT** -- to include them in **TYPE_DEF** as well would have introduced multiple class memberships. Instead, three new nodes were introduced into

TYPE_DEF; each has a single structural attribute denoting the actual constraint.

Class **TYPE_SPEC** is the complement of **TYPE_DEF**; it represents the Ada concept of types and subtypes. The nodes comprising **TYPE_SPEC** are compact representations of types and subtypes, suitable for semantic processing. It is not necessary to traverse a lengthy chain of nodes in order to obtain all of the pertinent information concerning the type/subtype, nor are special cases (i.e. different structures) introduced by irrelevant syntactic differences. The nodes comprising class **TYPE_SPEC** do not record source text; they contain semantic attributes only, and are not accessible through structural attributes.

Because the **TYPE_SPEC** nodes are not designed to record source code, but are intended to represent the concept of types and subtypes, there is not necessarily a one-to-one correspondence between the types and subtypes declared in the source and the **TYPE_SPEC** nodes included in the DIANA tree. An implementation must represent each of the universal types (which cannot be explicitly declared in an Ada program), and additional nodes may be created to represent various anonymous types (to be described later). Consequently, it is not possibly to store the type specification information within the nodes denoting defining occurrences of types and subtypes. Thus the **type_id** and **subtype_id** nodes of class **DEF_NAME** represent the NAMES of types and subtypes, not the types and subtypes themselves. Access to the corresponding type specification is provided by means of the sm_type_spec attribute.

The construction of new **TYPE_SPEC** nodes for a DIANA tree is governed by two basic principles:

1. Each distinct type or subtype is represented by a distinct node from class **TYPE_SPEC**.

2. There are never two **TYPE_SPEC** nodes for the same type or subtype

These principles ensure that the only action needed to determine whether or not two entities have the same subtype or the same base type is the comparison of the associated **TYPE_SPEC** nodes. If both denote the same node (not equivalent nodes, but the SAME node) then the types are the same; if they reference different nodes then the types are not the same.

Since a type definition always creates a new type, a new **TYPE_SPEC** node is created for every type definition. This is not necessarily true for subtype declarations. If a subtype declaration does not include a constraint then it does not introduce a new subtype (it in effect renames the subtype denoted by the type mark), therefore a new **TYPE_SPEC** node is not introduced. In this case the sm_type_spec attribute of the **subtype_id** denotes the **TYPE_SPEC** node associated with the type mark.

All anonymous types described in the Ada Reference Manual are represented in DIANA; i.e.

(a) anonymous derived types created by numeric type definitions [ARM, sections 3.5.4, 3.5.7, and 3.5.9]

The nodes representing constrained types have an additional attribute, sm_depends_on_dscrmt, which indicates whether or not the component subtype depends on a discriminant. A subtype of a record component depends on a discriminant if it has a constraint which contains a reference to a discriminant of the enclosing record type. Within a record type definition, the only forms of constraints which can contain a reference to a discriminant are index and discriminant constraints. Since the only nodes for which this attribute could ever be true are the **constrained_array**, **constrained_record**, and **constrained_access** nodes, it was not necessary to define an sm_depends_on_dscrmt attribute for any other **TYPE_SPEC** nodes (although a component subtype may be a private type with a discriminant constraint, such a subtype is represented by a **constrained_record** node rather than a **PRIVATE_SPEC** node, as discussed in section 4.4.4).

The sm_depends_on_dscrmt attribute was defined because otherwise it would not be easy to determine whether or not a component subtype depended on a discriminant if the constraint were sufficiently complicated. This information is essential because at certain times a component subtype that depends on a discriminant is treated differently from one that does not. For instance, the elaboration time of a component subtype is determined by whether or not it depends on a discriminant. If the component subtype does not depend on a discriminant then it is elaborated when the enclosing record type is elaborated; otherwise the component subtype is not fully elaborated until a discriminant constraint is imposed on the enclosing record type (the expressions in the component subtype indication which are not discriminants are evaluated during the elaboration of the enclosing record type).

4.4.2 UNIVERSAL TYPES

The Ada programming language defines three universal types -- universal integer, universal real, and universal fixed. The **TYPE_SPEC** nodes for the universal types have no attributes of their own since their properties are fixed -- they are not implementation dependent, nor can they be declared by a user. For example, there is no need for the sm_is_anonymous attribute because universal types are always anonymous. The universal types are used as the types of named numbers and certain static expressions.

4.4.3 DERIVED TYPES

All types other than the universal types may be derived, although restrictions may be placed on the location of certain kinds of derived type definitions. For instance, a derived type definition involving a private type is not allowed within the package specification declaring that private type [ARM, 7.4.1]; however, that private type may be derived outside of the package specification. Hence the attribute sm_derived is defined for class **DERIVABLE_SPEC**. If a type is derived then sm_derived references the **TYPE_SPEC** node of the parent type (not the parent subtype); otherwise the attribute is **void**.

A derived type definition creates a new base type whose properties are derived from the parent type. In addition, it defines a subtype of the derived type. A derived type definition in DIANA always results in the creation of a new TYPE_SPEC node for the new base type. Since its characteristics are derived from the parent type it will need the same kinds of attributes in order to represent the appropriate values, thus the base type is represented by the same kind of node as the parent type.

If the parent type and the parent subtype of a derived type do not have the same constraints, then a new TYPE_SPEC node is created for the subtype of the derived type. This node will record the new constraint, and its base type will be the newly created base type. The name of the derived type will denote this subtype, hence all references to the derived type will denote the type specification of the subtype. As a result, the base type is anonymous.

If the base type is a record or enumeration type then a representation clause may be given for the derived type if a representation clause was not given for the parent type BEFORE the derived type definition. Hence it is possible for the derived type to have a different representation from that of the parent type. The information given in an enumeration representation clause is recorded in the nodes for the literals of the enumeration type; the information from the component clauses is encoded in the nodes for the components (including discriminants) of the record type. Due to the possibility of different representations, it is not always feasible for the derived type to share the enumeration literals or record components of the parent type.

DIANA requires that copies be made of the defining occurrences of the enumeration literals, unless the parent type is a generic formal discrete type, which does not have any literals. The new literals reference the derived type as the type to which they belong, but have the same position number as the corresponding original literals. If a representation clause is not given for the derived type then the sm_rep attribute will also have the same value. The node for the derived type denotes these new defining occurrences as its literals. Duplication has an additional advantage for enumeration literals -- since the literal of the derived type overloads the corresponding literal of the parent type, it is convenient to have two different defining occurrences when processing used occurrences of the literals.

DIANA also requires the duplication of the discriminant part and the component list for a derived record type if a representation clause is given for that type. If an implementation determines that no such clause is given, it can choose whether to copy the defining occurrences or reference the structure of the parent type. Because the defining occurrences do not reference the record type to which they belong, no inconsistencies are introduced if the structure is not copied when the representation does not change.

4.4.4 PRIVATE, LIMITED PRIVATE, AND LIMITED TYPES

A private type declaration separates the properties of the type that may be used outside of the package from those which are hidden from the user. A private type has two points of declaration -- the first declaration is the private one, occurring in the visible part of the package specification; the second is a full type declaration that appears in the private part of the package. Private and limited private types are represented by nodes from **PRIVATE_SPEC**, and complete type specifications by those belonging to **FULL_TYPE_SPEC**.

A (limited) private declaration introduces a **private** or **l_private** node, and the subsequent full type declaration introduces the appropriate node from class **FULL_TYPE_SPEC**. The (limited) private specification rather than the full type specification is referenced as the type of an object, expression, etc. In addition, all used occurrences of the type name will denote the **type_id** of the private declaration. The **PRIVATE_SPEC** nodes have an sm_type_spec attribute that provides access to the full type specification. In this way the distinction between private and full types is preserved for the kinds of semantic processing which require knowledge of whether or not a type is private, but the information recorded in the full type specification is available for the processing which needs it.

The specification of a (limited) private type may be viewed as being distributed over two nodes, rather than being represented by two different nodes. The full type specification can never be referenced as the type of an object, expression, etc., hence the principle that there are never two **TYPE_SPEC** nodes for the same type or subtype is not violated by this representation of (limited) private types.

An alternate solution was considered. It was proposed that all references to the (limited) private type occurring either outside of the package or within the visible part of the package denote the **PRIVATE_SPEC** node, and those references occurring within either the private part of the package or the package body denote the **FULL_TYPE_SPEC** node. Although there would be two **TYPE_SPEC** nodes for one type, within a given area (the two areas being the one in which the type structure is hidden and the one in which the type structure is visible) it would appear as if there were only one node. With this approach the uses of the type would reflect whether or not the structure of the type was visible at that point in the source code. Unfortunately, upon closer examination the previous assumptions proved to be untrue.

Consider the case of a subprogram declared in the visible part of a package. Suppose the subprogram has a parameter of a private type that is declared in the visible part of the same package. Although it is possible for the parameter in the subprogram declaration to denote the private specification as its type, and the parameter in the subprogram body declaration to denote the full type specification as its type, ALL references to both the subprogram and its parameters denote the first defining occurrences -- those in the package specification, which reference the private specification. Suppose an object of the private type were declared inside the subprogram body; it would refer to the full type specification as its type. The subprogram body would then contain a mixture of references to the private type -- some to the full type specification, others to the private specification. It would no longer be possible to simply compare **TYPE_SPEC** nodes to determine it two entities have the same type. As a consequence, this solution was rejected.

The **private** and **l_private** nodes always represent base types. Although a subtype of a (limited) private type may be introduced, it will be represented by a node from class **FULL_TYPE_SPEC** rather than one from **PRIVATE_SPEC**. Due to the restrictions placed on the creation of new TYPE_SPEC nodes, a new node may be created for such a subtype only if a new constraint is imposed upon it (in other words, the subtype is not a renaming of another type or subtype).

The kinds of constraints which may be imposed upon a (limited) private type are restricted in those regions where the full structure of the type is hidden. The structure (and therefore the class) of a private or limited private type without discriminants is not visible outside of the package or in the visible part of the package, therefore no new constraints may be imposed on such types in these regions. If a private type has discriminants then its full type must be a record type, and a discriminant constraint is permitted even in the locations where the structure of the rest of the record is unknown. That subtype is represented by a **constrained_record** node. If the declaration occurs within the private part of the package or the package body then the structure of the private type is visible, and the subtype is represented by the appropriate node from class **FULL_TYPE_SPEC**.

The **l_private** node represents types which are limited private, not types which are limited. Types which are limited include task types, composite types having a subcomponent which is limited, and types derived from a limited type. Because these types are not explicitly declared to be limited, they are not represented by a distinct node kind as the limited private types are (to do so would require semantic analysis to determine when the distinct node kind was appropriate). Instead, an additional attribute is defined where necessary.

Task types are always limited, hence there is no need to record that information in the form of an additional attribute. This is not true for arrays and records. Determining whether or not an array or record has any limited subcomponents could be a very time-consuming process if the structure of the composite type is very complicated. As a consequence, the sm_is_limited attribute was defined for the class **UNCONSTRAINED_COMPOSITE**. It has a boolean value indicating whether or not the type is limited. Since derived types are represented by the same kind of nodes as their parent types, the fact that a derived type is limited can be recorded in same way that it was recorded for the parent type.

On the surface it may seem that a problem similar to that discussed for composite types having limited components exists for composite types having private components. A composite type that has private subcomponents and is declared outside of the package containing the private type definition has certain restrictions placed on the operations allowed for the composite type. The only operations permitted are those which are dependent on the characteristics of the private declaration alone (see section 7.4.2 of the Ada Reference Manual).

A closer examination reveals that at most it is necessary to check a component type (as opposed to component types and subcomponent types) to determine if an operation is legal or not. The operations allowed for types which are composites of composites are also allowed for composite types with private components (assignment, aggregates, catenation, etc.). Operations involving the private component rather than the composite type as a whole may be restricted; for instance, a selected component involving a component of the private component is not allowed. Since the type of the private component is determined during type resolution of the sub-expression, no lengthy searches are required to determine that the component is private. Certain operations that are allowed for arrays of non-composite objects, such as the relational operators for arrays of scalar components and the logical operators for arrays of boolean components, would not be allowed under the circumstances described above because it would not be possible to determine if the component type were indeed a scalar type or a boolean type. However, such a check involves only a single component type.

A need could not be found for an attribute indicating that an array or record has private subcomponents; hence none was defined.

4.4.5 INCOMPLETE TYPES

An incomplete type definition allows the definition of "mutually dependent and recursive access types" [ARM, 3.8.1]. Like a private type, it has two points of definition: one for the incomplete type, and a second for the full type specification.

Although the uses of the name of an incomplete type are restricted when they occur before the end of the subsequent full type declaration (the name may appear only in the subtype indication of an access type definition), the incomplete type becomes an ordinary full type once its structure has been given. Since there is no need to distinguish the incomplete type from a full type once the structure of the full type is known, the solution adopted for private types is not appropriate for incomplete types. In general, incomplete types are not represented as such in DIANA; their full type specifications are represented by nodes from class FULL_TYPE_SPEC, and attributes denoting the incomplete type actually reference the full type specification.

Only one sort of incomplete type is represented by a distinct node in the DIANA tree. Included in the class TYPE_SPEC is the node **incomplete**, which was introduced to handle an anomoly in the Ada programming language. The language places the following restrictions on the placement of the full type declaration:

"If the incomplete type declaration occurs immediately within either a declarative part or the visible part of a package specification, then the full type declaration must occur later and immediately within the same declarative part or visible part. If the incomplete type declaraton occurs immediately within the private part of a package, then the full type declaration must occur later and immediately within either the private part itself, or the declarative part of the corresponding package body." [ARM, 3.8.1]

Because a package body may be in a separate compilation unit, it is possible for the full type specification of an incomplete type declared in the private part of a package to be in a separate compilation unit. In this case it is not possible for references to the incomplete type which occur in the package specification to denote the full type specification; DIANA forbids forward references of that sort. It was necessary to define a node to represent the incomplete type in this special case, hence the **incomplete** node was introduced. It has a single attribute, sm_discriminant_s, to represent any discriminants belonging to the incomplete type. If the full type specification is not in a different compilation unit the **incomplete** node is not used to represent the incomplete type.

This problem does not arise for private types. The Ada language requires that the full type specification of a private type be given in the private part of the package specification, thus it may never occur in a separate compilation unit.

DIANA does not specify the manner in which the full type specification may be accessed from the incomplete type specification in this special case -- to do so would impose restrictions on an implementation. All references to the specification of this incomplete type will reference the **incomplete** node, even those occurring after the full type declaration. Since an implementation must provide some solution to the problem of reaching the full type specification for references to the incomplete type occurring within the package specification, there seemed to be no reason to deviate from the DIANA requirement that all references to a particular entity denote the same node.

It should be noted that it is possible to reach the incomplete type specification from the subsequent full type declaration. The sm_first attribute of the **type_id** introduced by the full type declaration denotes the **type_id** of the incomplete type declaration. Both **type_id** nodes have an sm_type_spec attribute denoting their respective type specifications.

4.4.6 GENERIC FORMAL TYPES

Although "a generic formal type denotes the subtype supplied as the corresponding actual parameter in a generic instantiation" [ARM, 12.1.2], a generic formal type is viewed as being unique within the generic unit. Hence a new **TYPE_SPEC** node is introduced by each generic type declaration, and the attributes of the node are set as if the generic type were a new type.

A generic formal type is represented by the **DERIVABLE_SPEC** node that is appropriate for its kind. The values of the attributes are set in a manner which reflects the properties of the generic type within the generic unit. For instance, sm_base_type contains a self-reference, sm_derived is **void**, and sm_is_anonymous is false, regardless of the whether or not any of the actual subtypes have the same attribute values. A representation specification cannot be given for a generic formal type; this restriction is reflected in the values of all attributes which record representation information (sm_size is **void**, **sm_is_packed** is false, etc.).

Some of the attributes of a node representing a generic formal type may be undefined because they require knowledge of the actual subtype. Since there may be numerous instantiations it is not possible to set these attributes in the node representing the generic type. For example, a generic formal discrete type is represented by an **enumeration** node; the attributes sm_literal_s and sm_range are not defined because they depend on the actual subtype. The information recorded by such attributes is not necessary for the semantic processing of the generic type within the generic unit.

The TYPE_SPEC nodes corresponding to generic formal types contain no indication that they are indeed generic formal types. This information can be deduced from the context of the declarations and recorded by an implementation in whatever manner it finds to be most convenient.

4.4.7 REPRESENTATION INFORMATION

Representation specifications can be given for certain types and first named subtypes through pragmas and representation clauses. Although occurrences of these pragmas and representation clauses remain in the DIANA tree to enable the source to be reconstructed, they are additionally recorded with the TYPE_SPEC nodes corresponding to the type structures that they affect.

The occurrences of the language pragmas PACK and CONTROLLED are recorded with the attributes sm_is_packed (for array and record types) and sm_is_controlled (for access types).

A representation clause may be given for a record type, giving storage information for the record itself and/or its components; a reference to this specification is recorded in the semantic attributes of the TYPE_SPEC node representing the record type as well as the defining occurrences of the discriminants and components. Similarly for enumeration types, information from representation specifications for the enumeration literals is recorded with the defining occurrences of the enumeration literals.

Length clauses may be applied to various types. The presence of a length clause specifying the storage size for a task or access type is recorded with the sm_storage_size attribute. A length clause may also be used to place a limit on the number of bits allocated for objects of a particular type or first named subtype. A size specification is indicated by one of two different attributes, depending on the kind of type a particular node represents. The TYPE_SPEC nodes representing non-scalar types have an sm_size attribute which is of type EXP; it references the actual expression given in the length clause, and is **void** if no length clause is given.

TYPE_SPEC nodes for scalar types have a cd_impl_size attribute, which is of the private type **value**. Unlike the attributes corresponding to other kinds of representation clauses, cd_impl_size does not necessarily contain the value given in a length clause. It was introduced to facilitate the evaluation of static expressions. DIANA always records the value of static expressions; however, the static values of certain Ada attributes are implementation dependent. Since these attributes are related to static types, it is convenient to store this information with the node representing the type.

One such attribute is SIZE, which returns the actual number of bits required to store any object of that type. The value of this attribute is recorded with the cd_impl_size attribute, which has a value even if no length clause is given for the type. A length clause merely specifies the upper bound for the size of objects of that type, hence it is possible for the value of cd_impl_size to be smaller than that given in a length clause. Because the Ada programming language restricts static types to the scalar types, this implementation dependent attribute is not necessary for the nodes representing non-scalar types.

The other implementation dependent attribute defined in DIANA is the cd_impl_small attribute for nodes representing fixed point types. It contains the value to be returned for the Ada attribute SMALL. The user may specify in a length clause a value for "small" (the smallest positive model number for the type or first named subtype); this value is used in representing values of that fixed point type, and may affect storage requirements for objects of that type. If no length clause is given, then cd_impl_small will contain the value for "small" selected by the implementation; in this case "small" for the base type may differ from "small" for subtypes of that base type [ARM, 3.5.9].

Section 4.5

CONSTRAINTS

4.5 CONSTRAINTS

The Ada programming language defines five kinds of constraints: range, floating point, fixed point, index, and discriminant. Because constraints are generally imposed on types or subtypes, DIANA handles constraints in a manner that is similar to the way in which types and subtypes are treated.

There are both syntactic and semantic representations of certain constraints in DIANA. However, the differences between the two are not as rigidly observed for constraints as for types and subtypes. This is due to an effort to reduce the number of nodes in the DIANA tree, and to the fact that in many cases the syntactic representation of a constraint is also suitable for semantic processing.

As a result, there are not two distinct classes for representing constraints. In general, the class **CONSTRAINT** represents the syntax of the various constraints; there is no class defined to represent a semantic version of a constraint. Although certain **TYPE_SPEC** nodes (which represent the semantic concept of a type or subtype) define an attribute to denote a constraint, if a node from class **CONSTRAINT** is appropriate then it is referenced rather than requiring a new "semantic" structure to be built.

To facilitate the process of constraint checking, an effort was made to represent the constraints in DIANA in as consistent a manner as possible. The **CONSTRAINT** node is not always suitable for the following kinds of constraints: discriminant, floating point, fixed point, and index.

A discriminant constraint is a series of discriminant associations. The sequence of associations may contain a mixture of **EXP** and **assoc** nodes (i.e. expressions and named associations); if named associations are used then the associations do not even have to appear in the order in which the discriminants are declared. Thus an additional sequence, designated by the sm_normalized_dscrmt_s attribute of the **constrained_record** node, is created for a discriminant constraint. This sequence is a normalized version of the syntactic sequence -- all named associations are replaced by the associated expressions, in the order in which the corresponding discriminants are declared. In the interest of economy, if the discriminant constraint appears in the source text in the normalized form, then the record subtype specification may reference the same sequence of expressions that the discriminant constraint denotes.

A different problem arises for fixed or floating point constraints in **TYPE_SPEC** nodes. A type specification in DIANA records the applicable constraint. Because a fixed or floating point constraint contains two parts, either of which is optional in a subtype declaration, it is possible for the accuracy definition and the range constraint to be given in two different constraints. Thus it is not sufficient for a **REAL** node to reference a **REAL_CONSTRAINT** node. Instead, the accuracy definition and the range constraint are recorded by separate attributes (sm_accuracy and sm_range) in the **REAL** node. Though the type specification does not reference a **REAL_CONSTRAINT** node, it may possibly reference one or both of the constituents of a **REAL_CONSTRAINT** node.

The final kind of constraint to have an additional semantic representation is the index constraint. DIANA adheres to the semantics of the Ada language in its representation of arrays created by constrained array definitions. An index constraint for a **constrained_array** node introduced by a constrained array definition is a sequence of discrete type specifications; if an index subtype is given by a type mark then the type specification corresponding to the type mark appears at that index position. Otherwise, an anonymous subtype is created for that particular index position. To allow array subtypes to be treated in a uniform manner, the same approach is taken for the index constraints of all constrained array subtypes -- those introduced by subtype declarations, slices, aggregates, etc. (some of these may be anonymous). The new sequence is denoted by the sm_index_subtype_s attribute of the **constrained_array** node.

It may be necessary to make copies of some constraints. The Ada programming language allows multiple object declarations, which are equivalent to a series of single object declarations. If the multiple object declaration contains a constrained array definition then the type of each object is unique; if the declaration contains a subtype indication with a constraint, then the subtype of each object is unique. In either case, a new **TYPE_SPEC** node is created for each object in the identifier list. If the constraint is non-static then each type specification has a unique constraint. Because the constraint designated by the as_type_def attribute of the object declaration is not designed to be used for semantic processing, that constraint may be "shared" with one of the TYPE_SPEC nodes.

Due to structural similarities, the class **RANGE** represents both an Ada range and an Ada range constraint. The difference can always be determined from the context. If the **RANGE** node is introduced by an as_constraint attribute, as in the case of a numeric type definition or a subtype indication, then it represents a range constraint. Otherwise, it is a simple range (i.e. it is introduced by a loop iteration scheme, a membership operator, an entry declaration, a choice, or a slice). A **RANGE** node appearing DIRECTLY in a sequence of **DISCRETE_RANGE** nodes (corresponding to an index constraint) is also a simple range.

In order to avoid a multiple class membership for the class **RANGE**, which when representing a range constraint should belong to class **CONSTRAINT**, and when denoting a simple range should be a member of class **DISCRETE_RANGE**, the classes **CONSTRAINT** and **DISCRETE_RANGE** were merged. Consequently, **CONSTRAINT** is a combination of the Ada syntactic categories "constraint" and "discrete_range". By including **DISCRETE_RANGE** in class **CONSTRAINT**, the **discrete_subtype** node was introduced into the class representing constraints. It was therefore necessary to add a restriction in the semantic specification of DIANA prohibiting an attribute having the type **CONSTRAINT** from referencing a **discrete_subtype** node.

Discrete subtype indications are represented by the node **discrete_subtype**. Although discrete subtype indications are syntactically identical to any other kind of subtype indication, the **subtype_indication** node could not be included in class **DISCRETE_RANGE** because it is already a member of class **TYPE_DEF**; to do so would have introduced multiple membership for node **subtype_indication**. Hence the **discrete_subtype** node was introduced. It has an as_subtype_indication attribute which denotes the actual subtype indication, thus **discrete_subtype** serves as an intermediate node.

When a range constraint, a floating point constraint, or a fixed point constraint is imposed on a type or subtype, it is necessary to perform constraint checks to insure that the constraint is compatible with the subtype given by the type mark. Unfortunately, the information required to do this is not incorporated in the corresponding type specification. Although a **SCALAR** node does have an sm_base_type attribute, it does not necessarily denote the type specification corresponding to the type mark in the subtype indication (a scalar subtype is constructed from the BASE TYPE of the type mark, not the type mark itself).

To make the type specification corresponding to the type mark accessible from the type specification of the new subtype, the sm_type_spec attribute was defined for the classes **RANGE** and **REAL_CONSTRAINT**. Although a range constraint may be part of a floating point or fixed point constraint, it was not sufficient to add sm_type_spec to the **RANGE** node alone; the accuracy definition must be available as well. The definition of this attribute for both classes results in redundancy for the range constraints which are part of fixed or floating point constraints. The sm_type_spec attributes of the **REAL_CONSTRAINT** node and the corresponding **RANGE** node (if there is one) always denote the same type specification.

If the constraints are associated with a subtype indication then sm_type_spec denotes the type specification of the type mark; however, **RANGE** and **REAL_CONSTRAINT** nodes can appear in other contexts. For instance, both may appear in type definitions. The expressions for the bounds of a range constraint associated with a type definition are not required to belong to the same type, therefore it is not feasible for sm_type_spec to reference a previously defined type. In this case sm_type_spec designates the type specification of the new base type.

RANGE nodes representing (discrete) ranges rather than range constraints can appear as a part of slices, entry family declarations, loop iteration schemes, membership operators, index constraints, and choices. In each of these cases the bounds must be of the same type, hence sm_type_spec denotes the appropriate base type, as specified in the Ada Reference Manual. For example, the Ada Reference Manual states that "for a membership test with a range, the simple expression and the bounds of the range must be of the same scalar type" [ARM, 4.5.2]; therefore sm_type_spec for a **RANGE** node associated with a membership operator denotes the type specification of that scalar type.

A RANGE attribute is represented by a different kind of node from the other Ada attributes. Unlike the others (except for BASE, which is another special case), the RANGE attribute does not return an expression; thus the attributes sm_value and sm_exp_type (defined for the other kinds of Ada attributes) do not apply. In addition, the RANGE attribute does not appear in the same contexts as other Ada attributes. Consequently it is represented by a special **range_attribute** node.

Section 4.6

EXPRESSIONS

4.6 EXPRESSIONS

Expressions in a DIANA structure are represented by nodes from class **EXP**. **EXP** also contains the class **NAME** because certain names can appear in expressions.

The nodes representing expressions record both the syntax and the semantics of expressions; therefore nodes in this class contain both structural and semantic attributes, and may be denoted by both structural and semantic attributes.

There are two kinds of expressions which have a syntactic component that may vary: the membership operator may contain either a type mark or a range, and the allocator may contain either a qualified expression or a subtype indication. Unfortunately, in each case the variants do not belong to the same DIANA class, therefore a single attribute could not be defined to represent the syntactic component. The DIANA solution was to define two variants for each of these expressions, each variant having the appropriate structural attribute to record the syntax of that particular variant. The membership operator is represented by the **range_membership** and **type_membership** nodes, and the allocator is represented by the **qualified_allocator** and **subtype_allocator** nodes.

All DIANA nodes representing expressions have an sm_exp_type attribute; it denotes the subtype of the expression. The subtype is referenced rather than the base type because the type specification of the subtype contains both the applicable constraint AND a direct path to the specification of the base type. In this way, all nodes representing expressions contain the information necessary for semantic checking, constraint checking, etc. It should be noted that this does not imply that sm_exp_type can never denote a base type -- in the Ada programming language a type is not only a type, but an unconstrained subtype as well.

Some expressions can have static values (see section 4.9 of the Ada Reference Manual). The sm_value attribute was defined for nodes which can represent static expressions to permit the static value to be obtained without traversing any corresponding subtrees. If the value of an expression represented by a node having an sm_value attribute is not static, then sm_value must have a distinguished value indicating that it is not evaluated.

Due to syntactic similarities, various nodes in class **EXP** can represent entities other than expressions. The **selected** node not only represents selected components of records, it represents expanded names as well. The **indexed** node represents an indexed component; however, it may also denote a member of an entry family. These nodes have the attributes sm value and sm exp type because they can represent expressions; however, since these attributes are meaningless for anything other than an expression, they are undefined if the node does not represent an expression (an expanded name denoting an object or a literal is considered an expression).

4.6.1 EXPRESSIONS WHICH INTRODUCE ANONYMOUS SUBTYPES

Certain expressions (such as slices) impose a new constraint on a type. To enable the subtypes of expressions to be treated in a consistent manner, DIANA requires that anonymous subtypes be created for these expressions. The expressions which may introduce anonymous subtypes are slices, aggregates, string literals, and allocators.

Although a constraint for one of these expressions may be explicit, it is not necessarily different from an existing constraint. In the interest of efficiency, DIANA allows an existing type specification to be referenced as the subtype of the expression if can be STATICALLY determined that the constraints are identical; however, an implementation is free to create an anonymous subtype for each such expression if it finds that approach to be more convenient. For example, if it can be determined statically that a slice has the same bounds as the array prefix, then the **slice** node is allowed (but not required) to denote the type specification of the array prefix as its subtype. If it can be determined statically that an aggregate or string literal has the same constraints as the context type (which is required to be determinable from the context alone) then the type specification of the context type may be referenced as the subtype of the expression.

The anonymous subtype is constructed from the appropriate base type and the new constraint. The base type for a slice is obtained from the array prefix, and the constraint is the discrete range. The base type for an aggregate or a string literal is taken from the context; determining the constraint for these expressions is more complicated (the constraint is not necessarily explicit). Sections 4.2 and 4.3 of the Ada Reference Manual discuss this procedure in detail.

An allocator containing a subtype indication with an explicit constraint introduces an anonymous subtype. This subtype is not necessarily that of the object created by the allocator (for further details, see section 4.6.5); however, a new type specification is created for it because constraint checks must be performed to ensure that the constraint is compatible with the type mark.

Unlike the other expressions which create anonymous subtypes, the allocator does not introduce the anonymous subtype via the sm exp type attribute. Though an allocator creates a new object, it RETURNS an access value. The anonymous subtype is not the subtype of the access value returned by the allocator, hence it cannot be denoted by sm exp type. The sm desig type attribute was defined for allocators containing subtype indications; it denotes the type specification corresponding to the subtype indication (if an explicit constraint is not given then sm desig type references the type specification of the type mark).

4.6.2 FUNCTION CALLS AND OPERATORS

The Ada programming language allows operators (both predefined and user-defined) to be given in either prefix or infix form. In addition, a function can be renamed as an operator, and an operator renamed as a function. Consequently, the form of use of a function or operator implies nothing about whether the function or operator is predefined or user-defined. Since it serves no semantic purpose to distinguish function calls from operators, all function calls and operators are represented as function calls in a DIANA structure.

There are two exceptions to this method of representation: the short-circuit operators and the membership operators. These operators cannot be represented as functions, therefore they cannot be overloaded. Unlike the parameters of a function call, all of which are evaluated before the call takes place, the evaluation of the second relation of a short-circuit operator is dependent upon the result of the evaluation of the first relation. The second relation is not evaluated when the first relation of an "and then" operator is "true" or when the first relation of an "or else" operator is "false". A membership operator requires either a type mark or a range, neither of which is an expression, hence neither can be represented as a parameter. These operators are represented by the **short_circuit** and **MEMBERSHIP** nodes rather than **function_call** nodes.

The name of the function (a used occurrence) provides access to the defining occurrence of the function or operator, making it possible to determine the kind of function or operator represented by the **function_call**. The lx_prefix attribute records whether the call is given in prefix or infix form; this information is required for subprogram specification conformance rules (the default values of a parameter of mode "in" might be a function call or operator).

The subtype of a function call is considered to be the return type. If the function call is a predefined operator then the return type is the appropriate base type, as specified in section 4.5 of the Ada Reference Manual. This means that the subtypes of certain function calls may be unconstrained; for example, the result of a catenation is always of an unconstrained array subtype. Since it is not always possible to determine statically the constraints on a value returned by a function call, it is not feasible to require an anonymous subtype to be created for a call to a function with an unconstrained return type.

4.6.3 IMPLICIT CONVERSIONS

The Ada programming language defines various kinds of implicit type conversions, some of which are recorded in a DIANA structure, while others are not.

An implicit conversion of an operand of a universal type to a numeric type may be required for an operand that is a numeric literal, a named number, or an attribute. Although this implicit conversion is not recorded by the introduction of a distinct node, it is in a sense recorded by the value of the sm_exp_type attribute. If the context requires an implicit conversion of an operand of a universal type, then the sm_exp_type attribute of the

numeric_literal, **used_object_id**, or **attribute** node denotes the target type rather than the universal type.

By allowing the sm_exp_type attribute to reflect the result of the implicit conversion, all of the information necessary to perform the conversion is recorded in the node representing the literal, named number, or attribute; no additional context information is required. In addition, the fact that an expression is the operand of an implicit conversion can now be determined easily by a DIANA user. For instance, a numeric literal is the operand of an implicit conversion if sm_exp_type does not denote a universal type. If sm_exp_type did not reflect the conversion, then in any context in which an operand of a universal type would not be appropriate it would be necessary to check for the existence of a convertible universal operand. Since scalar operands can appear in numerous contexts that require non-universal types, a substantial amount of checking would be involved. The DIANA approach localizes the checking for an implicit conversion to the nodes which may represent convertible universal operands.

The semantics of the Ada language force the determination of the existence of an implicit type conversion during the semantic checking phase (an implicit conversion is applied only if the innermost complete context requires the conversion for a legal interpretation); recording information that is already available should not impose a hardship on an implementation. The **numeric_literal**, **used_object_id**, and **attribute** node all represent used occurrences; hence no conflicts should arise as a result of this representation of implicit conversions (i.e. the sm_obj_type attribute of the **number_id** still denotes a universal type).

As a result of the DIANA representation of implicit conversions, the used occurrences of a named number cannot always be represented by a single node, since the sm_exp_type attribute of the **used_object_id** may reflect an implicit conversion. However, a single used occurrence having a particular target type may represent all used occurrences of that named number requiring that particular type.

If the variable to the right of the assignment operator in an assignment statement is an array, then the expression to the right of the assignment operator is implicitly converted to the subtype of the array variable. This implicit subtype conversion may also be performed on the initial value in an array variable declaration. Many kinds of expressions produce anonymous array subtypes, which have a DIANA representation. Since this representation is introduced by the sm_exp_type attribute of the corresponding expression, the solution adopted for scalar operands is not suitable for arrays. Due to the fact that the implicit subtype conversion can occur only in two well-defined contexts, it was decided that it was not necessary to record the need for an implicit conversion.

Certain type conversions take place during a call to a derived subprogram. For formal parameters of the parent type the following conversions are performed: the actual parameters corresponding to parameters of mode "in" and "in out" are converted to the parent type before the call takes place; parameters of mode "in out" and "out" are converted to the derived type after the call takes place. If the result of a derived function is of the parent type then the result is converted to the derived type.

The conversion of parameters described above cannot be represented in the sequence of actual parameter associations corresponding to the source code without interfering with source reconstruction; however, these conversions could be incorporated into the normalized actual parameter list. It was decided not to record these conversions because the need for such a conversion is easily detected by comparing the base types of the formal and actual parameters. Since an implementation is already required to compare the (sub)types of formal and actual parameters to determine which constraint checks are needed, checking for the need for implicit conversions should impose no hardship. Requiring these conversions to be represented would force calls to derived subprograms to be treated as special cases when constructing a DIANA structure. The conversion of a return value of the parent type is not represented for the same reasons.

4.6.4 PARENTHESIZED EXPRESSIONS

Under some circumstances parentheses have a semantic effect in the Ada programming language. Consider the following procedure call:

```
P ( (A) );
```

The parentheses around the actual parameter "A" make it an expression rather than a variable, hence the corresponding formal parameter must be of mode "in", or the program containing this statement is in error. In addition, certain parentheses (such as those contained in default expressions for formal parameters of mode "in") must be preserved in order to perform conformance checks. Hence DIANA defines a **parenthesized** node. Not only does it contain a reference to the expression that it encloses, it records the value (if the value is static) and the subtype of that expression.

Certain kinds of processing are not affected by the presence or absence of parentheses. To allow the **parenthesized** node to be easily discarded as the DIANA is read in, a restriction was added to the semantic specification of DIANA: a semantic attribute which denotes an expression can never reference a **parenthesized** node; it must designate the node representing the actual expression instead. This principle also applies to sequences which are created expressly for semantic attributes and may contain expressions, such as the various normalized sequences. As a consequence of this restriction, a **parenthesized** node can be referenced by only one attribute -- a structural one. Since many of the semantic attributes were introduced as "shortcuts", it would be inappropriate for them to denote a **parenthesized** node anyway.

4.6.5 ALLOCATORS

The subtype of an object created by an allocator is determined in one of two ways, depending on the class of the object. The subtype of an array or a discriminated object is determined by the qualified expression, subtype indication, or default discriminant values. The subtype of any other kind of object is "the subtype defined by the subtype indication of the access type definition" [ARM, 4.8]; i.e. it is the subtype determined by the context (the Ada language requires this type to be determinable from the context alone).

As a result of these requirements, the sm exp type attribute of an allocator creating an object that is not an array or a discriminated object denotes the type specification of the context subtype. Unfortunately, the value of sm exp type is not as easily determined in the other case -- an appropriate subtype is not always available for the sm exp type attribute of an allocator creating an array or a discriminated object. Since an allocator can create an object with a unique constraint, a collection that is compatible with that object may not exist. Consider the following declarations:

```
type AC is access STRING(1..10);

FIVE : POSITIVE := 5;

OBJ : AC := new STRING(1..FIVE);
```

Although the initialization of OBJ will result in a constraint error, the declaration of OBJ is legal, and hence must be represented in the DIANA structure.

It may seem that it would be simple to make an anonymous subtype for this sort of allocator, just as anonymous subtypes are created for other kinds of expressions. But due to the way in which access types are constrained, the construction of an anonymous subtype cannot always be performed as it would be for other classes of types.

The anonymous subtypes for other expressions are constructed from the base type of the context type and the new constraint. The base type of an array or record type cannot have a constraint already imposed upon it (constrained array type definitions create anonymous unconstrained base types, and the syntax of a record type definition does not allow a constraint); therefore the imposition of a constraint on the base type does not cause an inconsistency.

The base type of an access type is not always unconstrained, nor does the Ada language define an anonymous unconstrained base type for a constrained access type. Associated with an access base type is a collection containing the objects which are referenced by access values of that type. If that base type is constrained (i.e. the designated subtype is constrained), then all of the objects in its collection must have the same constraints. It would be inappropriate to introduce an anonymous base type having an unconstrained designated subtype.

Unfortunately, this means that there is no existing type that would be an appropriate base for the anonymous subtype of the allocator in the previous example. The objects which may be referenced by OBJ and the object created by the evaluation of the allocator do not belong to the same collection, therefore they should not have the same base type. One solution would be to create an anonymous BASE type for the allocator; however, it cannot always be determined statically whether or not the object created by an allocator belongs to the collection of the context type. For instance, if the variable FIVE had the value 10 rather than 5, then it would be inconsistent to construct an anonymous base type for the allocator, since the object it creates belongs to the collection associated with AR.

It was decided that in the case of an allocator creating an array or a discriminated object the sm_exp_type attribute would denote the context subtype, just as it does for other kinds of allocators. Within the context of the allocator it can easily be determined what constraint checks need to be performed by comparing the subtype of the qualified expression or the subtype introduced by the allocator with the designated subtype of the context type.

4.6.6 AGGREGATES AND STRING LITERALS

The Ada programming language allows the component associations of an aggregate to be given in two forms: named and positional. If named associations are used then the associations do not necessarily appear in the same order as the associated components. To simplify subsequent processing of the aggregate, the **aggregate** node contains a normalized list of component associations.

Since records have a static number of components (the expression for a discriminant governing a variant part must be static in an aggregate), it is possible for the component associations to be replaced by a sequence of expressions in the order of the components to which they correspond.

Unfortunately, the associations of array aggregates are not necessarily static. In addition, it is not always desirable to replace a static range by the corresponding number of component expressions, particularly if the range is large. Hence the normalized list of component associations for an array aggregate does not necessarily consist of expressions alone (obviously all positional associations will remain as expressions in the normalized sequence).

A single component association may contain several choices. Since the component associations in the normalized sequence must be in the proper order, and since the original choices do not necessarily correspond to components which are contiguous (much less in the proper order), each component association containing more than one choice is decomposed into two or more associations. The normalized sequence does not correspond to source code, hence the only requirements imposed on the decomposition process are that the resulting associations be semantically equivalent to the original ones, and that each association be either the component expression itself or a named association having a single choice.

An "others" choice does not necessarily denote consecutive components, therefore it is treated as if it were an association with multiple choices. Each component or range of components represented by the "others" choice is represented by a component expression or a named association in the normalized sequence.

If a choice in an array aggregate is given by a simple expression, and it can be determined statically that the expression belongs to the corresponding index subtype then that association may be replaced by the component expression.

A subaggregate is syntactically identical to an aggregate, therefore it is represented in a DIANA structure by the same kind of node. The only problem arising from this representation is caused by the sm_exp_type attribute. A subaggregate is an aggregate corresponding to a sub-dimension of a multidimensional array aggregate. An aggregate corresponding to an array component or a record component is NOT a subaggregate. Since a subaggregate corresponds to a dimension rather than a component, it does not have a subtype. A subaggregate does, however, have bounds (although the bounds may be implicit, as specified in section 4.3.2 of the Ada Reference Manual). In order to correctly represent the subaggregate, the sm_discrete_range attribute was defined for the **aggregate** node; it denotes the bounds of the subaggregate, and is **void** for an aggregate that is not a subaggregate. The sm_exp_type attribute of a subaggregate is **void**.

A string literal is not syntactically like an aggregate, therefore it is represented by a **string_literal** node. However, a string literal may be a subaggregate if it occurs "in a multidimensional aggregate at the place of a one-dimensional array of a character type" [ARM, 4.3.2]. To accomodate this case, the **string_literal** and **aggregate** nodes were placed in the class **AGG_EXP**, and the sm_discrete_range attribute was defined for both nodes.

As previously stated, an aggregate may have an anonymous subtype. In most cases the constraints for the subtype are obtained from the aggregate itself with no conflict as to which constraints to use. However, in the case of an aggregate which contains more than one subaggregate for a particular dimension, the choice is not clear. To add to the confusion, the bounds of the subaggregates for a particular dimension are not necessarily the same. Though the Ada language requires a check to be made that all of the (n-1)-dimensional subaggregates of an n-dimensional multidimensional array aggregate have the same bounds, a program containing a violation of this condition is not in error; instead, a constraint error is raised when the aggregate is evaluated during execution.

DIANA does not specify which subaggregate the constraint for a particular dimension is taken from. If all of the subaggregates have the same bounds then it does not matter which is chosen. If the bounds are not the same then it still does not matter, since the constraint error will be detected regardless of which bounds are selected for the anonymous subtype.

4.7 PROGRAM UNITS

Numerous kinds of declarations exist for package and subprograms -- renaming declarations, generic instantiations, etc. The information peculiar to each kind of declaration must be accessible from the defining occurrence of that entity. Rather than have a different kind of defining occurrence with different attributes for each kind of declaration, DIANA has only one for a package and one for each kind of subprogram. Each such defining occurrence has an sm_unit_desc attribute which denotes a **UNIT_DESC** node that not only indicates the form of declaration, but records pertinent information related to the entity as well. The **UNIT_DESC** nodes for special kinds of package and subprogram declarations are discussed in detail in the following sections.

The defining occurrence of a package or subprogram that is introduced by an ordinary declaration does not denote a **UNIT_DESC** node defined exclusively for a particular kind of declaration. Instead, it denotes the body of the subprogram or package, if it is in the same compilation unit. Although this information is not vital for a defining occurrence that does not correspond to a body declaration, this "shortcut" may be used for optimization purposes.

4.7.1 RENAMED UNITS

The Ada programming language allows renaming declarations for packages, subprograms, and entries. These declarations introduce new names for the original entities. In a few special cases an entity may even be renamed as another kind of entity. A package or subprogram renaming declaration has the same DIANA structure as an ordinary package or subprogram declaration; the fact that it is a renaming is indicated by the as_unit_kind attribute, which denotes a **renames_unit** node.

If the entity is being renamed as the same kind of entity (i.e. a package is being renamed as a package, a procedure as a procedure, etc.) then uses of the new name will have the same syntactic structure as uses of the old name, and can appear in the same kinds of context. For instance, a used occurrence of the name of a function which is renamed as a function will appear as a function call within the context of an expression. The function call must be given in prefix form, just as a function call containing the old name must. A **function_id** can represent the new name without conveying any incorrect semantic information, and used occurrences of this name can refer to the **function_id** without introducing any inconsistencies in the DIANA tree.

In such cases the new name is represented by the same kind of **DEF_NAME** node as the original entity, the sm_unit_kind attribute of which denotes a **renames_unit** node. Because the defining occurrence represents a new name rather than a new entity, the remainder of the semantic attributes, except for sm_spec for a subprogram name, have the same values as those of the original entity. Since a new formal part is given in the renaming of a subprogram, the sm_spec attribute must denote the formal part corresponding to the new name. Access to the defining occurrence of the original unit is provided through the as_name attribute of the **renames_unit** node.

Entities which are renamed as other kinds of entities present special cases. Consider a function renamed as an operator. Although a used occurrence of the new name will still appear as a function call within the context of an expression, a function call using the new name may be given in either infix or prefix form. If a **function_id** were used to represent the new name rather than an **operator_id** then the information conveyed by the type of the defining occurrence node would not be correct. Though the entity is the same function, its new name must be viewed as the name of an operator. The same is true for an attribute renamed as a function -- though a used occurrence returns the value of the attribute, it will look like a function call, not an attribute.

An entry renamed as a procedure presents a different problem. The syntax for procedure calls and entry calls is identical; however, from a semantic perspective, call statements using the new name are procedure calls, not entry calls. A call statement containing the new name cannot be used for the entry call statement in a conditional or timed entry call, nor can it be the prefix for a COUNT attribute.

With the exception of an enumeration literal renamed as a function, all entities which are renamed as other kinds of entities are represented by the **DEF_NAME** node which is appropriate for the new name. Applicable attributes in the defining occurrence for the new name have the same values as the corresponding attributes in the original entity. For instance, the **operator_id** and **function_id** nodes have the same attributes, so that all semantic attributes except for sm_unit_kind and sm_spec may be copied. On the other hand, none of the semantic attributes in a **function_id** are applicable for an Ada attribute, hence they should have the appropriate values; i.e. sm_is_inline is false, sm_address is void, etc.

The only entity which can be renamed as another kind of entity without changing either the syntactic or the semantic properties associated with the use of the name is an enumeration literal that is renamed as a function. An enumeration literal and a parameterless function call have the same appearance, and there are no semantic restrictions placed on the use of the new name. The new name can be represented by an **enumeration_id**, and used occurrences can be denoted by **used_object_id** nodes which reference that **enumeration_id** (rather than a **function_id**) as the defining occurrence. The values of the semantic attributes of the new **enumeration_id** are copies of those of the original **enumeration_id**.

4.7.2 GENERIC INSTANTIATIONS

The Ada language defines a set of rules for an instantiation, specifying which entity is denoted by each kind of generic formal parameter within the generic unit. For example, the name of a generic formal parameter of mode "in out" actually denotes the variable given as the corresponding generic actual parameter. An obvious implementation of generic instantiations would copy the generic unit and substitute the generic actual parameters for all uses of the generic formal parameters in the body of the unit; however, this substitution cannot be done if the body of the generic unit is compiled separately. In addition, a more sophisticated implementation may try to optimize instantiations by sharing code between several instantiations. Therefore the body of a generic unit is not copied in DIANA in order to avoid constraining an implementation and to avoid introducing an inconsistency in the event of a separately compiled body.

Generic formal parameters may appear in the specification portion of the generic unit; for instance, a formal parameter of a generic subprogram may be declared to be of a generic formal type. The specification portion of the instantiated unit will necessarily be involved in certain kinds of semantic processing whenever the instantiated unit or a part of its specification is referenced. For example, when that instantiated subprogram is called it is necessary to know the types of its parameters. Semantic processing would be facilitated if the entities given in the specification could be treated in a "normal" fashion; i.e. it is desirable that the appropriate semantic information be obtainable without a search for the generic actual parameter every time semantic information is needed. Because there may be numerous instantiations of a particular generic unit, it is not possible to simply add an additional attribute to the defining occurrences of the generic formal parameters in order to denote the corresponding actual parameters.

DIANA provides a solution in two steps, the first of which is the addition of a normalized list of the generic parameters, including entries for all default parameters. Within this sequence (sm_decl_s of the **instantiation** node) each parameter entry is represented by a declarative node which does not correspond to source code. Each declarative node introduces a new defining occurrence node; the name (lx_symrep) corresponds to the formal parameter, however, the values of the semantic attributes are determined by the actual parameter as well as the kind of declarative node introducing the defining occurrence.

After the normalized declaration list has been created the specification part of the generic unit is copied. Every reference to a generic formal parameter in the original generic specification is changed to reference the corresponding newly created defining occurrence. Since each **DEF_NAME** node contains the appropriate semantic information, specifications of instantiated units do not have to be treated as special cases.

A **DEF_NAME** node introduced by one of these special declarative nodes is not considered to be an additional defining occurrence of the generic formal parameter; should a defining occurrence that is introduced by such a declarative node have an sm_first attribute, it will reference itself, not the node for the formal parameter.

Since this list of declarative nodes is a normalized list, all of the object declarations which appear in it are SINGLE declarations, even though the generic formal parameter may have been declared originally in a multiple object declaration. The kind of declarative node created for a generic formal parameter is determined by the kind of parameter as well as by the entity denoted by the parameter.

The name of a formal object of mode "in" denotes "a constant whose value is a copy of the value of the associated generic actual parameter" [ARM, 12.3]. Thus a formal object of mode "in" is represented by a constant declaration in the normalized parameter list. The initial value is either either the actual parameter or the default value, and the subtype of the constant is that of the actual parameter.

The name of a formal object of mode "in out" denotes "the variable named by the associated actual parameter" [ARM, 12.3]. Hence a formal object of mode "in out" appears in the normalized parameter list as a renaming declaration in which the renamed object is the actual parameter. The values of the attributes of the new **variable_id** are determined just as they would be for an ordinary renaming.

The declarative nodes for both constant and variable declarations have an attribute for the type of the object being declared. Unfortunately, as_type_def is normally used to record syntax, but because the declarative node does not correspond to source code, there is no syntax to record. A possible solution would be for as_type_def to reference the TYPE_DEF structure belonging to the declaration of the actual parameter; however, this structure is not always appropriate. If the context of the declaration of the actual parameter and that of the instantiation is not the same, then an expanded name rather than a simple name might be required in the TYPE_DEF structure for the special declarative node. Rather than force an implementation to construct a new TYPE_DEF structure in order to adhere to the Ada visibility rules, DIANA allows the value of as_type_def in an OBJECT_DECL node generated by an instantiation to be undefined. Since these declarative nodes are introduced to facilitate semantic processing, not to record syntax, this solution should not cause any problems. Declarative nodes for objects in the copy of the specification are treated in the same manner.

The name of a formal type denotes "the subtype named by the associated generic actual parameter (the actual subtype)" [ARM, 12.3]. A generic formal type is represented in the normalized list by a subtype declaration. The name in the subtype indication corresponds to the generic actual parameter, and the subtype indication does not have a constraint, hence the declaration effectively renames the actual subtype as the formal type. The sm_type_spec attribute of the **subtype_id** references the TYPE_SPEC node associated with the actual parameter.

The name of a formal subprogram denotes "the subprogram, enumeration literal, or entry named by the associated generic actual parameter (the actual subprogram)" [ARM, 12.3]. A generic formal subprogram appears in the normalized list as a renaming declaration in which the newly created subprogram renames either the subprogram given in the association list or that chosen by the analysis as the default. The values of the attributes of the new DEF_NAME node are determined just as they would be for an ordinary renaming, with the exception of the HEADER node, which is discussed in one of the subsequent paragraphs.

References to generic formal parameters are not the only kind of references that are replaced in the copy of the generic specification. Substitutions must also be made for references to the discriminants of a generic formal private type, and for references to the formal parameters of a generic formal subprogram.

The name of a discriminant of a generic formal type denotes "the corresponding discriminant (there must be one) of the actual type associated with the generic formal type" [ARM, 12.3]. If a formal type has discriminants, references to them are changed to designate the corresponding discriminants of the base type of the newly created subtype (i.e. the base type of the actual type). Since the new **subtype_id** references the type specification of the actual subtype, any direct manipulation of the **subtype_id** will automatically access the correct discriminants.

The name of a formal parameter of a generic formal subprogram denotes "the corresponding formal parameter of the actual subprogram associated with the formal subprogram" [ARM, 12.3]. If a formal subprogram has a formal part, the declarative node and defining occurrence node for the newly created subprogram reference the **HEADER** node of the the actual subprogram. Any references to a formal parameter are changed in the copy of the generic unit specification to denote the corresponding formal parameter of the actual subprogram.

Consider the following example:

```
procedure EXAMPLE is

   OBJECT : INTEGER := 10;

   function FUNC ( DUMMY : INTEGER ) return BOOLEAN is
      begin
         return TRUE;
      end FUNC;

   generic
      FORMAL_OBJ : INTEGER;
      with function FORMAL_FUNC( X : INTEGER ) return BOOLEAN;
   package GENERIC_PACK is

      PACK_OBJECT : BOOLEAN := FORMAL_FUNC ( X => FORMAL_OBJ );

   end GENERIC_PACK;

   package body GENERIC_PACK is separate;

   package NEW_PACK is new GENERIC_PACK ( FORMAL_OBJ => OBJECT,
                                          FORMAL_FUNC => FUNC );

begin
   null;
end EXAMPLE;
```

If a DIANA structure were created for package EXAMPLE, then the normalized parameter list for package NEW_PACK would contain two declarative nodes. The first would be a constant declaration for a new FORMAL_OBJ, which would be initialized with the INTEGER value 10. The second would be a renaming declaration for a new FORMAL_FUNC; the original entity would be FUNC, and the header of the new FORMAL_FUNC would actually be that of FUNC. The specification for package NEW_PACK would be a copy of that of GENERIC_PACK; however, the references to FORMAL_FUNC and FORMAL_OBJ would be changed to references to the newly declared entities, and the reference to X would be changed to a reference to DUMMY.

4.7.3 TASKS

The definition of the Ada programming language specifies that "each task depends on at least one master" [ARM, 9.4]. Two kinds of direct dependence are described in the following excerpt (section 9.4) from the Ada Reference Manual:

(a) The task designated by a task object that is the object, or a subcomponent of the object created by the evaluation of an allocator depends on the master that elaborates the corresponding access type definition.

(b) The task designated by any other task object depends on the master whose execution creates the task object.

Because of the dynamic nature of the second kind of dependency, DIANA does not attempt to record any information about the masters of such task objects. The first kind of dependency, however, requires some sort of information about the static nesting level of the corresponding access type definition; hence the sm_master attribute was added to the type specification of access types. Its value is defined only for those access types which have designated types that are task types. This attribute provides access to the construct that would be the master of a task created by the evaluation of an allocator returning a value of that particular access type.

A master may be one of the following:

(a) a task

(b) a currently executing block statement

(c) a currently executing subprogram

(d) a library package

A problem arose over the type of sm_master -- there is no one class in DIANA that includes all of these constructs. The class ALL_DECL contains declarative nodes for tasks, subprograms, and packages; therefore it seemed appropriate to add a "dummy" node representing a block statement to this class. The **block_master** node, which contains a reference to the actual block statement, was added to ALL_DECL at the highest possible level, so that it would not be possible to have **block_master** nodes appearing in declarative parts, etc. Only one attribute (as_all_decl) other than sm_master has the class ALL_DECL as its type; restrictions on the value of this attribute were added to the semantic specification.

4.7.4 USER-DEFINED OPERATORS

The Ada programming language allows the user to overload certain operators by declaring a function with an operator symbol as the designator. Because these user-declared operators have user-declared bodies, etc., they are represented by a different kind of node from the predefined operators. The predefined operators are represented by **bltn_operator_id** nodes, which do not have the facility to record all of the information needed for user-defined operators. A user-defined operator is represented by an **operator_id** node; it has the same set of attributes as the **function_id**.

A special case arises for the inequality operator. The user is not allowed to explicitly overload the inequality operator; however, by overloading the equality operator, the user IMPLICITLY overloads the corresponding inequality operator. The result returned by the overloaded inequality operator is the complement of that returned by the overloaded equality operator.

Since the declaration of the overloaded inequality operator is implicit, the declaration is not represented in the DIANA tree (to do so would interfere with source reconstruction). At first glance it may seem that a simple implementation of the implicitly declared inequality operator would be to replace all uses of the operator by a combination of the "not" operator and the equality operator (i.e. "X /= Y" would be replaced by "not (X = Y)"). While this approach may be feasible for occurrences of the inequality operator within expressions, it will not work for occurrences in other contexts. For instance, this representation would not be appropriate for a renaming of the implicitly declared inequality operator, or for an implicitly declared operator that is used as a generic actual parameter.

In order for used occurrences (**used_name_id** nodes) to have a defining occurrence to reference, the implicitly declared inequality operator is represented by an **operator_id**. Unfortunately, this operator does not have a header or a body to be referenced by the attributes of the **operator_id**; some indication that this operator is a special case is needed. Thus the **implicit_not_eq** node was defined. Instead of referencing a body, the as_unit_desc attribute of an **operator_id** corresponding to an implicitly declared inequality operator denotes an **implicit_not_eq** node, which provides access to the body of the corresponding equality operator. The as_header attribute of the **operator_id** designates either the header of the corresponding equality operator, or a copy of it.

4.7.5 DERIVED SUBPROGRAMS

A derived type definition introduces a derived subprogram for each subprogram that is an operation of the parent type (i.e. each subprogram having either a parameter or a result of the parent type) and is derivable. A subprogram that is an operation of a parent type is derivable if both the parent type and the subprogram itself are declared immediately within the visible part of the same package (the subprogram must be explicitly declared, and becomes derivable at the end of the visible part). If the parent type is also a derived type, and it has derived subprograms, then those derived subprograms are also derivable.

The derived subprogram has the same designator as the corresponding derivable subprogram; however, it does not have the same parameter and result type profile. It should be noted that it would be possible to perform semantic checking without an explicit representation of the derived subprogram. All used occurrences of the designator could reference the defining occurrence of the corresponding derivable subprogram. When processing a subprogram call with that designator, the parameter and result type profile of the derivable subprogram could be checked. If the profile of the derivable subprogram was not appropriate, and a derived type was involved, then a check could be made to see if the subprogram was derivable for that particular type (i.e. that a derived subprogram does exist).

Unfortunately, the circumstances under which a derived subprogram is created are complex; it would be very difficult and inefficient to repeatedly calculate whether or not a derived subprogram existed. Hence derived subprograms are explicitly represented in DIANA. The appropriate defining occurrence node is created, and the sm_unit_desc attribute denotes a **derived_subprog** node, thereby distinguishing the derived subprogram from other kinds of subprograms. Once the new specification has been created, the derived subprogram can be treated as any other subprogram is treated; it is no longer a special case.

The specification of the derived subprogram is a copy of that of the derivable subprogram, with substitutions made to compensate for the type changes. As outlined in section 3.4 of the Ada Reference Manual, all references to the parent type are changed to references to the derived type, and any expression of the parent type becomes the operand of a type conversion that has the derived type as the target type. The specification of the derived subprogram deviates from the specification described in the Ada Reference Manual in one respect. The manual states that "any subtype of the parent type is likewise replaced by a subtype of the derived type with a similar constraint" [ARM, 3.4]. If this suggestion were followed, both an anonymous subtype and a new constraint would have to be created. Fortunately, both the requirements for semantic checking and the semantics of calls to the derived subprogram allow a representation which does not require the construction of new nodes (or subtypes).

All references to subtypes of the parent type are changed to references to the derived type in the specification of the derived subprogram. Because semantic checking requires only the base type, this representation provides all of the information needed to perform the checks. A call to a derived subprogram is equivalent to a call to the corresponding derivable subprogram, with appropriate conversions to the parent type for actual parameters and return values of the derived type. Though the derived subprogram has its own specification, it does not have its own body, thus the the type conversions described in section 3.4 of the Ada Reference Manual are necessary. In addition to performing the required type conversions to the parent type, an implementation could easily perform conversions to subtypes of the parent type when appropriate, thereby eliminating the need to create an anonymous subtype of the derived type. The **derived_subprog** node provides access to the defining occurrence of the corresponding derivable subprogram (and hence to the types and subtypes of its formal parameters).

Although the defining occurrence of a derived subprogram is represented in DIANA, its declaration is not, even though the Ada Reference Manual states that the implicit declaration of the derived subprogram follows the declarations of the operations of the derived type (which follow the derived type declaration itself). Consequently the defining occurrence of a derived subprogram can be referenced by semantic attributes alone.

4.8 PRAGMAS

The Ada programming language allows pragmas to occur in numerous places, most of which may be in sequences (sequences of statements, declarations, variants, etc.). To take advantage of this fact, several DIANA classes have been expanded to allow pragmas -- in particular, those classes which are used as sequence element types and which denote syntactic constructs marking places at which a pragma may appear. For instance, the class **STM_ELEM** contains the node **stm_pragma** and the class **STM**. All constructs which are defined as sequences of statements in the Ada syntax are represented in DIANA by a sequence containing nodes of type **STM_ELEM**.

The approach taken for the representation of comments could have been applied to pragmas; i.e. adding an attribute by which pragmas could be attached to each node denoting a construct that could be adjacent to a pragma. This approach has two disadvantages: there is a need to decide if a pragma should be associated with the construct preceding it or the one following it; and the attribute is "wasted" when a pragma is not adjacent to the node (which will be the most common case). Since the set of classes needing expansion is a small subset of the DIANA classes, it was decided to allow the nodes representing pragmas to appear directly in the associated sequences, exactly as given in the source.

The **pragma** node could not be added directly to each class needing it without introducing multiple membership for the **pragma** node. Since the DIANA classes are arranged in a hierarchy (if one excludes class the node **void**) such a situation would be highly undesirable. Instead, the **pragma** node is included in class **USE_PRAGMA**, which is contained in class **DECL**, and an intermediate node is included in the other classes. This intermediate node has an as_pragma attribute denoting the actual **pragma** node. The **stm_pragma** node mentioned at the beginning of this section is an intermediate node.

Sequences of the following constructs may contain pragmas:

(a) declarations (**decl_s** and **item_s**)

(b) statements (**stm_s**)

(c) variants (**variant_s**)

(d) select alternatives (**test_clause_elem_s**)

(e) case statement alternatives (**alternative_s**)

(f) component clauses (**comp_rep_s**)

(g) context clauses (**context_elem_s**)

(h) use clauses (**use_pragma_s**)

Unfortunately pragmas do not ALWAYS appear in sequences. In a few cases it was necessary to add an as_pragma_s attribute to nodes representing portions of source code which can contain pragmas. These cases are discussed in the following paragraphs.

The **comp_list** node (which corresponds to a component list in a record type definition) has an as_pragma_s attribute to represent the pragmas occurring between the variant part and the end of the record type definition (i.e. between the "end case" and the "end record").

The **labeled** node, which represents a labeled statement, has an as_pragma_s attribute to denote the pragmas appearing between the label or labels and the statement itself.

Pragmas may occur before an alignment clause in a record representation clause (i.e. between the "use record" and the "at mod"), hence the **alignment_clause** node also has an as_pragma_s attribute. If a record representation clause does not have an alignment clause then a pragma occurring after the reserved words "use record" is represented by an intermediate **comp_rep_pragma** node in the **comp_rep_** sequence (in this case the **comp_rep_s** sequence will have to be constructed whether any component clauses exist or not).

Finally, the **compilation_unit** node defines an as_pragma_s attribute which denotes a non-empty sequence in one of two cases. A compilation may consist of pragmas alone, in which case the as_pragma_s denotes the pragmas given for the compilation, and the other attributes are empty sequences or void.

If the compilation contains a compilation unit then as_pragma_s represents the pragmas which follow the compilation unit and are not associated with the following compilation unit (if there is a compilation unit following it at all). INLINE and INTERFACE pragmas occurring between compilation units must be associated with the preceding compilation unit according to the rules of the Ada programming language. LIST and PAGE pragmas may be associated with either unit unless they precede or follow a pragma which forces an association (i.e. a LIST pragma preceding an INLINE pragma must be associated with the previous compilation unit, since pragmas in DIANA must appear in the order given, and the INLINE pragma belongs with the previous unit). These four pragmas are the only ones which may follow a compilation unit.

Certain pragmas may be applied to specific entities. Although the presence of these pragmas must be recorded as they occur in the source (to enable the source to be constructed), it would be convenient if the information that they conveyed were readily available during semantic processing of the associated entity. Hence DIANA defines additional attributes to record pertinent pragma information in the nodes representing defining occurrences of certain entities to which pragmas may be applied. The following pragmas have corresponding semantic attributes:

(a) CONTROLLED
sm_is_controlled in the **access** node

(b) INLINE
sm_is_inline in the **generic_id** and **SUBPROG_NAME** nodes

(c) INTERFACE
sm_interface in the **SUBPROG_NAME** nodes

(d) PACK
sm_is_packed in the **UNCONSTRAINED_COMPOSITE** nodes

(e) SHARED
sm_is_shared in the **variable_id** node

Although it may seem that the pragmas OPTIMIZE, PRIORITY, and SUPPRESS should also have associated attributes, they do not. Each of these pragmas applies to the enclosing block or unit. The information conveyed by the OPTIMIZE and PRIORITY pragmas could easily be incorporated into the DIANA as it is read in. The SUPPRESS pragma is more complicated -- not only is a particular constraint check specified, but the name of a particular entity may be given as well. SUPPRESS is too dependent upon the constraint checking mechanism of an implementation to be completely specified by DIANA; in fact, the omission of the constraint checks is optional.

CHAPTER 5

EXAMPLES

This chapter consists of examples of DIANA structures. Each example contains a segment of Ada source code and an illustration of the resulting DIANA structure. Each node is represented by a box, with its type appearing in the upper left-hand corner. Structural attributes are represented as labeled arcs which connect the nodes. All other kinds of attributes appear inside the node itself; code and semantic attributes are represented by a name and a value, while lexical attributes representing names or numbers appear as strings (inside of quotes). All sequences are depicted as having a header node, even if the sequence is empty. If the copying of a node is optional, it is NOT copied in these examples.

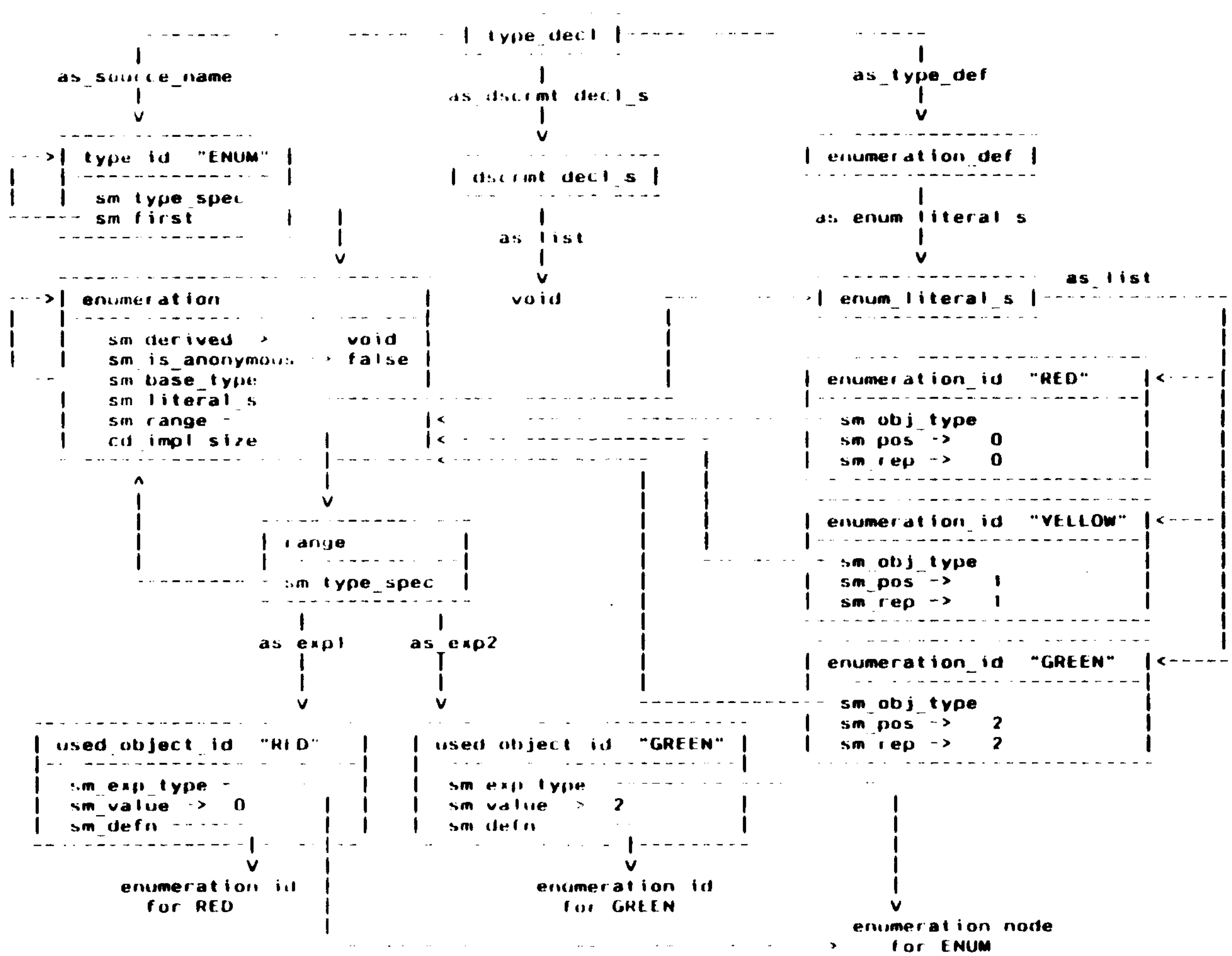
These illustrations DO NOT imply that all DIANA representations of these particular Ada code segments must consist of the same combination of nodes and arcs. For instance, an implementation is not required to have a heade node for a sequence. The format for these examples was selected because it seemed to be the most straightforward and easy to understand.

In certain instances an arc may point to a short text sequence describing the node that is referenced rather than pointing to the node itself. This is done for any of the following reasons:

- o the node is pictured in an example on another page
- o the node is not pictured in any of the examples
- o the node represents a predefined entity which cannot be depicted because it is implementation-dependent
- o the node is on the same page, but pointing to it would cause arcs to cross and result in a picture that would be difficult to understand

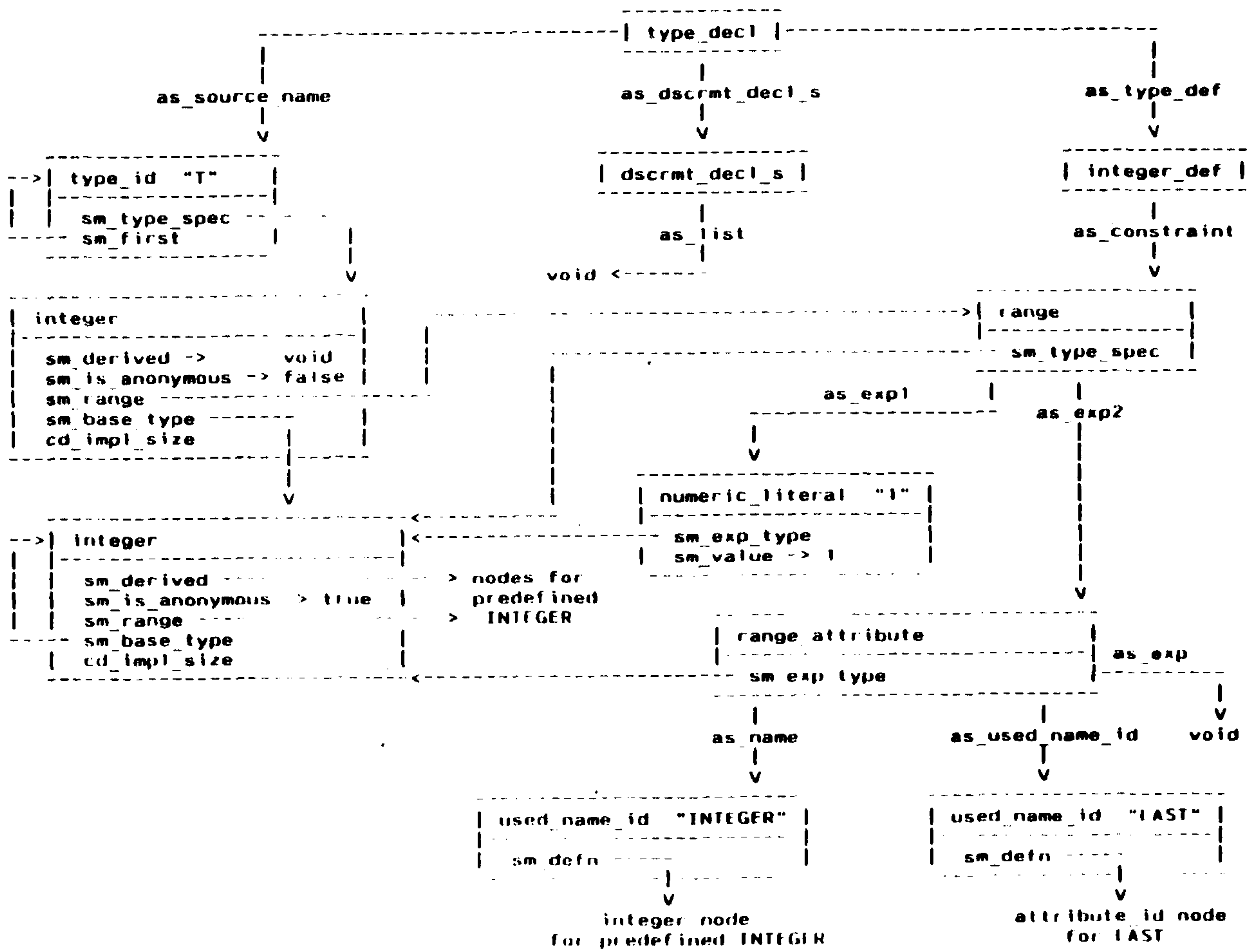
LIST OF EXAMPLES

type ENUM is (RED, YELLOW, GREEN);



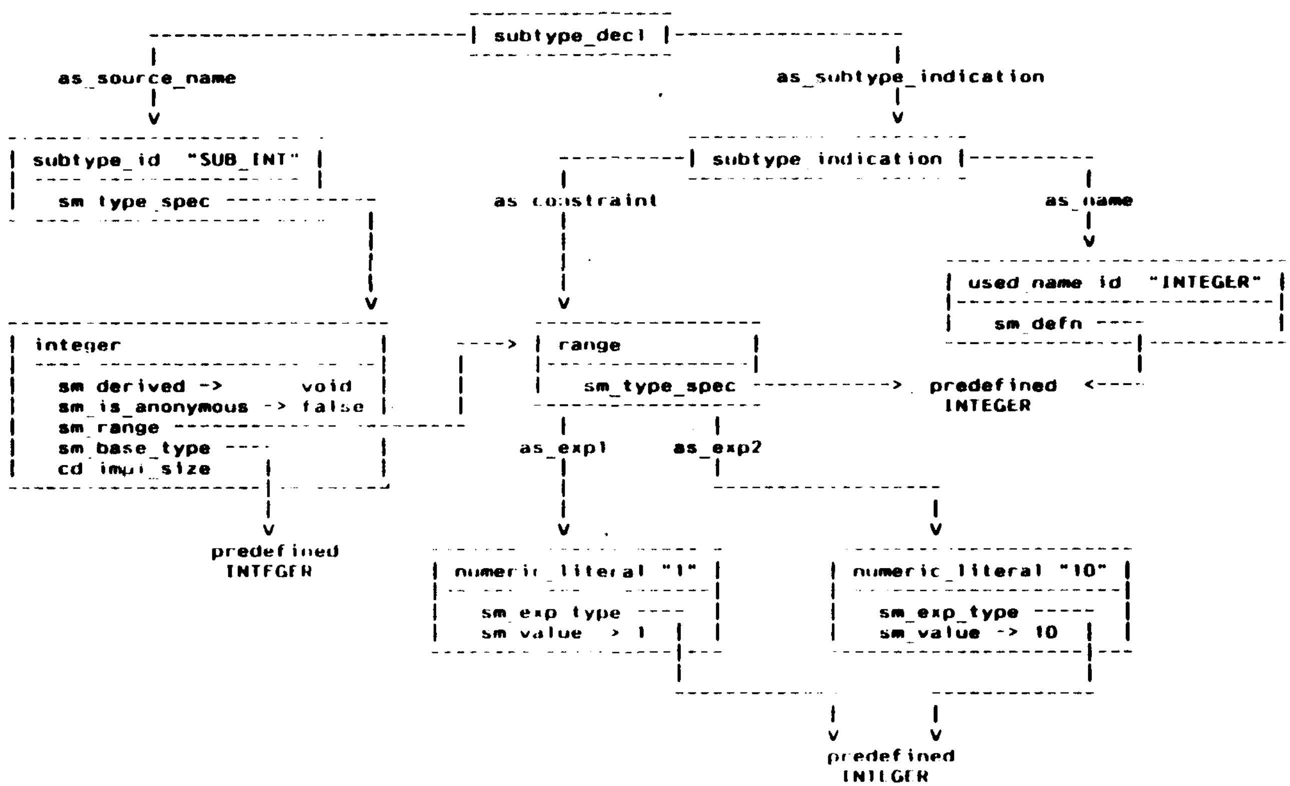
Example 1 Enumeration Type Definition

type T is range 1 .. INTEGER ' LAST;



Example 2 Integer Type Definition

subtype SUB_INT is INTEGER range 1..10;



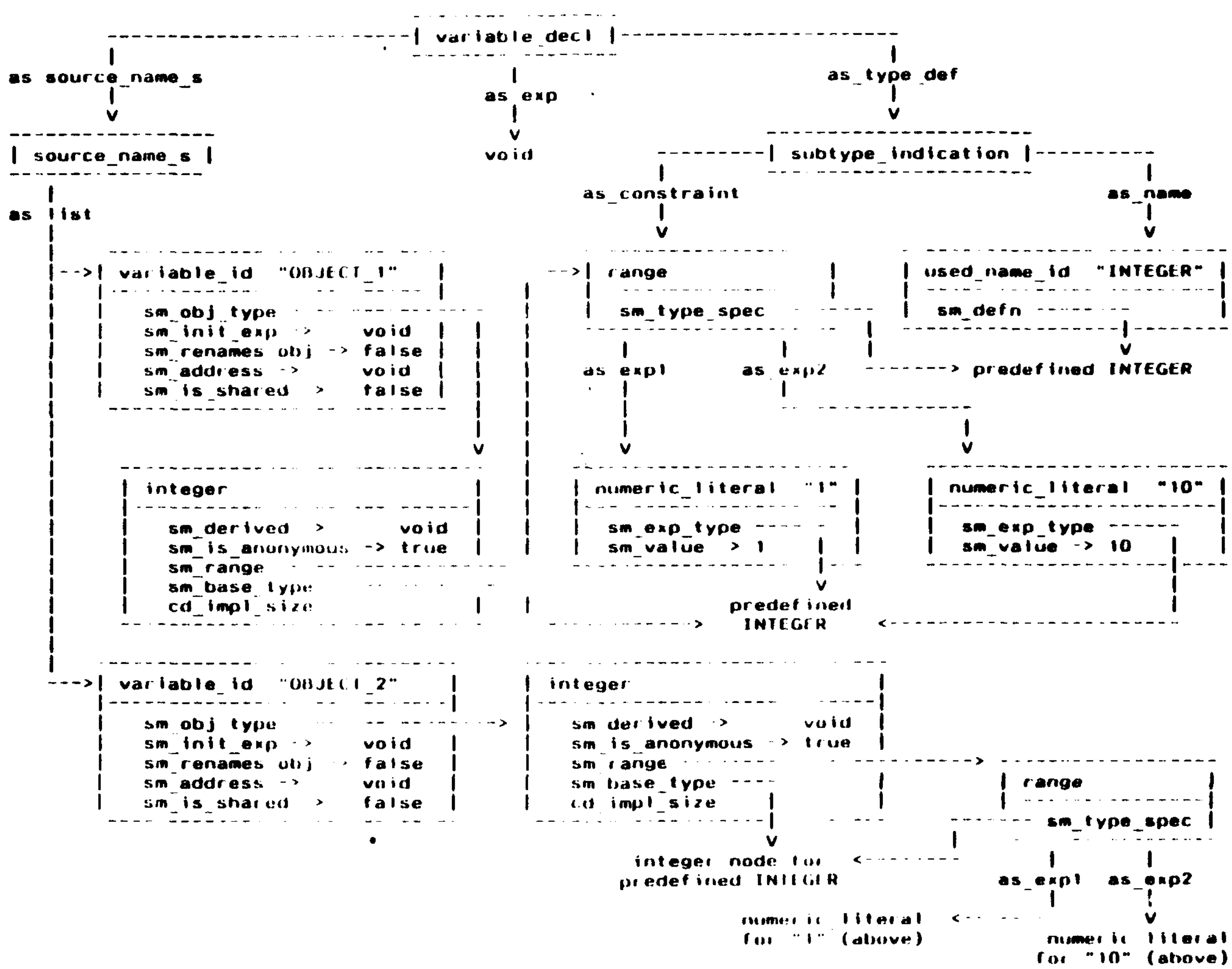
Example 3a Subtype Declaration

OBJECT_1, OBJECT_2 : SUB_INT;

```
           ------------------------------ | variable_decl |------------------------------
           |                                      |                                      |
   as_source_name_s                            as_exp                               as_type_def
           |                                      |                                      |
           V                                      V                                      V
   ---------------------                        void                      --------------------------
   | source_name_s |                                          ------------| subtype_indication |---------
   ---------------------                                      |           --------------------------     |
           |                                            as_constraint                                 as_name
        as_list                                               |                                        |
           |                                                  V                                        V
           |     ---------------------------------          void                    ---------------------------------
           |-->| variable_id   "OBJECT_1"     |                                    | used_name_id   "SUB_INT" |
           |   |--------------------------------|                                    |-------------------------------|
           |   |   sm_obj_type  -----------------------------|                       |   sm_defn -------           |
           |   |   sm_init_exp  ->     void   |              |                       ------------|------------------
           |   |   sm_renames_obj  ->  false  |              |                                   V
           |   |   sm_address  ->      void   |              |                       subtype_id node for
           |   |   sm_is_shared  >     false  |              |                            SUB_INT
           |   ---------------------------------             |
           |                                                 |------>  integer node for
           |                                                 |------>      SUB_INT
           |     ---------------------------------           |          in Example 3a
           --->| variable_id   "OBJECT_2"     |              |
               |--------------------------------|            |
               |   sm_obj_type  -----------------------------|
               |   sm_init_exp  ->     void   |
               |   sm_renames_obj  ->  false  |
               |   sm_address  ->      void   |
               |   sm_is_shared  >     false  |
               ---------------------------------
```

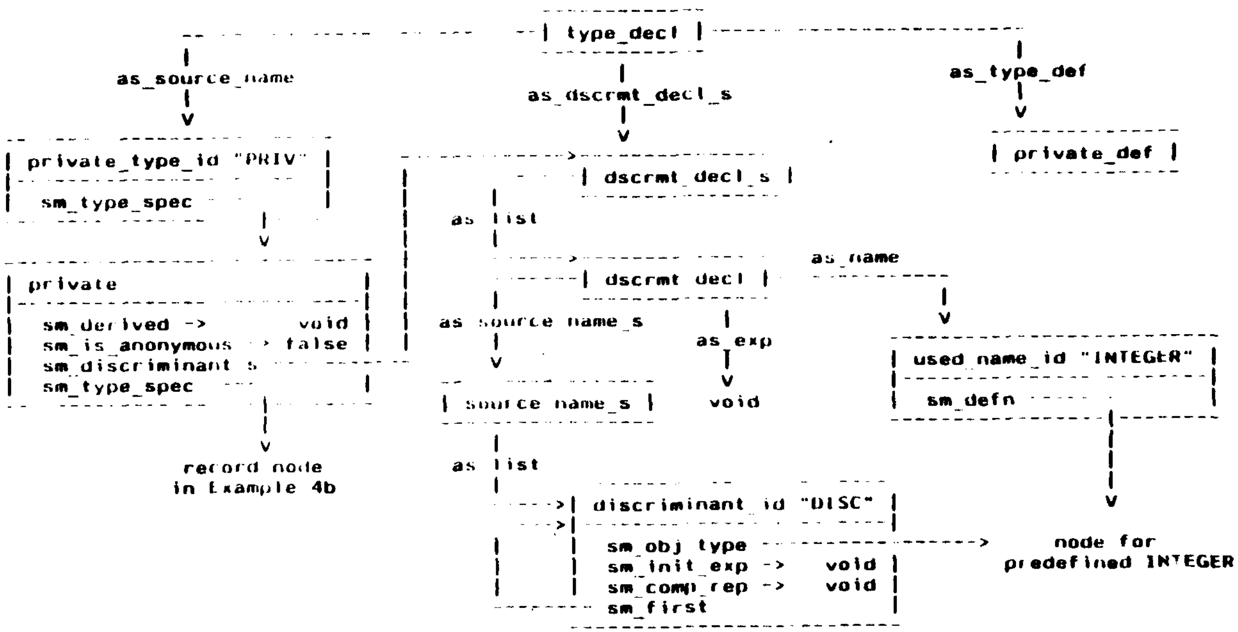
Example 3b   Multiple Object Declaration

```
OBJECT_1, OBJECT_2 : INTEGER range 1 .. 10;
```



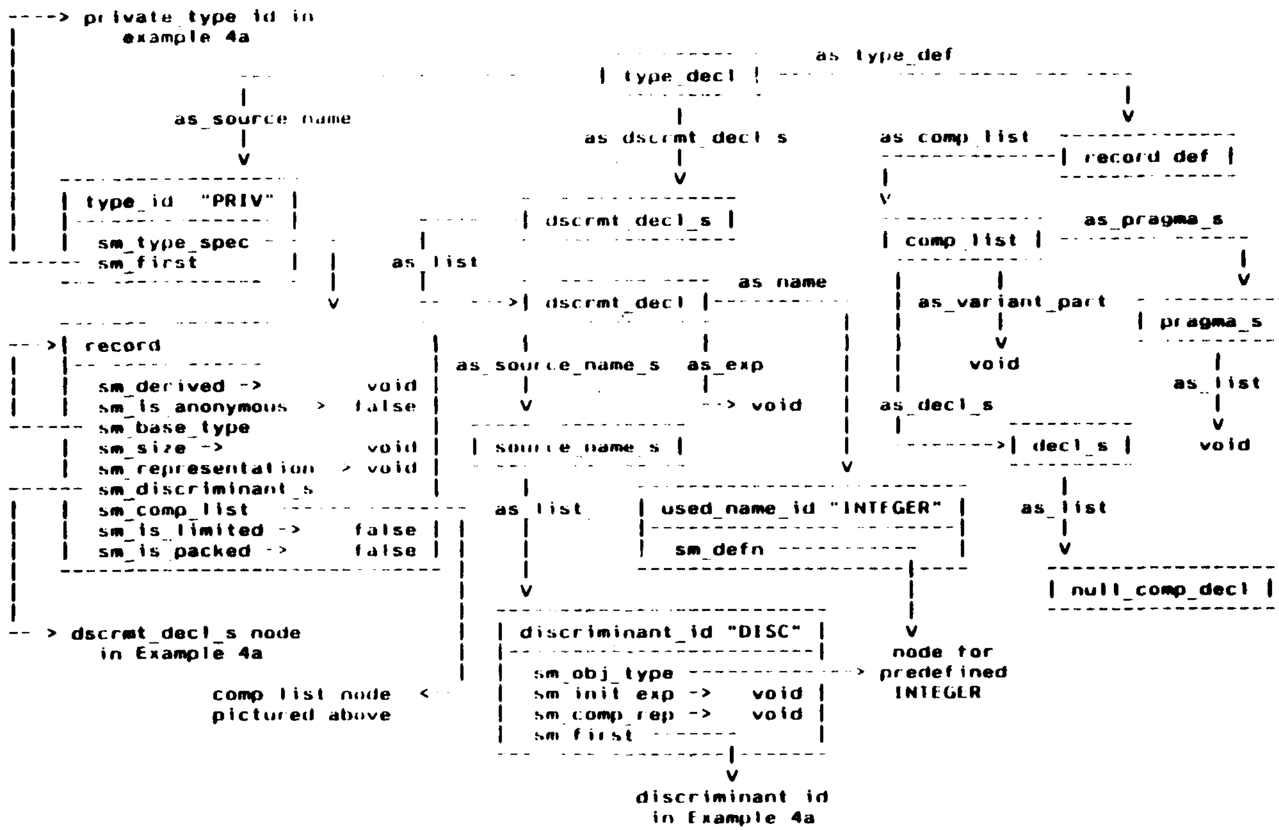
Example 3: Multiple Object Declaration with Anonymous Subtype

type PRIV (DISC : INTEGER) is private;



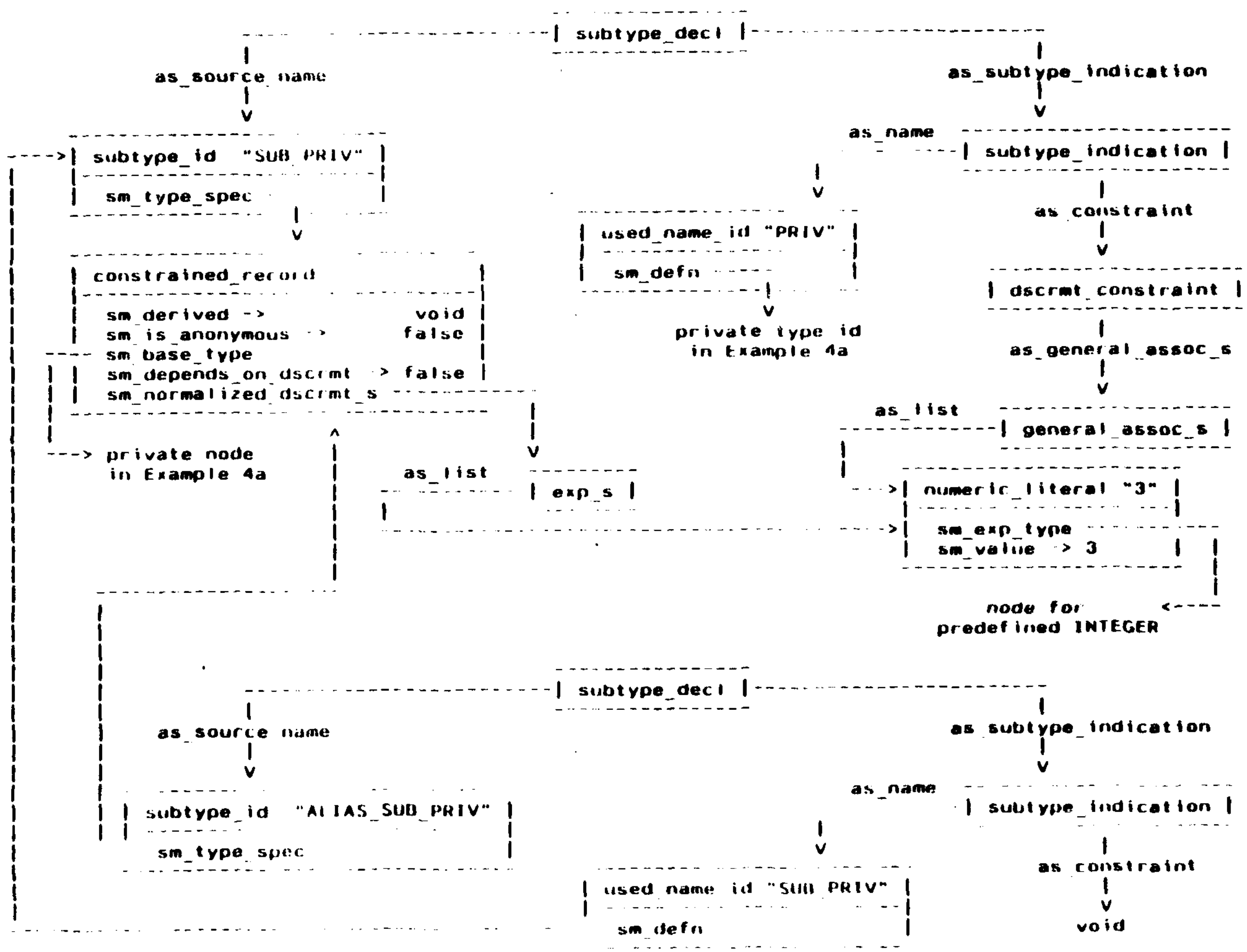
Example 4a - Private Type Declaration

```
type PRIV ( DISC : INTEGER ) is record
   null;
end;
```



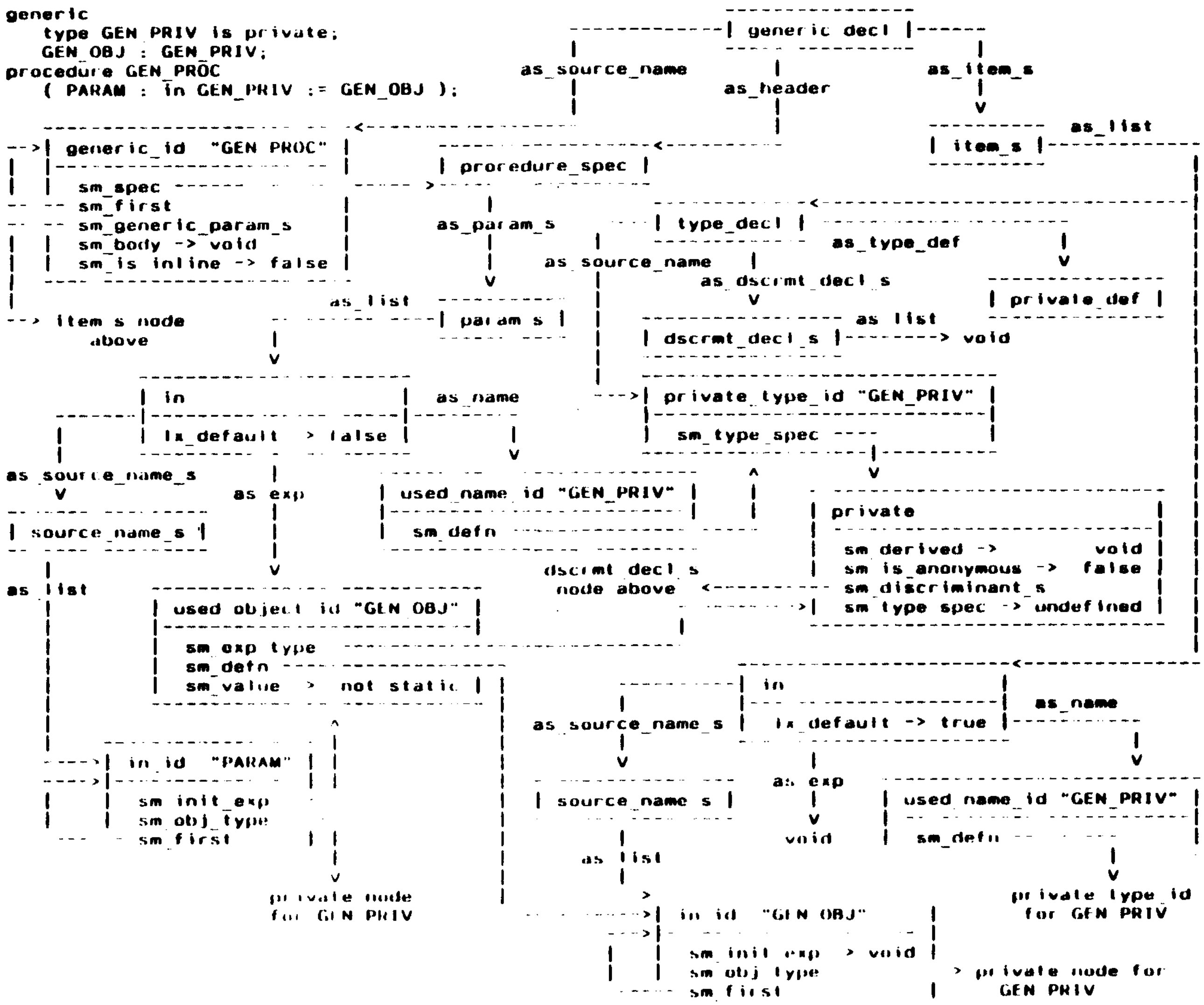
Example 4b - Full Record Type Declaration

```
subtype SUB_PRIV is PRIV ( 3 );
subtype ALIAS_SUB_PRIV is SUB_PRIV;
```

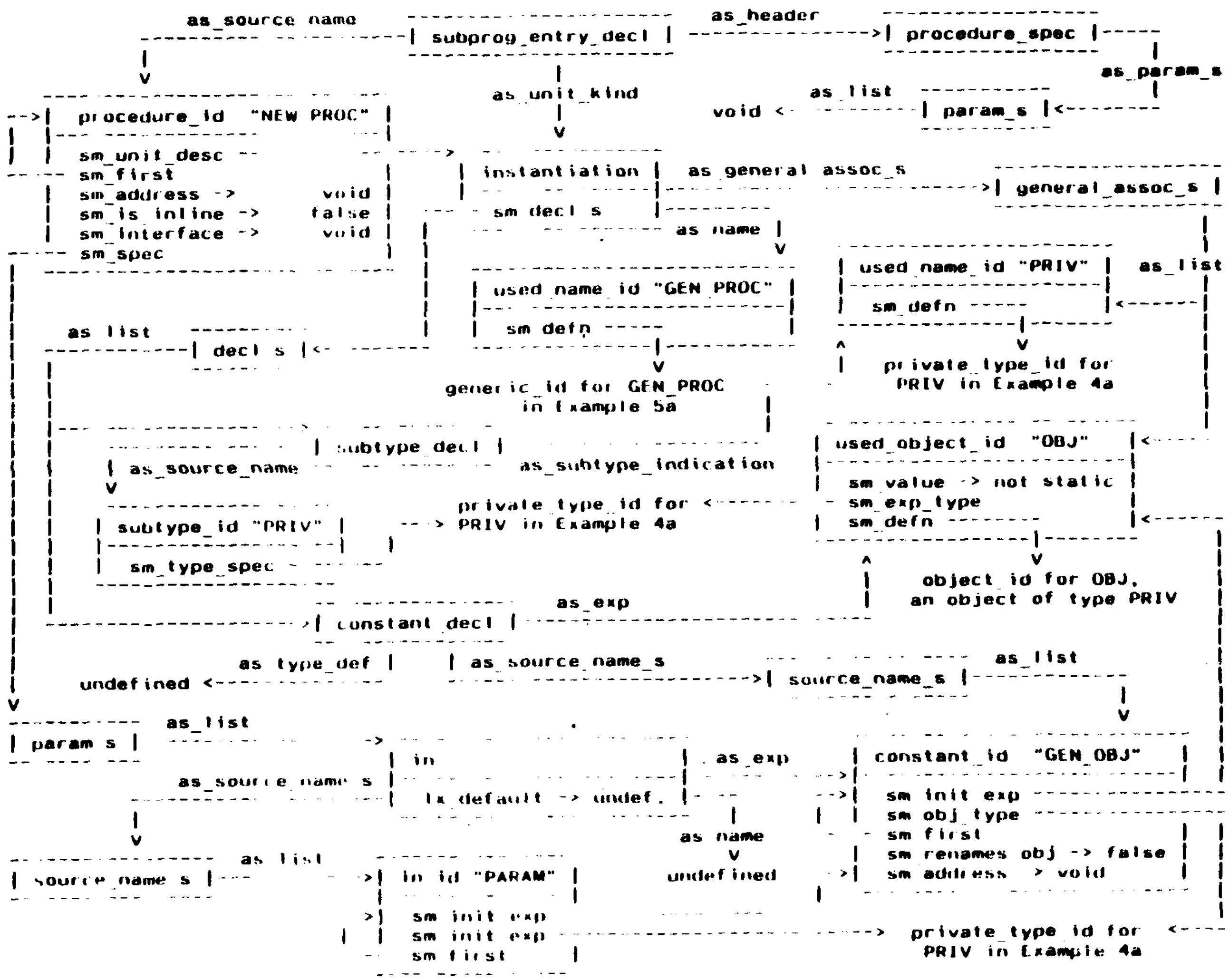


Example 4c Declarations of Subtype of Private Type

```
generic
   type GEN_PRIV is private;
   GEN_OBJ : GEN_PRIV;
procedure GEN_PROC
   ( PARAM : in GEN_PRIV := GEN_OBJ );
```

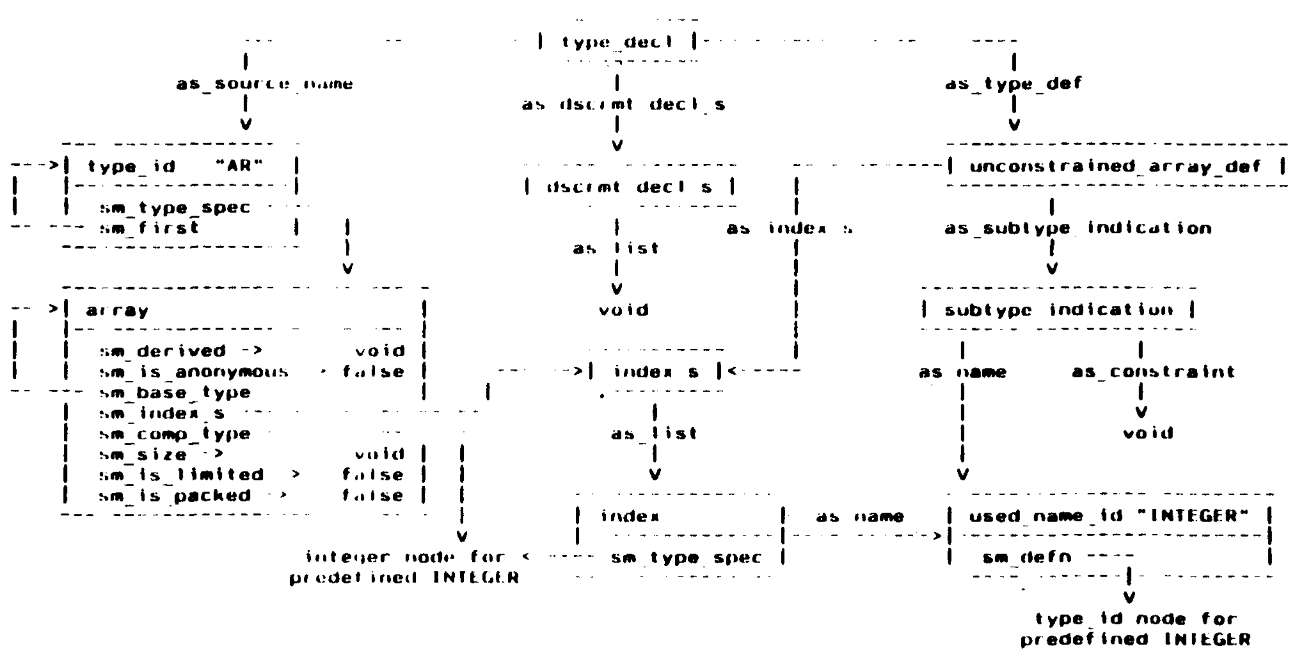


Example 5a - Generic Procedure Declaration procedure NEW_PROC is new GEN_PROC (PRIV, OBJ); -- declaration of OBJ is not shown



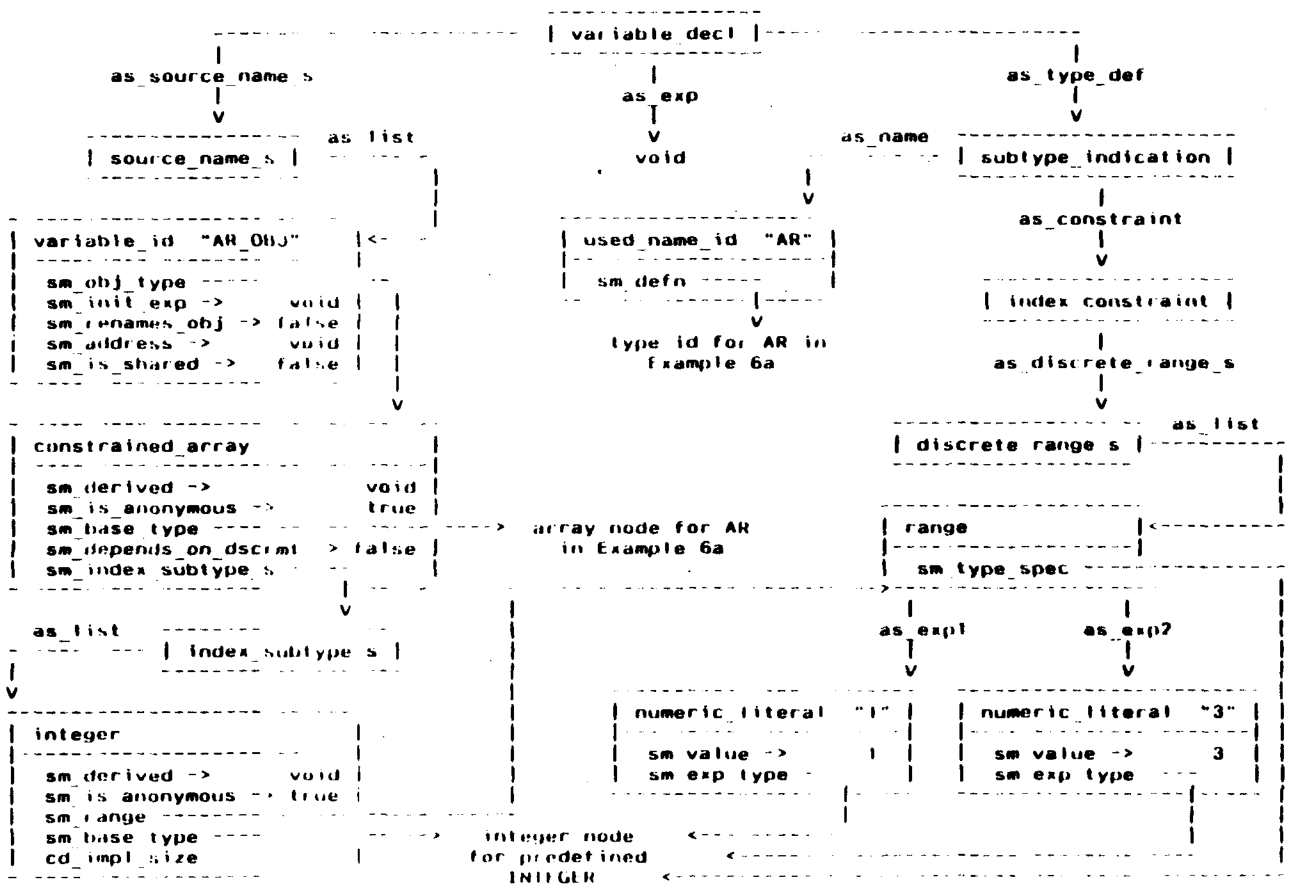
Example 5b - Generic Instantiation

type AR is array (INTEGER range <>) of INTEGER;



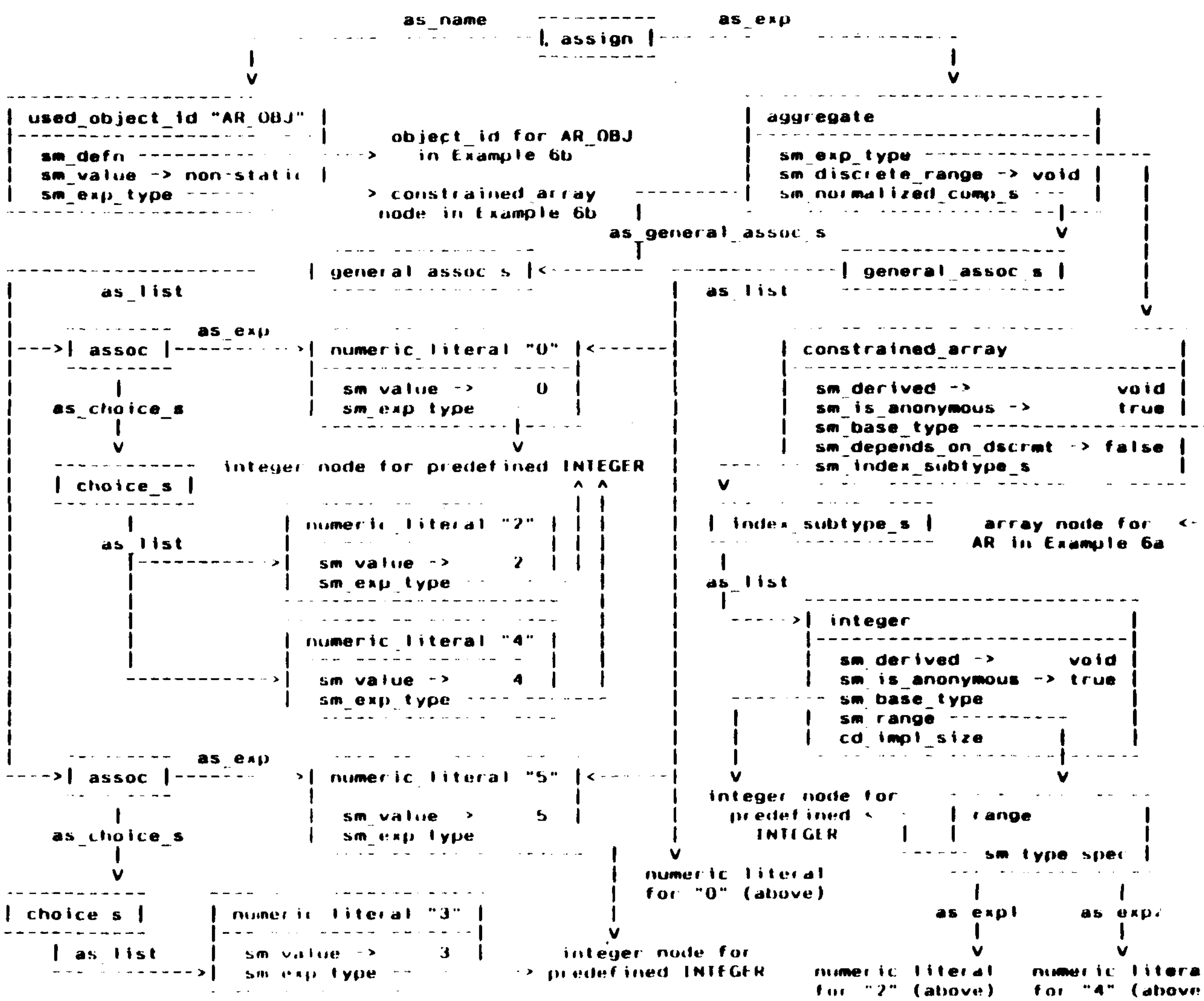
Example 6a Array Type Definition

AR_OBJ : AR (1 .. 3);



Example 6b Object Declaration with Anonymous Array Subtype

AR_OBJ := (2 | 4 => 0, 3 => 5);



Example 6c Assignment of an Array Aggregate

CHAPTER 6

EXTERNAL REPRESENTATION OF DIANA

The contents of this chapter will be included at a later date.

CHAPTER 7

THE DIANA PACKAGE IN ADA

The contents of this chapter will be included at a later date.

APPENDIX A

DIANA CROSS-REFERENCE GUIDE

PARTITIONS : 2

STRICT CLASSES : 94

STRICT CLASSES NOT DEFINING ATTRIBUTES : 35

STRICT CLASSES THAT DO NOT SERVE AS TYPES : 55

STRICT CLASSES THAT DO NOT SERVE AS TYPES
AND DO NOT DEFINE ATTRIBUTES : 3

LEAF NODES : 207

LEAF NODES NOT DEFINING ATTRIBUTES : 92

ATTRIBUTES : 135

PARTITIONS (UNINCLUDED CLASSES)

ALL_SOURCE
TYPE_SPEC

STRICT CLASSES THAT DO NOT SERVE AS TYPES AND DO NOT DEFINE ATTRIBUTES

FULL_TYPE_SPEC
GENERIC_PARAM
SEQUENCES

STRICT CLASSES THAT DO NOT DEFINE ATTRIBUTES

ALIGNMENT_CLAUSE
ALL_DECL
ALTERNATIVE_ELEM
BODY
CHOICE
COMP_REP_ELEM
CONSTRAINT
CONTEXT_ELEM
DECL
DISCRETE_RANGE
EXP
FULL_TYPE_SPEC
GENERAL_ASSOC
GENERIC_PARAM
HEADER
ITEM
ITERATION
MEMBERSHIP_OP
NAME
PARAM
PREDEF_NAME
SEQUENCES
SHORT_CIRCUIT_OP
SOURCE_NAME
STM
STM_ELEM
TEST_CLAUSE_ELEM
TYPE_DEF
TYPE_SPEC
UNIT_DESC
UNIT_KIND
USE_PRAGMA
USED_NAME
VARIANT_ELEM
VARIANT_PART

STRICT CLASSES THAT DO NOT SERVE AS TYPES

AGG_EXP
ALL_SOURCE
ARR_ACC_DER_DEF
BLOCK_LOOP
CALL_STM
CLAUSES_STM
COMP_NAME
CONSTRAINED
CONSTRAINED_DEF
DERIVABLE_SPEC
DSCRMT_PARAM_DECL
ENTRY_STM
EXP_DECL
EXP_EXP
EXP_VAL
EXP_VAL_EXP
FOR_REV
FULL_TYPE_SPEC
GENERIC_PARAM
ID_DECL
ID_S_DECL
INIT_OBJECT_NAME
LABEL_NAME
MEMBERSHIP
NAME_EXP
NAME_VAL
NAMED_ASSOC
NAMED_REP
NON_GENERIC_DECL
NON_TASK
NON_TASK_NAME
OBJECT_DECL
OBJECT_NAME
PARAM_NAME
PRIVATE_SPEC
QUAL_CONV
REAL
REAL_CONSTRAINT
RENAME_INSTANT
SEQUENCES
SIMPLE_RENAME_DECL
STM_WITH_EXP
STM_WITH_EXP_NAME
STM_WITH_NAME
SUBP_ENTRY_HEADER
SUBPROG_NAME
SUBPROG_PACK_NAME
TEST_CLAUSE
TYPE_NAME
UNCONSTRAINED
UNCONSTRAINED_COMPOSITE
UNIT_DECL
UNIT_NAME
USED_OBJECT
VC_NAME

LEAF NODES (CLASSES WITHOUT MEMBERS)

abort
accept
access
access_def
address
aggregate
alignment
all
alternative
alternative_pragma
alternative_s
and_then
argument_id
argument_id_s
array
assign
assoc
attribute
attribute_id
block
block_body
block_loop_id
block_master
bltn_operator_id
box_default
case
character_id
choice_exp
choice_others
choice_range
choice_s
code
comp_list
comp_rep
comp_rep_pragma
comp_rep_s
compilation
compilation_unit
compltn_unit_s
component_id
cond_clause
cond_entry
constant_decl
constant_id
constrained_access
constrained_array
constrained_array_def
constrained_record
context_elem_s
context_pragma
conversion
decl_s
deferred_constant_decl
delay
derived_def
derived_subprog
discrete_range_s
discrete_subtype
discriminant_id
dscrmt_constraint
dscrmt_decl
dscrmt_decl_s
entry
entry_call
entry_id
enum_literal_s
enumeration
enumeration_def
enumeration_id
exception_decl
exception_id
exit
exp_s
fixed
fixed_constraint
fixed_def
float
float_constraint
float_def
for
formal_dscrt_def
formal_fixed_def
formal_float_def
formal_integer_def
function_call
function_id
function_spec
general_assoc_s
generic_decl
generic_id
goto
if
implicit_not_eq
in
in_id
in_op
in_out
in_out_id
incomplete
index
index_constraint
index_s

indexed
instantiation
integer
integer_def
item_s
iteration_id
l_private
l_private_def
l_private_type_id
label_id
labeled
length_enum_rep
loop
name_default
name_s
named
no_default
not_in
null_access
null_comp_decl
null_stm
number_decl
number_id
numeric_literal
operator_id
or_else
out
out_id
package_body
package_decl
package_id
package_spec
param_s
parenthesized
pragma
pragma_id
pragma_s
private
private_def
private_type_id
procedure_call
procedure_id
procedure_spec
qualified
qualified_allocator
raise
range
range_attribute
range_membership
record
record_def
record_rep
renames_exc_decl
renames_obj_decl
renames_unit
return
reverse
scalar_s
select_alt_pragma
select_alternative
selected
selective_wait
short_circuit
slice
source_name_s
stm_pragma
stm_s
string_literal
stub
subprog_entry_decl
subprogram_body
subtype_allocator
subtype_decl
subtype_id
subtype_indication
subunit
task_body
task_body_id
task_decl
task_spec
terminate
test_clause_elem_s
timed_entry
type_decl
type_id
type_membership
unconstrained_array_def
universal_fixed
universal_integer
universal_real
use
use_pragma_s
used_char
used_name_id
used_object_id
used_op
variable_decl
variable_id
variant
variant_part
variant_pragma
variant_s
void
while
with

LEAF NODES THAT DO NOT DEFINE ATTRIBUTES

access_def
address
all
and_then
argument_id
assign
attribute_id
block_loop_id
box_default
character_id
choice_others
code
component_id
cond_clause
cond_entry
constant_decl
conversion
delay
derived_def
dscrmt_decl
entry_call
enumeration_id
exception_decl
fixed_constraint
fixed_def
float
float_constraint
float_def
for
formal_dscrt_def
formal_fixed_def
formal_float_def
formal_integer_def
function_id
goto
if
in_id
in_op
in_out
in_out_id
integer
integer_def
iteration_id
l_private
l_private_def
l_private_type_id
label_id
length_enum_rep
no_default
not_in
null_access
null_comp_decl
null_stm
number_decl
number_id
operator_id
or_else
out
out_id
package_body
package_decl
package_id
parenthesized
private
private_def
private_type_id
procedure_call
procedure_id
procedure_spec
qualified
raise
renames_exc_decl
renames_unit
return
reverse
select_alternative
selective_wait
stub
subprog_entry_decl
subtype_id
task_body
terminate
timed_entry
universal_fixed
universal_integer
universal_real
used_char
used_name_id
used_object_id
used_op
variable_decl
void

PREDEFINED AND USER-DEFINED TYPES

source_position IS THE DECLARED TYPE OF:

ALL_SOURCE.lx_srcpos

comments IS THE DECLARED TYPE OF:

ALL_SOURCE.lx_comments

symbol_rep IS THE DECLARED TYPE OF:

DEF_NAME.lx_symrep

DESIGNATOR.lx_symrep

string_literal.lx_symrep

value IS THE DECLARED TYPE OF:

fixed.cd_impl_small

REAL.sm_accuracy

EXP_VAL.sm_value

NAME_VAL.sm_value

USED_OBJECT.sm_value

operator IS THE DECLARED TYPE OF:

bltn_operator_id.sm_operator

number_rep IS THE DECLARED TYPE OF:

numeric_literal.lx_numrep

Boolean IS THE DECLARED TYPE OF:

in.lx_default

function_call.lx_prefix

CONSTRAINED.sm_depends_on_dscrmt

DERIVABLE_SPEC.sm_is_anonymous

access.sm_is_controlled

generic_id.sm_is_inline

SUBPROG_NAME.sm_is_inline

UNCONSTRAINED_COMPOSITE.sm_is_limited

UNCONSTRAINED_COMPOSITE.sm_is_packed

variable_id.sm_is_shared

VC_NAME.sm_renames_obj

Integer IS THE DECLARED TYPE OF:

SCALAR.cd_impl_size

ENUM_LITERAL.sm_pos

ENUM_LITERAL.sm_rep

ATTRIBUTES

```
as_alignment_clause      : ALIGNMENT_CLAUSE
     <= record_rep

as_all_decl              : ALL_DECL
     <= compilation_unit

as_alternative_s         : alternative_s
     <= block_body
     <= case

as_block_body            : block_body
     <= block

as_body                  : BODY
     <= SUBUNIT_BODY

as_choice_s              : choice_s
     <= alternative
     <= named
     <= variant

as_comp_list             : comp_list
     <= record_def
     <= variant

as_comp_rep_s            : comp_rep_s
     <= record_rep

as_compltn_unit_s        : compltn_unit_s
     <= compilation

as_constraint            : CONSTRAINT
     <= constrained_array_def
     <= CONSTRAINED_DEF

as_context_elem_s        : context_elem_s
     <= compilation_unit

as_decl_s                : decl_s
     <= comp_list
     <= task_decl

as_decl_s1               : decl_s
     <= package_spec

as_decl_s2               : decl_s
     <= package_spec

as_designator            : DESIGNATOR
     <= selected
```

```
as_discrete_range        : DISCRETE_RANGE
     <= choice_range
     <= entry
     <= FOR_REV
     <= slice

as_discrete_range_s      : discrete_range_s
     <= index_constraint

as_dscrmt_decl_s         : dscrmt_decl_s
     <= type_decl

as_enum_literal_s        : enum_literal_s
     <= enumeration_def

as_exp                   : EXP
     <= alignment
     <= attribute
     <= choice_exp
     <= comp_rep
     <= DSCRMT_PARAM_DECL
     <= EXP_DECL
     <= EXP_VAL_EXP
     <= NAMED_ASSOC
     <= NAMED_REP
     <= range_attribute
     <= REAL_CONSTRAINT
     <= STM_WITH_EXP
     <= TEST_CLAUSE
     <= while

as_exp1                  : EXP
     <= range
     <= short_circuit

as_exp2                  : EXP
     <= range
     <= short_circuit

as_exp_s                 : exp_s
     <= indexed

as_general_assoc_s       : general_assoc_s
     <= aggregate
     <= CALL_STM
     <= dscrmt_constraint
     <= function_call
     <= instantiation
     <= pragma

as_header                : HEADER
     <= subprogram_body
     <= UNIT_DECL
```

```
as_index_s              : index_s
    <= unconstrained_array_def

as_item_s               : item_s
    <= block_body
    <= generic_decl

as_iteration            : ITERATION
    <= loop

as_list                 : Seq Of GENERAL_ASSOC
    <= general_assoc_s

as_list                 : Seq Of SOURCE_NAME
    <= source_name_s

as_list                 : Seq Of ENUM_LITERAL
    <= enum_literal_s

as_list                 : Seq Of DISCRETE_RANGE
    <= discrete_range_s

as_list                 : Seq Of SCALAR
    <= scalar_s

as_list                 : Seq Of index
    <= index_s

as_list                 : Seq Of dscrmt_decl
    <= dscrmt_decl_s

as_list                 : Seq Of VARIANT_ELEM
    <= variant_s

as_list                 : Seq Of CHOICE
    <= choice_s

as_list                 : Seq Of ITEM
    <= item_s

as_list                 : Seq Of EXP
    <= exp_s

as_list                 : Seq Of STM_ELEM
    <= stm_s

as_list                 : Seq Of ALTERNATIVE_ELEM
    <= alternative_s

as_list                 : Seq Of PARAM
    <= param_s

as_list                 : Seq Of DECL
    <= decl_s
```

```
as_list                     : Seq Of TEST_CLAUSE_ELEM
     <= test_clause_elem_s

as_list                     : Seq Of NAME
     <= name_s

as_list                     : Seq Of compilation_unit
     <= compltn_unit_s

as_list                     : Seq Of pragma
     <= pragma_s

as_list                     : Seq Of CONTEXT_ELEM
     <= context_elem_s

as_list                     : Seq Of USE_PRAGMA
     <= use_pragma_s

as_list                     : Seq Of COMP_REP_ELEM
     <= comp_rep_s

as_list                     : Seq Of argument_id
     <= argument_id_s

as_membership_op            : MEMBERSHIP_OP
     <= MEMBERSHIP

as_name                     : NAME
     <= accept
     <= comp_rep
     <= deferred_constant_decl
     <= DSCRMT_PARAM_DECL
     <= function_spec
     <= index
     <= name_default
     <= NAME_EXP
     <= QUAL_CONV
     <= range_attribute
     <= RENAME_INSTANT
     <= REP
     <= SIMPLE_RENAME_DECL
     <= STM_WITH_EXP_NAME
     <= STM_WITH_NAME
     <= subtype_indication
     <= subunit
     <= type_membership
     <= variant_part

as_name_s                   : name_s
     <= abort
     <= use
     <= with
```

```
as_param_s                : param_s
     <= accept
     <= SUBP_ENTRY_HEADER

as_pragma                 : pragma
     <= alternative_pragma
     <= comp_rep_pragma
     <= context_pragma
     <= select_alt_pragma
     <= stm_pragma
     <= variant_pragma

as_pragma_s               : pragma_s
     <= alignment
     <= comp_list
     <= compilation_unit
     <= labeled

as_qualified              : qualified
     <= qualified_allocator

as_range                  : RANGE
     <= comp_rep
     <= range_membership
     <= REAL_CONSTRAINT

as_short_circuit_op       : SHORT_CIRCUIT_OP
     <= short_circuit

as_source_name            : SOURCE_NAME
     <= BLOCK_LOOP
     <= FOR_REV
     <= ID_DECL
     <= SUBUNIT_BODY

as_source_name_s          : source_name_s
     <= DSCRMT_PARAM_DECL
     <= ID_S_DECL
     <= labeled

as_stm                    : STM
     <= labeled

as_stm_s                  : stm_s
     <= accept
     <= alternative
     <= block_body
     <= CLAUSES_STM
     <= loop
     <= TEST_CLAUSE

as_stm_sl                 : stm_s
     <= ENTRY_STM
```

```
as_stm_s2               : stm_s
    <= ENTRY_STM

as_subtype_indication   : subtype_indication
    <= ARR_ACC_DER_DEF
    <= discrete_subtype
    <= subtype_allocator
    <= subtype_decl

as_subunit_body         : SUBUNIT_BODY
    <= subunit

as_test_clause_elem_s   : test_clause_elem_s
    <= CLAUSES_STM

as_type_def             : TYPE_DEF
    <= OBJECT_DECL
    <= type_decl

as_type_mark_name       : NAME
    <= renames_obj_decl

as_unit_kind            : UNIT_KIND
    <= NON_GENERIC_DECL

as_use_pragma_s         : use_pragma_s
    <= with

as_used_name            : USED_NAME
    <= assoc

as_used_name_id         : used_name_id
    <= attribute
    <= pragma
    <= range_attribute

as_variant_part         : VARIANT_PART
    <= comp_list

as_variant_s            : variant_s
    <= variant_part

cd_impl_size            : Integer
    <= SCALAR

cd_impl_small           : value
    <= fixed

lx_comments             : comments
    <= ALL_SOURCE

lx_default              : Boolean
    <= in
```

```
lx_numrep                : number_rep
     <= numeric_literal

lx_prefix                : Boolean
     <= function_call

lx_srcpos                : source_position
     <= ALL_SOURCE

lx_symrep                : symbol_rep
     <= DEF_NAME
     <= DESIGNATOR
     <= string_literal

sm_accuracy              : value
     <= REAL

sm_address               : EXP
     <= entry_id
     <= SUBPROG_PACK_NAME
     <= task_spec
     <= VC_NAME

sm_argument_id_s         : argument_id_s
     <= pragma_id

sm_base_type             : TYPE_SPEC
     <= NON_TASK

sm_body                  : BODY
     <= generic_id
     <= task_body_id
     <= task_spec

sm_comp_list             : comp_list
     <= record

sm_comp_rep              : COMP_REP_ELEM
     <= COMP_NAME

sm_comp_type             : TYPE_SPEC
     <= array

sm_decl_s                : decl_s
     <= instantiation
     <= task_spec

sm_defn                  : DEF_NAME
     <= DESIGNATOR

sm_depends_on_dscrmt     : Boolean
     <= CONSTRAINED

sm_derivable             : SOURCE_NAME
```

```
        <= derived_subprog

sm_derived              : TYPE_SPEC
        <= DERIVABLE_SPEC

sm_desig_type           : TYPE_SPEC
        <= access
        <= constrained_access
        <= subtype_allocator

sm_discrete_range       : DISCRETE_RANGE
        <= AGG_EXP

sm_discriminant_s       : dscrmt_decl_s
        <= incomplete
        <= PRIVATE_SPEC
        <= record

sm_equal                : SOURCE_NAME
        <= implicit_not_eq

sm_exp_type             : TYPE_SPEC
        <= EXP_EXP
        <= NAME_EXP
        <= USED_OBJECT

sm_first                : DEF_NAME
        <= constant_id
        <= discriminant_id
        <= PARAM_NAME
        <= type_id
        <= UNIT_NAME

sm_generic_param_s      : item_s
        <= generic_id

sm_index_s              : index_s
        <= array

sm_index_subtype_s      : scalar_s
        <= constrained_array

sm_init_exp             : EXP
        <= INIT_OBJECT_NAME

sm_interface            : PREDEF_NAME
        <= SUBPROG_NAME

sm_is_anonymous         : Boolean
        <= DERIVABLE_SPEC

sm_is_controlled        : Boolean
        <= access
```

```
sm_is_inline               : Boolean
     <= generic_id
     <= SUBPROG_NAME

sm_is_limited              : Boolean
     <= UNCONSTRAINED_COMPOSITE

sm_is_packed               : Boolean
     <= UNCONSTRAINED_COMPOSITE

sm_is_shared               : Boolean
     <= variable_id

sm_literal_s               : enum_literal_s
     <= enumeration

sm_master                  : ALL_DECL
     <= access

sm_normalized_comp_s       : general_assoc_s
     <= aggregate

sm_normalized_dscrmt_s     : exp_s
     <= constrained_record

sm_normalized_param_s      : exp_s
     <= CALL_STM
     <= function_call

sm_obj_type                : TYPE_SPEC
     <= OBJECT_NAME

sm_operator                : operator
     <= bltn_operator_id

sm_pos                     : Integer
     <= ENUM_LITERAL

sm_range                   : RANGE
     <= SCALAR

sm_renames_exc             : NAME
     <= exception_id

sm_renames_obj             : Boolean
     <= VC_NAME

sm_rep                     : Integer
     <= ENUM_LITERAL

sm_representation          : REP
     <= record

sm_size                    : EXP
```

```
        <= task_spec
        <= UNCONSTRAINED

    sm_spec                 : HEADER
        <= entry_id
        <= NON_TASK_NAME

    sm_stm                  : STM
        <= block_master
        <= exit
        <= LABEL_NAME

    sm_storage_size         : EXP
        <= access
        <= task_spec

    sm_type_spec            : TYPE_SPEC
        <= index
        <= PRIVATE_SPEC
        <= RANGE
        <= REAL_CONSTRAINT
        <= task_body_id
        <= TYPE_NAME

    sm_unit_desc            : UNIT_DESC
        <= SUBPROG_PACK_NAME

    sm_value                : value
        <= EXP_VAL
        <= NAME_VAL
        <= USED_OBJECT
```

NODES AND CLASSES

```
** abort

     IS INCLUDED IN:
            STM
            STM_ELEM
            ALL_SOURCE
     NODE ATTRIBUTES:
         (NODE SPECIFIC):
                as_name_s                  : name_s
         (INHERITED FROM ALL_SOURCE):
                lx_srcpos                  : source_position
                lx_comments                : comments

** accept

     IS INCLUDED IN:
            STM
            STM_ELEM
            ALL_SOURCE
     NODE ATTRIBUTES:
         (NODE SPECIFIC):
                as_name                    : NAME
                as_stm_s                   : stm_s
                as_param_s                 : param_s
         (INHERITED FROM ALL_SOURCE):
                lx_srcpos                  : source_position
                lx_comments                : comments

** access

     IS INCLUDED IN:
            UNCONSTRAINED
            NON_TASK
            FULL_TYPE_SPEC
            DERIVABLE_SPEC
            TYPE_SPEC
     NODE ATTRIBUTES:
         (NODE SPECIFIC):
                sm_storage_size            : EXP
                sm_master                  : ALL_DECL
                sm_desig_type              : TYPE_SPEC
                sm_is_controlled           : Boolean
         (INHERITED FROM UNCONSTRAINED):
                sm_size                    : EXP
         (INHERITED FROM NON_TASK):
                sm_base_type               : TYPE_SPEC
         (INHERITED FROM DERIVABLE_SPEC):
                sm_derived                 : TYPE_SPEC
                sm_is_anonymous            : Boolean

** access_def
```

```
        IS INCLUDED IN:
                ARR_ACC_DER_DEF
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ARR_ACC_DER_DEF):
                    as_subtype_indication   : subtype_indication
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** address

```
        IS INCLUDED IN:
                NAMED_REP
                REP
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM NAMED_REP):
                    as_exp                  : EXP
            (INHERITED FROM REP):
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** AGG_EXP

```
        CLASS MEMBERS:
                aggregate
                string_literal
        IS INCLUDED IN:
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_discrete_range       : DISCRETE_RANGE
            (INHERITED FROM EXP_EXP):
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** aggregate

```
        IS INCLUDED IN:
                AGG_EXP
                EXP_EXP
                EXP
```

```
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_general_assoc_s        : general_assoc_s
                    sm_normalized_comp_s      : general_assoc_s
            (INHERITED FROM AGG_EXP):
                    sm_discrete_range         : DISCRETE_RANGE
            (INHERITED FROM EXP_EXP):
                    sm_exp_type               : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                 : source_position
                    lx_comments               : comments
```

** alignment

```
        IS INCLUDED IN:
                ALIGNMENT_CLAUSE
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_pragma_s               : pragma_s
                    as_exp                    : EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                 : source_position
                    lx_comments               : comments
```

** ALIGNMENT_CLAUSE

```
        CLASS MEMBERS:
                alignment
                void
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                 : source_position
                    lx_comments               : comments
        IS THE DECLARED TYPE OF:
                record_rep.as_alignment_clause
```

** all

```
        IS INCLUDED IN:
                NAME_EXP
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM NAME_EXP):
                    as_name                   : NAME
                    sm_exp_type               : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
```

```
                lx_srcpos             : source_position
                lx_comments           : comments

**  ALL_DECL

        CLASS MEMBERS:
                block_master
                void
                ITEM
                subunit
                DSCRMT_PARAM_DECL
                DECL
                SUBUNIT_BODY
                dscrmt_decl
                PARAM
                ID_S_DECL
                ID_DECL
                null_comp_decl
                REP
                USE_PRAGMA
                subprogram_body
                task_body
                package_body
                in
                in_out
                out
                EXP_DECL
                deferred_constant_decl
                exception_decl
                type_decl
                UNIT_DECL
                task_decl
                subtype_decl
                SIMPLE_RENAME_DECL
                NAMED_REP
                record_rep
                use
                pragma
                OBJECT_DECL
                number_decl
                generic_decl
                NON_GENERIC_DECL
                renames_obj_decl
                renames_exc_decl
                length_enum_rep
                address
                constant_decl
                variable_decl
                subprog_entry_decl
                package_decl
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
```

```
            lx_srcpos              : source_position
            lx_comments            : comments
    IS THE DECLARED TYPE OF:
          compilation_unit.as_all_decl
          access.sm_master
```

** ALL_SOURCE

```
    CLASS MEMBERS:
          DEF_NAME
          index
          compilation_unit
          compilation
          comp_list
          VARIANT_PART
          ALIGNMENT_CLAUSE
          VARIANT_ELEM
          CONTEXT_ELEM
          COMP_REP_ELEM
          ALTERNATIVE_ELEM
          ITERATION
          SHORT_CIRCUIT_OP
          MEMBERSHIP_OP
          TEST_CLAUSE_ELEM
          UNIT_DESC
          HEADER
          CHOICE
          CONSTRAINT
          GENERAL_ASSOC
          STM_ELEM
          SEQUENCES
          TYPE_DEF
          ALL_DECL
          SOURCE_NAME
          PREDEF_NAME
          variant_part
          void
          alignment
          variant
          variant_pragma
          context_pragma
          with
          comp_rep
          comp_rep_pragma
          alternative
          alternative_pragma
          FOR_REV
          while
          and_then
          or_else
          in_op
          not_in
          TEST_CLAUSE
          select_alt_pragma
```

UNIT_KIND
derived_subprog
implicit_not_eq
BODY
SUBP_ENTRY_HEADER
package_spec
choice_exp
choice_others
choice_range
DISCRETE_RANGE
dscrmt_constraint
index_constraint
REAL_CONSTRAINT
NAMED_ASSOC
EXP
STM
stm_pragma
alternative_s
variant_s
use_pragma_s
test_clause_elem_s
stm_s
source_name_s
scalar_s
pragma_s
param_s
name_s
index_s
item_s
exp_s
enum_literal_s
discrete_range_s
general_assoc_s
dscrmt_decl_s
decl_s
context_elem_s
compltn_unit_s
comp_rep_s
choice_s
argument_id_s
enumeration_def
record_def
ARR_ACC_DER_DEF
CONSTRAINED_DEF
private_def
l_private_def
formal_dscrt_def
formal_float_def
formal_fixed_def
formal_integer_def
block_master
ITEM
subunit
OBJECT_NAME

LABEL_NAME
UNIT_NAME
TYPE_NAME
entry_id
exception_id
attribute_id
bltn_operator_id
argument_id
pragma_id
for
reverse
cond_clause
select_alternative
RENAME_INSTANT
GENERIC_PARAM
block_body
stub
procedure_spec
function_spec
entry
RANGE
discrete_subtype
float_constraint
fixed_constraint
named
assoc
NAME
EXP_EXP
labeled
null_stm
abort
STM_WITH_EXP
STM_WITH_NAME
accept
ENTRY_STM
BLOCK_LOOP
CLAUSES_STM
terminate
constrained_array_def
derived_def
access_def
unconstrained_array_def
subtype_indication
integer_def
fixed_def
float_def
DSCRMT_PARAM_DECL
DECL
SUBUNIT_BODY
INIT_OBJECT_NAME
ENUM_LITERAL
iteration_id
label_id
block_loop_id

NON_TASK_NAME
task_body_id
type_id
subtype_id
private_type_id
l_private_type_id
renames_unit
instantiation
name_default
no_default
box_default
range
range_attribute
DESIGNATOR
NAME_EXP
EXP_VAL
subtype_allocator
qualified_allocator
AGG_EXP
return
delay
STM_WITH_EXP_NAME
case
goto
raise
CALL_STM
cond_entry
timed_entry
loop
block
if
selective_wait
dscrmt_decl
PARAM
ID_S_DECL
ID_DECL
null_comp_decl
REP
USE_PRAGMA
subprogram_body
task_body
package_body
VC_NAME
number_id
COMP_NAME
PARAM_NAME
enumeration_id
character_id
SUBPROG_PACK_NAME
generic_id
USED_OBJECT
USED_NAME
NAME_VAL
all

slice
indexed
short_circuit
numeric_literal
EXP_VAL_EXP
null_access
aggregate
string_literal
assign
code
exit
entry_call
procedure_call
in
in_out
out
EXP_DECL
deferred_constant_decl
exception_decl
type_decl
UNIT_DECL
task_decl
subtype_decl
SIMPLE_RENAME_DECL
NAMED_REP
record_rep
use
pragma
variable_id
constant_id
component_id
discriminant_id
in_id
out_id
in_out_id
SUBPROG_NAME
package_id
used_char
used_object_id
used_op
used_name_id
attribute
selected
function_call
MEMBERSHIP
QUAL_CONV
parenthesized
OBJECT_DECL
number_decl
generic_decl
NON_GENERIC_DECL
renames_obj_decl
renames_exc_decl
length_enum_rep

```
                address
                procedure_id
                operator_id
                function_id
                range_membership
                type_membership
                conversion
                qualified
                constant_decl
                variable_decl
                subprog_entry_decl
                package_decl
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  alternative

        IS INCLUDED IN:
                ALTERNATIVE_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_choice_s             : choice_s
                    as_stm_s                : stm_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  ALTERNATIVE_ELEM

        CLASS MEMBERS:
                alternative
                alternative_pragma
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                alternative_s.as_list [Seq Of]

**  alternative_pragma

        IS INCLUDED IN:
                ALTERNATIVE_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_pragma               : pragma
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
```

```
                lx_comments             : comments

**  alternative_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_list                 : Seq Of ALTERNATIVE_ELEM
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                block_body.as_alternative_s
                case.as_alternative_s

**  and_then

        IS INCLUDED IN:
                SHORT_CIRCUIT_OP
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  argument_id

        IS INCLUDED IN:
                PREDEF_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                argument_id_s.as_list [Seq Of]

**  argument_id_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_list                 : Seq Of argument_id
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
```

```
                pragma_id.sm_argument_id_s

**  ARR_ACC_DER_DEF

        CLASS MEMBERS:
                constrained_array_def
                derived_def
                access_def
                unconstrained_array_def
        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_subtype_indication     : subtype_indication
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments

**  array

        IS INCLUDED IN:
                UNCONSTRAINED_COMPOSITE
                UNCONSTRAINED
                NON_TASK
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_index_s                : index_s
                sm_comp_type              : TYPE_SPEC
            (INHERITED FROM UNCONSTRAINED_COMPOSITE):
                sm_is_limited             : Boolean
                sm_is_packed              : Boolean
            (INHERITED FROM UNCONSTRAINED):
                sm_size                   : EXP
            (INHERITED FROM NON_TASK):
                sm_base_type              : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                sm_derived                : TYPE_SPEC
                sm_is_anonymous           : Boolean

**  assign

        IS INCLUDED IN:
                STM_WITH_EXP_NAME
                STM_WITH_EXP
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM STM_WITH_EXP_NAME):
                as_name                   : NAME
```

```
        (INHERITED FROM STM_WITH_EXP):
                as_exp                  : EXP
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** assoc

```
    IS INCLUDED IN:
            NAMED_ASSOC
            GENERAL_ASSOC
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_used_name            : USED_NAME
        (INHERITED FROM NAMED_ASSOC):
                as_exp                  : EXP
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** attribute

```
    IS INCLUDED IN:
            NAME_VAL
            NAME_EXP
            NAME
            EXP
            GENERAL_ASSOC
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_used_name_id         : used_name_id
                as_exp                  : EXP
        (INHERITED FROM NAME_VAL):
                sm_value                : value
        (INHERITED FROM NAME_EXP):
                as_name                 : NAME
                sm_exp_type             : TYPE_SPEC
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** attribute_id

```
    IS INCLUDED IN:
            PREDEF_NAME
            DEF_NAME
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
```

```
                lx_comments              : comments

** block

     IS INCLUDED IN:
            BLOCK_LOOP
            STM
            STM_ELEM
            ALL_SOURCE
     NODE ATTRIBUTES:
         (NODE SPECIFIC):
                as_block_body            : block_body
         (INHERITED FROM BLOCK_LOOP):
                as_source_name           : SOURCE_NAME
         (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

** block_body

     IS INCLUDED IN:
            BODY
            UNIT_DESC
            ALL_SOURCE
     NODE ATTRIBUTES:
         (NODE SPECIFIC):
                as_item_s                : item_s
                as_alternative_s         : alternative_s
                as_stm_s                 : stm_s
         (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments
     IS THE DECLARED TYPE OF:
            block.as_block_body

** BLOCK_LOOP

     CLASS MEMBERS:
            loop
            block
     IS INCLUDED IN:
            STM
            STM_ELEM
            ALL_SOURCE
     NODE ATTRIBUTES:
         (NODE SPECIFIC):
                as_source_name           : SOURCE_NAME
         (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

** block_loop_id

     IS INCLUDED IN:
```

```
                LABEL_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM LABEL_NAME):
                    sm_stm                  : STM
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  block_master

        IS INCLUDED IN:
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_stm                  : STM
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  bltn_operator_id

        IS INCLUDED IN:
                PREDEF_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_operator             : operator
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  BODY

        CLASS MEMBERS:
                block_body
                void
                stub
        IS INCLUDED IN:
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
```

```
                generic_id.sm_body
                SUBUNIT_BODY.as_body
                task_body_id.sm_body
                task_spec.sm_body

**  box_default

        IS INCLUDED IN:
                GENERIC_PARAM
                UNIT_KIND
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  CALL_STM

        CLASS MEMBERS:
                entry_call
                procedure_call
        IS INCLUDED IN:
                STM_WITH_NAME
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_general_assoc_s       : general_assoc_s
                    sm_normalized_param_s    : exp_s
            (INHERITED FROM STM_WITH_NAME):
                    as_name                  : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  case

        IS INCLUDED IN:
                STM_WITH_EXP
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_alternative_s         : alternative_s
            (INHERITED FROM STM_WITH_EXP):
                    as_exp                   : EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  character_id
```

```
        IS INCLUDED IN:
                ENUM_LITERAL
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ENUM_LITERAL):
                sm_pos                  : Integer
                sm_rep                  : Integer
            (INHERITED FROM OBJECT_NAME):
                sm_obj_type             : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  CHOICE

        CLASS MEMBERS:
                choice_exp
                choice_others
                choice_range
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                choice_s.as_list [Seq Of]

**  choice_exp

        IS INCLUDED IN:
                CHOICE
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_exp                  : EXP
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  choice_others

        IS INCLUDED IN:
                CHOICE
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
```

```
                lx_comments            : comments

**  choice_range

        IS INCLUDED IN:
                CHOICE
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_discrete_range      : DISCRETE_RANGE
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos              : source_position
                lx_comments            : comments

**  choice_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_list                : Seq Of CHOICE
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos              : source_position
                lx_comments            : comments
        IS THE DECLARED TYPE OF:
                alternative.as_choice_s
                named.as_choice_s
                variant.as_choice_s

**  CLAUSES_STM

        CLASS MEMBERS:
                if
                selective_wait
        IS INCLUDED IN:
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_test_clause_elem_s  : test_clause_elem_s
                as_stm_s               : stm_s
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos              : source_position
                lx_comments            : comments

**  code

        IS INCLUDED IN:
                STM_WITH_EXP_NAME
                STM_WITH_EXP
                STM
                STM_ELEM
```

```
            ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM STM_WITH_EXP_NAME):
                    as_name                 : NAME
            (INHERITED FROM STM_WITH_EXP):
                    as_exp                  : EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** comp_list

```
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_decl_s               : decl_s
                    as_pragma_s             : pragma_s
                    as_variant_part         : VARIANT_PART
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                variant.as_comp_list
                record.sm_comp_list
                record_def.as_comp_list
```

** COMP_NAME

```
        CLASS MEMBERS:
                component_id
                discriminant_id
        IS INCLUDED IN:
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_comp_rep             : COMP_REP_ELEM
            (INHERITED FROM INIT_OBJECT_NAME):
                    sm_init_exp             : EXP
            (INHERITED FROM OBJECT_NAME):
                    sm_obj_type             : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** comp_rep

```
        IS INCLUDED IN:
```

```
                COMP_REP_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name                 : NAME
                    as_range                : RANGE
                    as_exp                  : EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** COMP_REP_ELEM

```
        CLASS MEMBERS:
                comp_rep
                void
                comp_rep_pragma
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                comp_rep_s.as_list [Seq Of]
                COMP_NAME.sm_comp_rep
```

** comp_rep_pragma

```
        IS INCLUDED IN:
                COMP_REP_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_pragma               : pragma
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** comp_rep_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                 : Seq Of COMP_REP_ELEM
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                record_rep.as_comp_rep_s
```

** compilation

```
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_compltn_unit_s        : compltn_unit_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  compilation_unit

        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_context_elem_s        : context_elem_s
                    as_pragma_s              : pragma_s
                    as_all_decl              : ALL_DECL
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments
        IS THE DECLARED TYPE OF:
                compltn_unit_s.as_list [Seq Of]

**  compltn_unit_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                  : Seq Of compilation_unit
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments
        IS THE DECLARED TYPE OF:
                compilation.as_compltn_unit_s

**  component_id

        IS INCLUDED IN:
                COMP_NAME
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM COMP_NAME):
                    sm_comp_rep              : COMP_REP_ELEM
            (INHERITED FROM INIT_OBJECT_NAME):
                    sm_init_exp              : EXP
            (INHERITED FROM OBJECT_NAME):
```

```
                sm_obj_type             : TYPE_SPEC
        (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** cond_clause

```
    IS INCLUDED IN:
            TEST_CLAUSE
            TEST_CLAUSE_ELEM
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM TEST_CLAUSE):
                as_exp                  : EXP
                as_stm_s                : stm_s
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** cond_entry

```
    IS INCLUDED IN:
            ENTRY_STM
            STM
            STM_ELEM
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM ENTRY_STM):
                as_stm_s1               : stm_s
                as_stm_s2               : stm_s
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** constant_decl

```
    IS INCLUDED IN:
            OBJECT_DECL
            EXP_DECL
            ID_S_DECL
            DECL
            ITEM
            ALL_DECL
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM OBJECT_DECL):
                as_type_def             : TYPE_DEF
        (INHERITED FROM EXP_DECL):
                as_exp                  : EXP
        (INHERITED FROM ID_S_DECL):
                as_source_name_s        : source_name_s
        (INHERITED FROM ALL_SOURCE):
```

```
                lx_srcpos              : source_position
                lx_comments            : comments

**  constant_id

        IS INCLUDED IN:
                VC_NAME
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_first               : DEF_NAME
            (INHERITED FROM VC_NAME):
                sm_renames_obj         : Boolean
                sm_address             : EXP
            (INHERITED FROM INIT_OBJECT_NAME):
                sm_init_exp            : EXP
            (INHERITED FROM OBJECT_NAME):
                sm_obj_type            : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                lx_symrep              : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos              : source_position
                lx_comments            : comments

**  CONSTRAINED

        CLASS MEMBERS:
                constrained_array
                constrained_access
                constrained_record
        IS INCLUDED IN:
                NON_TASK
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_depends_on_dscrmt   : Boolean
            (INHERITED FROM NON_TASK):
                sm_base_type           : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                sm_derived             : TYPE_SPEC
                sm_is_anonymous        : Boolean

**  constrained_access

        IS INCLUDED IN:
                CONSTRAINED
                NON_TASK
                FULL_TYPE_SPEC
```

```
            DERIVABLE_SPEC
            TYPE_SPEC
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
            sm_desig_type           : TYPE_SPEC
        (INHERITED FROM CONSTRAINED):
            sm_depends_on_dscrmt    : Boolean
        (INHERITED FROM NON_TASK):
            sm_base_type            : TYPE_SPEC
        (INHERITED FROM DERIVABLE_SPEC):
            sm_derived              : TYPE_SPEC
            sm_is_anonymous         : Boolean
```

** constrained_array

```
    IS INCLUDED IN:
            CONSTRAINED
            NON_TASK
            FULL_TYPE_SPEC
            DERIVABLE_SPEC
            TYPE_SPEC
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
            sm_index_subtype_s      : scalar_s
        (INHERITED FROM CONSTRAINED):
            sm_depends_on_dscrmt    : Boolean
        (INHERITED FROM NON_TASK):
            sm_base_type            : TYPE_SPEC
        (INHERITED FROM DERIVABLE_SPEC):
            sm_derived              : TYPE_SPEC
            sm_is_anonymous         : Boolean
```

** constrained_array_def

```
    IS INCLUDED IN:
            ARR_ACC_DER_DEF
            TYPE_DEF
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
            as_constraint           : CONSTRAINT
        (INHERITED FROM ARR_ACC_DER_DEF):
            as_subtype_indication   : subtype_indication
        (INHERITED FROM ALL_SOURCE):
            lx_srcpos               : source_position
            lx_comments             : comments
```

** CONSTRAINED_DEF

```
    CLASS MEMBERS:
            subtype_indication
            integer_def
            fixed_def
            float_def
```

```
        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_constraint           : CONSTRAINT
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  constrained_record

        IS INCLUDED IN:
                CONSTRAINED
                NON_TASK
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_normalized_dscrmt_s  : exp_s
            (INHERITED FROM CONSTRAINED):
                sm_depends_on_dscrmt    : Boolean
            (INHERITED FROM NON_TASK):
                sm_base_type            : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                sm_derived              : TYPE_SPEC
                sm_is_anonymous         : Boolean

**  CONSTRAINT

        CLASS MEMBERS:
                void
                DISCRETE_RANGE
                dscrmt_constraint
                index_constraint
                REAL_CONSTRAINT
                RANGE
                discrete_subtype
                float_constraint
                fixed_constraint
                range
                range_attribute
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                constrained_array_def.as_constraint
                CONSTRAINED_DEF.as_constraint

**  CONTEXT_ELEM
```

```
        CLASS MEMBERS:
                context_pragma
                with
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                context_elem_s.as_list [Seq Of]
```

** context_elem_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                     : Seq Of CONTEXT_ELEM
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                compilation_unit.as_context_elem_s
```

** context_pragma

```
        IS INCLUDED IN:
                CONTEXT_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_pragma                   : pragma
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** conversion

```
        IS INCLUDED IN:
                QUAL_CONV
                EXP_VAL_EXP
                EXP_VAL
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM QUAL_CONV):
                    as_name                     : NAME
            (INHERITED FROM EXP_VAL_EXP):
                    as_exp                      : EXP
```

```
        (INHERITED FROM EXP_VAL):
                sm_value                    : value
        (INHERITED FROM EXP_EXP):
                sm_exp_type                 : TYPE_SPEC
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
```

** DECL

```
    CLASS MEMBERS:
            ID_S_DECL
            ID_DECL
            null_comp_decl
            REP
            USE_PRAGMA
            EXP_DECL
            deferred_constant_decl
            exception_decl
            type_decl
            UNIT_DECL
            task_decl
            subtype_decl
            SIMPLE_RENAME_DECL
            void
            NAMED_REP
            record_rep
            use
            pragma
            OBJECT_DECL
            number_decl
            generic_decl
            NON_GENERIC_DECL
            renames_obj_decl
            renames_exc_decl
            length_enum_rep
            address
            constant_decl
            variable_decl
            subprog_entry_decl
            package_decl
    IS INCLUDED IN:
            ITEM
            ALL_DECL
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
    IS THE DECLARED TYPE OF:
            decl_s.as_list [Seq Of]
```

** decl_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                 : Seq Of DECL
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                instantiation.sm_decl_s
                task_spec.sm_decl_s
                task_decl.as_decl_s
                package_spec.as_decl_s1
                            .as_decl_s2
                comp_list.as_decl_s
```

** DEF_NAME

```
        CLASS MEMBERS:
                SOURCE_NAME
                PREDEF_NAME
                OBJECT_NAME
                LABEL_NAME
                UNIT_NAME
                TYPE_NAME
                void
                entry_id
                exception_id
                attribute_id
                bltn_operator_id
                argument_id
                pragma_id
                INIT_OBJECT_NAME
                ENUM_LITERAL
                iteration_id
                label_id
                block_loop_id
                NON_TASK_NAME
                task_body_id
                type_id
                subtype_id
                private_type_id
                l_private_type_id
                VC_NAME
                number_id
                COMP_NAME
                PARAM_NAME
                enumeration_id
                character_id
                SUBPROG_PACK_NAME
                generic_id
                variable_id
                constant_id
```

```
                component_id
                discriminant_id
                in_id
                out_id
                in_out_id
                SUBPROG_NAME
                package_id
                procedure_id
                operator_id
                function_id
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    lx_symrep                   : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                PARAM_NAME.sm_first
                DESIGNATOR.sm_defn
                discriminant_id.sm_first
                type_id.sm_first
                constant_id.sm_first
                UNIT_NAME.sm_first

**  deferred_constant_decl

        IS INCLUDED IN:
                ID_S_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name                     : NAME
            (INHERITED FROM ID_S_DECL):
                    as_source_name_s            : source_name_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments

**  delay

        IS INCLUDED IN:
                STM_WITH_EXP
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM STM_WITH_EXP):
                    as_exp                      : EXP
            (INHERITED FROM ALL_SOURCE):
```

```
                lx_srcpos                 : source_position
                lx_comments               : comments

**  DERIVABLE_SPEC

        CLASS MEMBERS:
                FULL_TYPE_SPEC
                PRIVATE_SPEC
                task_spec
                NON_TASK
                private
                l_private
                SCALAR
                CONSTRAINED
                UNCONSTRAINED
                enumeration
                REAL
                integer
                constrained_array
                constrained_access
                constrained_record
                UNCONSTRAINED_COMPOSITE
                access
                float
                fixed
                array
                record
        IS INCLUDED IN:
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_derived                : TYPE_SPEC
                sm_is_anonymous           : Boolean

**  derived_def

        IS INCLUDED IN:
                ARR_ACC_DER_DEF
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ARR_ACC_DER_DEF):
                as_subtype_indication     : subtype_indication
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments

**  derived_subprog

        IS INCLUDED IN:
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
```

```
                sm_derivable              : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments
```

** DESIGNATOR

```
        CLASS MEMBERS:
                USED_OBJECT
                USED_NAME
                used_char
                used_object_id
                used_op
                used_name_id
        IS INCLUDED IN:
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_defn                   : DEF_NAME
                lx_symrep                 : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments
        IS THE DECLARED TYPE OF:
                selected.as_designator
```

** DISCRETE_RANGE

```
        CLASS MEMBERS:
                RANGE
                discrete_subtype
                range
                void
                range_attribute
        IS INCLUDED IN:
                CONSTRAINT
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments
        IS THE DECLARED TYPE OF:
                entry.as_discrete_range
                FOR_REV.as_discrete_range
                AGG_EXP.sm_discrete_range
                slice.as_discrete_range
                choice_range.as_discrete_range
                discrete_range_s.as_list [Seq Of]
```

** discrete_range_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_list                 : Seq Of DISCRETE_RANGE
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                index_constraint.as_discrete_range_s
```

** discrete_subtype

```
        IS INCLUDED IN:
                DISCRETE_RANGE
                CONSTRAINT
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_subtype_indication   : subtype_indication
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** discriminant_id

```
        IS INCLUDED IN:
                COMP_NAME
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_first                : DEF_NAME
            (INHERITED FROM COMP_NAME):
                sm_comp_rep             : COMP_REP_ELEM
            (INHERITED FROM INIT_OBJECT_NAME):
                sm_init_exp             : EXP
            (INHERITED FROM OBJECT_NAME):
                sm_obj_type             : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** dscrmt_constraint

```
        IS INCLUDED IN:
                CONSTRAINT
                ALL_SOURCE
```

```
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_general_assoc_s      : general_assoc_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  dscrmt_decl

        IS INCLUDED IN:
                DSCRMT_PARAM_DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DSCRMT_PARAM_DECL):
                    as_source_name_s        : source_name_s
                    as_exp                  : EXP
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                dscrmt_decl_s.as_list [Seq Of]

**  dscrmt_decl_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                 : Seq Of dscrmt_decl
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                PRIVATE_SPEC.sm_discriminant_s
                incomplete.sm_discriminant_s
                record.sm_discriminant_s
                type_decl.as_dscrmt_decl_s

**  DSCRMT_PARAM_DECL

        CLASS MEMBERS:
                dscrmt_decl
                PARAM
                in
                in_out
                out
        IS INCLUDED IN:
                ITEM
                ALL_DECL
                ALL_SOURCE
```

```
NODE ATTRIBUTES:
    (NODE SPECIFIC):
            as_source_name_s        : source_name_s
            as_exp                  : EXP
            as_name                 : NAME
    (INHERITED FROM ALL_SOURCE):
            lx_srcpos               : source_position
            lx_comments             : comments
```

** entry

```
IS INCLUDED IN:
        SUBP_ENTRY_HEADER
        HEADER
        ALL_SOURCE
NODE ATTRIBUTES:
    (NODE SPECIFIC):
            as_discrete_range       : DISCRETE_RANGE
    (INHERITED FROM SUBP_ENTRY_HEADER):
            as_param_s              : param_s
    (INHERITED FROM ALL_SOURCE):
            lx_srcpos               : source_position
            lx_comments             : comments
```

** entry_call

```
IS INCLUDED IN:
        CALL_STM
        STM_WITH_NAME
        STM
        STM_ELEM
        ALL_SOURCE
NODE ATTRIBUTES:
    (INHERITED FROM CALL_STM):
            as_general_assoc_s      : general_assoc_s
            sm_normalized_param_s   : exp_s
    (INHERITED FROM STM_WITH_NAME):
            as_name                 : NAME
    (INHERITED FROM ALL_SOURCE):
            lx_srcpos               : source_position
            lx_comments             : comments
```

** entry_id

```
IS INCLUDED IN:
        SOURCE_NAME
        DEF_NAME
        ALL_SOURCE
NODE ATTRIBUTES:
    (NODE SPECIFIC):
            sm_spec                 : HEADER
            sm_address              : EXP
    (INHERITED FROM DEF_NAME):
            lx_symrep               : symbol_rep
```

```
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                  : source_position
                    lx_comments                : comments

**  ENTRY_STM

        CLASS MEMBERS:
                cond_entry
                timed_entry
        IS INCLUDED IN:
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_stm_s1                  : stm_s
                    as_stm_s2                  : stm_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                  : source_position
                    lx_comments                : comments

**  ENUM_LITERAL

        CLASS MEMBERS:
                enumeration_id
                character_id
        IS INCLUDED IN:
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_pos                     : Integer
                    sm_rep                     : Integer
            (INHERITED FROM OBJECT_NAME):
                    sm_obj_type                : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep                  : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                  : source_position
                    lx_comments                : comments
        IS THE DECLARED TYPE OF:
                enum_literal_s.as_list [Seq Of]

**  enum_literal_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                    : Seq Of ENUM_LITERAL
            (INHERITED FROM ALL_SOURCE):
```

```
                lx_srcpos                : source_position
                lx_comments              : comments
        IS THE DECLARED TYPE OF:
                enumeration.sm_literal_s
                enumeration_def.as_enum_literal_s
```

** enumeration

```
        IS INCLUDED IN:
                SCALAR
                NON_TASK
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_literal_s             : enum_literal_s
            (INHERITED FROM SCALAR):
                sm_range                 : RANGE
                cd_impl_size             : Integer
            (INHERITED FROM NON_TASK):
                sm_base_type             : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                sm_derived               : TYPE_SPEC
                sm_is_anonymous          : Boolean
```

** enumeration_def

```
        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_enum_literal_s        : enum_literal_s
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments
```

** enumeration_id

```
        IS INCLUDED IN:
                ENUM_LITERAL
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ENUM_LITERAL):
                sm_pos                   : Integer
                sm_rep                   : Integer
            (INHERITED FROM OBJECT_NAME):
                sm_obj_type              : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                lx_symrep                : symbol_rep
```

```
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  exception_decl

    IS INCLUDED IN:
            ID_S_DECL
            DECL
            ITEM
            ALL_DECL
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM ID_S_DECL):
                as_source_name_s        : source_name_s
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  exception_id

    IS INCLUDED IN:
            SOURCE_NAME
            DEF_NAME
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                sm_renames_exc          : NAME
        (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  exit

    .IS INCLUDED IN:
            STM_WITH_EXP_NAME
            STM_WITH_EXP
            STM
            STM_ELEM
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                sm_stm                  : STM
        (INHERITED FROM STM_WITH_EXP_NAME):
                as_name                 : NAME
        (INHERITED FROM STM_WITH_EXP):
                as_exp                  : EXP
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  EXP
```

```
CLASS MEMBERS:
        void
        NAME
        EXP_EXP
        DESIGNATOR
        NAME_EXP
        EXP_VAL
        subtype_allocator
        qualified_allocator
        AGG_EXP
        USED_OBJECT
        USED_NAME
        NAME_VAL
        all
        slice
        indexed
        short_circuit
        numeric_literal
        EXP_VAL_EXP
        null_access
        aggregate
        string_literal
        used_char
        used_object_id
        used_op
        used_name_id
        attribute
        selected
        function_call
        MEMBERSHIP
        QUAL_CONV
        parenthesized
        range_membership
        type_membership
        conversion
        qualified
IS INCLUDED IN:
        GENERAL_ASSOC
        ALL_SOURCE
NODE ATTRIBUTES:
    (INHERITED FROM ALL_SOURCE):
            lx_srcpos                : source_position
            lx_comments              : comments
IS THE DECLARED TYPE OF:
        comp_rep.as_exp
        alignment.as_exp
        NAMED_REP.as_exp
        entry_id.sm_address
        task_spec.sm_storage_size
                 .sm_size
                 .sm_address
        SUBPROG_PACK_NAME.sm_address
        while.as_exp
```

```
                TEST_CLAUSE.as_exp
                STM_WITH_EXP.as_exp
                EXP_VAL_EXP.as_exp
                short_circuit.as_exp1
                             .as_exp2
                NAMED_ASSOC.as_exp
                attribute.as_exp
                exp_s.as_list [Seq Of]
                access.sm_storage_size
                choice_exp.as_exp
                DSCRMT_PARAM_DECL.as_exp
                REAL_CONSTRAINT.as_exp
                range_attribute.as_exp
                range.as_exp1
                     .as_exp2
                UNCONSTRAINED.sm_size
                VC_NAME.sm_address
                INIT_OBJECT_NAME.sm_init_exp
                EXP_DECL.as_exp

**  EXP_DECL

        CLASS MEMBERS:
                OBJECT_DECL
                number_decl
                constant_decl
                variable_decl
        IS INCLUDED IN:
                ID_S_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp                  : EXP
            (INHERITED FROM ID_S_DECL):
                    as_source_name_s        : source_name_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  EXP_EXP

        CLASS MEMBERS:
                EXP_VAL
                subtype_allocator
                qualified_allocator
                AGG_EXP
                short_circuit
                numeric_literal
                EXP_VAL_EXP
                null_access
                aggregate
```

```
                string_literal
                MEMBERSHIP
                QUAL_CONV
                parenthesized
                range_membership
                type_membership
                conversion
                qualified
        IS INCLUDED IN:
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** exp_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                 : Seq Of EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                function_call.sm_normalized_param_s
                CALL_STM.sm_normalized_param_s
                indexed.as_exp_s
                constrained_record.sm_normalized_dscrmt_s
```

** EXP_VAL

```
        CLASS MEMBERS:
                short_circuit
                numeric_literal
                EXP_VAL_EXP
                null_access
                MEMBERSHIP
                QUAL_CONV
                parenthesized
                range_membership
                type_membership
                conversion
                qualified
        IS INCLUDED IN:
                EXP_EXP
                EXP
                GENERAL_ASSOC
```

```
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_value                : value
            (INHERITED FROM EXP_EXP):
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** EXP_VAL_EXP

```
        CLASS MEMBERS:
                MEMBERSHIP
                QUAL_CONV
                parenthesized
                range_membership
                type_membership
                conversion
                qualified
        IS INCLUDED IN:
                EXP_VAL
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp                  : EXP
            (INHERITED FROM EXP_VAL):
                    sm_value                : value
            (INHERITED FROM EXP_EXP):
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** fixed

```
        IS INCLUDED IN:
                REAL
                SCALAR
                NON_TASK
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    cd_impl_small           : value
            (INHERITED FROM REAL):
                    sm_accuracy             : value
            (INHERITED FROM SCALAR):
                    sm_range                : RANGE
                    cd_impl_size            : Integer
```

```
        (INHERITED FROM NON_TASK):
                sm_base_type            : TYPE_SPEC
        (INHERITED FROM DERIVABLE_SPEC):
                sm_derived              : TYPE_SPEC
                sm_is_anonymous         : Boolean
```

** fixed_constraint

```
    IS INCLUDED IN:
            REAL_CONSTRAINT
            CONSTRAINT
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM REAL_CONSTRAINT):
                sm_type_spec            : TYPE_SPEC
                as_exp                  : EXP
                as_range                : RANGE
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** fixed_def

```
    IS INCLUDED IN:
            CONSTRAINED_DEF
            TYPE_DEF
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM CONSTRAINED_DEF):
                as_constraint           : CONSTRAINT
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** float

```
    IS INCLUDED IN:
            REAL
            SCALAR
            NON_TASK
            FULL_TYPE_SPEC
            DERIVABLE_SPEC
            TYPE_SPEC
    NODE ATTRIBUTES:
        (INHERITED FROM REAL):
                sm_accuracy             : value
        (INHERITED FROM SCALAR):
                sm_range                : RANGE
                cd_impl_size            : Integer
        (INHERITED FROM NON_TASK):
                sm_base_type            : TYPE_SPEC
        (INHERITED FROM DERIVABLE_SPEC):
                sm_derived              : TYPE_SPEC
                sm_is_anonymous         : Boolean
```

```
**  float_constraint

        IS INCLUDED IN:
                REAL_CONSTRAINT
                CONSTRAINT
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM REAL_CONSTRAINT):
                sm_type_spec            : TYPE_SPEC
                as_exp                  : EXP
                as_range                : RANGE
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  float_def

        IS INCLUDED IN:
                CONSTRAINED_DEF
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM CONSTRAINED_DEF):
                as_constraint           : CONSTRAINT
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  for

        IS INCLUDED IN:
                FOR_REV
                ITERATION
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM FOR_REV):
                as_source_name          : SOURCE_NAME
                as_discrete_range       : DISCRETE_RANGE
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  FOR_REV

        CLASS MEMBERS:
                for
                reverse
        IS INCLUDED IN:
                ITERATION
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_source_name          : SOURCE_NAME
```

```
                as_discrete_range        : DISCRETE_RANGE
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

**  formal_dscrt_def

        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

**  formal_fixed_def

        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

**  formal_float_def

        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

**  formal_integer_def

        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

**  FULL_TYPE_SPEC

        CLASS MEMBERS:
                task_spec
                NON_TASK
                SCALAR
                CONSTRAINED
                UNCONSTRAINED
                enumeration
```

```
                REAL
                integer
                constrained_array
                constrained_access
                constrained_record
                UNCONSTRAINED_COMPOSITE
                access
                float
                fixed
                array
                record
        IS INCLUDED IN:
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (INHERITED FROM DERIVABLE_SPEC):
                    sm_derived              : TYPE_SPEC
                    sm_is_anonymous         : Boolean
```

** function_call

```
        IS INCLUDED IN:
                NAME_VAL
                NAME_EXP
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_general_assoc_s      : general_assoc_s
                    sm_normalized_param_s   : exp_s
                    lx_prefix               : Boolean
            (INHERITED FROM NAME_VAL):
                    sm_value                : value
            (INHERITED FROM NAME_EXP):
                    as_name                 : NAME
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** function_id

```
        IS INCLUDED IN:
                SUBPROG_NAME
                SUBPROG_PACK_NAME
                NON_TASK_NAME
                UNIT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM SUBPROG_NAME):
```

```
                sm_is_inline              : Boolean
                sm_interface              : PREDEF_NAME
        (INHERITED FROM SUBPROG_PACK_NAME):
                sm_unit_desc              : UNIT_DESC
                sm_address                : EXP
        (INHERITED FROM NON_TASK_NAME):
                sm_spec                   : HEADER
        (INHERITED FROM UNIT_NAME):
                sm_first                  : DEF_NAME
        (INHERITED FROM DEF_NAME):
                lx_symrep                 : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments
```

** function_spec

```
    IS INCLUDED IN:
            SUBP_ENTRY_HEADER
            HEADER
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_name                   : NAME
        (INHERITED FROM SUBP_ENTRY_HEADER):
                as_param_s                : param_s
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments
```

** GENERAL_ASSOC

```
    CLASS MEMBERS:
            NAMED_ASSOC
            EXP
            named
            assoc
            void
            NAME
            EXP_EXP
            DESIGNATOR
            NAME_EXP
            EXP_VAL
            subtype_allocator
            qualified_allocator
            AGG_EXP
            USED_OBJECT
            USED_NAME
            NAME_VAL
            all
            slice
            indexed
            short_circuit
            numeric_literal
```

```
                EXP_VAL_EXP
                null_access
                aggregate
                string_literal
                used_char
                used_object_id
                used_op
                used_name_id
                attribute
                selected
                function_call
                MEMBERSHIP
                QUAL_CONV
                parenthesized
                range_membership
                type_membership
                conversion
                qualified
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                general_assoc_s.as_list [Seq Of]

**  general_assoc_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                 : Seq Of GENERAL_ASSOC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                instantiation.as_general_assoc_s
                function_call.as_general_assoc_s
                CALL_STM.as_general_assoc_s
                aggregate.as_general_assoc_s
                         .sm_normalized_comp_s
                dscrmt_constraint.as_general_assoc_s
                pragma.as_general_assoc_s

**  generic_decl

        IS INCLUDED IN:
                UNIT_DECL
                ID_DECL
                DECL
                ITEM
```

```
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_item_s                : item_s
            (INHERITED FROM UNIT_DECL):
                as_header                : HEADER
            (INHERITED FROM ID_DECL):
                as_source_name           : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments
```

** generic_id

```
        IS INCLUDED IN:
                NON_TASK_NAME
                UNIT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_generic_param_s       : item_s
                sm_is_inline             : Boolean
                sm_body                  : BODY
            (INHERITED FROM NON_TASK_NAME):
                sm_spec                  : HEADER
            (INHERITED FROM UNIT_NAME):
                sm_first                 : DEF_NAME
            (INHERITED FROM DEF_NAME):
                lx_symrep                : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments
```

** GENERIC_PARAM

```
        CLASS MEMBERS:
                name_default
                no_default
                box_default
        IS INCLUDED IN:
                UNIT_KIND
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments
```

** goto

```
        IS INCLUDED IN:
```

```
                STM_WITH_NAME
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM STM_WITH_NAME):
                    as_name                  : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  HEADER

        CLASS MEMBERS:
                SUBP_ENTRY_HEADER
                package_spec
                procedure_spec
                function_spec
                entry
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments
        IS THE DECLARED TYPE OF:
                entry_id.sm_spec
                subprogram_body.as_header
                NON_TASK_NAME.sm_spec
                UNIT_DECL.as_header

**  ID_DECL

        CLASS MEMBERS:
                type_decl
                UNIT_DECL
                task_decl
                subtype_decl
                SIMPLE_RENAME_DECL
                generic_decl
                NON_GENERIC_DECL
                renames_obj_decl
                renames_exc_decl
                subprog_entry_decl
                package_decl
        IS INCLUDED IN:
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_source_name           : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
```

```
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  ID_S_DECL

        CLASS MEMBERS:
                EXP_DECL
                deferred_constant_decl
                exception_decl
                OBJECT_DECL
                number_decl
                constant_decl
                variable_decl
        IS INCLUDED IN:
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_source_name_s         : source_name_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  if

        IS INCLUDED IN:
                CLAUSES_STM
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM CLAUSES_STM):
                    as_test_clause_elem_s    : test_clause_elem_s
                    as_stm_s                 : stm_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  implicit_not_eq

        IS INCLUDED IN:
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_equal                 : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                : source_position
                    lx_comments              : comments

**  in
```

```
        IS INCLUDED IN:
                PARAM
                DSCRMT_PARAM_DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                lx_default                : Boolean
            (INHERITED FROM DSCRMT_PARAM_DECL):
                as_source_name_s          : source_name_s
                as_exp                    : EXP
                as_name                   : NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments

**  in_id

        IS INCLUDED IN:
                PARAM_NAME
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM PARAM_NAME):
                sm_first                  : DEF_NAME
            (INHERITED FROM INIT_OBJECT_NAME):
                sm_init_exp               : EXP
            (INHERITED FROM OBJECT_NAME):
                sm_obj_type               : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                lx_symrep                 : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments

**  in_op

        IS INCLUDED IN:
                MEMBERSHIP_OP
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                 : source_position
                lx_comments               : comments

**  in_out

        IS INCLUDED IN:
                PARAM
                DSCRMT_PARAM_DECL
```

```
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DSCRMT_PARAM_DECL):
                    as_source_name_s        : source_name_s
                    as_exp                  : EXP
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** in_out_id

```
        IS INCLUDED IN:
                PARAM_NAME
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM PARAM_NAME):
                    sm_first                : DEF_NAME
            (INHERITED FROM INIT_OBJECT_NAME):
                    sm_init_exp             : EXP
            (INHERITED FROM OBJECT_NAME):
                    sm_obj_type             : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** incomplete

```
        IS INCLUDED IN:
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_discriminant_s       : dscrmt_decl_s
```

** index

```
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name                 : NAME
                    sm_type_spec            : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
```

```
                index_s.as_list [Seq Of]

**  index_constraint

        IS INCLUDED IN:
                CONSTRAINT
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_discrete_range_s     : discrete_range_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  index_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                 : Seq Of index
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                array.sm_index_s
                unconstrained_array_def.as_index_s

**  indexed

        IS INCLUDED IN:
                NAME_EXP
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp_s                : exp_s
            (INHERITED FROM NAME_EXP):
                    as_name                 : NAME
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  INIT_OBJECT_NAME

        CLASS MEMBERS:
                VC_NAME
                number_id
                COMP_NAME
                PARAM_NAME
```

```
                variable_id
                constant_id
                component_id
                discriminant_id
                in_id
                out_id
                in_out_id
        IS INCLUDED IN:
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_init_exp             : EXP
            (INHERITED FROM OBJECT_NAME):
                    sm_obj_type             : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** instantiation

```
        IS INCLUDED IN:
                RENAME_INSTANT
                UNIT_KIND
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_general_assoc_s      : general_assoc_s
                    sm_decl_s               : decl_s
            (INHERITED FROM RENAME_INSTANT):
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** integer

```
        IS INCLUDED IN:
                SCALAR
                NON_TASK
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (INHERITED FROM SCALAR):
                    sm_range                : RANGE
                    cd_impl_size            : Integer
            (INHERITED FROM NON_TASK):
                    sm_base_type            : TYPE_SPEC
```

```
        (INHERITED FROM DERIVABLE_SPEC):
                sm_derived              : TYPE_SPEC
                sm_is_anonymous         : Boolean

**  integer_def

    IS INCLUDED IN:
            CONSTRAINED_DEF
            TYPE_DEF
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM CONSTRAINED_DEF):
                as_constraint           : CONSTRAINT
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  ITEM

    CLASS MEMBERS:
            DSCRMT_PARAM_DECL
            DECL
            SUBUNIT_BODY
            dscrmt_decl
            PARAM
            ID_S_DECL
            ID_DECL
            null_comp_decl
            REP
            USE_PRAGMA
            subprogram_body
            task_body
            package_body
            in
            in_out
            out
            EXP_DECL
            deferred_constant_decl
            exception_decl
            type_decl
            UNIT_DECL
            task_decl
            subtype_decl
            SIMPLE_RENAME_DECL
            void
            NAMED_REP
            record_rep
            use
            pragma
            OBJECT_DECL
            number_decl
            generic_decl
            NON_GENERIC_DECL
            renames_obj_decl
```

```
                renames_exc_decl
                length_enum_rep
                address
                constant_decl
                variable_decl
                subprog_entry_decl
                package_decl
        IS INCLUDED IN:
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                item_s.as_list [Seq Of]

**  item_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                 : Seq Of ITEM
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                generic_id.sm_generic_param_s
                generic_decl.as_item_s
                block_body.as_item_s

**  ITERATION

        CLASS MEMBERS:
                void
                FOR_REV
                while
                for
                reverse
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                loop.as_iteration

**  iteration_id

        IS INCLUDED IN:
                OBJECT_NAME
```

```
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM OBJECT_NAME):
                    sm_obj_type             : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  l_private

        IS INCLUDED IN:
                PRIVATE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (INHERITED FROM PRIVATE_SPEC):
                    sm_discriminant_s       : dscrmt_decl_s
                    sm_type_spec            : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                    sm_derived              : TYPE_SPEC
                    sm_is_anonymous         : Boolean

**  l_private_def

        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  l_private_type_id

        IS INCLUDED IN:
                TYPE_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM TYPE_NAME):
                    sm_type_spec            : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  label_id
```

```
        IS INCLUDED IN:
                LABEL_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM LABEL_NAME):
                    sm_stm                  : STM
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** LABEL_NAME

```
        CLASS MEMBERS:
                label_id
                block_loop_id
        IS INCLUDED IN:
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_stm                  : STM
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** labeled

```
        IS INCLUDED IN:
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_source_name_s        : source_name_s
                    as_stm                  : STM
                    as_pragma_s             : pragma_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** length_enum_rep

```
        IS INCLUDED IN:
                NAMED_REP
                REP
                DECL
                ITEM
```

```
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM NAMED_REP):
                    as_exp                  : EXP
            (INHERITED FROM REP):
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** loop

```
        IS INCLUDED IN:
                BLOCK_LOOP
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_iteration            : ITERATION
                    as_stm_s                : stm_s
            (INHERITED FROM BLOCK_LOOP):
                    as_source_name          : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** MEMBERSHIP

```
        CLASS MEMBERS:
                range_membership
                type_membership
        IS INCLUDED IN:
                EXP_VAL_EXP
                EXP_VAL
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_membership_op        : MEMBERSHIP_OP
            (INHERITED FROM EXP_VAL_EXP):
                    as_exp                  : EXP
            (INHERITED FROM EXP_VAL):
                    sm_value                : value
            (INHERITED FROM EXP_EXP):
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** MEMBERSHIP_OP

```
        CLASS MEMBERS:
                in_op
                not_in
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                MEMBERSHIP.as_membership_op

**  NAME

        CLASS MEMBERS:
                DESIGNATOR
                NAME_EXP
                void
                USED_OBJECT
                USED_NAME
                NAME_VAL
                all
                slice
                indexed
                used_char
                used_object_id
                used_op
                used_name_id
                attribute
                selected
                function_call
        IS INCLUDED IN:
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                comp_rep.as_name
                REP.as_name
                name_default.as_name
                exception_id.sm_renames_exc
                subunit.as_name
                name_s.as_list [Seq Of]
                accept.as_name
                RENAME_INSTANT.as_name
                renames_obj_decl.as_type_mark_name
                SIMPLE_RENAME_DECL.as_name
                deferred_constant_decl.as_name
                function_spec.as_name
                STM_WITH_NAME.as_name
```

```
                STM_WITH_EXP_NAME.as_name
                QUAL_CONV.as_name
                type_membership.as_name
                NAME_EXP.as_name
                variant_part.as_name
                DSCRMT_PARAM_DECL.as_name
                index.as_name
                range_attribute.as_name
                subtype_indication.as_name

**  name_default

        IS INCLUDED IN:
                GENERIC_PARAM
                UNIT_KIND
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  NAME_EXP

        CLASS MEMBERS:
                NAME_VAL
                all
                slice
                indexed
                attribute
                selected
                function_call
        IS INCLUDED IN:
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name                 : NAME
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  name_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
```

```
                as_list                       : Seq Of NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                     : source_position
                lx_comments                   : comments
        IS THE DECLARED TYPE OF:
                with.as_name_s
                abort.as_name_s
                use.as_name_s
```

** NAME_VAL

```
        CLASS MEMBERS:
                attribute
                selected
                function_call
        IS INCLUDED IN:
                NAME_EXP
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_value                      : value
            (INHERITED FROM NAME_EXP):
                as_name                       : NAME
                sm_exp_type                   : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                     : source_position
                lx_comments                   : comments
```

** named

```
        IS INCLUDED IN:
                NAMED_ASSOC
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_choice_s                   : choice_s
            (INHERITED FROM NAMED_ASSOC):
                as_exp                        : EXP
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                     : source_position
                lx_comments                   : comments
```

** NAMED_ASSOC

```
        CLASS MEMBERS:
                named
                assoc
        IS INCLUDED IN:
                GENERAL_ASSOC
                ALL_SOURCE
```

```
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp                  : EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** NAMED_REP

```
        CLASS MEMBERS:
                length_enum_rep
                address
        IS INCLUDED IN:
                REP
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp                  : EXP
            (INHERITED FROM REP):
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** no_default

```
        IS INCLUDED IN:
                GENERIC_PARAM
                UNIT_KIND
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** NON_GENERIC_DECL

```
        CLASS MEMBERS:
                subprog_entry_decl
                package_decl
        IS INCLUDED IN:
                UNIT_DECL
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_unit_kind            : UNIT_KIND
```

```
        (INHERITED FROM UNIT_DECL):
                as_header               : HEADER
        (INHERITED FROM ID_DECL):
                as_source_name          : SOURCE_NAME
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  NON_TASK

    CLASS MEMBERS:
            SCALAR
            CONSTRAINED
            UNCONSTRAINED
            enumeration
            REAL
            integer
            constrained_array
            constrained_access
            constrained_record
            UNCONSTRAINED_COMPOSITE
            access
            float
            fixed
            array
            record
    IS INCLUDED IN:
            FULL_TYPE_SPEC
            DERIVABLE_SPEC
            TYPE_SPEC
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                sm_base_type            : TYPE_SPEC
        (INHERITED FROM DERIVABLE_SPEC):
                sm_derived              : TYPE_SPEC
                sm_is_anonymous         : Boolean

**  NON_TASK_NAME

    CLASS MEMBERS:
            SUBPROG_PACK_NAME
            generic_id
            SUBPROG_NAME
            package_id
            procedure_id
            operator_id
            function_id
    IS INCLUDED IN:
            UNIT_NAME
            SOURCE_NAME
            DEF_NAME
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
```

```
                sm_spec                 : HEADER
            (INHERITED FROM UNIT_NAME):
                sm_first                : DEF_NAME
            (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  not_in

        IS INCLUDED IN:
                MEMBERSHIP_OP
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  null_access

        IS INCLUDED IN:
                EXP_VAL
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM EXP_VAL):
                sm_value                : value
            (INHERITED FROM EXP_EXP):
                sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  null_comp_decl

        IS INCLUDED IN:
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  null_stm

        IS INCLUDED IN:
                STM
                STM_ELEM
                ALL_SOURCE
```

```
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  number_decl

        IS INCLUDED IN:
                EXP_DECL
                ID_S_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM EXP_DECL):
                    as_exp                  : EXP
            (INHERITED FROM ID_S_DECL):
                    as_source_name_s        : source_name_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  number_id

        IS INCLUDED IN:
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM INIT_OBJECT_NAME):
                    sm_init_exp             : EXP
            (INHERITED FROM OBJECT_NAME):
                    sm_obj_type             : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  numeric_literal

        IS INCLUDED IN:
                EXP_VAL
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    lx_numrep               : number_rep
            (INHERITED FROM EXP_VAL):
```

```
                sm_value                    : value
            (INHERITED FROM EXP_EXP):
                sm_exp_type                 : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
```

** OBJECT_DECL

```
        CLASS MEMBERS:
                constant_decl
                variable_decl
        IS INCLUDED IN:
                EXP_DECL
                ID_S_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_type_def                 : TYPE_DEF
            (INHERITED FROM EXP_DECL):
                as_exp                      : EXP
            (INHERITED FROM ID_S_DECL):
                as_source_name_s            : source_name_s
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
```

** OBJECT_NAME

```
        CLASS MEMBERS:
                INIT_OBJECT_NAME
                ENUM_LITERAL
                iteration_id
                VC_NAME
                number_id
                COMP_NAME
                PARAM_NAME
                enumeration_id
                character_id
                variable_id
                constant_id
                component_id
                discriminant_id
                in_id
                out_id
                in_out_id
        IS INCLUDED IN:
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
```

```
            (NODE SPECIFIC):
                    sm_obj_type                 : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep                   : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** operator_id

```
        IS INCLUDED IN:
                SUBPROG_NAME
                SUBPROG_PACK_NAME
                NON_TASK_NAME
                UNIT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM SUBPROG_NAME):
                    sm_is_inline                : Boolean
                    sm_interface                : PREDEF_NAME
            (INHERITED FROM SUBPROG_PACK_NAME):
                    sm_unit_desc                : UNIT_DESC
                    sm_address                  : EXP
            (INHERITED FROM NON_TASK_NAME):
                    sm_spec                     : HEADER
            (INHERITED FROM UNIT_NAME):
                    sm_first                    : DEF_NAME
            (INHERITED FROM DEF_NAME):
                    lx_symrep                   : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** or_else

```
        IS INCLUDED IN:
                SHORT_CIRCUIT_OP
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** out

```
        IS INCLUDED IN:
                PARAM
                DSCRMT_PARAM_DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
```

```
        (INHERITED FROM DSCRMT_PARAM_DECL):
                as_source_name_s        : source_name_s
                as_exp                  : EXP
                as_name                 : NAME
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** out_id

```
    IS INCLUDED IN:
            PARAM_NAME
            INIT_OBJECT_NAME
            OBJECT_NAME
            SOURCE_NAME
            DEF_NAME
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM PARAM_NAME):
                sm_first                : DEF_NAME
        (INHERITED FROM INIT_OBJECT_NAME):
                sm_init_exp             : EXP
        (INHERITED FROM OBJECT_NAME):
                sm_obj_type             : TYPE_SPEC
        (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** package_body

```
    IS INCLUDED IN:
            SUBUNIT_BODY
            ITEM
            ALL_DECL
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM SUBUNIT_BODY):
                as_source_name          : SOURCE_NAME
                as_body                 : BODY
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** package_decl

```
    IS INCLUDED IN:
            NON_GENERIC_DECL
            UNIT_DECL
            ID_DECL
            DECL
            ITEM
            ALL_DECL
```

```
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM NON_GENERIC_DECL):
                    as_unit_kind            : UNIT_KIND
            (INHERITED FROM UNIT_DECL):
                    as_header               : HEADER
            (INHERITED FROM ID_DECL):
                    as_source_name          : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** package_id

```
        IS INCLUDED IN:
                SUBPROG_PACK_NAME
                NON_TASK_NAME
                UNIT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM SUBPROG_PACK_NAME):
                    sm_unit_desc            : UNIT_DESC
                    sm_address              : EXP
            (INHERITED FROM NON_TASK_NAME):
                    sm_spec                 : HEADER
            (INHERITED FROM UNIT_NAME):
                    sm_first                : DEF_NAME
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** package_spec

```
        IS INCLUDED IN:
                HEADER
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_decl_s1              : decl_s
                    as_decl_s2              : decl_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** PARAM

```
        CLASS MEMBERS:
                in
                in_out
                out
```

```
        IS INCLUDED IN:
                DSCRMT_PARAM_DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DSCRMT_PARAM_DECL):
                as_source_name_s         : source_name_s
                as_exp                   : EXP
                as_name                  : NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments
        IS THE DECLARED TYPE OF:
                param_s.as_list [Seq Of]
```

** PARAM_NAME

```
        CLASS MEMBERS:
                in_id
                out_id
                in_out_id
        IS INCLUDED IN:
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_first                 : DEF_NAME
            (INHERITED FROM INIT_OBJECT_NAME):
                sm_init_exp              : EXP
            (INHERITED FROM OBJECT_NAME):
                sm_obj_type              : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                lx_symrep                : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments
```

** param_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_list                  : Seq Of PARAM
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments
        IS THE DECLARED TYPE OF:
                accept.as_param_s
```

```
                SUBP_ENTRY_HEADER.as_param_s

**  parenthesized

        IS INCLUDED IN:
                EXP_VAL_EXP
                EXP_VAL
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM EXP_VAL_EXP):
                as_exp                  : EXP
            (INHERITED FROM EXP_VAL):
                sm_value                : value
            (INHERITED FROM EXP_EXP):
                sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  pragma

        IS INCLUDED IN:
                USE_PRAGMA
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_used_name_id         : used_name_id
                as_general_assoc_s      : general_assoc_s
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                comp_rep_pragma.as_pragma
                context_pragma.as_pragma
                pragma_s.as_list [Seq Of]
                select_alt_pragma.as_pragma
                alternative_pragma.as_pragma
                stm_pragma.as_pragma
                variant_pragma.as_pragma

**  pragma_id

        IS INCLUDED IN:
                PREDEF_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
```

```
                sm_argument_id_s        : argument_id_s
            (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** pragma_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_list                 : Seq Of pragma
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                alignment.as_pragma_s
                compilation_unit.as_pragma_s
                labeled.as_pragma_s
                comp_list.as_pragma_s
```

** PREDEF_NAME

```
        CLASS MEMBERS:
                attribute_id
                void
                bltn_operator_id
                argument_id
                pragma_id
        IS INCLUDED IN:
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                SUBPROG_NAME.sm_interface
```

** private

```
        IS INCLUDED IN:
                PRIVATE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (INHERITED FROM PRIVATE_SPEC):
                sm_discriminant_s       : dscrmt_decl_s
                sm_type_spec            : TYPE_SPEC
```

```
        (INHERITED FROM DERIVABLE_SPEC):
                sm_derived              : TYPE_SPEC
                sm_is_anonymous         : Boolean
```

** private_def

```
    IS INCLUDED IN:
            TYPE_DEF
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** PRIVATE_SPEC

```
    CLASS MEMBERS:
            private
            i_private
    IS INCLUDED IN:
            DERIVABLE_SPEC
            TYPE_SPEC
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                sm_discriminant_s       : dscrmt_decl_s
                sm_type_spec            : TYPE_SPEC
        (INHERITED FROM DERIVABLE_SPEC):
                sm_derived              : TYPE_SPEC
                sm_is_anonymous         : Boolean
```

** private_type_id

```
    IS INCLUDED IN:
            TYPE_NAME
            SOURCE_NAME
            DEF_NAME
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM TYPE_NAME):
                sm_type_spec            : TYPE_SPEC
        (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** procedure_call

```
    IS INCLUDED IN:
            CALL_STM
            STM_WITH_NAME
            STM
            STM_ELEM
            ALL_SOURCE
```

```
        NODE ATTRIBUTES:
            (INHERITED FROM CALL_STM):
                    as_general_assoc_s      : general_assoc_s
                    sm_normalized_param_s   : exp_s
            (INHERITED FROM STM_WITH_NAME):
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** procedure_id

```
        IS INCLUDED IN:
                SUBPROG_NAME
                SUBPROG_PACK_NAME
                NON_TASK_NAME
                UNIT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM SUBPROG_NAME):
                    sm_is_inline            : Boolean
                    sm_interface            : PREDEF_NAME
            (INHERITED FROM SUBPROG_PACK_NAME):
                    sm_unit_desc            : UNIT_DESC
                    sm_address              : EXP
            (INHERITED FROM NON_TASK_NAME):
                    sm_spec                 : HEADER
            (INHERITED FROM UNIT_NAME):
                    sm_first                : DEF_NAME
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** procedure_spec

```
        IS INCLUDED IN:
                SUBP_ENTRY_HEADER
                HEADER
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM SUBP_ENTRY_HEADER):
                    as_param_s              : param_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** QUAL_CONV

```
        CLASS MEMBERS:
                conversion
```

```
            qualified
        IS INCLUDED IN:
            EXP_VAL_EXP
            EXP_VAL
            EXP_EXP
            EXP
            GENERAL_ASSOC
            ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_name                 : NAME
            (INHERITED FROM EXP_VAL_EXP):
                as_exp                  : EXP
            (INHERITED FROM EXP_VAL):
                sm_value                : value
            (INHERITED FROM EXP_EXP):
                sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** qualified

```
        IS INCLUDED IN:
            QUAL_CONV
            EXP_VAL_EXP
            EXP_VAL
            EXP_EXP
            EXP
            GENERAL_ASSOC
            ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM QUAL_CONV):
                as_name                 : NAME
            (INHERITED FROM EXP_VAL_EXP):
                as_exp                  : EXP
            (INHERITED FROM EXP_VAL):
                sm_value                : value
            (INHERITED FROM EXP_EXP):
                sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
            qualified_allocator.as_qualified
```

** qualified_allocator

```
        IS INCLUDED IN:
            EXP_EXP
            EXP
            GENERAL_ASSOC
            ALL_SOURCE
        NODE ATTRIBUTES:
```

```
            (NODE SPECIFIC):
                    as_qualified            : qualified
            (INHERITED FROM EXP_EXP):
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** raise

```
        IS INCLUDED IN:
                STM_WITH_NAME
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM STM_WITH_NAME):
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** RANGE

```
        CLASS MEMBERS:
                range
                void
                range_attribute
        IS INCLUDED IN:
                DISCRETE_RANGE
                CONSTRAINT
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_type_spec            : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                comp_rep.as_range
                range_membership.as_range
                REAL_CONSTRAINT.as_range
                SCALAR.sm_range
```

** range

```
        IS INCLUDED IN:
                RANGE
                DISCRETE_RANGE
                CONSTRAINT
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp1                 : EXP
```

```
                as_exp2                 : EXP
          (INHERITED FROM RANGE):
                sm_type_spec            : TYPE_SPEC
          (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** range_attribute

```
      IS INCLUDED IN:
            RANGE
            DISCRETE_RANGE
            CONSTRAINT
            ALL_SOURCE
      NODE ATTRIBUTES:
          (NODE SPECIFIC):
                as_name                 : NAME
                as_exp                  : EXP
                as_used_name_id         : used_name_id
          (INHERITED FROM RANGE):
                sm_type_spec            : TYPE_SPEC
          (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** range_membership

```
      IS INCLUDED IN:
            MEMBERSHIP
            EXP_VAL_EXP
            EXP_VAL
            EXP_EXP
            EXP
            GENERAL_ASSOC
            ALL_SOURCE
      NODE ATTRIBUTES:
          (NODE SPECIFIC):
                as_range                : RANGE
          (INHERITED FROM MEMBERSHIP):
                as_membership_op        : MEMBERSHIP_OP
          (INHERITED FROM EXP_VAL_EXP):
                as_exp                  : EXP
          (INHERITED FROM EXP_VAL):
                sm_value                : value
          (INHERITED FROM EXP_EXP):
                sm_exp_type             : TYPE_SPEC
          (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** REAL

```
      CLASS MEMBERS:
            float
```

```
            fixed
        IS INCLUDED IN:
            SCALAR
            NON_TASK
            FULL_TYPE_SPEC
            DERIVABLE_SPEC
            TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_accuracy             : value
            (INHERITED FROM SCALAR):
                sm_range                : RANGE
                cd_impl_size            : Integer
            (INHERITED FROM NON_TASK):
                sm_base_type            : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                sm_derived              : TYPE_SPEC
                sm_is_anonymous         : Boolean

**  REAL_CONSTRAINT

        CLASS MEMBERS:
            float_constraint
            fixed_constraint
        IS INCLUDED IN:
            CONSTRAINT
            ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_type_spec            : TYPE_SPEC
                as_exp                  : EXP
                as_range                : RANGE
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  record

        IS INCLUDED IN:
            UNCONSTRAINED_COMPOSITE
            UNCONSTRAINED
            NON_TASK
            FULL_TYPE_SPEC
            DERIVABLE_SPEC
            TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_discriminant_s       : dscrmt_decl_s
                sm_representation       : REP
                sm_comp_list            : comp_list
            (INHERITED FROM UNCONSTRAINED_COMPOSITE):
                sm_is_limited           : Boolean
                sm_is_packed            : Boolean
            (INHERITED FROM UNCONSTRAINED):
```

```
                sm_size                    : EXP
            (INHERITED FROM NON_TASK):
                sm_base_type               : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                sm_derived                 : TYPE_SPEC
                sm_is_anonymous            : Boolean
```

** record_def

```
        IS INCLUDED IN:
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_comp_list               : comp_list
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                  : source_position
                lx_comments                : comments
```

** record_rep

```
        IS INCLUDED IN:
                REP
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_alignment_clause        : ALIGNMENT_CLAUSE
                as_comp_rep_s              : comp_rep_s
            (INHERITED FROM REP):
                as_name                    : NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                  : source_position
                lx_comments                : comments
```

** RENAME_INSTANT

```
        CLASS MEMBERS:
                renames_unit
                instantiation
        IS INCLUDED IN:
                UNIT_KIND
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_name                    : NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                  : source_position
                lx_comments                : comments
```

** renames_exc_decl

```
        IS INCLUDED IN:
                SIMPLE_RENAME_DECL
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM SIMPLE_RENAME_DECL):
                    as_name                 : NAME
            (INHERITED FROM ID_DECL):
                    as_source_name          : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** renames_obj_decl

```
        IS INCLUDED IN:
                SIMPLE_RENAME_DECL
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_type_mark_name       : NAME
            (INHERITED FROM SIMPLE_RENAME_DECL):
                    as_name                 : NAME
            (INHERITED FROM ID_DECL):
                    as_source_name          : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** renames_unit

```
        IS INCLUDED IN:
                RENAME_INSTANT
                UNIT_KIND
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM RENAME_INSTANT):
                    as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** REP

```
        CLASS MEMBERS:
```

```
                void
                NAMED_REP
                record_rep
                length_enum_rep
                address
        IS INCLUDED IN:
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name                     : NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                record.sm_representation
```

** return

```
        IS INCLUDED IN:
                STM_WITH_EXP
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM STM_WITH_EXP):
                    as_exp                      : EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** reverse

```
        IS INCLUDED IN:
                FOR_REV
                ITERATION
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM FOR_REV):
                    as_source_name              : SOURCE_NAME
                    as_discrete_range           : DISCRETE_RANGE
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** SCALAR

```
        CLASS MEMBERS:
                enumeration
                REAL
                integer
                float
```

```
            fixed
      IS INCLUDED IN:
            NON_TASK
            FULL_TYPE_SPEC
            DERIVABLE_SPEC
            TYPE_SPEC
      NODE ATTRIBUTES:
         (NODE SPECIFIC):
               sm_range                : RANGE
               cd_impl_size            : Integer
         (INHERITED FROM NON_TASK):
               sm_base_type            : TYPE_SPEC
         (INHERITED FROM DERIVABLE_SPEC):
               sm_derived              : TYPE_SPEC
               sm_is_anonymous         : Boolean
      IS THE DECLARED TYPE OF:
            scalar_s.as_list [Seq Of]

**  scalar_s

      IS INCLUDED IN:
            SEQUENCES
            ALL_SOURCE
      NODE ATTRIBUTES:
         (NODE SPECIFIC):
               as_list                 : Seq Of SCALAR
         (INHERITED FROM ALL_SOURCE):
               lx_srcpos               : source_position
               lx_comments             : comments
      IS THE DECLARED TYPE OF:
            constrained_array.sm_index_subtype_s

**  select_alt_pragma

      IS INCLUDED IN:
            TEST_CLAUSE_ELEM
            ALL_SOURCE
      NODE ATTRIBUTES:
         (NODE SPECIFIC):
               as_pragma               : pragma
         (INHERITED FROM ALL_SOURCE):
               lx_srcpos               : source_position
               lx_comments             : comments

**  select_alternative

      IS INCLUDED IN:
            TEST_CLAUSE
            TEST_CLAUSE_ELEM
            ALL_SOURCE
      NODE ATTRIBUTES:
         (INHERITED FROM TEST_CLAUSE):
               as_exp                  : EXP
               as_stm_s                : stm_s
```

```
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  selected

    IS INCLUDED IN:
            NAME_VAL
            NAME_EXP
            NAME
            EXP
            GENERAL_ASSOC
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_designator           : DESIGNATOR
        (INHERITED FROM NAME_VAL):
                sm_value                : value
        (INHERITED FROM NAME_EXP):
                as_name                 : NAME
                sm_exp_type             : TYPE_SPEC
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  selective_wait

    IS INCLUDED IN:
            CLAUSES_STM
            STM
            STM_ELEM
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM CLAUSES_STM):
                as_test_clause_elem_s   : test_clause_elem_s
                as_stm_s                : stm_s
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  SEQUENCES

    CLASS MEMBERS:
            alternative_s
            variant_s
            use_pragma_s
            test_clause_elem_s
            stm_s
            source_name_s
            scalar_s
            pragma_s
            param_s
            name_s
            index_s
```

```
                item_s
                exp_s
                enum_literal_s
                discrete_range_s
                general_assoc_s
                dscrmt_decl_s
                decl_s
                context_elem_s
                compltn_unit_s
                comp_rep_s
                choice_s
                argument_id_s
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  short_circuit

        IS INCLUDED IN:
                EXP_VAL
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp1                 : EXP
                    as_exp2                 : EXP
                    as_short_circuit_op     : SHORT_CIRCUIT_OP
            (INHERITED FROM EXP_VAL):
                    sm_value                : value
            (INHERITED FROM EXP_EXP):
                    sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  SHORT_CIRCUIT_OP

        CLASS MEMBERS:
                and_then
                or_else
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                short_circuit.as_short_circuit_op
```

```
**  SIMPLE_RENAME_DECL

        CLASS MEMBERS:
                renames_obj_decl
                renames_exc_decl
        IS INCLUDED IN:
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_name                  : NAME
            (INHERITED FROM ID_DECL):
                as_source_name           : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

**  slice

        IS INCLUDED IN:
                NAME_EXP
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_discrete_range        : DISCRETE_RANGE
            (INHERITED FROM NAME_EXP):
                as_name                  : NAME
                sm_exp_type              : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                : source_position
                lx_comments              : comments

**  SOURCE_NAME

        CLASS MEMBERS:
                OBJECT_NAME
                LABEL_NAME
                UNIT_NAME
                TYPE_NAME
                void
                entry_id
                exception_id
                INIT_OBJECT_NAME
                ENUM_LITERAL
                iteration_id
                label_id
                block_loop_id
                NON_TASK_NAME
```

```
                task_body_id
                type_id
                subtype_id
                private_type_id
                l_private_type_id
                VC_NAME
                number_id
                COMP_NAME
                PARAM_NAME
                enumeration_id
                character_id
                SUBPROG_PACK_NAME
                generic_id
                variable_id
                constant_id
                component_id
                discriminant_id
                in_id
                out_id
                in_out_id
                SUBPROG_NAME
                package_id
                procedure_id
                operator_id
                function_id
        IS INCLUDED IN:
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DEF_NAME):
                    lx_symrep              : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos              : source_position
                    lx_comments            : comments
        IS THE DECLARED TYPE OF:
                SUBUNIT_BODY.as_source_name
                implicit_not_eq.sm_equal
                derived_subprog.sm_derivable
                FOR_REV.as_source_name
                BLOCK_LOOP.as_source_name
                source_name_s.as_list [Seq Of]
                ID_DECL.as_source_name

**  source_name_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                : Seq Of SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos              : source_position
                    lx_comments            : comments
```

```
        IS THE DECLARED TYPE OF:
                labeled.as_source_name_s
                DSCRMT_PARAM_DECL.as_source_name_s
                ID_S_DECL.as_source_name_s

**  STM

        CLASS MEMBERS:
                labeled
                null_stm
                abort
                STM_WITH_EXP
                STM_WITH_NAME
                accept
                ENTRY_STM
                BLOCK_LOOP
                CLAUSES_STM
                terminate
                return
                delay
                STM_WITH_EXP_NAME
                case
                goto
                raise
                CALL_STM
                cond_entry
                timed_entry
                loop
                block
                if
                selective_wait
                assign
                code
                exit
                entry_call
                procedure_call
        IS INCLUDED IN:
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                block_master.sm_stm
                exit.sm_stm
                LABEL_NAME.sm_stm
                labeled.as_stm

**  STM_ELEM

        CLASS MEMBERS:
                STM
                stm_pragma
```

```
                labeled
                null_stm
                abort
                STM_WITH_EXP
                STM_WITH_NAME
                accept
                ENTRY_STM
                BLOCK_LOOP
                CLAUSES_STM
                terminate
                return
                delay
                STM_WITH_EXP_NAME
                case
                goto
                raise
                CALL_STM
                cond_entry
                timed_entry
                loop
                block
                if
                selective_wait
                assign
                code
                exit
                entry_call
                procedure_call
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                stm_s.as_list [Seq Of]
```

** stm_pragma

```
        IS INCLUDED IN:
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_pragma               : pragma
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** stm_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
```

```
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_list                   : Seq Of STM_ELEM
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                 : source_position
                    lx_comments               : comments
        IS THE DECLARED TYPE OF:
                ENTRY_STM.as_stm_s1
                         .as_stm_s2
                accept.as_stm_s
                block_body.as_stm_s
                loop.as_stm_s
                alternative.as_stm_s
                TEST_CLAUSE.as_stm_s
                CLAUSES_STM.as_stm_s
```

** STM_WITH_EXP

```
        CLASS MEMBERS:
                return
                delay
                STM_WITH_EXP_NAME
                case
                assign
                code
                exit
        IS INCLUDED IN:
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp                    : EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                 : source_position
                    lx_comments               : comments
```

** STM_WITH_EXP_NAME

```
        CLASS MEMBERS:
                assign
                code
                exit
        IS INCLUDED IN:
                STM_WITH_EXP
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name                   : NAME
            (INHERITED FROM STM_WITH_EXP):
                    as_exp                    : EXP
            (INHERITED FROM ALL_SOURCE):
```

```
                lx_srcpos               : source_position
                lx_comments             : comments

**  STM_WITH_NAME

        CLASS MEMBERS:
                goto
                raise
                CALL_STM
                entry_call
                procedure_call
        IS INCLUDED IN:
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_name                 : NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  string_literal

        IS INCLUDED IN:
                AGG_EXP
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                lx_symrep               : symbol_rep
            (INHERITED FROM AGG_EXP):
                sm_discrete_range       : DISCRETE_RANGE
            (INHERITED FROM EXP_EXP):
                sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  stub

        IS INCLUDED IN:
                BODY
                UNIT_DESC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  SUBP_ENTRY_HEADER
```

```
        CLASS MEMBERS:
                procedure_spec
                function_spec
                entry
        IS INCLUDED IN:
                HEADER
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_param_s                  : param_s
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments

**  subprog_entry_decl

        IS INCLUDED IN:
                NON_GENERIC_DECL
                UNIT_DECL
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM NON_GENERIC_DECL):
                as_unit_kind                : UNIT_KIND
            (INHERITED FROM UNIT_DECL):
                as_header                   : HEADER
            (INHERITED FROM ID_DECL):
                as_source_name              : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments

**  SUBPROG_NAME

        CLASS MEMBERS:
                procedure_id
                operator_id
                function_id
        IS INCLUDED IN:
                SUBPROG_PACK_NAME
                NON_TASK_NAME
                UNIT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_is_inline                : Boolean
                sm_interface                : PREDEF_NAME
            (INHERITED FROM SUBPROG_PACK_NAME):
                sm_unit_desc                : UNIT_DESC
```

```
                sm_address              : EXP
        (INHERITED FROM NON_TASK_NAME):
                sm_spec                 : HEADER
        (INHERITED FROM UNIT_NAME):
                sm_first                : DEF_NAME
        (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** SUBPROG_PACK_NAME

```
    CLASS MEMBERS:
            SUBPROG_NAME
            package_id
            procedure_id
            operator_id
            function_id
    IS INCLUDED IN:
            NON_TASK_NAME
            UNIT_NAME
            SOURCE_NAME
            DEF_NAME
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                sm_unit_desc            : UNIT_DESC
                sm_address              : EXP
        (INHERITED FROM NON_TASK_NAME):
                sm_spec                 : HEADER
        (INHERITED FROM UNIT_NAME):
                sm_first                : DEF_NAME
        (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** subprogram_body

```
    IS INCLUDED IN:
            SUBUNIT_BODY
            ITEM
            ALL_DECL
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_header               : HEADER
        (INHERITED FROM SUBUNIT_BODY):
                as_source_name          : SOURCE_NAME
                as_body                 : BODY
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
```

```
                lx_comments             : comments

**  subtype_allocator

        IS INCLUDED IN:
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_subtype_indication   : subtype_indication
                sm_desig_type           : TYPE_SPEC
            (INHERITED FROM EXP_EXP):
                sm_exp_type             : TYPE_SPEC
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  subtype_decl

        IS INCLUDED IN:
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_subtype_indication   : subtype_indication
            (INHERITED FROM ID_DECL):
                as_source_name          : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  subtype_id

        IS INCLUDED IN:
                TYPE_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM TYPE_NAME):
                sm_type_spec            : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  subtype_indication
```

```
        IS INCLUDED IN:
                CONSTRAINED_DEF
                TYPE_DEF
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_name                 : NAME
            (INHERITED FROM CONSTRAINED_DEF):
                as_constraint           : CONSTRAINT
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                subtype_allocator.as_subtype_indication
                discrete_subtype.as_subtype_indication
                subtype_decl.as_subtype_indication
                ARR_ACC_DER_DEF.as_subtype_indication

**  subunit

        IS INCLUDED IN:
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_name                 : NAME
                as_subunit_body         : SUBUNIT_BODY
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  SUBUNIT_BODY

        CLASS MEMBERS:
                subprogram_body
                task_body
                package_body
        IS INCLUDED IN:
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_source_name          : SOURCE_NAME
                as_body                 : BODY
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                subunit.as_subunit_body

**  task_body

        IS INCLUDED IN:
```

```
                SUBUNIT_BODY
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM SUBUNIT_BODY):
                    as_source_name              : SOURCE_NAME
                    as_body                     : BODY
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** task_body_id

```
        IS INCLUDED IN:
                UNIT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_type_spec                : TYPE_SPEC
                    sm_body                     : BODY
            (INHERITED FROM UNIT_NAME):
                    sm_first                    : DEF_NAME
            (INHERITED FROM DEF_NAME):
                    lx_symrep                   : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** task_decl

```
        IS INCLUDED IN:
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_decl_s                   : decl_s
            (INHERITED FROM ID_DECL):
                    as_source_name              : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** task_spec

```
        IS INCLUDED IN:
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
```

```
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_decl_s               : decl_s
                    sm_storage_size         : EXP
                    sm_size                 : EXP
                    sm_address              : EXP
                    sm_body                 : BODY
            (INHERITED FROM DERIVABLE_SPEC):
                    sm_derived              : TYPE_SPEC
                    sm_is_anonymous         : Boolean
```

** terminate

```
        IS INCLUDED IN:
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** TEST_CLAUSE

```
        CLASS MEMBERS:
                cond_clause
                select_alternative
        IS INCLUDED IN:
                TEST_CLAUSE_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp                  : EXP
                    as_stm_s                : stm_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** TEST_CLAUSE_ELEM

```
        CLASS MEMBERS:
                TEST_CLAUSE
                select_alt_pragma
                cond_clause
                select_alternative
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                test_clause_elem_s.as_list [Seq Of]
```

```
**  test_clause_elem_s

        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_list                 : Seq Of TEST_CLAUSE_ELEM
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
        IS THE DECLARED TYPE OF:
                CLAUSES_STM.as_test_clause_elem_s

**  timed_entry

        IS INCLUDED IN:
                ENTRY_STM
                STM
                STM_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ENTRY_STM):
                as_stm_s1               : stm_s
                as_stm_s2               : stm_s
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  type_decl

        IS INCLUDED IN:
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_dscrmt_decl_s        : dscrmt_decl_s
                as_type_def             : TYPE_DEF
            (INHERITED FROM ID_DECL):
                as_source_name          : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  TYPE_DEF

        CLASS MEMBERS:
                enumeration_def
                record_def
                ARR_ACC_DER_DEF
                CONSTRAINED_DEF
```

```
                void
                private_def
                l_private_def
                formal_dscrt_def
                formal_float_def
                formal_fixed_def
                formal_integer_def
                constrained_array_def
                derived_def
                access_def
                unconstrained_array_def
                subtype_indication
                integer_def
                fixed_def
                float_def
        IS INCLUDED IN:
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
        IS THE DECLARED TYPE OF:
                type_decl.as_type_def
                OBJECT_DECL.as_type_def
```

** type_id

```
        IS INCLUDED IN:
                TYPE_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_first                : DEF_NAME
            (INHERITED FROM TYPE_NAME):
                    sm_type_spec            : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** type_membership

```
        IS INCLUDED IN:
                MEMBERSHIP
                EXP_VAL_EXP
                EXP_VAL
                EXP_EXP
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
```

```
        (NODE SPECIFIC):
              as_name                      : NAME
        (INHERITED FROM MEMBERSHIP):
              as_membership_op             : MEMBERSHIP_OP
        (INHERITED FROM EXP_VAL_EXP):
              as_exp                       : EXP
        (INHERITED FROM EXP_VAL):
              sm_value                     : value
        (INHERITED FROM EXP_EXP):
              sm_exp_type                  : TYPE_SPEC
        (INHERITED FROM ALL_SOURCE):
              lx_srcpos                    : source_position
              lx_comments                  : comments

**  TYPE_NAME

     CLASS MEMBERS:
            type_id
            subtype_id
            private_type_id
            l_private_type_id
     IS INCLUDED IN:
            SOURCE_NAME
            DEF_NAME
            ALL_SOURCE
     NODE ATTRIBUTES:
        (NODE SPECIFIC):
              sm_type_spec                 : TYPE_SPEC
        (INHERITED FROM DEF_NAME):
              lx_symrep                    : symbol_rep
        (INHERITED FROM ALL_SOURCE):
              lx_srcpos                    : source_position
              lx_comments                  : comments

**  TYPE_SPEC

     CLASS MEMBERS:
            DERIVABLE_SPEC
            incomplete
            void
            universal_integer
            universal_real
            universal_fixed
            FULL_TYPE_SPEC
            PRIVATE_SPEC
            task_spec
            NON_TASK
            private
            l_private
            SCALAR
            CONSTRAINED
            UNCONSTRAINED
            enumeration
            REAL
```

```
                integer
                constrained_array
                constrained_access
                constrained_record
                UNCONSTRAINED_COMPOSITE
                access
                float
                fixed
                array
                record
        IS THE DECLARED TYPE OF:
                task_body_id.sm_type_spec
                PRIVATE_SPEC.sm_type_spec
                subtype_allocator.sm_desig_type
                EXP_EXP.sm_exp_type
                USED_OBJECT.sm_exp_type
                NAME_EXP.sm_exp_type
                constrained_access.sm_desig_type
                access.sm_desig_type
                index.sm_type_spec
                array.sm_comp_type
                REAL_CONSTRAINT.sm_type_spec
                RANGE.sm_type_spec
                NON_TASK.sm_base_type
                DERIVABLE_SPEC.sm_derived
                TYPE_NAME.sm_type_spec
                OBJECT_NAME.sm_obj_type

**  UNCONSTRAINED

        CLASS MEMBERS:
                UNCONSTRAINED_COMPOSITE
                access
                array
                record
        IS INCLUDED IN:
                NON_TASK
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_size                 : EXP
            (INHERITED FROM NON_TASK):
                    sm_base_type            : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                    sm_derived              : TYPE_SPEC
                    sm_is_anonymous         : Boolean

**  unconstrained_array_def

        IS INCLUDED IN:
                ARR_ACC_DER_DEF
                TYPE_DEF
```

```
            ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_index_s                  : index_s
            (INHERITED FROM ARR_ACC_DER_DEF):
                as_subtype_indication       : subtype_indication
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
```

** UNCONSTRAINED_COMPOSITE

```
        CLASS MEMBERS:
                array
                record
        IS INCLUDED IN:
                UNCONSTRAINED
                NON_TASK
                FULL_TYPE_SPEC
                DERIVABLE_SPEC
                TYPE_SPEC
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_is_limited               : Boolean
                sm_is_packed                : Boolean
            (INHERITED FROM UNCONSTRAINED):
                sm_size                     : EXP
            (INHERITED FROM NON_TASK):
                sm_base_type                : TYPE_SPEC
            (INHERITED FROM DERIVABLE_SPEC):
                sm_derived                  : TYPE_SPEC
                sm_is_anonymous             : Boolean
```

** UNIT_DECL

```
        CLASS MEMBERS:
                generic_decl
                NON_GENERIC_DECL
                subprog_entry_decl
                package_decl
        IS INCLUDED IN:
                ID_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_header                   : HEADER
            (INHERITED FROM ID_DECL):
                as_source_name              : SOURCE_NAME
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
```

** UNIT_DESC

```
CLASS MEMBERS:
        UNIT_KIND
        derived_subprog
        implicit_not_eq
        BODY
        void
        RENAME_INSTANT
        GENERIC_PARAM
        block_body
        stub
        renames_unit
        instantiation
        name_default
        no_default
        box_default
IS INCLUDED IN:
        ALL_SOURCE
NODE ATTRIBUTES:
    (INHERITED FROM ALL_SOURCE):
            lx_srcpos              : source_position
            lx_comments            : comments
IS THE DECLARED TYPE OF:
        SUBPROG_PACK_NAME.sm_unit_desc
```

** UNIT_KIND

```
CLASS MEMBERS:
        void
        RENAME_INSTANT
        GENERIC_PARAM
        renames_unit
        instantiation
        name_default
        no_default
        box_default
IS INCLUDED IN:
        UNIT_DESC
        ALL_SOURCE
NODE ATTRIBUTES:
    (INHERITED FROM ALL_SOURCE):
            lx_srcpos              : source_position
            lx_comments            : comments
IS THE DECLARED TYPE OF:
        NON_GENERIC_DECL.as_unit_kind
```

** UNIT_NAME

```
CLASS MEMBERS:
        NON_TASK_NAME
        task_body_id
        SUBPROG_PACK_NAME
```

```
                generic_id
                SUBPROG_NAME
                package_id
                procedure_id
                operator_id
                function_id
        IS INCLUDED IN:
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_first                : DEF_NAME
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  universal_fixed

        IS INCLUDED IN:
                TYPE_SPEC
        NODE ATTRIBUTES:

**  universal_integer

        IS INCLUDED IN:
                TYPE_SPEC
        NODE ATTRIBUTES:

**  universal_real

        IS INCLUDED IN:
                TYPE_SPEC
        NODE ATTRIBUTES:

**  use

        IS INCLUDED IN:
                USE_PRAGMA
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name_s               : name_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments

**  USE_PRAGMA
```

```
        CLASS MEMBERS:
                use
                pragma
        IS INCLUDED IN:
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                use_pragma_s.as_list [Seq Of]
```

** use_pragma_s

```
        IS INCLUDED IN:
                SEQUENCES
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                as_list                     : Seq Of USE_PRAGMA
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                with.as_use_pragma_s
```

** used_char

```
        IS INCLUDED IN:
                USED_OBJECT
                DESIGNATOR
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM USED_OBJECT):
                sm_exp_type                 : TYPE_SPEC
                sm_value                    : value
            (INHERITED FROM DESIGNATOR):
                sm_defn                     : DEF_NAME
                lx_symrep                   : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
```

** USED_NAME

```
        CLASS MEMBERS:
                used_op
                used_name_id
```

```
        IS INCLUDED IN:
                DESIGNATOR
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DESIGNATOR):
                sm_defn                     : DEF_NAME
                lx_symrep                   : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                assoc.as_used_name

**  used_name_id

        IS INCLUDED IN:
                USED_NAME
                DESIGNATOR
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DESIGNATOR):
                sm_defn                     : DEF_NAME
                lx_symrep                   : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos                   : source_position
                lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                attribute.as_used_name_id
                range_attribute.as_used_name_id
                pragma.as_used_name_id

**  USED_OBJECT

        CLASS MEMBERS:
                used_char
                used_object_id
        IS INCLUDED IN:
                DESIGNATOR
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                sm_exp_type                 : TYPE_SPEC
                sm_value                    : value
            (INHERITED FROM DESIGNATOR):
                sm_defn                     : DEF_NAME
```

```
                lx_symrep              : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos              : source_position
                lx_comments            : comments
```

** used_object_id

```
        IS INCLUDED IN:
                USED_OBJECT
                DESIGNATOR
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM USED_OBJECT):
                sm_exp_type            : TYPE_SPEC
                sm_value               : value
            (INHERITED FROM DESIGNATOR):
                sm_defn                : DEF_NAME
                lx_symrep              : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos              : source_position
                lx_comments            : comments
```

** used_op

```
        IS INCLUDED IN:
                USED_NAME
                DESIGNATOR
                NAME
                EXP
                GENERAL_ASSOC
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM DESIGNATOR):
                sm_defn                : DEF_NAME
                lx_symrep              : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                lx_srcpos              : source_position
                lx_comments            : comments
```

** variable_decl

```
        IS INCLUDED IN:
                OBJECT_DECL
                EXP_DECL
                ID_S_DECL
                DECL
                ITEM
                ALL_DECL
                ALL_SOURCE
        NODE ATTRIBUTES:
            (INHERITED FROM OBJECT_DECL):
```

```
                as_type_def             : TYPE_DEF
        (INHERITED FROM EXP_DECL):
                as_exp                  : EXP
        (INHERITED FROM ID_S_DECL):
                as_source_name_s        : source_name_s
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** variable_id

```
    IS INCLUDED IN:
            VC_NAME
            INIT_OBJECT_NAME
            OBJECT_NAME
            SOURCE_NAME
            DEF_NAME
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                sm_is_shared            : Boolean
        (INHERITED FROM VC_NAME):
                sm_renames_obj          : Boolean
                sm_address              : EXP
        (INHERITED FROM INIT_OBJECT_NAME):
                sm_init_exp             : EXP
        (INHERITED FROM OBJECT_NAME):
                sm_obj_type             : TYPE_SPEC
        (INHERITED FROM DEF_NAME):
                lx_symrep               : symbol_rep
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** variant

```
    IS INCLUDED IN:
            VARIANT_ELEM
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_choice_s             : choice_s
                as_comp_list            : comp_list
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
```

** VARIANT_ELEM

```
    CLASS MEMBERS:
            variant
            variant_pragma
    IS INCLUDED IN:
            ALL_SOURCE
```

```
    NODE ATTRIBUTES:
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
    IS THE DECLARED TYPE OF:
            variant_s.as_list [Seq Of]

**  VARIANT_PART

    CLASS MEMBERS:
            variant_part
            void
    IS INCLUDED IN:
            ALL_SOURCE
    NODE ATTRIBUTES:
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments
    IS THE DECLARED TYPE OF:
            comp_list.as_variant_part

**  variant_part

    IS INCLUDED IN:
            VARIANT_PART
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_name                 : NAME
                as_variant_s            : variant_s
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  variant_pragma

    IS INCLUDED IN:
            VARIANT_ELEM
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_pragma               : pragma
        (INHERITED FROM ALL_SOURCE):
                lx_srcpos               : source_position
                lx_comments             : comments

**  variant_s

    IS INCLUDED IN:
            SEQUENCES
            ALL_SOURCE
    NODE ATTRIBUTES:
        (NODE SPECIFIC):
                as_list                 : Seq Of VARIANT_ELEM
```

```
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
        IS THE DECLARED TYPE OF:
                variant_part.as_variant_s
```

** VC_NAME

```
        CLASS MEMBERS:
                variable_id
                constant_id
        IS INCLUDED IN:
                INIT_OBJECT_NAME
                OBJECT_NAME
                SOURCE_NAME
                DEF_NAME
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    sm_renames_obj              : Boolean
                    sm_address                  : EXP
            (INHERITED FROM INIT_OBJECT_NAME):
                    sm_init_exp                 : EXP
            (INHERITED FROM OBJECT_NAME):
                    sm_obj_type                 : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep                   : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos                   : source_position
                    lx_comments                 : comments
```

** void

```
        IS INCLUDED IN:
                PREDEF_NAME
                COMP_REP_ELEM
                ALIGNMENT_CLAUSE
                ALL_DECL
                BODY
                UNIT_KIND
                NAME
                ITERATION
                SOURCE_NAME
                TYPE_SPEC
                TYPE_DEF
                VARIANT_PART
                REP
                RANGE
                CONSTRAINT
                EXP
                DEF_NAME
                ALL_SOURCE
                UNIT_DESC
                DECL
```

```
                DISCRETE_RANGE
                GENERAL_ASSOC
                ITEM
        NODE ATTRIBUTES:
            (INHERITED FROM REP):
                    as_name                 : NAME
            (INHERITED FROM RANGE):
                    sm_type_spec            : TYPE_SPEC
            (INHERITED FROM DEF_NAME):
                    lx_symrep               : symbol_rep
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** while

```
        IS INCLUDED IN:
                ITERATION
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_exp                  : EXP
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

** with

```
        IS INCLUDED IN:
                CONTEXT_ELEM
                ALL_SOURCE
        NODE ATTRIBUTES:
            (NODE SPECIFIC):
                    as_name_s               : name_s
                    as_use_pragma_s         : use_pragma_s
            (INHERITED FROM ALL_SOURCE):
                    lx_srcpos               : source_position
                    lx_comments             : comments
```

APPENDIX B

REFERENCES

[1] P.F. Albrecht, P.E. Garrison, S.L. Graham, R.H. Hyerle, P. Ip, and B. Krieg-Bruckner. "Source-to-Source Translation: Ada to Pascal and Pascal to Ada." In Symposium on the Ada Programming Language, pages 183-193. ACM- SIGPLAN, Boston, December, 1980.

[2] B. M. Brosgol, J.M. Newcomer, D.A. Lamb, D. Levine, M. S. Van Deusen, and W.A. Wulf. TCOLada : Revised Report on An Intermediate Representation for the Preliminary Ada Language. Technical Report CMU-CS-80-105, Carnegie-Mellon University, Computer Science Department, February, 1980.

[3] J.N. Buxton. Stoneman: Requirements for Ada Programming Support Environments. Technical Report, DARPA, February, 1980.

[4] M. Dausmann, S. Drossopoulou, G. Goos, G. Persch, G. Winterstein. AIDA Introduction and User Manual. Technical Report Nr. 38/80, Institut fuer Informatik II, Universitaet Karlsruhe, 1980.

[5] M. Dausmann, S. Drossopoulou, G. Persch, G. Winterstein. On Reusing Units of Other Program Libraries. Technical Report Nr. 31/80, Institut fuer Informatik II, Universitaet Karlsruhe, 1980.

[6] Formal Definition of the Ada Programming Language November 1980 edition, Honeywell, Inc., Cii Honeywell Bull, INRIA, 1980.

[7] 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 The revised reference manual, July 1980 edition, Honeywell, Inc., and Cii-Honeywell Bull, 1980.

[8] 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.

[9] 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, August, 1981.

[10] G. Persch, G. Winterstein, M. Dausmann, S. Drossopoulou, G. Goos. AIDA Reference Manual. Technical Report Nr. 39/80, Institut fuer Informatik II, Universitaet Karlsruhe, November, 1980.

[11] Author unknown. Found on a blackboard at Eglin Air Force Base.