# GRAPHITE REFERENCE MANUAL

Alexander L. Wolf

COINS Technical Report 87–109
Arcadia Design Document UM–87–07
October 1987

*Software Development Laboratory*
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

*This manual corresponds, except where indicated, to* GRAPHITE *Version 2.*

Ada is a registered trademark of the US Government (Ada Joint Program Office).

# Contents

# List of Tables

# List of Figures

# Preface

GRAPHITE is a tool intended to aid in the development, reuse, and documentation of graph data objects. GRAPHITE consists of a language, called GDL, for specifying classes of graphs and a processor for automatically generating abstract data types in the language Ada that are implementations of the specified classes. The GRAPHITE processor is itself implemented in Ada. Motivation and rationale for the design of GRAPHITE can be found in [Clarke, Wileden, & Wolf, 1986]. A user manual for GRAPHITE appears as [Tarr, 1987].

GRAPHITE was designed and implemented by members of the Software Development Laboratory, Computer and Information Science Department, University of Massachusetts at Amherst. The principal designers were Lori Clarke, Jack Wileden, and Alexander Wolf. Robert Graham and Steven Zeil also contributed to the design. The principal implementors of Version 1 were Mary Burdick, Alexander Wolf, and Peri Tarr. Version 2 was implemented by Peri Tarr and William Rosenblatt.

This work was supported in part by the following grants: Rome Air Development Center, SCEEE–PDP/85–0037 and F30602–86–C–0006; National Science Foundation, DCR–84–04217, DCR–84–08143, and DCR–85–00332; and the Defense Advanced Research Projects Agency (ARPA Order 6104, Program Code 7E20) through National Science Foundation grant CCR–87–04478.

# Chapter 1

# Introduction

Many of the data objects manipulated by software systems are *graphs*. Moreover, an individual instance of a graph may be shared by several different components of a system. For example, in a software develpment environment, parse trees, abstract syntax trees, control flow graphs, and call graphs are all classes of graphs that are likely to be manipulated by tools in the environment; an abstract syntax tree may be created by a syntax analyzer, modified by a semantic analyzer, and referenced by a pretty printer, among other tools.

We employ the following terminology for graphs. A *class* of graphs is defined by specifying a set of *node kinds*. Each node kind is associated with a set of *attributes*. Attributes are used to describe the properties of the objects represented by the nodes in the graph, and each such attribute has a type, referred to here as an *attribute type*. Some of the attribute types are actually node kinds, which makes it possible to connect nodes into graph structures. An instance of a node kind is a set of values, one for each attribute associated with that node's kind. A particular graph, which is a member of some class of graphs, is then just a set of instances of node kinds in that class.

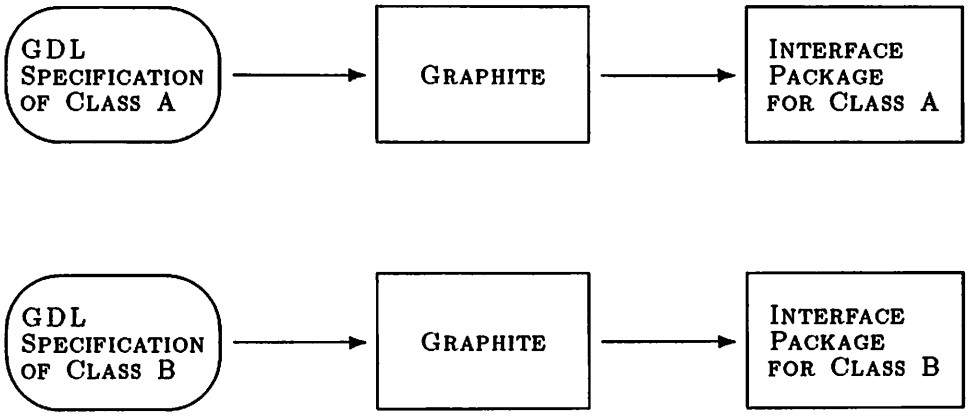
GRAPHITE is a tool intended to aid in the development, reuse, and documentation of graph data objects. GRAPHITE consists of a graph description language, called GDL, for specifying a class of graphs, and a processor that produces an implementation of an abstract data type for that class in the programming language Ada. The abstract data type is encapsulated in a package referred to as an *interface package*. The interface package defines the graph and the operations that can manipulate the graph. The operations include those to create nodes, get and put attribute values, and read and write nodes and graphs, as well as those to ascertain the kind of a node and its associated attributes.

Using the interface package produced by the GRAPHITE processor, a system component can create and/or access a number of different graphs of a particular class. Any or all of these

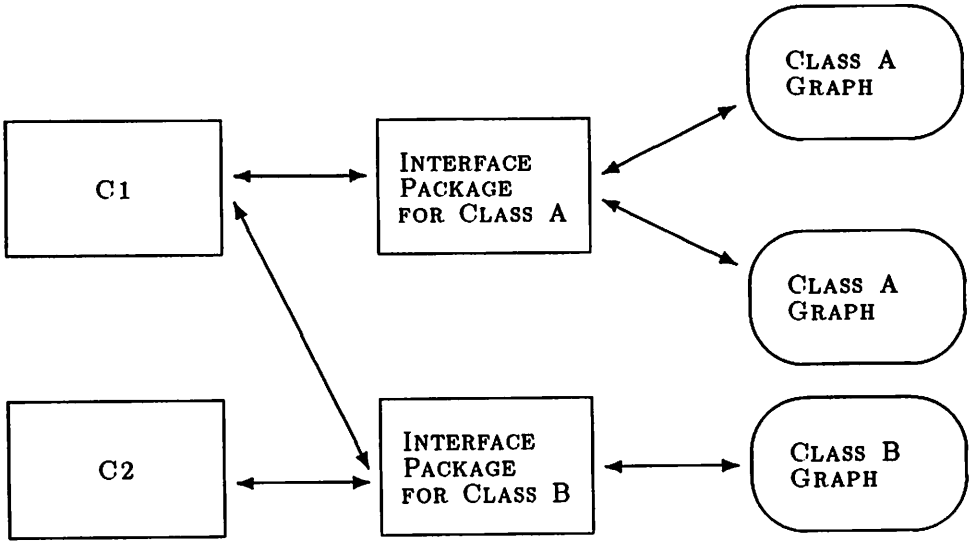
could be manipulated by other components, which would also access these graphs through the operations provided in the interface package. Moreover, a component may use more than one interface package in order to access more than one class of graphs. Figure 1.1a illustrates how GRAPHITE can be used to create interface packages for two different classes of graphs, called Class A and Class B. Figure 1.1b then shows how two components of a system, C1 and C2, might use these packages; C1 to manipulate two instances of Class A and both C1 and C2 to manipulate an instance of Class B.

The GRAPHITE processor actually produces two different kinds of interface packages. One supports experimental development of software systems. Referred to as a *development interface package*, it is designed so that when developers modify the definitions of graph classes there is a minimal effect on components of the system, even on those components that use this modified class. The second interface package is designed for efficient manipulation of graphs and is referred to as a *production interface package*. When the definition of a graph class has been finalized, the second kind of interface package can be substituted for the first so that a more efficient, but less flexible, version of the system can be created. Thus we have support for both development and production versions of the interface package and a process for easily going from the development to the production version.

This manual describes GRAPHITE in detail. Specifically, Chapter 2 describes the syntax and semantics of GDL, Chapter 3 describes the functionality of generated interface packages in terms of the types, constants, subprograms, and exceptions defined in such packages, and Chapter 4 describes the facilities in GRAPHITE that allow users to control, at least to a certain extent, the code generated for an interface package. Appendices are provided that summarize the reserved words and syntax of GDL. Motivation and rationale for the design of GRAPHITE can be found in [Clarke, Wileden, & Wolf, 1986]. A user manual for GRAPHITE, describing its installation, invocation, error messages, and limits, appears as [Tarr, 1987].




(a)




(b)

Figure 1.1: Creating and Using Graph Interface Packages.

# Chapter 2

# GDL—The Graph Description Language

This chapter describes the Graph Description Language, GDL, which is used to specify a class of graphs. This specification is the input to the GRAPHITE processor, which uses it to generate an interface package for creating and manipulating graphs of that class. Relevant portions of the GDL syntax are given at the beginning of sections that introduce particular constructs. The complete syntax appears in Appendix B, which also details the modified Backus-Naur Form used here as the syntax description language.

GDL facilitates the development and maintenance of large software systems by providing a common medium for communicating the definition of a graph class among developers. To make this medium as natural to use and easy to understand as possible for Ada developers, GDL is strongly modeled after Ada. This is evident in GDL's lexical rules, syntactic style, type structure, and declaration-order rules. It is also evident in GDL's module structure, which uses a package-like *class* declaration to encapsulate the description of a given class of graphs.

## 2.1 Lexical Elements

The text of a GDL specification follows the lexical rules of Ada as given in Chapter 2 of [DoD, 1983], except for *reserved words* and *pragmas*.

### 2.1.1 Reserved Words

The reserved words in GDL include all the Ada reserved words plus the following four identifiers.

**class** **node**
**group** **sequence**

A complete list of the reserved words of GDL appears in Appendix A.

*Notes:*

An effort was made to define keywords for GDL that are already used as reserved words in Ada so that developers are given the same freedom as in Ada for choosing identifiers. Reserved words of Ada that are not used in GDL, such as **task**, are not permitted as identifiers in GDL to avoid any conflicts with Ada implementations of interface packages.

#### Restricted Identifiers

In addition to reserved words, certain other identifiers are not available for declaration in GDL, since they would conflict with identifier declarations automatically generated for an interface package. These *restricted identifiers* fall into the following two categories:[1]

- *Identifiers beginning with the four characters GDL_*

  All identifiers local to an interface package (i.e., declared in the body part of an interface package) begin with these four characters.

- *Identifiers that are provided by an interface package*

  These include the names of certain types, constants, subprograms, and exceptions for manipulating the specified class of graphs {§3}.

*Notes:*

The restrictions on identifiers described above do not apply to names appearing in GDL *with clauses* {§2.4.3}. This is because the use of names in GDL *with clauses* do not cause any declarations to be generated within an interface package that might conflict with other, automatically generated declarations. Rather, such uses are simply references to externally defined entities {§3.1}.

[1] The current version of the GRAPHITE processor does not flag declarations of these identifiers as being in error.

### 2.1.2 Pragmas[2]

```
pragma  ::=  pragma identifier [(argument_association {, argument_association})];

argument_association  ::=
        [argument_identifier =>] name
    |   [argument_identifier =>] expression
```

Pragmas are used in GDL to convey information to the GRAPHITE processor in the same way that pragmas are used in Ada to convey information to an Ada compiler (see [DoD, 1983], Section 2.8). The GDL pragmas are described in Chapter 4.

*Examples:*

```
pragma InterfaceVersion ( Development );
pragma MaximumFamilySize ( 10 );
pragma Suppress ( NodeKind_Check );
```

## 2.2 Classes

```
class_declaration  ::=
    class identifier is
        package_name_declaration
        {declarative_item}
    end class_simple_name;

package_name_declaration  ::=  package identifier;

declarative_item  ::=
        Ada_type_declaration      |  Ada_constant_declaration
    |   node_kind_declaration     |  group_declaration
    |   sequence_declaration      |  commonly_available_attribute_declaration
```

A class declaration introduces a class of graphs, encapsulating declarations of the node kinds, attributes, and attribute types that make up the class. More specifically, a class consists of a series of declarative items, where a declarative item is an Ada type, an Ada constant, a node kind, a node group, a node sequence, or a commonly-available attribute. The various kinds of declarative items are described in subsequent sections.

[2]The current version of the GRAPHITE processor does not support pragmas.

The first declaration in a class must be a package name declaration, which introduces a name for the interface package generated for the class {§3.1}.

A class declaration must contain at least one node kind declaration.

The simple name appearing at the end of a class declaration must be the same as the class identifier.

*Examples:*

Figure 2.1 presents a skeleton GDL specification in which ExampleGraph is the name of the class of graphs being defined and Example is the name of the interface package that is to be generated.

This figure is used throughout Chapter 2 to illustrate various features of GDL.

## 2.3 Node Kinds and Attributes

```
node_kind_declaration ::= full_node_kind_declaration | incomplete_node_kind_declaration

full_node_kind_declaration ::=
    node identifier_list is
      attribute_declaration_list
    end node;

attribute_declaration_list ::= null; | attribute_declaration {attribute_declaration}

attribute_declaration ::=
        identifier_list [(<>)] : subtype_indication [:= expression];
    |   identifier_list [(<>)] : node [:= expression];
    |   commonly_available_attribute_simple_name [(<>)] [:= expression];

subtype_indication ::= type_mark [constraint]

type_mark ::= type_name | subtype_name

identifier_list ::= identifier {, identifier_list}
```

Node kinds are the primary building blocks of GDL. To define a node kind, the developer specifies the name of that node kind and the attributes that make up nodes of that kind. The format for defining a node kind is similar to that of an Ada record declaration, with attributes acting as record fields. The similarity between node kind declarations and record declarations,

```
class ExampleGraph is
  package Example;

  with Lexical.( Comment, Position );

  type LexicalInformation is
     record
        SourceComment : Lexical.Comment;
        SourcePosition  : Lexical.Position;
     end record;

  LowWeight : constant Integer := -10;
  type BranchWeight is new Integer range LowWeight .. 10;

  group Statement;        -- complete definition given below
  type StatementSequence is sequence of Statement;

  node ConditionClause; -- complete definition given below

  node ExpressionNode is
     . . .
  end node;

  SourceConnection : LexicalInformation;
  ExecutionCount   : Natural;

  node ConditionClause is
     SourceConnection;
     Weight     : BranchWeight;
     Condition  : ExpressionNode;
     Statements : StatementSequence;
  end node;

  node IfStatement is
     SourceConnection;
     ExecutionCount;
     IfBranch          : ConditionClause;
     ElsIfBranch (<>) : ConditionClause;
     ElseBranch        : StatementSequence;
  end node;

  . . .

  group Statement is ( IfStatement, WhileStatement, CaseStatement, ... );

end ExampleGraph;
```

Figure 2.1: Skeleton GDL Specification for a Class of Graphs.

however, is purely syntactic. In particular, node kinds are not necessarily implemented as records, and record-oriented operations, such as field selection, cannot be applied directly to nodes.

A node kind declaration that has an identifier list with more than one element is equivalent to a series of node kind declarations, each having one element in its identifier list. Similarly, an attribute declaration that has an identifier list with more than one element is equivalent to a series of attribute declarations, each having one element in its identifier list.

There are restrictions on the type of an attribute. In particular, an attribute cannot be an unconstrained array. Furthermore, if an attribute is of an unconstrained type with discriminants, then defaults for those discriminants must have been given.

The form of attribute declaration that includes the box compound delimiter enclosed in parentheses—(<>)—indicates the declaration of an *attribute family* {§2.3.2}.

The form of attribute declaration that includes the reserved word **node** is used to indicate that the type of the attribute(s) is the predefined node group that contains as its members all node kinds in the given class {§2.4.4}.

The form of node kind declaration that includes the reserved word **null** indicates the declaration of a node kind having no attributes.

*Examples:*

ConditionClause and IfStatement are two node kinds declared in Figure 2.1.

*Notes:*

If a GDL specification exclusively contains node kinds that do not have any attributes, then the class contains no attributes and no attribute types.

### 2.3.1 Commonly-available Attribute Declarations

```
commonly_available_attribute_declaration  ::=
        identifier_list : subtype_indication [:= expression];
    |   identifier_list : node [:= expression];
```

GDL provides a syntactic shorthand (not available to Ada records) intended for situations in which two or more node kind declarations contain an identical attribute declaration (i.e., the name and type of the declared attribute are the same). This syntactic shorthand allows an attribute declaration to be made in one place and then used in various node kind declarations

by simply listing the identifier given in the attribute declaration. Such an attribute declaration appears outside of any node kind declaration and is referred to as a *commonly-available* attribute declaration. Whether or not the shorthand is used, each attribute is only associated with one node kind; it is simply the type information that is shared using commonly-available attribute declarations.

A commonly-available attribute declaration that has an identifier list with more than one element is equivalent to a series of commonly-available attribute declarations, each having one element in its identifier list.

*Examples:*

In Figure 2.1, node kinds ConditionClause and IfStatement contain identical declarations for an attribute named SourceConnection.

### 2.3.2 Attribute Families

An attribute declaration whose form includes the box compound delimiter enclosed in parentheses—(<>)—indicates the declaration of a *family* of attributes, all of the same type. The number of members of this family is determined at the time an instance of the node kind containing the attribute declaration is created {§3.3}; different instances of the same node kind may have different numbers of family members.

In this context, the "family name" is considered to be the attribute name.

The number of family members can range from 0 (zero) to 31 (thirty-one). This limit can be changed for particular attribute families using pragma MaximumFamilySize {§4}.

Members of an attribute family are (conceptually) referred to by a tuple consisting of the attribute (i.e., family) name and an *index*. The index for a given family in a given instance of a node kind containing the family ranges from 1 (one) to the number of members in that family.

*Examples:*

In Figure 2.1, node kind IfStatement contains the declaration of an attribute family ElsIfBranch; the number of ElsIfBranch family members in a given IfStatement node is determined when that node is created. For instance, the following call on an interface package that is generated for class ExampleGraph would establish 10 (ten) family members:

```
N := Create ( IfStatement, 10 );
```

The following call gives the value M to the third family member:

```
PutAttribute ( N, ElsIfBranch, 3, M );
```

(Subprograms Create and PutAttribute are described in Section 3.3.)

## 2.4 Types

GDL supports four categories of types: user-definable Ada types, predefined Ada types, imported Ada types, and GDL-specific types. The values and operations associated with types in the first three categories are simply those described in [DoD, 1983]. The values and operations associated with GDL-specific types are described in Section 2.4.4 and Chapter 3.

When a type is used in the subtype indication of an attribute declaration, it is referred to as an *attribute type*. When a type is used in the declaration of another type, it is said to be used to *form* that type and, transitively, any other type formed from that type. The formed type might itself be used as an attribute type or in the declaration of yet another type.

A given type can be used both as an attribute type and to form an attribute type. Different restrictions apply to a type depending on whether it is used as an attribute type or is used to form an attribute type {§2.3}{§2.4.3}.

### 2.4.1 User-Definable Ada Types

```
Ada_type_declaration ::=
    full_type_declaration | subtype_declaration | imported_Ada_type_declaration

full_type_declaration ::= type identifier [discriminant_part] is type_definition;

type_definition ::=
        enumeration_type_definition | integer_type_definition
    |   real_type_definition        | array_type_definition
    |   record_type_definition      | derived_type_definition

subtype_declaration ::= subtype identifier is subtype indication [:= expression];
```

Types can be declared using Ada type constructors such as *integer*, *enumeration*, and *array*. All Ada user-definable types are allowed in a class declaration except *access* and *private*

types. Access and private types must instead be defined outside of the class declaration and "imported" {§2.4.3}. Following is a complete list of the allowed Ada type constructors along with the relevant section numbers in [DoD, 1983].

| | | | |
|---|---|---|---|
| *Subtype* | (3.3.2) | *Integer* | (3.5.4) |
| *Derived* | (3.4) | *Float* | (3.5.7) |
| *Enumeration* | (3.5.1) | *Fixed* | (3.5.9) |
| *Character* | (3.5.2) | *Array* | (3.6) |
| *Boolean* | (3.5.3) | *Record* | (3.7) |

*Notes:*

The GDL-specific types *node kind*, *node group*, and *node sequence* cannot be used in the declarations of user-defined Ada types (e.g., as components of a record type) {§2.4.4}.

### 2.4.2 Predefined Ada Types

The predefined Ada types (and subtypes), which are listed below with a reference to the relevant sections in [DoD, 1983], can be used directly in a class declaration.

| | | | |
|---|---|---|---|
| Character | (3.5.2) | String | (3.6.3) |
| Boolean | (3.5.3) | Duration | (9.6) |
| Integer | (3.5.4) | Natural | (C) |
| Float | (3.5.7) | Positive | (C) |

*Notes:*

The predefined Ada types, just like other types, can be redeclared.[3]

[3]The current version of GRAPHITE does not actually support this. The identifiers corresponding to the Ada predefined types are treated much like reserved words.

### 2.4.3 Imported Ada Types

```
imported_Ada_type_declaration ::=
    with imported_type_description {, imported_type_description};

imported_type_description ::=
        package_name.identifier [subtype_clause]
    |   package_name.identifier external_form_clause operation_clauses
    |   package_name.identifier assignment_operation_clause
    |   package_name.( name_shortened_type_description {, name_shortened_type_description} )

name_shortened_type_description ::=
        expanded_name [subtype_clause]
    |   expanded_name external_form_clause operation_clauses
    |   expanded_name assignment_operation_clause

package_name ::= expanded_name
```

In Ada, a *with clause* attached to a package indicates that some entities defined in other, external package(s) are to be imported. In GDL, the *with clause* performs a very similar function; a *with clause* inside a class declaration describes a set of externally defined and packaged Ada types, possibly including access and private types, that are expected to be used as, or to form, attribute types.

The syntax of the GDL *with clause* expands upon that of the Ada *with clause*, which only allows mention of the name of a package that provides some type. In particular, the GDL *with clause* explicitly identifies the type being imported[4] and includes information necessary to automatically generate Ada code for manipulating instances of that type when it is used as an attribute type.

As a syntactic shorthand, the description of an imported Ada type can take a form in which the information about the type is enclosed in parentheses and preceded by a package name selector. This is equivalent to a form in which there are no parentheses and the package name selector is prepended to all names appearing in the description. A list of imported Ada type descriptions can be used within the same parenthesized form to indicate that the names in all of those descriptions share the same package name selector.

[4]This information is needed in GDL because the specification parts of the packages providing the types are not assumed to be available to the GRAPHITE processor, as they would be to an Ada compiler. Moreover, explicit mention of the types requested (i.e., imported) from a package serves to better describe the relationship between the providing and requesting packages, since the set of types requested from a package is in general a proper subset of the types provided by that package [Wolf, Clarke, & Wileden, 1985].

The name corresponding to an imported Ada type can denote either the name of a subtype or the name of a type.

*Examples:*

In Figure 2.1, types Position and Comment are imported from package Lexical and used in defining the record type LexicalInformation. A more complete example is given in Figure 2.2, the details of which are explained below.

*Notes:*

All names appearing in a *with clause* must denote externally defined entities.

Nodes in a class $C$ can refer to nodes in another class $D$ by importing the type for designating nodes in class $D$ {§3.2.2}, and then using that type as an attribute type.

The order of imported Ada type descriptions given in GDL *with clauses* (unlike the order of package names given in Ada with clauses) may be significant. In particular, if types T and ST are imported (using the same or different *with clauses*) and ST is a subtype of type T, then the description of T must appear before the description of ST {§2.7}.

Because of the way it is interpreted, the parenthesized form should not be used with the predefined Ada types. That is,

```
with P.( T subtype of Integer );
```

is equivalent to

```
with P.T subtype of P.Integer;
```

and not, as might have been intended

```
with P.T subtype of Integer;
```

### Imported Subtypes

---

subtype_clause ::= **subtype of** expanded_name

---

If the identifier corresponding to an imported Ada type denotes the name of a subtype, then the name of the base type of that subtype must be given in a *subtype clause* following the

```
with Pac.SomeType1 subtype of Integer,

     Pac.SomeType2,

     Pac.( SomeType3/ExternalSomeType3
               := => SomeType3Assign
               in  => InternalizeSomeType3
               out => ExternalizeSomeType3 ),

     Pac.( SomeType4 subtype of SomeType3 );
```

Figure 2.2: Example Specifications of Imported Ada Types.

name of the imported Ada type.[5] If the base type is other than one of the predefined Ada types, then it must itself have been (previously) imported.

An imported subtype and its base type need not be defined in the same package.

*Examples:*

In Figure 2.2, SomeType1 is described as a subtype of the predefined Ada type Integer and SomeType4 as a subtype of the imported Ada type Pac.SomeType3.

### Imported Types

```
external_form_clause  ::=  / expanded_name

operation_clauses  ::=
    (       internalize_operation_clause
        ∧   externalize_operation_clause
        ∧   [assignment_operation_clause]  )

internalize_operation_clause  ::=  in => expanded_name

externalize_operation_clause  ::=  out => expanded_name

assignment_operation_clause  ::=  := => expanded_name  |  := => :=
```

[5]This information is needed because an interface package generated from a GDL specification provides subprograms to get and put attribute values that are overloaded on the basis of the attributes' base types {§3}; base types are used because overloading in Ada is legal for types and not subtypes.

If the identifier corresponding to an imported Ada type denotes the name of a type, and if that type is used as an attribute type, then additional information may be necessary to automatically generate Ada code for manipulating attributes of that type. (This information is not required for an imported subtype because it can be inferred from the base type of that subtype.) The additional information concerns facilities for input and output of nodes containing attributes of that type and the assignment operation associated with the type.

There are restrictions on an imported Ada type that is used to *form* an attribute type. Such a type must be readable and writeable using the Ada direct input/output facilities ([DoD, 1983], Section 14.2.4) and cannot be a limited type ([DoD, 1983], Section 7.4.4).

The remainder of this description of imported types applies only to those imported types that are used as attribute types.

*Input/Output Information*

Interface packages generated by the GRAPHITE processor use instantiations of the generic Ada direct input/output package Direct_IO (see [DoD, 1983], Section 14.2.4) to read and write graph instances to and from operating system files {§3.3.4}. Values of all types except access types and types formed from access types are directly readable and writeable.

If an imported type is formed from an access type, then an *external-form type* must be defined to serve as an external representation of the imported type. The external-form type should hold (or allow the retrieval of) the same information as the imported type and be directly readable and writeable. In particular, it cannot be formed from access types.

In addition to the external-form type, two dual-parameter procedures must be defined, serving to convert values between the imported type and the external-form type. These procedures are specified as follows.

- The procedure that converts a value of the imported type into a value of the external-form type is referred to as the *externalize operation*. The type of the first parameter is the external-form type and the type of the second parameter is the imported type.
- The procedure that converts a value of the external-form type into a value of the imported type is referred to as the *internalize operation*. The type of the first parameter is the imported type and the type of the second parameter is the external-form type.
- The modes of the parameters in both procedures should be **in out**.

An external-form type cannot be an unconstrained array. Furthermore, if an external-form type is of an unconstrained type with discriminants, then defaults for those discriminants must have been given.

An imported type, its external-form type, its internalize operation, and its externalize operation need not be defined in the same package.[6]

*Examples:*

In Figure 2.2, the imported type SomeType3 is described as having an external-form type ExternalSomeType3 and conversion procedures InternalizeSomeType3, which converts values of the external-form type to their "internal" form (SomeType3), and ExternalizeSomeType3, which converts values of the imported type to their external form (ExternalSomeType3). Imported type SomeType4, since it is a subtype of SomeType3, shares SomeType3's internalize and externalize operations.

*Notes:*

This facility can be used even when the imported type is itself directly readable and writeable. For example, it can be used to specify a more compact external representation.

The type for designating nodes {§3.2.2} in a class is not suitable for reading and writing. Interface packages, however, do not provide an external-form type corresponding to the type for designating nodes. Therefore, a node containing an attribute whose type is the type for designating nodes in some other class cannot be read and written.[7]

*Assignment Operation*

Interface packages generated by the GRAPHITE processor make use of the assignment operation defined for an imported type. By default, this operation has the syntax of the built-in assignment operator ":=". If an imported type is a *limited* type, however, then the assignment operator is not valid ([DoD, 1983], Section 7.4.4) and must be explicitly defined.

An explicitly defined assignment operation associated with an imported type must be a dual-parameter procedure whose first parameter would correspond to the left operand of the built-in assignment operator and whose second parameter would correspond to the right operand of that operator.

*Examples:*

[6] This does not actually apply to the current version of GRAPHITE, which does in fact require that all those entities be defined in the same package.

[7] This restriction will be corrected in a future version of GRAPHITE.

In Figure 2.2, imported type SomeType1, a subtype of the predefined Ada type Integer, does not require an indication of its assignment operation, since the built-in operator ":=" is the default for values of this type. Imported type SomeType2 apparently has the built-in operator associated with it as well. Imported type SomeType3, on the other hand, is associated with an explicitly defined assignment operation, SomeType3Assign. Finally, SomeType4, since it is a subtype of SomeType3, shares SomeType3's operations, including the assignment operation, SomeType3Assign.

### 2.4.4 GDL-specific Types

There are three GDL-specific types: *node kind*, *node group*, and *node sequence*. Each is an attribute type constructor that facilitates the description of a graph class and, thus, has no counterpart in Ada.

Node kinds, node groups, and node sequences cannot be used in the declarations of user-defined Ada types (e.g., as components of a record type).

#### Node Kind

A node kind is a named collection of attributes, as described in Section 2.3. The operations appropriate to values of node kinds (i.e., nodes) are described in Chapter 3.

Every node kind in a given class shares a distinguished value referred to as the *null node value*, which signifies a node kind attribute that designates no node at all. This value is denoted by an identifier that is derived from the name of the class by prepending the characters Null onto the class name. The null node value is the default initial value for all attributes that are node kinds and node groups {§2.6}.

*Examples:*

For the class of Figure 2.1, the null node value is denoted by the identifier NullExampleGraph.

*Notes:*

The null node value is provided by the interface package as a constant {§3.2.2}.

## Node Group

---

group_declaration ::= full_group_declaration | incomplete_group_declaration

full_group_declaration ::=
**group** identifier **is** ( group_member {, group_member} ) [:= expression];

group_member ::= *node_kind*_simple_name | *group*_simple_name

---

A node group is used as an attribute type for an attribute whose values can be any one of a number of different node kinds. The operations appropriate to values of a node group are the same as those for a node kind, since a value of a node group is simply a node.

A given node kind can be a member of more than one node group.

Node groups can be nested. That is, a member of a node group can itself be a node group. A node group G1 that is a member of another node group G2 can be replaced by its members in the declaration of G2 without any effect on the users of G2.

Circularity in the definition of a node group is not permitted. That is, a node group cannot be a (direct or indirect) member of itself.

GDL provides a predefined node group that contains as its members all node kinds in the given class. This node group is denoted by the reserved word **node**. Node group **node** cannot be redeclared.

*Examples:*

In Figure 2.1, the node group Statement is defined to range over a set of node kinds representing statements.

*Notes:*

Since a value of a node group is a node kind, node groups share the null node value.

The rules for membership in node groups allow for a given node kind to be defined more than once as a member of a given node group. The same effect is obtained whether that node kind is mentioned once or more than once as a member of that node group.

A node group containing (either directly or indirectly) the predefined node group **node** contains all node kinds in the given class and can be effectively replaced by **node**.

## Node Sequence

---

sequence_declaration ::= **type** identifier **is sequence of** sequence_element;

sequence_element ::= *node_kind*_simple_name | *group*_simple_name | **node**

---

A node sequence is used to indicate an ordered collection of nodes. Operations on values of a node sequence include those to create a sequence, retrieve an element of a sequence, and determine whether or not a sequence is empty. These operations are described in Chapter 3.

If the simple name in a sequence declaration denotes a node kind, then an element of that sequence can be of only that node kind. Alternatively, if the simple name in a sequence declaration denotes a node group, then an element of that sequence can be of any node kind in the group.

Every node sequence in a given class shares a distinguished value referred to as the *null node sequence value*, which signifies a node sequence attribute that designates no node sequence at all. This value is denoted by an identifier that is derived from the name of the class by prepending the characters Null and appending the characters Sequence onto the name. The null node sequence value is the default initial value for all attributes that are node sequences {§2.6}.

*Examples:*

In Figure 2.1, type StatementSequence is declared to be an ordered list of nodes, where each node must be one of the node kinds declared in the group Statement.

For the class of Figure 2.1, the null node sequence value is denoted by the identifier NullExampleGraphSequence.

*Notes:*

The null node sequence value is provided by the interface package as a constant {§3.2.2}.

The null node sequence value is not the same as an empty node sequence, which is the value returned by the create operation for node sequences {§3.3.3}.

### Incomplete Declarations

---

```
incomplete_node_kind_declaration  ::=  node identifier_list;

incomplete_group_declaration  ::=  group identifier;
```

---

Node kinds and node groups can be used as attribute types, members of node groups, and as elements of node sequences before their full declarations are given. To do so requires prior incomplete declarations for those node kinds and node groups.

For each incomplete node kind declaration, a corresponding full declaration with the same identifier list must be given within the class declaration. (While the elements of such identifier lists must be the same, the order of those elements is not significant.) Similarly, for each incomplete node group declaration, a corresponding full declaration with the same identifier must be given within the class declaration.

*Examples:*

In Figure 2.1, group Statement and node kind ConditionClause are introduced through incomplete declarations.

*Notes:*

The use of incomplete node kinds and node groups allows the definition of recursive graph structures.

## 2.5 Constants

---

```
Ada_constant_declaration  ::=  constant_declaration  |  number_declaration

constant_declaration  ::=
        identifier_list : constant subtype_indication := expression;
    |   identifier_list : constant constrained_array_definition := expression;

number_declaration  ::=  identifier_list : constant := universal_static_expression;
```

---

A GDL class declaration can include the declaration of an Ada constant, which may be a constant Ada object or a named number ([DoD, 1983], Section 3.2). Such a constant can then be used in further declarations within the class declaration.

An Ada constant declaration that has an identifier list with more than one element is equivalent to a series of Ada constant declarations, each having one element in its identifier list.

*Examples:*

In Figure 2.1, LowWeight is defined as a constant and used in the subsequent definition of BranchWeight.

*Notes:*

Although the syntax of an Ada constant declaration closely resembles that of a commonly-available attribute declaration, the two declarations are semantically quite different. In particular, an Ada constant cannot be used as a commonly-available attribute—that is, it cannot be listed as an attribute of a node kind in a node kind declaration.

## 2.6 Attribute Initialization

All attributes in a GDL class declaration can be given initial values. In the absence of a user-specified initial value, an attribute that is a node kind, a node group, or a node sequence is given a "null" initial value by default (see below). User-specified initial values can be given to attributes of all attribute types except node sequences.[8]

A user-specified initial value is given as an expression of the appropriate type; syntactically, it is the same as an initialization expression for Ada objects. In the case of a node kind, the form taken is that of a record aggregate. In the case of a node group, the form is that of a qualified record aggregate, where the qualifier is the name of the node kind in the group to which the attribute is being initialized.

An initial value for an attribute can be given in three places:

- in the declaration of an attribute type, except for an imported Ada type;
- in the declaration of an attribute, both within a node kind declaration and in a commonly-available attribute declaration; and
- in the use of a commonly-available attribute declaration within a node kind declaration.

In the event that more than one initial value is given for a particular attribute, the precedence order from highest to lowest is: node kind declaration, commonly-available attribute

[8]The current version of the GRAPHITE processor does not actually support user initialization of node kinds and node groups.

declaration, attribute type declaration. For example, if an attribute type AT is declared and an initial value $i$ given, then that value holds for all attributes of that type, except for those attributes of type AT initialized to a different value $j$ in their attribute declarations. If the attribute declaration of type AT is commonly available, then that second initial value $j$ can be overridden by an initial value $k$ given in a node kind declaration that uses the commonly-available declaration.

The default initial value for a node kind or a node group is the null node value for the class {§2.4.4}. The default initial value for a node sequence is the null node sequence value for the class {§2.4.4}.

*Notes:*

A consequence of the rules for determining the initial value of an attribute is that any initial values given in types used to *form* an attribute type are ignored.

GDL differs from Ada in that GDL allows initializations in all user-definable Ada attribute type declarations, except in those involving unconstrained arrays. Ada allows initialization only of fields of record types.

## 2.7 Declaration Order

The rules governing the order of declarations in a GDL class declaration follow Ada's linear elaboration rules; an entity cannot be referred to before it is declared (see [DoD, 1983], Section 3.9). Mutually dependent or recursive node kind declarations can be specified by using incomplete declarations {§2.4.4}.

## 2.8 Identifier Visibility

The rules governing the visibility of identifiers in GDL are similar to the rules found in Ada (see [DoD, 1983], Chapter 8).[9] In particular, the identifiers serving as names for the declarative items in a class declaration {§2.2} cannot be redeclared within the same class declaration. The exception is the identifier associated with a commonly-available attribute declaration {§2.3.1}, which can be reused in a (local) attribute declaration within a node. The identifiers serving as the names of record fields, enumeration literals, and (local) attributes can also be reused within a class declaration.

[9]The current version of the GRAPHITE processor supports neither the redeclaration of predefined Ada types and subtypes {§2.4.2}, nor the reuse of imported identifiers {§2.4.3}.

# Chapter 3

# Interface Packages

Having described the input to the GRAPHITE processor in the previous chapter, we now turn to the output, namely the Ada interface package that implements a given GDL specification of a class of graphs. As mentioned in Chapter 1, the GRAPHITE processor is designed to generate either a development version or a production version of such a package.[1] The versions are similar in that they realize a graph as an abstract data type and provide the same set of operations on graphs. The versions differ in how they resolve the often conflicting goals of flexibility and efficiency; while the express purpose of the development interface package is to support flexibility by minimizing the impact of changes on clients of an interface package, the intent of the production interface package is to provide efficient access to a relatively stable class of graphs.

This chapter describes the specification parts of development and production interface packages. Knowledge of the entities defined in the specification parts are necessary for developing programs that use an interface package. (The contents of the body parts, on the other hand, should be irrelevant.) First, an overview of the compilation units generated by the GRAPHITE processor is given. This is followed by descriptions of the types, constants, subprograms, and exceptions defined in interface package specification parts. Finally, considerations for compilation of an interface package and its clients and considerations for programming of clients are presented.

[1] The current version of the GRAPHITE processor does not actually support the generation of a production interface package.

## 3.1 Compilation Units

From a given GDL specification, the GRAPHITE processor generates two Ada compilation units. These compilation units correspond to the specification and body parts of the interface package. The name of the package is specified by the package name declaration appearing in the class declaration {§2.2}.

The compilation unit corresponding to the specification part of an interface package may begin with an Ada *with clause*. Such a *with clause* is used to import any types and subtypes defined within the class declaration as imported Ada types or subtypes {§2.4.3}.

*Examples:*

Figure 3.1 shows a portion of the specification part of the development interface package that would be generated by GRAPHITE from the GDL specification given in Figure 2.1 on page 9. Notice that the name of the interface package is Example and that there is an Ada *with clause* mentioning Lexical, which is the package that defines the types Comment and Position.

This figure is used throughout Chapter 3 to illustrate various features of interface packages.

## 3.2 Types and Constants

### 3.2.1 Attribute Subtypes and Attribute Base Types

As part of the process of generating an interface package, the attribute types defined and/or used in a GDL specification are mapped to Ada types defined and/or used in the interface package. Table 3.1 summarizes these mappings, which are explained in greater detail in subsequent sections.

In Table 3.1 and in the subsequent discussion, a distinction is made between an *attribute subtype* and an *attribute base type*. An attribute subtype is the subtype indication given as part of an attribute declaration in a GDL specification {§2.3}. An attribute base type is the base type of the subtype (see [DoD, 1983], Section 3.3) and is used to communicate attribute values in get/put subprograms of an interface package {§3.3}.

All node kinds and node groups of a given class are considered to be subtypes of a single private type, which is declared in the interface package. Similarly, all node sequences of a given class are considered to be subtypes of another, single private type declared in the interface package.

*Notes:*

```
   -- imported types and subtypes
      with Lexical;

package Example is
   -- GDL-specific types and constants
      type ExampleGraph          is private;
      type ExampleGraphSequence  is private;

      NullExampleGraph           : constant ExampleGraph;
      NullExampleGraphSequence   : constant ExampleGraphSequence;

   -- user-defined types, subtypes, and constants
      type LexicalInformation is
        record
          SourceComment : Lexical.Comment;
          SourcePosition : Lexical.Position;
        end record;
      LowWeight : constant Integer := -10;
      type BranchWeight is new Integer range LowWeight .. 10;
      . . .

   -- types for communicating names
      type NodeKindName          is new String;
      type NodeSequenceName      is new String;
      type AttributeName         is new String;
      type AttributeSubtypeName  is new String;
      type AttributeBaseTypeName is ( LexicalInformation_AT, BranchWeight_AT, ExampleGraph_AT,
                                      ExampleGraphSequence_AT, Standard_Integer_AT, ... );

   -- type for communicating attribute family counts
      type AttributeFamilyCountList is array ( Positive range <> ) of Natural;

   -- types for listing a node's attributes
      type AttributeNamePointer is access AttributeName;
      type AttributeNameList is array ( Positive range <> ) of AttributeNamePointer;

   -- subprograms to ascertain a node's definition
      function Kind ( TheNode : ExampleGraph ) return NodeKindName;
      function AttributeSubtype ( TheAttribute : AttributeName; TheNodeKind : NodeKindName )
                                  return AttributeSubtypeName;
      function AttributeBaseType ( TheAttribute : AttributeName; TheNodeKind : NodeKindName )
                                   return AttributeBaseTypeName;
      function NodeKindAttributes ( TheNodeKind : NodeKindName ) return AttributeNameList;

   -- subprograms to manipulate a node
      function   Create ( TheNodeKind : NodeKindName ) return ExampleGraph;
      function   Create ( TheNodeKind : NodeKindName; TheList : AttributeFamilyCountList )
                        return ExampleGraph;
      function   Create ( TheNodeKind : NodeKindName; TheList : Natural )
                        return ExampleGraph;
```

Figure 3.1: Specification Part of Development Interface Package for Class of Figure 2.1.

```
procedure DeleteNode ( TheNode : in out ExampleGraph );
procedure SubstituteUnchecked ( OldNode : ExampleGraph; NewNode : ExampleGraph );
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheValue : LexicalInformation );
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex: Positive; TheValue : LexicalInformation );
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                         return LexicalInformation;
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex : Positive ) return LexicalInformation;
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheValue : BranchWeight );
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex: Positive; TheValue : BranchWeight );
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                         return BranchWeight;
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex : Positive ) return BranchWeight;
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheValue : ExampleGraph );
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex: Positive; TheValue : ExampleGraph );
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                         return ExampleGraph;
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex : Positive ) return ExampleGraph;
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                         TheValue : ExampleGraphSequence );
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                         TheIndex: Positive; TheValue : ExampleGraphSequence );
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                         return ExampleGraphSequence;
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex : Positive ) return ExampleGraphSequence;
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheValue : Standard.Integer );
procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex: Positive; TheValue : Standard.Integer );
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                         return Standard.Integer;
function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                         TheIndex : Positive ) return Standard.Integer;
...
```

Figure 3.1: (continued).

```
   -- subprograms to manipulate node sequences
      function  Create ( TheSequenceKind : NodeSequenceName ) return ExampleGraphSequence;
      function  Kind ( TheSequence : ExampleGraphSequence ) return NodeSequenceName;
      procedure Insert ( TheSequence : ExampleGraphSequence; ThePosition : Positive;
                         TheNode : ExampleGraph );
      function  Length ( TheSequence : ExampleGraphSequence ) return Natural;
      function  Retrieve ( TheSequence : ExampleGraphSequence; ThePosition : Positive )
                           return ExampleGraph;
      procedure Remove ( TheSequence : ExampleGraphSequence; ThePosition : Positive );
   -- subprograms to input and output graphs
      procedure ReadGraph ( FileName : String; TheGraph : in out ExampleGraph );
      procedure WriteGraph ( FileName : String; TheGraph : in out ExampleGraph );
   -- exceptions
      UnknownNodeKind         : exception;
      UnknownAttribute        : exception;
      UnexpectedNodeKind      : exception;
      UnexpectedAttribute     : exception;
      UnknownNodeSequence     : exception;
      NodeSequenceError       : exception;
      UnexpectedNodeSequence  : exception;
      AttributeFamilyCountError : exception;
private
   -- (partial) representations for private types and values for deferred constants
      . . .
end Example;
```

Figure 3.1: (continued).

| ATTRIBUTE SUBTYPES IN GDL SPECIFICATIONS | ATTRIBUTE BASE TYPES IN INTERFACE PACKAGES |
|---|---|
| Ada Base Type | Same Base Type |
| Ada Subtype | Base Type of Subtype† |
| Imported Ada Base Type | Same Base Type |
| Imported Ada Subtype | Base Type of Subtype |
| All Node Kinds and Node Groups | One Private Type |
| All Node Sequences | One Private Type |

†The *declaration* of the user-defined subtype nonetheless appears in the interface package {§3.2.6}.

Table 3.1: Mappings of Types in GDL Specifications to Types in Interface Packages.

The mappings of attribute subtypes to attribute base types are used in only certain situations, which are described below. In particular, the constraints on values implied by subtypes are preserved by interface packages.

### 3.2.2 GDL-specific Types and Constants

The name of the class given in a GDL specification is used as the name of an Ada private type whose objects designate nodes in graphs of the class. This type is the base type of all the attribute subtypes formed from the GDL-specific attribute type constructor *node kind* {§2.4.4}.

The name of the class is also used to form the name of an Ada private type whose objects designate sequences of nodes in the class. In particular, for a class C, the name of the type would be CSequence. This type is the base type of all the attribute subtypes formed from the GDL-specific attribute type constructor *node sequence* {§2.4.4}.

Although one (base) type is used to designate nodes of all kinds, an interface package will guarantee at run time that a node is used in a manner consistent with its kind. For instance, if a node kind has an attribute *A* whose (sub)type is another node kind *NK*, then only nodes of kind *NK* will be allowed as values of attribute *A*. The same guarantee is made for node sequences, which, like nodes, are designated by objects of one (base) type.

The semantics of node and node sequence assignment is *sharing* (as opposed to *copy*). Similarly, the semantics of the equality/inequality test for nodes and node sequences is based on designation of the *same* node or node sequence (as opposed to designation of an *equivalent* node or node sequence).

The null node value for a class is provided by a constant whose identifier is derived from the name of the class by prepending the characters Null onto the class name. The null node value is the default initial value for all attributes that are node kinds and node groups {§2.6}.

The null node sequence value for a class is provided by a constant whose identifier is derived from the name of the class by prepending the characters Null and appending the characters Sequence onto the class name. The null node sequence value is the default initial value for all attributes that are node sequences {§2.6}.

*Examples:*

In Figure 3.1, the type for designating nodes is ExampleGraph and the type for designating node sequences is ExampleGraphSequence. The null node value is denoted by the constant NullExampleGraph and the null node sequence value is denoted by the constant NullExample-GraphSequence.

*Notes:*

Because the types for designating nodes and node sequences are private, the only operations that can be performed on them (other than assignment and the equality/inequality tests) are those realized by the visible subprograms defined elsewhere in the specification part {§3.3}.

The information conveyed by the GDL-specific attribute type constructor *node group* is used only to constrain the possible values of an attribute that designates a node {§2.4.4}. Therefore, the private type used as the attribute base type for nodes is also used for node groups.

### 3.2.3 Types for Communicating Names

Five types are used to communicate the names of node kinds, node sequences, attributes, attribute subtypes, and attribute base types between an interface package and the clients of that package. These are NodeKindName, NodeSequenceName, AttributeName, AttributeSubtypeName, and AttributeBaseTypeName, respectively.

#### NodeKindName, NodeSequenceName, and AttributeName

In a development interface package, NodeKindName, NodeSequenceName, and AttributeName are derived from the type String. The valid values of these types, for a given GDL specification, are character strings identical to the names used in that GDL specification, with case ignored. Given a character string $S$ used as a value of one of these types, the interface package checks the validity of $S$ upon invocation of a subprogram that takes $S$ as a parameter. The following exceptions may be raised as a result this check {§3.4}:

- UnknownNodeKind is raised if $S$ is not a valid node kind name;
- UnknownNodeSequence is raised if $S$ is not a valid node sequence name; and
- UnknownAttribute is raised if $S$ is not a valid attribute name.

Checking for valid names can be suppressed using the GDL pragma Suppress {§4}.

In a production interface package, NodeKindName, NodeSequenceName, and AttributeName are enumeration types whose literals are identical to the names used in the GDL specification. Given an enumeration literal $E$ used as the value of one of these types, the validity of $E$ is checked by the Ada compiler. (Thus, the exceptions UnknownNodeKind, UnknownNodeSequence, and UnknownAttribute can never be raised by a production interface package.)

In the case of an attribute family, the "family name" is considered to be the attribute name {§2.3.2}.

Type **NodeSequenceName** is generated only if the GDL-specific type *node sequence* is used to construct at least one attribute type {§2.4.4}.

If a GDL specification exclusively contains node kinds that do not have any attributes {§2.3}, then **AttributeName** is given a single, "dummy" value: **Null_AN**.

*Examples:*

A production interface package generated from the GDL specification given in Figure 2.1 would include the following definitions:

```
type NodeKindName     is ( ConditionClause, ExpressionNode, IfStatement, ... );
type NodeSequenceName is ( StatementSequence, ... );
type AttributeName    is ( SourceConnection, ExecutionCount, Weight, ...,
                           ElsIfBranch, Condition, ... );
```

*Notes:*

It is recommended that when using a development interface package, string constants be used to represent values of types **NodeKindName**, **NodeSequenceName**, and **AttributeName** {§3.6.2}.

### AttributeSubtypeName and AttributeBaseTypeName

In a development interface package, **AttributeSubtypeName** is derived from the type **String** and is used as the return type of function **AttributeSubtype** {§3.3}. Values of this type (returned by the function) take the following form:

[<*external package name*>_]<*attribute subtype name*>_AT

where the "name" of an attribute subtype is considered to be the identifier used in the subtype indication of an attribute declaration {§2.4}. The external package name appears only if the attribute type is imported {§2.4.3} or is a predefined Ada type or subtype {§2.4.2}. In the latter case, the package name used is **Standard** (see [DoD, 1983], Section 8.6).

When the predefined node group **node** is used as an attribute type, then the subtype name used in **AttributeSubtypeName** is **Node_AT** {§2.4.4}.

In a production interface package, AttributeSubtypeName is an enumeration type. The literals of this type are identical to the character strings used in the development interface package.

Pragma SubtypeName can be used to override the generated values of type AttributeSubtypeName described above {§4}.

AttributeBaseTypeName is an enumeration type (in both development and production interface packages) whose literals correspond to the base types of attribute subtypes {§3.2.1}. AttributeBaseTypeName is used as the return type of function AttributeBaseType {§3.3}. Values of this type take the following form:

[<*external package name*>_]<*attribute base type name*>_AT

The external package name appears only if the attribute base type is imported {§2.4.3} or is a predefined Ada type {§2.4.2}. In the latter case, the package name used is Standard (see [DoD, 1983], Section 8.6).

Pragma BaseTypeName can be used to override the default generated values of type AttributeBaseTypeName described above {§4}.

If a GDL specification exclusively contains node kinds that do not have any attributes {§2.3}, then AttributeSubtypeName and AttributeBaseTypeName are both given a single, "dummy" value: Null_AT.

*Examples:*

In the absence of SubtypeName and BaseTypeName pragmas, the attribute type LexicalInformation appearing in Figure 2.1 results in the character string LexicalInformation_AT of type AttributeSubtypeName and enumeration literal LexicalInformation_AT of type AttributeBaseTypeName (the latter appears in the specification part shown in Figure 3.1).

In the absence of SubtypeName and BaseTypeName pragmas, the attribute type Natural appearing in Figure 2.1 results in the character string Standard_Natural_AT of type AttributeSubtypeName and enumeration literal Standard_Integer_AT of type AttributeBaseTypeName (the latter appears in the specification part shown in Figure 3.1).

A production interface package generated from the GDL specification given in Figure 2.1 would include the following definition:

```
type AttributeSubtypeName is ( LexicalInformation_AT, Standard_Natural_AT,
                               BranchWeight_AT, ExpressionNode_AT,
                               StatementSequence_AT, ConditionClause_AT, ... );
```

*Notes:*

It is recommended that when using a development interface package, string constants be used to represent values of type AttributeSubtypeName {§3.6.2}.

The elements of AttributeBaseTypeName correspond to the sets of get/put subprograms provided by an interface package {§3.3}.

### 3.2.4 Type for Communicating Attribute Family Counts

An *attribute family count* is an integer of subtype Natural that represents the number of members of an attribute family {§2.3.2}. This number must be supplied at the time an instance of the node kind containing the attribute family is created.

#### AttributeFamilyCountList

A given node kind may contain declarations for one or more attribute families. The type AttributeFamilyCountList is an unconstrained, one-dimensional array whose elements represent attribute family counts. Its index subtype is Positive. AttributeFamilyCountList is used in an interface package as the type of one of the parameters to the overloaded function Create {§3.3}.

For a node kind that has $n$ attribute families, a value of type AttributeFamilyCountList should have $n$ elements. The order of the elements is the same as the order of the declarations of the corresponding attribute families.

*Examples:*

Given the following node kind declaration, which contains declarations for two attribute families:

```
node Node1 is
   Att1          : Integer;
   AttFam1 (<>) : Boolean;
   Att2          : Integer;
   AttFam2 (<>) : String ( 1 .. 10 );
end node;
```

a call on a (production) interface package to create a node of this kind having 3 (three) members of attribute family AttFam1 and 5 (five) members of attribute family AttFam2 would look like this:

```
D := Create ( Node1, ( 3, 5 ) );
```

(Function Create is described in Section 3.3.)

*Notes:*

For a parameter to function Create of type AttributeFamilyCountList, if the number of attribute families in the node being created is $n$ and the number of elements in the list is $n+m$, then the $m$ excess family counts in the list are ignored by Create {§3.3}.

### 3.2.5 Types for Listing a Node's Attributes

#### AttributeNameList and AttributeNamePointer

The type AttributeNameList is used to represent a list of the names of attributes declared for a node kind; AttributeNameList is used in an interface package as the return type of function NodeKindAttributes {§3.3}.

AttributeNameList is an unconstrained, one-dimensional array whose elements are pointers to attribute names. These pointers are represented by the type AttributeNamePointer, which is an access type whose designated type is AttributeName {§3.2.3}. The index subtype of AttributeNameList is Positive.

For a node kind that has $n$ attributes, a value of type AttributeNameList has $n$ elements. The order of the elements is the same as the order of the declarations of the corresponding attributes.

For a node kind that has no attributes, a value of type AttributeNameList is a null array (see [DoD, 1983], Section 3.6.1).

*Notes:*

It is recommended that clients of an interface package use an access type whose designated type is AttributeNameList, rather than using statically declared variables of type AttributeNameList. This avoids the problem of having to specify the size of such variables at the time of their declaration.

It is also recommended that values of type AttributeNameList be manipulated through Ada attributes[2] such as 'First and 'Last, which allow manipulation of arrays in a general-purpose fashion (see [DoD, 1983], Section 3.6.2).

[2]The term "attribute" has a meaning in Ada different from that in GRAPHITE. Its use here refers to the Ada meaning; an Ada attribute is akin to a function (see [DoD, 1983], Section 4.1.4).

### 3.2.6 User-defined Types, Subtypes, and Constants

The declarations of user-defined types, subtypes, and constants that appear in a GDL specification are reproduced as-is in the specification part of an interface package. The order of those declarations is preserved.

*Examples:*

The user-defined constant LowWeight and types LexicalInformation and BranchWeight declared in class ExampleGraph of Figure 2.1 appear as-is in the specification part of package Example of Figure3.1.

*Notes:*

The declarations of imported Ada types {§2.4.3} appearing in a GDL specification do not result in any type declarations in the specification part of an interface package. Instead, an Ada *with clause*, listing the names of the packages in which those imported types are declared, is attached to the specification part {§3.1}.

## 3.3 Subprograms

The subprograms provided by GRAPHITE-generated interface packages fall into four basic categories, as shown in Table 3.2.

### 3.3.1 Subprograms to Ascertain a Node's Definition

**Kind**

---

**function** Kind ( TheNode : <*class name*> ) **return** NodeKindName;

---

For node TheNode, function Kind retrieves the name of the node's kind {§3.2.3}, where <*class name*> is the type for designating nodes {§3.2.2}.

In a development interface package, the name is returned as a character string in upper case.

*Exceptions Raised:*

- Constraint_Error (predefined Ada exception)

| 1. SUBPROGRAMS TO ASCERTAIN A NODE'S DEFINITION | |
|---|---|
| Kind | retrieves the name of a node's kind |
| AttributeSubtype | retrieves the name of an attribute's subtype |
| AttributeBaseType | retrieves the name of an attribute's base type |
| NodeKindAttributes | retrieves the names of a node kind's attributes |
| **2. SUBPROGRAMS TO MANIPULATE A NODE** | |
| Create† | creates a new node of a given kind |
| DeleteNode | deletes a node |
| PutAttribute† | puts the value of an attribute of a given type |
| GetAttribute† | gets the value of an attribute of a given type |
| **3. SUBPROGRAMS TO MANIPULATE NODE SEQUENCES** | |
| Create | creates a given sequence |
| Kind | retrieves the name of a sequence |
| Insert | inserts a node into a sequence at a given position |
| Length | retrieves the length of a sequence |
| Retrieve | retrieves a node from a sequence at a given position |
| Remove | removes a node at a given position from a sequence |
| **4. SUBPROGRAMS TO INPUT AND OUTPUT GRAPHS** | |
| ReadGraph | reads a graph from a file |
| WriteGraph | writes a graph to a file |

†Overloaded within this category.

Table 3.2: Subprograms Provided by Interface Packages.

- TheNode is null

*Examples:*

The following call to function Kind of package Example in Figure 3.1:

```
Kind ( Create ( "IfStatement", 5 ) )
```

would yield the character string IFSTATEMENT. (Function Create is described below.)

### AttributeSubtype

---

```
function AttributeSubtype ( TheAttribute : AttributeName; TheNodeKind : NodeKindName )
                          return AttributeSubtypeName;
```

---

For attribute TheAttribute declared in node kind TheNodeKind, function AttributeSubtype retrieves the name of the attribute's subtype {§3.2.3}.

In a development interface package, the name is returned as a character string in upper case.

*Exceptions Raised:*

- UnknownAttribute
  - TheAttribute is not a valid attribute name (development interface package only) {§3.2.3}
- UnknownNodeKind
  - TheNodeKind is not a valid node kind name (development interface package only) {§3.2.3}

*Examples:*

The following call to function AttributeSubtype of package Example in Figure 3.1:

```
AttributeSubtype ( "ExecutionCount", "IfStatement" )
```

would yield the character string STANDARD_NATURAL_AT.

### AttributeBaseType

---

**function** AttributeBaseType ( TheAttribute : AttributeName; TheNodeKind : NodeKindName )
**return** AttributeBaseTypeName;

---

For attribute **TheAttribute** declared in node kind **TheNodeKind**, function **AttributeBaseType** retrieves the name of the attribute's base type {§3.2.3}.

*Exceptions Raised:*

- **UnknownAttribute**
  - **TheAttribute** is not a valid attribute name (development interface package only) {§3.2.3}
- **UnknownNodeKind**
  - **TheNodeKind** is not a valid node kind name (development interface package only) {§3.2.3}

*Examples:*

The following call to function **AttributeBaseType** of package **Example** in Figure 3.1:

**AttributeBaseType ( "ExecutionCount", "IfStatement" )**

would yield the enumeration literal **Standard_Integer_AT**.

### NodeKindAttributes

---

**function** NodeKindAttributes ( TheNodeKind : NodeKindName ) **return** AttributeNameList;

---

For node kind **TheNodeKind**, function **NodeKindAttributes** retrieves a list of the names of the node kinds's attributes {§3.2.3}.

If node kind **TheNodeKind** does not contain any attributes, then this function returns a null array (see [DoD, 1983], Section 3.6.1).

In a development interface package, the names are returned as character strings in upper case.

*Exceptions Raised:*

- **UnknownNodeKind**
  - **TheNodeKind** is not a valid node kind name (development interface package only) {§3.2.3}

*Examples:*

The following call to function **NodeKindAttributes** of package **Example** in Figure 3.1:

```
NodeKindAttributes ( "IfStatement" )
```

would yield the following list of character strings:

```
SOURCECONNECTION
EXECUTIONCOUNT
IFBRANCH
ELSIFBRANCH
ELSEBRANCH
```

### 3.3.2 Subprograms to Manipulate a Node

**Create**

---

```
function Create ( TheNodeKind : NodeKindName ) return <class name>;
function Create ( TheNodeKind : NodeKindName; TheList : AttributeFamilyCountList )
                  return <class name>;
function Create ( TheNodeKind : NodeKindName; TheList : Natural ) return <class name>;
```

---

Overloaded function **Create** creates a new node of kind **TheNodeKind**, where *<class name>* is the type for designating nodes in the class {§3.2.2}. The attributes of the node are given initial values, if such values have been specified {§2.6}.

The two forms of Create that take parameter TheList are used for creating instances of node kinds containing attribute families {§2.3.2}. The form that provides TheList as type AttributeFamilyCountList is for lists of counts for 1 (one) or more families {§3.2.4}. The other form is for creating instances of node kinds having exactly 1 (one) attribute family. This second form simply provides a syntactic convenience that avoids the need for parentheses to enclose a literal of type Natural, as would be required if the other form were used. For example, the following two calls to Create, the first of which is the form involving a parameter of type AttributeFamilyCountList and the second of which is the form involving a parameter of type Natural, are effectively equivalent:

```
Create ( SomeNodeKind, ( 3 ) )
Create ( SomeNodeKind,   3   )
```

For a call to function Create that includes attribute family counts, if the number of attribute families in the node being created is $n$ and the number of elements in parameter TheList is $n+m$, then the $m$ excess family counts in the list are ignored.

For a production interface package, the last two forms of function Create are generated only if at least one attribute family is declared in the GDL specification.

For a development interface package, the last two forms of function Create are always generated.

*Exceptions Raised:*

- AttributeFamilyCountError
  - TheList has too few elements for the node kind being created
  - an individual element of TheList (or the value itself, in the case of the third form of Create) is greater than the maximum allowed number of family members for that attribute
- Constraint_Error (predefined Ada exception)
  - an error occurs in an attribute initialization expression {§2.6}
  - the value of an element of TheList (or the value itself, in the case of the third form of Create) is not in the range of Natural
- Storage_Error (predefined Ada exception)
  - storage is not sufficient to accommodate the new node or one of its initialized attributes

- **UnknownNodeKind**
  - **TheNodeKind** is not a valid node kind name (development interface package only) {§3.2.3}

**DeleteNode**

---

```
procedure DeleteNode ( TheNode : in out <class name> );
```

---

For node **TheNode**, procedure **DeleteNode** attempts to release the storage allocated to that node, where *<class name>* is the type for designating nodes in the class {§3.2.2}.

The effect of calling **DeleteNode ( N )** is as follows:

- after executing **DeleteNode ( N )**, the value of **N** is the null node value {§3.2.2};
- **DeleteNode ( N )**, when **N** is already equal to the null node value, has no effect.

If **M** and **N** designate the same node, then referring to this node through **M** is erroneous if the reference is attempted after the call **DeleteNode ( N )**.

*Notes:*

Deletion of an entire graph can be performed by traversing the graph and deleting the individual nodes, using procedure **DeleteNode**, encountered in the traversal.

If a node **M** is an attribute of a node **N** and no other means of referring to **M** exists (e.g., through a variable or through an attribute of a node other than **N**), then after the call **DeleteNode ( N )**, node **M** is no longer accessible and its storage might be left unreclaimed.

Procedure **DeleteNode** makes use of the predefined Ada generic procedure **Unchecked_Deallocation** (see [DoD, 1983], Section 13.10.1). Its effect, therefore, is heavily dependent upon the Ada language system being used.

**PutAttribute**

---

```
procedure PutAttribute ( TheNode : <class name>; TheAttribute : AttributeName;
                         TheValue : <attribute base type> );
procedure PutAttribute ( TheNode : <class name>; TheAttribute : AttributeName;
                         TheIndex: Positive; TheValue : <attribute base type> );
```

---

For node TheNode, the first form of overloaded procedure PutAttribute puts value TheValue into attribute TheAttribute, where <*class name*> is the type for designating nodes in the class {§3.2.2} and <*attribute base type*> is the base type of the attribute {§3.2.1}. The second form is for putting a value into member TheIndex of an attribute family {§2.3.2}.

Procedure PutAttribute is actually overloaded along two dimensions. The first dimension, as described above, includes a pair of procedures, one for simple attributes and the other for members of attribute families. The second dimension of overloading is based on attribute base types. In particular, there is one pair of procedures in the first dimension for each attribute base type in the class.

For a production interface package, the second form of procedure PutAttribute is generated for a given attribute base type *ABT* only if an attribute family is declared in the GDL specification whose attribute type is an attribute subtype of *ABT*.

For a development interface package, the second form of procedure PutAttribute is generated for all attribute base types.

If a GDL specification exclusively contains node kinds that do not have any attributes, then procedure PutAttribute is not generated, since the class contains no attributes and, therefore, no attribute types {§2.3}.

*Exceptions Raised:*

- Constraint_Error (predefined Ada exception)
  - a subtype constraint has been violated for an attribute that is of a user-definable Ada type {§2.4.1}, predefined Ada type {§2.4.2}, or imported Ada type {§2.4.3} (i.e., attribute types other than the GDL-specific types *node kind*, *node group*, and *node sequence* {§2.4.4})
  - TheIndex is less than 1 (one) or greater than the number of family members
  - TheNode is null
- UnexpectedNodeKind
  - a subtype constraint has been violated for an attribute that is a node or node group {§2.4.4}
- UnexpectedNodeSequence
  - a subtype constraint has been violated for an attribute that is a node sequence {§2.4.4}
- UnexpectedAttribute

- TheAttribute is not an attribute of TheNode

- UnknownAttribute
  - TheAttribute is not a valid attribute name (development interface package only) {§3.2.3}

*Examples:*

The following are calls to overloaded procedure PutAttribute of package Example in Figure 3.1, where M and N are variables of type ExampleGraph and I is a variable of subtype Natural:

```
M := Create ( "ConditionClause" );
N := Create ( "IfStatement", 10 );
PutAttribute ( N, "ExecutionCount", I );
PutAttribute ( M, "Weight", BranchWeight'(5) );
PutAttribute ( N, "ElsIfBranch", 3, M );
```

Notice that the second call to PutAttribute requires a qualified expression to disambiguate the use of the literal 5 (see [DoD, 1983], Section 4.7).

*Notes:*

Although the type of parameter TheValue is an attribute base type, the constraints on values implied by attribute subtypes are preserved by interface packages.

The use of overloading underscores the similarities in the functionality of the operations. For instance, the perspective can be taken that there is only one "put" operation on nodes and that this operation works for any attribute. The fact that an interface package must actually provide several subprograms to realize this operation is hidden by the fact that the subprograms are overloaded.

GetAttribute

---

```
function GetAttribute ( TheNode : <class name>; TheAttribute : AttributeName )
                        return <attribute base type>;
function GetAttribute ( TheNode : <class name>; TheAttribute : AttributeName;
                        TheIndex : Positive ) return <attribute base type>;
```

---

For node TheNode, the first form of overloaded function GetAttribute returns the value of attribute TheAttribute, where <*class name*> is the type for designating nodes in the class {§3.2.2} and <*attribute base type*> is the base type of the attribute {§3.2.1}. The second form is for getting the value of member TheIndex of an attribute family {§2.3.2}.

Function GetAttribute is actually overloaded along two dimensions. The first dimension, as described above, includes a pair of functions, one for simple attributes and the other for members of attribute families. The second dimension of overloading is based on attribute base types. In particular, there is one pair of functions in the first dimension for each attribute base type in the class.

For a production interface package, the second form of function GetAttribute is generated for a given attribute base type $ABT$ only if an attribute family is declared in the GDL specification whose attribute type is an attribute subtype of $ABT$.

For a development interface package, the second form of function GetAttribute is generated for all attribute base types.

If a GDL specification exclusively contains node kinds that do not have any attributes, then function GetAttribute is not generated, since the class contains no attributes and, therefore, no attribute types {§2.3}.

*Exceptions Raised:*

- Constraint_Error (predefined Ada exception)
  - TheIndex is less than 1 (one) or greater than the number of family members
  - TheNode is null
- UnexpectedAttribute
  - TheAttribute is not an attribute of TheNode
- UnknownAttribute
  - TheAttribute is not a valid attribute name (development interface package only) {§3.2.3}

*Examples:*

The following are calls to overloaded function GetAttribute of package Example in Figure 3.1, where M and N are variables of type ExampleGraph, I is a variable of subtype Natural, and B is a variable of type BranchWeight:

```
I := GetAttribute ( N, "ExecutionCount" );
M= GetAttribute ( N, "ElsIfBranch", 3 );
B:= GetAttribute ( M, "Weight" );
```

*Notes:*

The use of overloading underscores the similarities in the functionality of the operations. For instance, the perspective can be taken that there is only one "get" operation on nodes and that this operation works for any attribute. The fact that an interface package must actually provide several subprograms to realize this operation is hidden by the fact that the subprograms are overloaded.

### 3.3.3 Subprograms to Manipulate Node Sequences

Node sequences are lists of nodes indexed by a position number of the predefined Ada subtype **Positive** (see [DoD, 1983], Appendix C). A given node may be an element of more than one node sequence.

An object of the type for designating node sequences in a class designates an *empty* sequence if the object does not designate the null node sequence value {§2.4.4} and if a call to the node sequence function **Length** returns the value 0 (zero).

The subprograms in this category are generated by GRAPHITE only if the GDL-specific type constructor *node sequence* is used in the GDL specification {§2.4.4}.

**Create**

---

**function** Create ( TheSequenceKind : NodeSequenceName ) **return** <*class name*>Sequence;

---

Function **Create** creates a new, empty sequence of kind **TheSequenceKind**, where <*class name*>Sequence is the type for designating node sequences in the class {§3.2.2}.

*Exceptions Raised:*

- Storage_Error (predefined Ada exception)
  - storage is not sufficient to accommodate the new node sequence

- UnknownNodeSequence
  - TheSequenceKind is not a valid node sequence name (development interface package only) {§3.2.3}

### Kind

---

**function** Kind ( TheSequence : <*class name*>Sequence ) **return** NodeSequenceName;

---

For node sequence TheSequence, function Kind retrieves the name of the sequence's kind {§3.2.3}, where <*class name*>Sequence is the type for designating node sequences {§3.2.2}.

In a development interface package, the name is returned as a character string in upper case.

*Exceptions Raised:*

- Constraint_Error (predefined Ada exception)
  - TheSequence is null

*Examples:*

The following call to function Kind of package Example in Figure 3.1:

```
Kind ( Create ( "StatementSequence" ) )
```

would yield the character string STATEMENTSEQUENCE. (Function Create is described above.)

### Insert

---

**procedure** Insert ( TheSequence : <*class name*>Sequence; ThePosition : Positive;
TheNode : <*class name*> );

---

Procedure Insert inserts node TheNode into position ThePosition of node sequence TheSequence, where <*class name*>Sequence is the type for designating node sequences {§3.2.2}.

*Exceptions Raised:*

- Constraint_Error (predefined Ada exception)
  - TheSequence is null
  - ThePosition is not in the range of Positive
- NodeSequenceError
  - ThePosition is greater than Length ( TheSequence ) + 1
- Storage_Error (predefined Ada exception)
  - storage is not sufficient to accommodate the new node sequence element
- UnexpectedNodeKind
  - the kind of TheNode violates the subtype constraint for TheSequence {§2.4.4}

*Examples:*

The following series of calls to procedure Insert of package Example in Figure 3.1 operates on a node sequence NS whose length is initially 5 (five); M and N are nodes.

```
Insert ( NS, 1, M );                -- insert at the front
Insert ( NS, 7, N );                -- insert at the rear
Insert ( NS, 3, NullExampleGraph ); -- insert a null node
```

**Length**

---

**function** Length ( TheSequence : <*class name*>Sequence ) **return** Natural;

---

Function **Length** returns the length of node sequence **TheSequence**, where <*class name*>Sequence is the type for designating node sequences {§3.2.2}.

*Exceptions Raised:*

- Constraint_Error (predefined Ada exception)
  - TheSequence is null

*Examples:*

The following call to procedure Insert of package Example in Figure 3.1 makes use of function Length to insert node N at the end of node sequence NS:

```
Insert ( NS, Length ( NS )+1, N );  -- insert at the rear
```

(Function Insert is described above.)

Retrieve

---

```
function Retrieve ( TheSequence : <class name>Sequence; ThePosition : Positive )
                  return <class name>;
```

---

Function Retrieve returns the node at position ThePosition of node sequence TheSequence, where *<class name>*Sequence is the type for designating node sequences {§3.2.2}. The sequence is left unchanged.

*Exceptions Raised:*

- Constraint_Error (predefined Ada exception)
  - TheSequence is null
  - ThePosition is not in the range of Positive
- NodeSequenceError
  - ThePosition is greater than Length ( TheSequence )

*Examples:*

The following series of calls to function Retrieve of package Example in Figure 3.1 operates on a node sequence NS whose length is 7 (seven); N is a node.

```
N := Retrieve ( NS, 1 );                -- retrieve first element
N := Retrieve ( NS, 7 );                -- retrieve last element
N := Retrieve ( NS, Length ( NS ) );    -- retrieve last element
```

(Function Length is described above.)

Remove

---

**procedure** Remove ( TheSequence : <*class name*>Sequence; ThePosition : Positive );

---

Procedure Remove removes the node at position ThePosition from node sequence TheSequence, where <*class name*>Sequence is the type for designating node sequences {§3.2.2}. The removed node is left unchanged.

*Exceptions Raised:*

- Constraint_Error (predefined Ada exception)
  - TheSequence is null
  - ThePosition is not in the range of Positive
- NodeSequenceError
  - ThePosition is greater than Length ( TheSequence )

*Examples:*

The following series of calls to procedure Remove of package Example in Figure 3.1 operates on a node sequence NS whose length is initially 7 (seven).

```
Remove ( NS, 1 );                -- remove first element
Remove ( NS, 3 );                -- remove third element
Remove ( NS, Length ( NS ) );    -- remove last element
```

(Function Length is described above.)

### 3.3.4 Subprograms to Input and Output Graphs

Input and output of graphs is based on *reachability* and on the use of the direct input/output facilities provided by the standard generic Ada package Direct_IO (see [DoD, 1983], Section 14.2.4). In particular, the nodes composing a *persistent* graph (i.e., a graph that can be "saved" on secondary storage) are defined to be all the nodes reachable, through attributes that are nodes, from some root node. The root node is a parameter to the write operation and is returned by the read operation. Any node can serve as a root node, but

only nodes reachable from that node will be saved. A given persistent graph is stored in a secondary-storage file that contains only that graph. The file's name is a parameter to both the read and write operations.

*Notes:*

Currently, the read and write operations place a restriction on the form of graphs that can be saved (i.e., made persistent). In particular, such graphs must be *oriented*. This restriction, however, does not apply to the other operations of interface packages; any form of graph, including disconnected graphs, can be created and manipulated using those other operations.

The use of package Direct_IO places restrictions on the attribute types of nodes that can be saved. These restrictions apply to imported Ada types and are described in Section 2.4.3.

Interface packages do not provide an external-form type corresponding to the type for designating nodes in a class, even though the type for designating nodes is not suitable for reading and writing {§2.4.3}. Therefore, a node containing an attribute whose type is the type for designating nodes in some other class cannot be read and written.[3]

### ReadGraph

---

**procedure** ReadGraph ( FileName : String; TheGraph : **In out** *<class name>* );

---

Procedure **ReadGraph** retrieves a graph from secondary-storage file **FileName**. Upon completion of an invocation of ReadGraph, **TheGraph** designates a root node of the graph, where *<class name>* is the type for designating nodes in the class {§3.2.2}. The root designated by **TheGraph** is the same one that was used to write the graph.

*Exceptions Raised:*

The exceptions raised during execution of **ReadGraph** are dependent upon the Ada compiler and operating system in use.

*Notes:*

The interpretation of the value of **FileName** is dependent upon the Ada compiler and operating system in use.

[3]This restriction will be corrected in a future version of GRAPHITE.

### WriteGraph

---

**procedure** WriteGraph ( FileName : String; TheGraph : **In out** <*class name*> );

---

Procedure **WriteGraph** creates a secondary-storage file whose name is **FileName**, and stores in that file all nodes reachable from the node designated by **TheGraph**, where <*class name*> is the type for designating nodes in the class {§3.2.2}.

*Exceptions Raised:*

The exceptions raised during execution of **WriteGraph** are dependent upon the Ada compiler and operating system in use.

*Notes:*

The interpretation of the value of **FileName** is dependent upon the Ada compiler and operating system in use.

## 3.4 Exceptions

Several exceptions are defined in, and potentially raised by, GRAPHITE-generated interface packages. This section describes those exceptions, indicating which subprograms may raise them, under what conditions they are raised, and the pragmas that can be used to suppress them from being raised.

### AttributeFamilyCountError

Exception **AttributeFamilyCountError** is raised if a list of attribute family counts has too few elements. It is also raised if an individual attribute family count is greater than the maximum allowed number of family members for that attribute.

*Subprogram that raises this exception:*

- Create (node operation)

### NodeSequenceError

Exception NodeSequenceError is raised when a position specification is inappropriate. It is only declared in an interface package if the GDL-specific type constructor *node sequence* is used in the GDL specification {§2.4.4}.

*Subprograms that raise this exception:*

- Insert

  Raised if ThePosition is greater than Length ( TheSequence ) + 1.

- Remove

  Raised if ThePosition is greater than Length ( TheSequence ).

- Retrieve

  Raised if ThePosition is greater than Length ( TheSequence ).

(Function Length is described in Section 3.3.3.)

### UnexpectedAttribute

Exception UnexpectedAttribute is raised when an attempt is made to put or get an attribute value of a node whose kind does not contain that attribute.

*Subprograms that raise this exception:*

- GetAttribute
- PutAttribute

### UnexpectedNodeKind

Given a node $N$ whose kind is $NK$, exception UnexpectedNodeKind is raised when

- an attempt is made give the value $N$ to an attribute whose attribute type is neither $NK$ nor a node group that contains $NK$;

- an attempt is made to insert $N$ into a node sequence whose type is neither $NK$ nor a node group that contains $NK$.

Pragma Suppress ( NodeKind_Check ) can be used to suppress this exception {§4}.

*Subprograms that raise this exception:*

- Insert
- PutAttribute

### UnexpectedNodeSequence

Given a node sequence $S$ whose kind is $NS$, exception UnexpectedNodeSequence is raised when an attempt is made to give the value $S$ to an attribute whose attribute type is not $NS$. It is declared in an interface package only if the GDL-specific type constructor *node sequence* is used in the GDL specification {§2.4.4}.

Pragma Suppress ( NodeSequence_Check ) can be used to suppress this exception {§4}.

*Subprogram that raises this exception:*

- PutAttribute

### UnknownAttribute

Exception UnknownAttribute is raised when a value of type AttributeName does not correspond to a valid attribute name {§3.2.3}. Only subprograms in development interface packages can raise this exception.

Pragma Suppress ( AttributeName_Check ) can be used to suppress this exception {§4}.

*Subprograms that raise this exception:*

- AttributeSubtype
- AttributeBaseType
- GetAttribute
- PutAttribute

UnknownNodeKind

Exception UnknownNodeKind is raised when a value of type NodeKindName does not correspond to a valid node kind name {§3.2.3}. Only subprograms in development interface packages can raise this exception.

Pragma Suppress ( NodeKindName_Check ) can be used to suppress this exception {§4}.

*Subprograms that raise this exception:*

- AttributeSubtype
- AttributeBaseType
- Create (node operation)
- NodeKindAttributes

UnknownNodeSequence

Exception UnknownNodeSequnce is raised when a value of type NodeSequenceName does not correspond to a valid node sequence name {§3.2.3}. It is declared in an interface package only if the GDL-specific type constructor *node sequence* is used in the GDL specification {§2.4.4}. Only subprograms in development interface packages can raise this exception.

Pragma Suppress ( NodeSequenceName_Check ) can be used to suppress this exception {§4}.

*Subprogram that raises this exception:*

- Create (node sequence operation)

## 3.5 Compilation Considerations

The output of the GRAPHITE processor is two Ada compilation units {§3.1}, which require processing by an Ada compilation system before the interface package they embody can actually be used. Under certain circumstances, however, (re)compilation of the specification part of an interface package can be avoided. This can have significant ramifications on development. In particular, the purpose of the development interface package is to facilitate

experimentation by reducing the impact of change {§1}; in essence, this means reducing the amount of compilation and recompilation of a system.

This section details the conditions under which the two compilation units of an interface package need to be compiled, and the possible effects on clients of that interface package. These conditions center on *changes* to the GDL specification from which an interface package is generated (since a GDL specification that results in an entirely new interface package will always require the compilation of both compilation units). It is assumed here that the name of the class, as well as the package name declaration appearing in the GDL specification, do not change. Familiarity with the compilation rules of Ada is also assumed (see [DoD, 1983], Section 10.3).

### 3.5.1 Interface Package Body Part

Any change to a GDL specification (other than pure format changes, such as addition of blank lines or modification of a comment) would require a compilation of the (newly) generated body part of an interface package. This holds for both development and production interface packages.

### 3.5.2 Interface Package Specification Part

The Ada compilation rules state that the compilation of a specification part of a package invalidates any previous compilations of clients of that package, forcing recompilation of those clients. The development and production interface packages differ in the degree to which changes to a GDL specification affect their specification parts, and by extension, the clients of the interface package.

#### Development Interface Package

Either of the following changes to a GDL specification would require a compilation of the (newly) generated specification part of a development interface package and, thus, recompilation of all the interface package's clients:

- addition of an attribute type whose base type is not the base type of any attribute type found in the old GDL specification; or
- deletion of an attribute type whose base type is not the base type of any attribute type remaining in the GDL specification.

*Examples:*

To appreciate some of the flexibility offered by the development interface package, consider the following scenario: System components $C_1$ through $C_n$ manipulate graphs of class ExampleGraph (Figure 2.1). Thus, they all refer to entities declared in package Example (Figure 3.1) and are compiled against the specification part of that package. Suppose that the developer of $C_1$ decides that it is necessary to have a new node kind, called SpecialC1Info, and that nodes of this kind are to be a new attribute, called C1Info, of node kind IfStatement. Suppose further that components $C_2$ through $C_n$ have no use for this new attribute. How extensive will be the effects of this change?

Interface package Example must certainly be reprogrammed to account for the new node kind and attribute. This activity, of course, is automated by GRAPHITE; the developer needs only to alter the existing GDL specification of class ExampleGraph and pass that altered specification through GRAPHITE. Component $C_1$ must also be reprogrammed if it is to make use of the new node kind and attribute. The remaining clients of the interface package, on the other hand, need not be reprogrammed, since they interact with IfStatement nodes, when and if necessary, through subprograms such as GetAttribute and PutAttribute, which allow a client to operate exclusively on the node kinds and attributes of interest. Now, the only difference between the newly generated interface package and the old interface package (assuming that all the attributes of SpecialC1Info were of previously used attribute types) is in their body parts; the specification parts of both packages are identical, since they do not contain any specific information about node kinds and attributes. Therefore, only component $C_1$ and the body part of Example must be recompiled to account for this change in the class definition. Because the specification part of Example is not recompiled, components $C_2$ through $C_n$ do not need to be recompiled.

### Production Interface Package

Any of the following changes to a GDL specification would require a compilation of the (newly) generated specification part of a production interface package and, thus, recompilation of all the clients of the interface package:

- addition or deletion of a node kind or node sequence;
- addition of an attribute whose name is not the same as any attribute in the old GDL specification;
- deletion of an attribute whose name is not the same as any attribute remaining in the new GDL specification;

- addition of an attribute type whose base type is not the base type of any attribute type found in the old GDL specification;
- deletion of an attribute type whose base type is not the base type of any attribute type remaining in the new GDL specification;
- addition of an attribute type whose subtype name is not the subtype name of any attribute type found in the old GDL specification;
- deletion of an attribute type whose subtype name is not the subtype name of any attribute type remaining in the new GDL specification;
- addition of an attribute family if the old GDL specification did not contain any attribute families of the same base type; or
- deletion of an attribute family if the new GDL specification does not contain any other attribute families.

## 3.6 Programming Considerations

### 3.6.1 Using the Development Interface

Using character strings to communicate the names of node kinds, node sequences, attributes, and attribute subtypes might appear to hinder the design of concise and efficient algorithms for clients of development interface packages. In contrast, enumeration literals have the advantage in Ada of being usable in more contexts than are character strings; enumeration types, of which enumeration literals are the values, are discrete and so their uses can include the expressions in case and loop statements, the indices of arrays, and the discriminants of variant records, while strings can be used in none of these contexts. This advantage argues strongly for defining node kind, node sequence, attribute, and attribute subtype names as literals of enumeration types instead of as character strings. Of course, such enumeration types cannot be declared in the specification part of development interface packages (as they are in production interface packages) without then tying those specification parts to particular class definitions.

A compromise scheme can be employed, however, that permits the use of enumeration literals and character strings in the places where they are most appropriate. The scheme involves the use of the predefined Ada function Value (see [DoD, 1983], Section 3.5.5). T'Value, for some discrete type T, converts a character string into a value of type T. For example, given an enumeration type ClassAttributes defined by

```
type ClassAttributes is ( Att1, Att2, Att3, ... )
```

ClassAttributes'Value("Att2") is equivalent to the enumeration literal Att2 of type ClassAttributes. Thus, the function is used in this scheme to permit the names of node kinds, node sequences, and attributes to be treated as enumeration literals *within* interface package clients, for concise and efficient local processing, while being treated as character strings *between* clients and interface packages for communicating names. More specifically, each client could have enumeration types representing the names of just those node kinds, node sequences, attributes, and attribute subtypes in which it was interested. Value would be used to convert a name from a character string to a value of the appropriate local enumeration type.

A predefined Ada function related to Value is Image (see [DoD, 1983], Section 3.5.5). T'Image, for some discrete type T, converts a value of type T into a character string of the predefined Ada type String. Thus, Image is the inverse of Value.

*Notes:*

Both Value and Image use the predefined Ada type String to represent character strings. A value of a type derived from String, such as NodeKindName, NodeSequenceName, AttributeName, and AttributeSubtypeName, can be converted into a value of a type String by an explicit type conversion (see [DoD, 1983], Section 4.6). For example, given the object A of type AttributeName, String ( A ) converts A to a value of type String. Similarly, a value of type String can be converted into a value of a type derived from String by an explicit type conversion. For example, given an object S of type String, AttributeName ( S ) converts S into a value of type AttributeName.

### 3.6.2 Moving From Development to Production

Once a graph class has stabilized to the point where experimentation with its definition and representation is no longer a primary activity, a developer can use GRAPHITE to generate a version of the interface package that is oriented more toward efficiency than flexibility. The design of the production interface package is intended to make this transition as easy as possible. Specifically, it minimizes the amount of reprogramming of interface package clients that would be needed to begin using the optimized interface. Hence, the specification part of a production interface package is very similar to that of a development interface package; graph objects are realized as abstract data types and the same set of subprograms is provided for operating on graphs. In fact, the only significant difference between the development and production interface package that is visible to clients involves the use of enumeration types. The production interface package, because it is not required to be as flexible as the development interface package, uses enumeration literals to communicate the names of node kinds, node sequences, attributes, and attribute subtypes between clients and interface packages {§3.2.3}.

With enumeration literals, the overhead of interpreting character strings to check their legality and to use them for processing is avoided. For instance, once a single, definitive enumeration type is available for each of the various kinds of names communicated between tools and the interface package, there is no longer a need to use the translation scheme employing the Ada function Value described above. Of course, moving from character strings to enumeration literals, since it involves a change in a visible type, does require the reprogramming of clients. The *amount* of that reprogramming, however, can be made almost negligible by the use of an appropriate programming discipline during development. That discipline has two components. First, a client that uses a local enumeration type should isolate the places where it refers to function Value. Second, *string constants* (see [DoD, 1983], sections 3.2.1 and 3.6.3), representing node kind, node sequence, attribute, and attribute subtype names within clients, should be used as parameters in calls to interface package subprograms. For example, a client that refers to node kind NK should, during development, use the following declaration:

```
NK : constant NodeKindName := "NK";
```

This string constant could then be used for all (relevant) invocations of graph operations, such as the following call to create a node of kind NK:

```
X := Create ( NK );
```

Notice that references to node kind, node sequence, attribute, and attribute subtype names through string constants are syntactically identical to references through enumeration literals. Thus, for a client that adheres to this discipline, moving from development to production would only involve the reprogramming necessary to remove calls to function Value and eliminate access to the string-constant declarations, which may simply mean deleting those string-constant declarations or, if the declarations reside in a separate package (perhaps shared by several clients), changing a context clause (see [DoD, 1983, Section 10.1.1).

# Chapter 4

# GDL Pragmas

Pragmas are used in GDL {§2.1.2} to convey information to the GRAPHITE processor in the same way that pragmas are used in Ada to convey information to an Ada compiler (see [DoD, 1983], Section 2.8). This information is used to guide, at least to a certain extent, the code generated for an interface package. This chapter summarizes the pragmas that can appear in a GDL specification.[1]

*Notes:*

An installation of GRAPHITE may also accept the information contained in a pragma through command line arguments; such a facility would be documented in the GRAPHITE User Manual [Tarr, 1987].

### InterfaceVersion

Pragma **InterfaceVersion** is used to instruct the GRAPHITE processor to generate either a development interface package or a production interface package. The single argument to this pragma is either the identifier **Development** or the identifier **Production**.

This pragma can appear anywhere a declarative item of a class declaration can appear {§2.2}.

In the absence of this pragma, a development interface package is generated.

[1]The current version of the GRAPHITE processor does not actually support pragmas.

### BaseTypeName

Pragma BaseTypeName is used to override the default value generated for an element of type AttributeBaseTypeName {§3.2.3}. This pragma takes two arguments. The first is the simple name of an attribute base type and the second is an identifier to be used as the value of the element.

This pragma can appear anywhere that a declarative item of a class declaration can appear {§2.2}, but it must appear after the first use of an attribute subtype of the base type to which it refers.

*Examples:*

For the attribute base type Integer of class ExampleGraph shown in Figure 2.1, the following pragma overrides the default generated value Standard_Integer_AT:

```
pragma BaseTypeName ( Integer, Integer_Type );
```

### SubtypeName

Pragma SubtypeName is used to override the default value generated for an element of type AttributeSubtypeName {§3.2.3}. This pragma takes two arguments. The first is the simple name of an attribute subtype and the second is an identifier to be used as the value of the element.

This pragma can appear anywhere that a declarative item of a class declaration can appear {§2.2}, but it must appear after the first use of the attribute subtype to which it refers.

*Examples:*

For the attribute subtype ConditionClause of class ExampleGraph shown in Figure 2.1, the following pragma overrides the default generated value ConditionClause_AT:

```
pragma SubtypeName ( ConditionClause, CondNode );
```

### MaximumFamilySize

Pragma MaximumFamilySize is used to override the default maximum number of members in an attribute family {§2.3.2}. The single argument to this pragma is a positive integer literal.

This pragma can only appear immediately after the declaration of an attribute family and refers, by context, to that family.

*Examples:*

The following use of pragma MaximumFamilySize specifies that the maximum number of elements in attribute family AF of node kind NK is 100:

```
node NK is
  AF (<>) : Integer;   pragma MaximumFamilySize ( 100 );
end node;
```

### Suppress

Pragma Suppress is used to omit certain run-time checks. The single argument to this pragma is any one of the following identifiers:

- AttributeName_Check

  Indicates checking for the validity of an attribute name {§3.2.3}. This (run-time) check is only meaningful in a development interface package and is ignored during generation of a production interface package.

- NodeKind_Check

  Indicates checking for constraints on attribute types and types of sequences that are node kinds or node groups {§2.4.4}.

- NodeKindName_Check

  Indicates checking for the validity of a node kind name {§3.2.3}. This (run-time) check is only meaningful in a development interface package and is ignored during generation of a production interface package.

- NodeSequence_Check

  Indicates checking for constraints on attribute types that are node sequences {§2.4.4}.

- NodeSequenceName_Check

  Indicates checking for the validity of a node sequence name {§3.2.3}. This (run-time) check is only meaningful in a development interface package and is ignored during generation of a production interface package.

If an error situation arises in the absence of the corresponding run-time check, the execution of the program is erroneous (i.e., the results are not defined).

*Examples:*

```
pragma Suppress ( NodeKind_Checking );
pragma Suppress ( NodeKindName_Checking );
```

# Appendix A

# Reserved Words

This appendix summarizes the reserved words of GDL.

- *All Ada reserved words* (see [DoD, 1983], Section 2.9):

| | | | | |
|---|---|---|---|---|
| **abort** | **declare** | **generic** | **of** | **select** |
| **abs** | **delay** | **goto** | **or** | **separate** |
| **accept** | **delta** | | **others** | **subtype** |
| **access** | **digits** | **if** | **out** | |
| **all** | **do** | **in** | | **task** |
| **and** | | **is** | **package** | **terminate** |
| **array** | | **pragma** | **then** | |
| **at** | **else** | | **private** | **type** |
| | **elsif** | **limited** | **procedure** | |
| | **end** | **loop** | | |
| **begin** | **entry** | | **raise** | **use** |
| **body** | **exception** | | **range** | |
| | **exit** | **mod** | **record** | **when** |
| | | | **rem** | **while** |
| | | **new** | **renames** | **with** |
| **case** | **for** | **not** | **return** | |
| **constant** | **function** | **null** | **reverse** | **xor** |

- *Additional GDL keywords*:

| | | | |
|---|---|---|---|
| **class** | **group** | **node** | **sequence** |

- *Identifiers beginning with the four characters GDL_, except when imported* {§2.4.3}.
- *Identifiers declared as types, constants, subprograms, and exceptions in an interface package, except when imported* {§2.4.3}:

| | | | |
|---|---|---|---|
| <*class name*> | DeleteNode | NodeKindAttributes | UnexpectedAttribute |
| <*class name*>Sequence | | NodeKindName | UnexpectedNodeKind |
| | | NodeSequenceName | UnexpectedNodeSequence |
| | | Null<*class name*> | UnkownAttribute |
| AttributeBaseType | GetAttribute | Null<*class name*>Sequence | UnkownNodeKind |
| AttributeBaseTypeName | | | UnkownNodeSequence |
| AttributeFamilyCountError | | | |
| AttributeFamilyCountList | | PutAttribute | |
| AttributeName | Insert | | |
| AttributeNameList | | | |
| AttributeNamePointer | | ReadGraph | |
| AttributeSubtype | | Remove | |
| AttributeSubtypeName | Kind | Retrieve | |
| | | | |
| Create | Length | SubstituteUnchecked | WriteGraph |

# Appendix B

# GDL Syntax Summary

This appendix presents the syntax of the graph description language, GDL. The notation used to describe the syntax is essentially the same modified Backus-Naur Form used in [DoD, 1983]:

- Lower case words, some containing embedded underlines, are used to denote syntactic categories.
- Boldface words are used to denote reserved words.
- Square brackets enclose optional items.
- Braces enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule.
- A vertical bar separates alternative items unless it occurs immediately after an opening brace, in which case it stands for itself.
- If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part; the italicized part is intended to convey some semantic information.

One additional notational aid has been added here:

- Angle brackets enclose "shuffled" items, which are separated by the wedge symbol ("$\wedge$"). Each of the items appears exactly once, but in any order.

```
graphic_character ::=
    basic_graphic_character | lower_case_letter | other_special_character

basic_graphic_character ::=
        upper_case_letter | digit
    |   special_character | space_character

basic_character ::= basic_graphic_character | format_effector

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

numeric_literal ::= decimal_literal | based_literal

decimal_literal ::= integer [.integer] [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | E - integer

based_literal ::= base # based_integer [.based_integer] # [exponent]

base ::= integer

based_integer ::= extended_digit {[underline] extended_digit}

extended_digit ::= digit | letter

character_literal ::= 'graphic_character'

string_literal ::= "{graphic_character}"

pragma ::= pragma identifier [(argument_association {, argument_association})]

argument_association ::=
        [argument_identifier =>] name
    |   [argument_identifier =>] expression


class_declaration ::=
```

```
    class identifier is
          package name declaration
          {declarative_item}
    end class_simple_name;

package_name_declaration  ::=  package identifier;

declarative_item  ::=
          Ada_type_declaration     |  Ada_constant_declaration
    |     node_kind_declaration    |  group_declaration
    |     sequence_declaration     |  commonly_available_attribute_declaration

identifier_list  ::=  identifier {, identifier_list}

Ada_constant_declaration  ::=  constant_declaration  |  number_declaration

constant_declaration  ::=
          identifier_list : constant subtype_indication := expression;
    |     identifier_list : constant constrained_array_definition := expression;

number_declaration  ::=  identifier_list : constant := universal_static_expression;

Ada_type_declaration  ::=
    full_type_declaration  |  subtype_declaration  |  imported_Ada_type_declaration

full_type_declaration  ::=  type identifier [discriminant_part] is type_definition;

type_definition  ::=
          enumeration_type_definition  |  integer_type_definition
    |     real_type_definition         |  array_type_definition
    |     record_type_definition       |  derived_type_definition

subtype_declaration  ::=  subtype identifier is subtype_indication [:= expression];

subtype_indication  ::=  type_mark [constraint]

type_mark  ::=  type_name  |  subtype_name

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

derived_type_definition  ::=  new subtype_indication

range_constraint  ::=  range range
```

```
range ::=
        range_Ada_attribute
    |   simple_expression .. simple_expression


enumeration type definition ::=
    (enumeration_literal_specification
        {, enumeration_literal_specification}) [:= expression]

enumeration_literal_specification ::= enumeration_literal

enumeration_literal ::= identifier | character_literal

integer_type_definition ::= range_constraint [:= expression]

real type_definition ::= floating_point_constraint | fixed_point_constraint

floating_point_constraint ::=
    floating_accuracy_definition [range_constraint] [:= expression]

floating_accuracy_definition ::= digits static_simple_expression

fixed_point_constraint ::=
    fixed_accuracy_definition [range_constraint] [:= expression]

fixed_accuracy_definition ::= delta static_simple_expression

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 [:= expression]

index_subtype_definition ::= type_mark range <>

index_constraint ::= (discrete_range {, discrete_range})

discrete_range ::= discrete_subtype_indication | range
```

```
record_type_definition  ::=
    record
      component_list
    end record

component_list  ::=
        component_declaration {component_declaration}
    |   {component_declaration} variant_part
    |   null;

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

component_subtype_definition  ::=  subtype_indication

discriminant_part  ::=  (discriminant_specification {; discriminant_specification})

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

discriminant_constraint  ::=  (discriminant_association {, discriminant_association})

discriminant_association  ::=
    [discriminant_simple_name {| discriminant_simple_name} =>] expression

variant_part  ::=
    case discriminant_simple_name is
      variant {variant}
    end case;

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

choice  ::=
        simple_expression          |  discrete_range
    |   component_simple_name      |  others

imported_Ada_type_declaration  ::=  with imported_type_description {, imported_type_description};

imported_type_description  ::=
        package_name.identifier [subtype_clause]
    |   package_name.identifier external_form_clause operation_clauses
    |   package_name.identifier assignment_operation_clause
    |   package_name.( name_shortened_type_description {, name_shortened_type_description} )

name_shortened_type_description  ::=
```

```
        expanded_name [subtype_clause]
    |   expanded_name external_form_clause operation_clauses
    |   expanded_name assignment_operation_clause

package_name  ::=  expanded_name

subtype_clause  ::=  subtype of expanded_name

external_form_clause  ::=  / expanded name

operation_clauses  ::=
    (       internalize_operation_clause
        ∧   externalize_operation_clause
        ∧   [assignment_operation_clause]   )

internalize_operation_clause  ::=  in => expanded_name

externalize_operation_clause  ::=  out => expanded_name

assignment_operation_clause  ::=  := => expanded name   |   := => :=



node_kind_declaration  ::=  full_node_kind_declaration   |   incomplete_node_kind_declaration

incomplete_node_kind_declaration  ::=  node identifier_list;

full_node_kind_declaration  ::=
    node identifier_list is
        attribute_declaration_list
    end node;

attribute_declaration_list  ::=  null;   |   attribute_declaration {attribute_declaration}

attribute_declaration  ::=
        identifier_list [(<>)] : subtype_indication [:= expression];
    |   identifier_list [(<>)] : node [:= expression];
    |   commonly_available_attribute_simple_name [(<>)] [:= expression];

commonly_available_attribute_declaration  ::=
        identifier_list : subtype_indication [:= expression];
    |   identifier_list : node [:= expression];

group_declaration  ::=  full_group_declaration   |   incomplete_group_declaration

incomplete_group_declaration  ::=  group identifier;
```

full_group_declaration ::= **group** identifier **is** ( group_member {, group_member} ) [:= expression];

group_member ::= *node_kind*_simple_name | *group*_simple_name

sequence_declaration ::= **type** identifier **is sequence of** sequence_element;

sequence_element ::= *node_kind*_simple_name | *group*_simple_name | **node**

name ::=
    simple_name | character_literal
  | operator_symbol | indexed_component
  | slice | selected_component
  | Ada_attribute

simple_name ::= identifier

expanded_name ::= identifier {. identifier}

operator_symbol ::= string_literal

prefix ::= name | function_call

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)

indexed_component ::= prefix(expression {, expression})

slice ::= prefix(discrete_range)

selected_component ::= prefix.selector

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

Ada_attribute ::= prefix'Ada_attribute_designator

```
Ada_attribute_designator ::= simple_name [(universal_static_expression)]

aggregate ::= (component_association {, component_association})

component_association ::= [choice {| choice} =>] expression

expression ::=
        relation {and relation}   |  relation {and then relation}
    |   relation {or relation}    |  relation {or else relation}
    |   relation {xor relation}

relation ::=
        simple_expression [relational_operator simple_expression]
    |   simple_expression [not] in range
    |   simple_expression [not] in type_mark

simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

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

primary ::=
        numeric_literal  |  null             |  aggregate
    |   string_literal   |  name             |  allocator
    |   function_call    |  type_conversion  |  qualified_expression
    |   (expression)

logical_operator ::= and  |  or  |  xor

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

binary_adding_operator ::= +  |  -  |  &

unary_adding_operator ::= +  |  -

multiplying_operator ::= *  |  /  |  mod  |  rem

highest_precedence_operator ::= **  |  abs  |  not

type_conversion ::= type_mark(expression)

qualified_expression ::= type_mark'(expression)  |  type_mark'aggregate

allocator ::= new subtype_indication  |  new qualified_expression
```

# Bibliography

[Clarke, Wileden, & Wolf, 1986]

L.A. Clarke, J.C. Wileden, and A.L. Wolf, GRAPHITE: *A Meta-tool for Ada Environment Development*, **Proc. IEEE Computer Society Second Inter. Conf. on Ada Applications and Environments**, Miami Beach, Florida, April 1986, pp. 81–90.

[DoD, 1983]

**Reference Manual for the Ada Programming Language** (ANSI/MIL–STD–1815A), United States Department of Defense, Washington, D.C., January 1983.

[Tarr, 1987]

P.L. Tarr, GRAPHITE *User Manual*, **Arcadia Design Document UM–87–08**, COINS Department, Univ. of Massachusetts, Amherst, Massachusetts, October 1987.

[Wolf, Clarke, & Wileden, 1985]

A.L. Wolf, L.A. Clarke, and J.C. Wileden, *Ada-Based Support for Programming-in-the-Large*, **IEEE Software**, Vol. 2, No. 2, March 1985, pp. 58–71.