# ASIS-for-GNAT User's Guide

**Ada Core Technologies, Inc.**

ASIS-for-GNAT User's Guide



ASIS-for-GNAT User's Guide

# About This Guide

This guide has two aims. The first one is to introduce you to the Ada Semantic Interface Specification (ASIS) and show you how you can build various useful tools on top of ASIS. The second is to describe the ASIS implementation for the GNAT Ada 95 compiler.

## What This Guide Contains

This guide contains the following chapters:

- Chapter 1 [Introduction], page 5, contains the general definition of ASIS and gives some examples of tools which can be built on top of ASIS;
- Chapter 2 [Getting Started], page 7, contains a short guided tour through the development and use of ASIS-for-GNAT-based tools;
- Chapter 3 [ASIS Overview], page 13 gives the general overview of the ASIS definition to help an ASIS newcomer to navigate through ASIS (readers already familiar with ASIS can just skip this section);
- Chapter 4 [ASIS Context], page 19 explains what is the meaning of ASIS Context in the case of ASIS-for-GNAT and what should be done to prepare a set of Ada components to be processed by an ASIS application;
- Chapter 6 [ASIS Application Templates], page 31 describes a set of Ada source components provided by the ASIS-for-GNAT distribution that may be used as a basis for developing simple ASIS applications;
- Chapter 7 [ASIS Tutorials], page 33 describes some examples included in the ASIS-for-GNAT distribution that comprise a hands-on tutorial to get some initial experience with ASIS;
- Chapter 8 [How to Build Efficient ASIS Applications], page 35 describes the problem of tree swapping as a possible source of a poor productivity of ASIS applications and explains how to avoid unnecessary tree swappings;
- Chapter 9 [Processing an Ada Library by an ASIS-Based Tool], page 39 touches the specific issues of processing Ada programs which use the pre-compiled Ada libraries by an ASIS tool;
- Chapter 10 [Compiling Binding and Linking Applications with ASIS-for-GNAT], page 41 explains how to compile an ASIS application with ASIS-for-GNAT and how to create an executable for it;
- Chapter 12 [File Naming Conventions and Applications Name Space], page 45 explains which names can and cannot be used as names of an ASIS application components.

## What You Should Know Before Reading This Guide

This User's Guide assumes that you are familiar with Ada 95 and that you have some basic experience in Ada programming with GNAT.

This User's Guide also assumes that you have ASIS-for-GNAT properly installed for your GNAT compiler, and that you are familiar with the structure of the ASIS-for-GNAT distribution (if not, see the top README file from the distribution, section 2)

This guide does not assume that you have any knowledge or experience in ASIS. If you indeed do not, you will learn some basic things about ASIS when reading this Guide, doing exercises, playing with examples and referring to the ASIS definition if needed.

## Related Information

To get to know more about GNAT, refer to the GNAT User's Guide.

Refer to ASIS-for-GNAT Installation Guide to learn how to install the ASIS implementation for your GNAT compiler.

The ASIS 95 definition now exists as ISO/IEC International Standard 15291.

To get more information about ASIS, visit the ASIS Working Group Web Pages (http://www.acm.org/sigada/wg/asiswg).

To read this Guide you will hardly need GNAT or ASIS-for-GNAT Reference Manual.

# 1 Introduction

## 1.1 What Is ASIS?

The Ada Semantic Interface Specification (ASIS) is an interface between an Ada environment (as defined by ISO/IEC 8652:1995) and any tool requiring information from it. An Ada environment includes valuable semantic and syntactic information. ASIS is an open and published callable interface which gives CASE tool and application developers access to this information. ASIS has been designed to be independent of underlying Ada environment implementations, thus supporting portability of software engineering tools while relieving tool developers from needing to understand the complexities of an Ada environment's proprietary internal representation.

Technically, ASIS is a hierarchy of the Ada package specifications. These packages define a set of Ada private types which implement basic notions needed to describe an Ada program. Operations for these types, called ASIS queries, give you statically determinable information about Ada compilation units in your environment.

You may use ASIS as a third-part Ada library to implement a number of useful program analysis tools.

## 1.2 ASIS Scope: What Kind of Tools Can Be Built with ASIS?

The following ASIS properties define the ASIS scope:

bullet ASIS is a read-only interface;

bullet ASIS provides only statically-determinable information about Ada programs;

bullet ASIS provides all the syntax and the basic semantic information from/about Ada programs. If some semantic property of a program cannot be directly queried by means of ASIS queries, an ASIS application can compute the needed piece of information itself from the information available through ASIS queries;

bullet Even though containing some implementation dependencies, ASIS provides information from/about Ada program in high-level terms which are well-conformed with RM 95 and which are Ada/ASIS-implementation-independent in their very nature.

Examples of tools that benefit from the ASIS interface include, but are not limited by: automated code monitors, browsers, call tree tools, code reformators, coding standards compliance tools, correctness verifiers, debuggers, dependency tree analysis tools, design tools, document generators, metrics tools, quality assessment tools, reverse engineering tools, re-engineering tools, style checkers, test tools, timing estimators, and translators.

# 2 Getting Started

In this section we go through the ASIS application development and usage cycle in a very simplified way: we take a sample problem to be solved with an ASIS application, then we present the code of the ASIS application which gives the solution for our problem, then we show how to compile it and build the executable for it with ASIS-for-GNAT and how to prepare an ASIS Context to be processed by the program, and finally we show the output produced by our program when it is applied to itself.

## 2.1 The Problem

Suppose our goal is to process some set of Ada compilation units, and for every unit to print its full expanded Ada name, whether this unit is a spec, a body or a subunit, and whether this unit is a user-defined unit, a predefined unit as defined by RM 95 or an implementation-specific unit (such as a part of a Run-Time Library).

## 2.2 An ASIS Application Which Solves the Problem

```
with Ada.Wide_Text_IO;        use Ada.Wide_Text_IO;
with Ada.Characters.Handling; use Ada.Characters.Handling;

--  ASIS-specific context clauses:
with Asis;
with Asis.Implementation;
with Asis.Ada_Environments;
with Asis.Compilation_Units;
with Asis.Exceptions;
with Asis.Errors;

procedure Example1 is
   My_Context : Asis.Context;
   --  ASIS Context is an abstraction of an Ada environment, it
   --  defines a set of ASIS Compilation Units available through
   --  ASIS queries

begin
   --  first, by initializing an ASIS implementation, we make it
   --  ready for work
   Asis.Implementation.Initialize;

   --  then we define our Context by making association with
   --  "physical" environment:
   Asis.Ada_Environments.Associate
    (My_Context, "My Asis Context", "-CA");
   --  See ASIS-for-GNAT Reference Manual for the description of the
   --  parameters of the Associate query, see also chapter
   --  "ASIS Context" for the description of different kinds of
   --  ASIS Context in case of ASIS-for-GNAT
```

```
--  by opening a Context we make it ready for processing by ASIS
--  queries
Asis.Ada_Environments.Open (My_Context);

Processing_Units: declare
   Next_Unit : Asis.Compilation_Unit;
   --  ASIS Compilation_Unit is the abstraction to represent Ada
   --  compilation units as described in RM 95

   All_Units : Asis.Compilation_Unit_List :=
   --  ASIS list are one-dimensional unconstrained arrays.
   --  Therefore, when declaring an object of an ASIS list type,
   --  we have to provide either a constraint or explicit
   --  initialization expression:

      Asis.Compilation_Units.Compilation_Units (My_Context);
   --  Compilation_Units query gives you a list of all the units
   --  contained in an ASIS Context
begin
   Put_Line
     ("A Context contains the following compilation units:");
   New_Line;
   for I in All_Units'Range loop
      Next_Unit := All_Units (I);
      Put ("   ");

      --  to get a unit name, we just need a Unit_Full_Name
      --  query. ASIS uses Wide_String as a string type,
      --  therefore we convert the result into String to use
      --  Ada.Text_IO
      Put (Asis.Compilation_Units.Unit_Full_Name (Next_Unit));

      --  to get more info about a unit, we ask about its class
      --  and about its origin

      case Asis.Compilation_Units.Unit_Kind (Next_Unit) is
         when Asis.A_Library_Unit_Body =>
            Put (" (body)");
         when Asis.A_Subunit =>
            Put (" (subunit)");
         when others =>
            Put (" (spec)");
      end case;

      case Asis.Compilation_Units.Unit_Origin (Next_Unit) is
         when Asis.An_Application_Unit =>
            Put_Line (" - user-defined unit");
```

```
            when Asis.An_Implementation_Unit =>
               Put_Line (" - implementation-specific unit");
            when Asis.A_Predefined_Unit =>
               Put_Line (" - Ada predefined unit");
            when Asis.Not_An_Origin =>
               Put_Line
                 (" - unit does not actually exist in a Context");
         end case;

      end loop;
   end Processing_Units;

   --  Cleaning up: we have to close out Context, to break its
   --  association with the external environment and to finalize
   --  our ASIS implementation to release all the resources used:
   Asis.Ada_Environments.Close (My_Context);
   Asis.Ada_Environments.Dissociate (My_Context);
   Asis.Implementation.Finalize;

exception
   when Asis.Exceptions.ASIS_Inappropriate_Context |
        Asis.Exceptions.ASIS_Inappropriate_Compilation_Unit |
        Asis.Exceptions.ASIS_Failed =>

      --  we check not for all the ASIS-defined exceptions, but only
      --  those of them which can actually be raised in our ASIS
      --  application.
      --
      --  If an ASIS exception is raised, we output the ASIS error
      --  status and the ASIS diagnosis string:

      Put_Line ("ASIS exception is raised:");
      Put_Line ("ASIS diagnosis is:");
      Put_Line (Asis.Implementation.Diagnosis);
      Put      ("ASIS error status is: ");
      Put_Line
        (Asis.Errors.Error_Kinds'Wide_Image
           (Asis.Implementation.Status));
end Example1;
```

## 2.3 Required Sequence of Calls

An ASIS application must use the following sequence of calls:

1. `Asis.Implementation.Initialize (...);`

   This call initializes the ASIS implementation internal data structures and prepares the ASIS implementation for work. For most of the ASIS queries, it is erroneous to call them if an ASIS implementation is not initialized.

2. `Asis.Ada_Environments.Associate (...);`

   This call is the only means to define a value of a variable of the ASIS limited private type Context. This value is some specific association of the ASIS Context with the "external world". The way of making this association and the meaning of the corresponding parameters of the Associate query are implementation-specific, but as soon as this association has been made and a Context variable is opened, the ASIS Context designated by this variable may be considered to be a set of ASIS Compilation Units available through the ASIS queries.

3. `Asis.Ada_Environments.Open (...);`

   Opening an ASIS Context variable makes the corresponding Context accessible for all ASIS queries.

   After opening the Context, an ASIS application can start obtaining ASIS Compilation Units from it, further analyze Compilation Units by decomposing them into ASIS Elements etc.

   ASIS relies on the content of a Context being frozen as long as the Context remains open. It is erroneous to change through some non-ASIS program any data structures used by an ASIS implementation to define and implement this Context while the Context is open.

4. `Asis.Ada_Environments.Close (...);`

   After closing the Context it is impossible to retrieve any information from it. All the values of the ASIS objects of Compilation_Unit, Element and Line types obtained when this Context was open become obsolete, and it is erroneous to use them after the Context was closed. The content of this Context need not be frozen while the Context remains closed. Note that a closed Context keeps its association with the "external world" and it may be opened again with the same association. Note also that the content (that is, the corresponding set of ASIS Compilation Units) of the Context may be different from what was in the Context before, because the "external world" may have changed while the Context remained closed.

5. `Asis.Ada_Environments.Dissociate (...);`

   This query breaks the association between the corresponding ASIS Context and the "external world", and the corresponding Context variable becomes undefined.

6. `Asis.Implementation.Finalize (...);`

   This releases all the resources used by an ASIS implementation.

An application can perform these steps in a loop. It may initialize and finalize an ASIS implementation several times, it may associate and dissociate the same Context several times while an ASIS implementation remains initialized, and it may open and close the same Context several times while the Context keeps its association with the "external world".

An application can have several ASIS Contexts opened at a time (the upper limit is implementation-specific), and for each open Context, an application can process several Compilation Units obtained from this Context at a time (the upper limit is also implementation-specific). ASIS-for-GNAT does not impose any special limitations on the number of ASIS Contexts and on the number of the ASIS Compilation Units processed at a time, as long as an ASIS application is within the general resource limitations of the underlying system.

## 2.4 Building the Executable for an ASIS application

The rest of this section assumes that you have ASIS-for-GNAT properly installed as an Ada library.

To get the executable for the ASIS application from subsection 1.2 (assuming that it is located in your current directory as the Ada source file named example1.adb), you have to call gnatmake as:

`gnatmake example1[.adb] -largs -lasis`

For more details concerning compiling ASIS applications and building executables for them with ASIS-for-GNAT see chapter Chapter 10 [Compiling Binding and Linking Applications with ASIS-for-GNAT], page 41.

## 2.5 Generating Trees for Input

To get information from an Ada environment being processed, ASIS-for-GNAT processes so-called tree files. A tree file is generated by GNAT, and it contains a snapshot of a compiler's internal data structures. For more details see section Section 4.1 [ASIS Context and Tree Files], page 19 of this Guide

To create a tree file for a unit contained in some source file, you should compile this file with '-gnatc -gnatt' compiler options. If we want to apply the application described in section Section 2.2 [An ASIS Application Which Solves the Problem], page 7 to itself, we have to compile the source of this application with a command

`gcc -c -gnatc -gnatt example1.adb`

and as a result, we will get the tree file named example1.adt in the current directory.

For more explanation how to generate and how to deal with tree files see chapters Chapter 4 [ASIS Context], page 19 and Chapter 7 [ASIS Tutorials], page 33.

## 2.6 Running an ASIS Application

To complete our example, let's execute our ASIS application. If you have followed all the steps described in chapter Chapter 2 [Getting Started], page 7, now you should have in your current directory the executable `example1` and the tree file `example1.atd`. (Note that a tree file contains information about a unit it was created for and about all the units upon which this unit depends semantically). If we run our application, it will process an ASIS Context defined by one tree file `example1.adt` (for more details about defining an ASIS context see chapter Chapter 4 [ASIS Context], page 19 ASIS-for-GNAT Reference Manual). The result will be:

```
A Context contains the following compilation units:

   Standard (spec) - Ada predefined unit
   Example1 (body) - user-defined unit
```

```
Ada.Text_IO (spec) - Ada predefined unit
Ada (spec) - Ada predefined unit
Ada.IO_Exceptions (spec) - Ada predefined unit
Ada.Streams (spec) - Ada predefined unit
System (spec) - Ada predefined unit
System.File_Control_Block (spec) - Ada predefined unit
System.Parameters (spec) - Ada predefined unit
Ada.Characters.Handling (spec) - Ada predefined unit
Ada.Characters (spec) - Ada predefined unit
Asis (spec) - user-defined unit
A4G.A_Types (spec) - user-defined unit
A4G (spec) - user-defined unit
Ada.Characters.Latin_1 (spec) - Ada predefined unit
A4G.Int_Knds (spec) - user-defined unit
Types (spec) - user-defined unit
Unchecked_Deallocation (spec) - Ada predefined unit
Asis.Implementation (spec) - user-defined unit
Asis.Errors (spec) - user-defined unit
Asis.Ada_Environments (spec) - user-defined unit
Asis.Compilation_Units (spec) - user-defined unit
Asis.Ada_Environments.Containers (spec) - user-defined unit
Asis.Exceptions (spec) - user-defined unit
```

In the current implementation, ASIS implementation components are considered user-defined, not implementation-specific, units. Note also, that some components of the GNAT Run-Time Library may be implicitly "withed" by some Ada units, and therefore they may be presented by a tree file, that is why you can see System.File_Control_Block in the list above.

# 3 ASIS Overview

This chapter contains a short overview of the ASIS definition as given in the ISO/IEC 15291:1999 ASIS Standard. This overview is aimed at helping an ASIS newcomer to find needed information in the ASIS definition and to navigate himself in it.

For more details look into the ASIS definition itself. To get some initial experience with ASIS, go through the ASIS Tutorials (see Chapter 7 [ASIS Tutorials], page 33).

## 3.1 Main ASIS Abstractions

ASIS is based on the three main abstractions used to describe Ada programs:

Context
: an ASIS Context is a logical handle to an Ada environment, as defined in RM 95 Chapter 10. To avoid the language-lawyer difficulties when trying to understand the formal relation between an ASIS Context and an Ada environment, an ASIS application developer may view an ASIS Context as a way to define a set of compilation units available through the ASIS queries.

Compilation Unit
: an ASIS Compilation Unit is a logical handle to an Ada compilation unit. It reflects practically one-to-one all the properties of compilation units defined by RM 95, and it also reflects some properties of "physical objects" treated by an underlying Ada implementation as compilation units (such as time of last update, the name of some object treated as containing the source text for a unit). An ASIS Compilation Unit provides the black-box view of a compilation unit, considering a unit as a whole. It may be decomposed and analyzed as a white-box by means of ASIS Elements.

Element
: an ASIS Element is a logical handle to syntax components of ASIS Compilation Units (both explicit and implicit).

Some ASIS components use additional abstractions needed for specific pieces of functionality provided by these components:

Container
: an ASIS Container (defined and used by the Asis.Ada_Environments.Containers package) provides means for structuring the content of an ASIS Context by grouping ASIS Compilation units into Containers.

Line
: an ASIS Line (defined and used by the Asis.Text package) is the abstraction of a line in an Ada source text. An ASIS Line has a length, a string image and a number.

Span
: an ASIS Span (defined and used by the Asis.Text package) defines the location of an Element, a Compilation Unit or a whole compilation in the corresponding source text.

Id
: An ASIS Id (defined and used by the Asis.Ids package) provides a way to store some "image" of an ASIS Element outside an ASIS application. An application may create an Id value from an Element value and store it in a file. After that, the same or another application may read this Id value in and try to convert it back into the corresponding Element value.

## 3.2 ASIS Package Hierarchy

ASIS is defined as a hierarchy of Ada package specifications. Below is the short description of this hierarchy.

Asis - this is the top package of the hierarchy. It defines the main ASIS abstractions - Context, Compilation_Unit and Element - as Ada private types. It also contains a set of enumeration types that define the classification hierarchy for ASIS Elements (which closely reflects the Ada syntax defined in RM 95) and classification of ASIS Compilation Units. This package does not contain any queries;

Asis.Implementation - contains subprograms that control an ASIS implementation: initializing and finalizing it, retrieving and resetting the diagnosis information. Its child package Asis.Implementation.Permissions contains boolean queries which tells you how ASIS implementation-specific features are implemented in your ASIS implementation;

Asis.Ada_Environments - contains queries that deal with an ASIS Context: associating and dissociating, opening and closing a Context;

Asis.Compilation_Units - contains queries that work with ASIS Compilation Units: obtaining units from a Context, getting semantic dependencies between Units and black-box Unit properties;

Asis.Compilation_Units.Relations - contains queries that return integrated semantic dependencies among ASIS Compilation Units, e.g. all the Units needed by a given Unit to be included in a partition;

Asis.Elements - contains queries working on Elements and implementing general Element properties: gateway queries from ASIS Compilation Units to ASIS Elements, queries defining the position of an Element in the Element classification hierarchy, queries which define for a given Element its Enclosing Compilation Unit and its Enclosing Element. It also contains queries that work on pragmas;

Asis.Declarations, Asis.Definitions, Asis.Statements, Asis.Expressions and ASIS.Clauses - each of these packages contains queries working on Elements of the corresponding kind - that is, representing Ada declarations, definitions, statements, expressions and clauses respectively;

Asis.Text - contains queries returning information about the source representation of ASIS Compilation Units and ASIS Elements;

Asis.Exceptions - defines ASIS exceptions;

Asis.Errors - defines possible ASIS error statuses.

## 3.3 Structural and Semantic Queries

Queries working on Elements and returning Elements or Element Lists are divided into structural and semantic queries.

Each structural query (except Enclosing_Element) implements one step of the parent-to-child decomposition of an Ada program according to the ASIS Element classification hierarchy. Asis.Elements.Enclosing_Element query implements the reverse child-to-parent step. (For implicit Elements obtained as results of semantic queries, Enclosing Element may not correspond to what could be expected from the Ada syntax and semantics as defined in RM 95, in this case the documentation of a semantic query also defines the effect of Enclosing_Element applied to its result).

A semantic query for a given Element returns the Element representing some semantic property of the first - e.g. type declaration for an expression as expression's type, a defining identifier as a definition for a simple name etc.

For example, if we have Element El representing an assignment statement:

```
X := A + B;
```

then we can get the structural components of this assignment statements by applying the appropriate structural queries:

```
El_Var  := Asis.Statements.Assignment_Variable_Name (El); --  X
El_Expr := Asis.Statements.Assignment_Expression    (El); --  A + B
```

And then we can analyze semantic properties of the variable name represented by El_Var and of the expression represented by El_Expr by means of appropriate semantic queries:

```
El_Var_Def   :=
   Asis.Expressions.Corresponding_Name_Definition (El_Var);
El_Expt_Type :=
   Asis.Expressions.Corresponding_Expression_Type (El_Expr);
```

As the result, El_Var_Def will be of A_Defining_Identifier kind and will represent the defining occurrence of X, while El_Expt_Type of a kind An_Ordinary_Type_Declaration will represent the declaration of the type of the expression A + B.

If we apply Asis.Elements.Enclosing_Element to El_Var or to El_Expr, we will get back to the Element representing the assignment statement.

An important thing about classifying queries working on Elements as structural and semantic is that all the structural queries cannot go outside one ASIS Compilation Unit, but for semantic queries it is quite usual that the argument of a query is in one ASIS Compilation Unit, but the result of this query is in another ASIS Compilation Unit.

## 3.4 ASIS Error Handling Policy

Only ASIS-defined exceptions (and the Ada predefined Storage_Error exception) are allowed to propagate outside the ASIS queries. ASIS exceptions are defined in the Asis.Exceptions package.

When an ASIS exception is raised, ASIS sets the Error Status (the possible ASIS error conditions are defined as the values of the Asis.Errors.Error_Kinds type) and forms the Diagnosis string. An application can query the current value of the ASIS Error Status by Asis.Implementation.Status query, and the current content of the Diagnosis string by Asis.Implementation.Diagnosis query. An application can reset the Error Status and Diagnosis by Asis.Implementation.Set_Status procedure.

## 3.5 Dynamic Validity Checking of the ASIS Queries

ASIS has just one type (Element) for all kinds of Ada syntax constructs, and just one type (Compilation_Unit) for all kinds of Ada compilation units. However, many of the queries working on Elements and Compilation Units can be applied only to specific kinds of

Elements and Compilation units respectively. (For example, it does not make sense and is illegal to query Assignment_Variable_Name for an Element of An_Ordinary_Type_Declaration kind).

ASIS is a dynamic validity checking interface. If a query working on Elements has a list of appropriate Element kinds in its documentation, this means that this query can work only on Elements of the kinds from this list. Such a query should raise Asis.Exceptions.ASIS_Inappropriate_Element exception with Asis.Errors.Value_Error error status set when called for any Element with a kind not from the list of the appropriate Element kinds.

If a query working on Compilation Units has a list of appropriate unit kinds in its documentation, then this query can work only on Compilation Units of the kinds from this list. Such a query should raise Asis.Exceptions.ASIS_Inappropriate_Compilation_Unit with Asis.Errors.Value_Error error status set when called for any Compilation_Unit with a kind not from the list of the appropriate unit kinds.

If a query has a list of expected Element kinds or expected Compilation Unit kinds in its documentation, this query does not raise any exception when called with any argument, but it produces a meaningful result only when called with an argument with the kind from this list. For example, if Asis.Elements.Statement_Kind query is called for an argument of A_Declaration kind, it just returns Not_A_Statement, but without raising any exception.

## 3.6 ASIS Iterator

ASIS provides a powerful mechanism to traverse an Ada code, the generic procedure Asis.Iterator.Traverse_Element. This procedure makes top-down left-to-right (or depth-first) traversal of the ASIS tree (that is, of the syntax structure of the Ada code represented by the hierarchy of ASIS Elements). In the course of this traversal, it applies to each Element the formal Pre_Operation procedure when visiting this element for the first time, and the formal Post_Operation procedure when leaving this Element. By providing his own instantiations for Pre_- and Post_Operation, the user gains the ability to automatically process all ASIS Elements found in a given ASIS tree.

For example, suppose we have an assignment statement:

```
X := F (Y);
```

When called for an Element representing this statement, a Traverse_Element instantiation does the following (below Pre_Op and Post_Op stand for actual procedures provided for formal Pre_Operation and Post_Operation, and numbers indicate the sequence of calls to Pre_Op and Post_Op during traversal):

```
                 (1 Pre_Op)  X := F (Y) (10 Post_Op)
                                |
                                |
               -----------------------------------
               |                                 |
    (2 Pre_Op) X (3 Post_Op)                     |
                                                 |
                                   (4 Pre_Op) F(Y) (9 Post_Op)
                                                 |
```

```
                               |
                 ---------------------------
                 |                         |
    (5 Pre_Op)   F (6 Post_Op)  (7 Pre_Op) Y (8 Post_Op)
```

To see in more detail how Traverse_Element may be used for fast-and-easy development of a number of useful ASiS applications, see ASIS tutorials provided as a part of ASIS-for-GNAT distribution (Section 4).

## 3.7 How to Navigate through the Asis Specification

The following hints and tips may be useful when looking for some specific information in the ASIS definition:

1. Use the short overview of the ASIS packages given in section Section 3.2 [ASIS Package Hierarchy], page 14 to limit your browsing to a smaller set of ASIS packages (e.g. if you are interested in what can be done with Compilation_Units - look only in Asis.Compilation_Units, if you are looking for queries which can be used to decompose and analyze declarations, limit your search to Asis.Declarations).
2. Inside Asis packages working with particular kinds of Elements (Asis.Declarations, Asis.Definitions, Asis.Statements, Asis.Expressions and ASIS.Clauses) queries are ordered according to the order of the description of the corresponding constructions in RM 95 (e.g., package Asis.Statements starts from a query retrieving labels and ends with the query decomposing a code statement).
3. The names of all the semantic queries (and only ones) start from Corresponding_... or Implicit_...
4. Use comment sentinels given in the specification of the ASIS packages. A sentinel of the form "–|ER" (from "Element Reference") introduces a new element kind, and it is followed by a group of sentinels of the form "–|CR" (from "Child Reference") which list queries yielding the child Elements for the Element just introduced.

# 4 ASIS Context

## 4.1 ASIS Context and Tree Files

From an ASIS application viewpoint we may view an ASIS Context as a set of ASIS Compilation Units accessible through the ASIS queries. The common ASIS implementation technique is to base an implementation of an ASIS Context on some persistent data structures created by the underlying Ada compiler when compiling Ada compilation units maintained by this compiler. An ASIS Context can only contain compilable (that is, legal) compilation units.

In case of ASIS-for-GNAT, an ASIS implementation is based on tree output files, or, simply, tree files. When called with the special option (-gnatt), GNAT creates and outputs a tree file in case if no error was detected during the compilation. The tree file is a kind of the snapshot of the compiler internal data structures (basically, of the Abstract Syntax Tree (AST)) in the very end of the successful compilation. ASIS then inputs tree files and recreates in its internal data structures just the same picture as the compiler had in the end of the corresponding successful compilation.

An important consequence of the GNAT source-based compilation model is that AST contains full information not only about the unit being compiled, but also about all the units upon which this unit depends semantically. Therefore, having read a tree file, ASIS can in general provide information about more than one unit. By processing a tree file information can be provided about the unit for which this tree was created and about all the units upon which it depends semantically. However, to process several units, ASIS sometimes has to change the tree being processed (in particular, it is the case when an application switches between units which do not semantically depend on each other). Therefore, in the course of an ASIS application, ASIS may read different tree files and it may read the same tree file more then once.

The name of a tree file is obtained from the name of the source file being compiled by replacing its suffix with '`.adt`'. For example, the tree file for `foo.adb` is named `foo.adt`.

## 4.2 Creating Tree Files for Use by ASIS

Neither GNAT nor `gnatmake` will create tree files automatically when you are working with your Ada program. It is the responsibility of a user of an ASIS application to create a set of tree files which would correctly reflect the set of the Ada components to be processed by ASIS/ASIS applications, as well as to maintain the consistency of the trees and the related source files.

To create a tree file for a certain source file, the corresponding source file must be compiled with -gnatc -gnatt flags:

```
gcc -c -gnatc -gnatt foo.adb
```

will produce foo.adt, provided that foo.adb contains the source of a legal Ada compilation unit. -gnatt generates a tree file, and -gnatc turns off tree expansion. ASIS needs tree files created without tree expansion, whereas to create an object file, GNAT needs expanded AST. Therefore it is impossible to produce tree files together with producing object files.

The following things are important to remember when generating and dealing with tree files:

1. ASIS-for-GNAT is distributed for a particular version of GNAT. All the trees to be processed by an ASIS application should be generated by this specific version of the compiler.
2. When creating a tree file for ASIS, use two options -gnatc and -gnatt separately, do not combine them into a single -gnatct or -gnattc option or it will result in a generation of a bogus object file.
3. The tree file is not created if an error has been detected during the compilation.
4. Opposite to object files, a tree file may be generated for any legal Ada compilation unit, including a library package declaration requiring a body and a subunit.
5. A set of tree files processed by ASIS (ASIS application) may be inconsistent, for example two tree files may have been created with the different versions of the source of the same unit. This will lead to inconsistencies in the corresponding ASIS Context. See section Section 4.4 [Consistency Problem], page 23 for more details.
6. Do not move tree, object and source files among directories in the underlying file system! It may confuse ASIS, and it may detect inconsistency between tree and source files when opening a Context or you may get wrong results when asking about the source or object file for a given ASIS Compilation Unit.
7. When calling `gcc` or `gnatmake` to create tree files, all file and directory names containing relative path information should start from "./" or "../" (".\" and "..\" respectively in Windows NT/95). That is, to create a tree file for the source file `foo.adb` located in the inner directory named "inner", you should call gcc as:

   ```
   >gcc -c -gnatc -gnatt .\inner\foo.adb
   ```

   but not as

   ```
   >gcc -c -gnatc -gnatt inner\foo.ads
   ```

   otherwise ASIS will get confused.
8. When reading in a tree file, ASIS checks that this tree file was created with '-gnatc' option, and it does not accept trees created without '-gnatc'.
9. Tree and ALI files. If called to create a tree, GNAT does not destroy an ALI file if the ALI file already exists for the unit being compiled and if this ALI file is up-to-date. Moreover, GNAT may use some information from the existing ALI file to put it into the tree file. So if you would like to have both object and tree files for your program, first create object files and then - tree files.
10. There is only one extension for tree files - .adt, whereas the standard GNAT name convention for the Ada source files uses two different extensions for a spec (.ads) and for a body (.adb). This means that if you first compile the body for the tree:

    ```
    >gcc -c -gnatc -gnatt foo.adb
    ```

    and then - compile the corresponding spec for the tree:

    ```
    >gcc -c -gnatc -gnatt foo.ads
    ```

    then the tree file `foo.adt` will be created twice - first for the body, and then - for the spec, the tree for the spec will override the tree for the body, and the information about the body will be lost for ASIS. If you first create the tree for a spec, and then for a body, the second tree will also override the first one, but no information will be lost for ASIS, because the tree for a body contains full information about the corresponding spec.

To avoid losing information when creating trees for a set of Ada sources, use the following rules:

- if a set of Ada components to process makes up a complete partition, use `gnatmake` (see section Section 8.4 [Using gnatmake to Create Tree Files], page 36 for more details);

- otherwise first create trees for specs, and then - for bodies:

```
>gcc -c -gnatc -gnatt *.ads
>gcc -c -gnatc -gnatt *.adb
```

11. Reading tree files is a time-consuming operation. Try to minimize the number of tree files to be processed by your application and to avoid unnecessary tree swappings. (See chapter Chapter 8 [How to Build Efficient ASIS Applications], page 35 for some tips).

Note that between opening and closing a Context, an ASIS application should not change its working directory (or restore it before making an ASIS call), otherwise application behavior may be erroneous.

### 4.2.1 Creating Trees for Data Decomposition Annex

Using the ASIS Data Decomposition Annex (DDA) does not require anything special to be done by an ASIS user, except one thing. The implementation of the ASIS DDA is based on some special annotation added by the compiler to the trees used by ASIS. An ASIS user should be aware of the fact, that trees created for subunits does not have this special annotation, therefore ASIS DDA queries do not work correctly on trees created for subunits (and these queries may not work correctly if a set of tree files making up a Context contain a tree created for a subunit).

So, when working with ASIS DDA, a user should avoid creating separate trees for subunits. Actually, it is not a limitation - to create a tree for a subunit, a user should also have the source of the parent body around. If in this situation a user creates the tree for the parent body, it will contain the full information (including DDA-specific annotation) for all the subunits which are around. From the other side, a tree created for a single subunit has to contain information about the parent body, so it is about of the same size as the tree for the parent body.

The best way to create trees when using ASIS DDA is to use gnatmake - it will never create separate trees for subunits.

## 4.3 Different Ways to Define an ASIS Context in ASIS-for-GNAT

The Asis.Ada_Environments.Associate query which defines a Context has the following profile:

```
procedure Associate
           (The_Context : in out Asis.Context;
```

```
                      Name        : in Wide_String;
                      Parameters  : in Wide_String := Default_Parameters);
```

In ASIS-for-GNAT Name does not have any special meaning, and all the properties of a Context being associated are set by the Parameters string.

When making an association of an ASIS Context in ASIS-for-GNAT, you may specify the following things in the Parameters string of the Asis.Ada_Environments.Associate query:

- the way of defining a set of tree files making up the Context (-C options);
- the way of dealing with tree files when opening the Context and when processing ASIS queries (-F options);
- the way of processing the source files during the consistency check when opening the Context (-S options):
- the search path for tree files making up the Context (-T options);
- the search path for source files used for calling GNAT to create a tree file on the fly (-I options);

Also the association parameters may (and in some cases - have to) contain the names of tree files or directories making up search paths for tree and/or source files. Below is the overview of the Context association parameters in ASIS-for-GNAT, for full details refer to the ASIS-for-GNAT Reference Manual.

Note that the set of options for the Context association is not frozen, we are open for discussing ASIS application developers' needs, and we can change or extend an existing set of options in future.

The way to define a set of tree files making up a Context; the following options are possible:

-C1 - "one tree" Context, defines a Context made up by a single tree file, this tree file name should be given explicitly in the Parameters string

-CN - "N-trees" Context, defines a Context made up by a set of tree files, the names of the tree files making up the Context should be given explicitly in the Parameters string

-CP - "partition" Context, this option is not implemented yet. The idea is to define a Context representing a complete partition, as defined in RM 95, 10.2;

-CA - "all trees" Context, defines a Context made up by all the tree files in the tree search path given in the same Parameters string, if this option is set together with -FM option, ASIS can also create new tree files on the fly when processing queries yielding ASIS Compilation units.

The way of dealing with tree files when opening the Context and when processing ASIS queries; the following options are possible:

-FS - all the trees considered as making up a given Context are created on the fly, whether or not the corresponding tree file already exists; once created, a tree file may then be reused while the Context remains open. This option can be set only with -CA option;

-FT - only pre-created trees are used, no tree file can be created by ASIS;

-FM - mixed approach: if a needed tree does not exist, the attempt to create it on the fly is made. This option can only be set with -CA option.

The way of processing the source files during the consistency check when opening the Context; the following options are possible:

-SA - source files for all the Compilation Units belonging to the Context (except the predefined Standard package) are taken into account for consistency check when opening the Context (see 3.4 concerning the consistency problem in ASIS-for-GNAT);

-SE - only existing source files for all the Compilation Units belonging to the Context are taken into account for consistency check when opening the Context (see 3.4 concerning the consistency problem in ASIS-for-GNAT);

-SN - none of the source files from the underlying file system are taken into account when checking the consistency of the set of tree files making up a Context.

The default options are -CA, -FT and -SA.

Note, that for -C1 Context, a parameter string should contain exactly one name of a tree file. Moreover, for such a Context if during the opening of the Context this tree file could not be successfully read in because of any reason, Asis_Failed is raised.

Using -I option for defining an ASIS Context is similar to using -I option when calling GNAT, -T option is used in the same way, but for tree files, for full details concerning using -T and -I options refer to the ASIS-for-GNAT Reference Manual. Note, that -T option is used only to locate existing tree files, and it has no effect for -FS Contexts. On the other side, -I option is used only to construct a set of arguments when ASIS calls GNAT to create a tree file "on the fly", it has no effect for -FT Contexts, and it cannot be used to tell ASIS where it should look for source files for ASIS Compilation Units.

## 4.4 Consistency Problem

There are two different kinds of consistency problems existing for ASIS-for-GNAT, and both of them can show up when opening an ASIS Context.

First, it may be a tree file created by another version of GNAT (see the top README file about the coordination between the GNAT and ASIS-for-GNAT versions). This means that there is an ASIS-for-GNAT installation problem.

Second, it may be that the tree files are inconsistent with the existing source files or with each other.

### 4.4.1 Inconsistent versions of ASIS and GNAT

When ASIS-for-GNAT reads a tree file created by the version of the compiler for which a given version of ASIS-for-GNAT is not supposed to be used, ASIS treats the situation as the ASIS-for-GNAT installation problem and raises PROGRAM_ERROR with the corresponding exception message. In this case, PROGRAM_ERROR is not caught by any ASIS query and propagates outside ASIS. Note that this is not a violation of the requirement stated in the ASIS definition that only ASIS-defined exceptions are allowed to propagate outside ASIS queries, because in this case you do not have ASIS-for-GNAT properly installed and therefore you do not have a valid ASIS implementation. Note also that the real cause may be some old tree file you have forgotten to remove when reinstalling ASIS-for-GNAT. This is also considered an installation error.

Be careful when using "when others" exception handler in your ASIS application: do not use it just to catch non-ASIS exceptions and to suppress them without any analysis.

### 4.4.2 Consistency of a set of tree and source files

When processing a set of more then one tree file making up the same Context, ASIS may face a consistency problem. A set of tree files is inconsistent if it contains two trees representing the same compilation unit and these trees were created with different versions of the source of this unit. A tree file is inconsistent with a source of a unit represented by this tree if the source file currently available for the unit differs from the source used to create the tree file.

When opening a Context (Asis.Ada_Environmens.Open query), ASIS does the following checks for all the tree files making up the Context:

- if -SA option is set for the Context, ASIS checks that for every Compilation Unit represented by a tree, the source file is available and it is the same as the source file used to create the tree (a tree file contains the reference for all the source files used to create this tree file;
- if -SE option is set for the Context, then if for a Compilation Unit represented by a tree a source file is available, ASIS checks that this source is the same as the source used to create the tree. If for a Compilation Unit belonging to a Context a source file is not available, ASIS checks that all the tree files containing this unit were created with the same version of the source of this unit.
- if -SN option is set for the Context, ASIS checks that all the trees were created from the same versions of the sources involved.

If any of these checks fail, Asis_Failed is raised as a result of opening a Context. If the Context has been successfully opened, it ensures that ASIS will process only consistent set of tree and object files until the Context is closed (provided that this set will not be changed by some non-ASIS actions).

## 4.5 Processing Several Contexts at a Time

If your application processes more then one open Context at a time, and if at least one of the Contexts is defined with -FS or -FM option, be aware of the fact that all the tree files created by ASIS on the fly are placed in the current directory. Therefore, to be on the safe side when processing several opened Contexts at a time, an ASIS application should have at most one Context defined with -FS or -FM option. If it has such a Context, all the other Context should not use tree files located in the current directory.

# 5 ASIS Interpreter asistant

## 5.1 asistant introduction

asistant is an interactive interface to ASIS queries. It allows a user to play around with ASIS without building his own ASIS applications. It provides a simple command language which allows to define variables of ASIS types and to assign them values by calling ASIS queries.

asistant may be very useful during learning ASIS: it allows to try different ASIS queries and to see immediately what are the results. asistant does not crash in case of any error in calling ASIS queries (such as calling a query for an inappropriate Element) - instead it reports an error and gives a user the possibility to try again.

asistant may also be useful as a debug and "ASIS visualization" tool in a real-life ASIS application project: if an ASIS programmer has some problems in finding out which query should be used in a given situation or why a given query does not work correctly with a given piece of Ada code, he may use asistant to reconstruct the situation which causes problems in his ASIS application and to do some experiments with ASIS queries.

Though primarily an interactive tool, asistant also can interpret sequences of asistant commands written to a file (called a script file below). asistant can also store in a file the log of an interactive section which can then be reused as a script file.

The full documentation of asistant may be found in the asistant Users' Guide (file asistant.ug in the asistant source directory). Here only a very short overview of asistant usage is presented.

The executable for asistant is created in the asistant source directory as a part of the standard procedure of installing ASIS-for-GNAT as an Ada library. Put this executable somewhere on your path, and then type 'asistant' to call asistant in an interactive mode. As a result, the program will output a brief information about itself and then the asistant prompt '>' will appear:

```
ASIStant - ASIS Tester And iNTerpreter, v1.2
(C) 1997-1999, Free Software Foundation, Inc.
  Asis Version: ASIS 2.0.R

>
```

Now a user can input asistant commands (asistant supports in its command language the same form of comments as Ada does, names in asistant are not case-sensitive):

```
>Initialize ("") -- the ASIS Initialize query is called with an
                 -- empty string as a parameter

>set (Cont) --  the non-initialized variable Cont of the ASIS
            --  Context type is created

>Associate (Cont, "", "") --  the ASIS Associate query with two empty
                          --  strings as parameters is called for Cont
```

```
>Open (Cont)  --  the ASIS Open query is called for Cont

>set (C_U, Compilation_Unit_Body ("Test", Cont)) -- the variable C_U
  --  of the ASIS Compilation_Unit type is created and initialized by
  --  the result of the call to the ASIS query Compilation_Unit_Body.
  --  As a result, C_U will represent an compilation unit named "Test"
  --  and contained in the ASIS Context named Cont

>set (Unit, Unit_Declaration (C_U))  --  the variable Unit of the ASIS
  --  Element type is created and initialized by the result of calling
  --  the ASIS Unit_Declaration query

>print (Unit) --  as a result of this command, the ASIS debug image of
              --  the current value of Unit will be printed:

Element Debug_Image:
A_PROCEDURE_BODY_DECLARATION
located in Test (body, Unit_Id = 2, Context_Id = 1)
text position : 1 : 1 - 9 : 7
   Nodes:
      Node            : 1363 - N_SUBPROGRAM_BODY
      R_Node          : 1363 - N_SUBPROGRAM_BODY
      Node_Field_1    : 0 - N_EMPTY
   Rel_Sloc           : -10
   obtained from the tree .\test.atb (Tree_Id = 1)

--  suppose now, that we do make an error - we call an ASIS query for
--  inappropriated element:

>set (Elem, Assignment_Expression (Unit))

--  ASIS will raise an exception, asistant will output the ASIS debug
--  information:

Exception is raised by ASIS query ASSIGNMENT_EXPRESSION.
Status : VALUE_ERROR
Diagnosis :
Inappropriate Element Kind in Asis.Statements.Assignment_Expression

--  it does not change any of the existing variables and it prompts
--  a user again:

> ...
```

## 5.2 `asistant` commands

The list of the `asistant` commands given in this section is incomplete and it is not supposed to be used as a reference manual for these commands. Its purpose is only to give some general feeling of what can be done with asistant:

Help [(name)] - outputs the profile of the ASIS query 'name', when calling with no argument, generates a general asistant help;

Set (name) - creates a (non-initialized) variable 'name' of the ASIS Context type;

Set (name, expr) - evaluates the expression 'expr' (it may be any legal asistant expression, a call to some ASIS query is the most common case in practice) and creates the variable 'name' of the type and with the value of 'expr';

Print (expr) - evaluate the expression 'expr' and outputs its value;

Run ("filename") - launches the script from a file "filename", reading further commands from it;

Pause - paused the current script and turns asistant into interactive mode;

Run - resumes a previously paused script;

Browse - switches asistant into step-by-step ASIS tree browsing;

Log ("filename") - opens a file "filename" for session logging

Log - closes the current log file

Quit [(exit-status)] - quits asistant

## 5.3 `asistant` variables

`asistant` variables have Ada-style (simple) names. Variables can be of any ASIS type and of conventional integer, boolean and string type. All the variables are created and assigned dynamically by the asistant Set command, there is no predefined variables.

There is no type checking in asistant: each call to a Set command may be considered as creating the first argument from scratch and initializing it by the value provided by the second argument.

## 5.4 Browsing an ASIS tree

Browser is invoked by calling the asistant service function BROWSE. BROWSE disables the asistant command interpreter and enables the command interpreter of Browser. The Browser 'Q' command switches back into the asistant environment by enabling asistant command interpreter and disabling the Browser interpreter.

BROWSE has a single parameter of Element type, and it starts browsing the ASIS tree starting from its argument Element. BROWSE returns the result of Element type, an Element on which the process of tree browsing was stopped. So, if a user types"

```
> set (e0, Browse (e1))
```

he will start ASIS tree browsing from e1, and when he finishes the browsing, e0 will represent the last Element being visited during the browsing.

If a user types

```
> Browse (e1)
```

he will be able to browse the ASIS tree, but the last element of the browsing will be discarded.

Browser displays the ASIS Element it currently points at and expects one of the following keystrokes:

U - one step up the ASIS tree (equivalent to calling the ASIS Enclosing_Element query);

D - one step down the ASIS tree, to the left-most component of the current Element

N - go to the right sibling (to the next element in the ASIS tree hierarchy)

P - go to the left sibling (to the previous element in the ASIS tree hierarchy)

\(D|d)(T|t) - change the form of displaying the current Element: 'D' turns ON displaying the debug image, 'd' turns it OFF. 'T' turns ON displaying the text image, 't' turns it OFF.

<SPACE><query> - call the <query> for the current Element (see 5.4.);

Q - back to the asistant environment, the Browser command interpreter is disabled and the asistant command interpreter is enabled with the current Element returned as a result of the call to BROWSE;

Browser immediately interprets the keystroke and displays the new current Element. If the message "Cannot go in this direction." appears, this means that traversal in this direction from current node is impossible (that is, the current node is either a terminal Element and it is not possible to go down, or it is the leftmost or the rightmost component of some element, and it is not possible to go left or right, or it is the top Element in its enclosing unit structure and it is not possible to go up).

It is possible to issue some ordinary ASIS queries from inside the Browser (for example, semantic queries). The legal queries are those that accept one parameter of type Element and return Element as a result.

When the user presses <SPACE>, he is asked to enter the query name. If the query is legal, the current Element is replaced by the result of the call to the given query with the current Element as a parameter.

## 5.5 Example

Suppose we have an ASIS compilation unit Demo in the source file demo.adb:

```
procedure Demo is
   function F (I : Integer) return Integer;

   function F (I : Integer) return Integer is
   begin
      return (I + 1);
   end F;

   N : Integer;

begin
 N := F (3);
```

```
    end Demo;
```

And suppose that the tree for this source is created in the current directory. Below is a sequence of asistant commands which does some work with this unit. asistant comments are used to explain what is doing:

```
initialize ("")

--  creating and opening a Context made up by all the tree files
--  in the current directory;

Set (Cont)
Associate (Cont, "", "")
Open (Cont)

-- getting a Compilation_Unit (body) named "Demo" from this Context;
Set (CU, Compilation_Unit_Body ("Demo", Cont))

--  going into the unit structure and getting to the expression
--  in the right part of the assignment statements in the unit body:
Set (Unit, Unit_Declaration (CU))
Set (Stmts, Body_Statements (Unit, False))
Set (Stmt, Stmts (1))
Set (Expr, Assignment_Expression (Stmt))

- outputting the debug image and the text image of this expression:
Print (Expr)
Print (Element_Image (Expr))

--  this expression is of A_Function_Call kind, so it's possible to ask
--  for the declaration of the called function:
Set (Corr_Caled_Fun, Corresponding_Called_Function (Expr))

--  the debug and the text image of the declaration of the called
--  function is printed:
Print (Corr_Caled_Fun)
Print (Element_Image (Corr_Caled_Fun))

-- the asistant session is closed:
Quit
```

# 6 ASIS Application Templates

The subdirectory 'templates' of the ASIS distribution contains a set of Ada source components that can be used as templates for developing simple ASIS applications. The general idea is that one can easily build an ASIS application by adding the code performing some specific ASIS analysis in well-defined places in these templates.

See the solutions provided for ASIS tutorial as the examples of the use of the templates.

For more information see the README file in the 'templates' subdirectory.

# 7 ASIS Tutorials

The subdirectory 'tutorial' of the ASIS distribution contains a simple hands-on ASIS tutorial which may be useful in getting the quick start with ASIS. The tutorial contains a set of simple tasks based on the asistant tool and on a set of the ASIS Application Templates provided as a part of the ASIS distribution. The complete solutions are provided for all the tasks, so the tutorial may also be considered as a set of ASIS examples.

At the moment the documentation of the tutorial exists as a set of README files in the 'tutorial' subdirectory and its subdirectories. This documentation will be moved into this Guide soon.

# 8 How to Build Efficient ASIS Applications

## 8.1 Tree Swapping as a Possible Cause of Poor Application Performance

If an ASIS Context is made up by more then one tree, then ASIS may switch between different trees during an ASIS application run. Switching between trees means that ASIS reads trees over and over again, and this may slow down an application considerably.

Basically, there are two causes for tree swapping:

1. Processing of semantically independent units. Suppose in Context Cont we have units P and Q which do not depend on each other, and Cont does not contain any third unit depending on both P and Q. This means, that P and Q cannot be represented by the same tree. To get some information about P, ASIS needs tree p.adt to be accessed, and to get some information about Q, ASIS needs `q.adt`. Therefore, if an applications retrieves some information from P, and then starts processing of Q, ASIS has to read `q.adt`.
2. The possibility for the same unit to be presented in more then one tree. A unit may be presented by the tree created for itself, and it also is presented by all the trees created for unit which semantically depend upon a given unit. Suppose we have a library procedure Proc depending on a library package Pack, and in the set of trees making up our Context we have trees `pack.adt` and `proc.adt`. Suppose we have got some Element representing some component of Pack, when `pack.adt` was accessed by ASIS, and suppose that because of some other actions undertaken by an application ASIS changed the tree being accessed to `proc.adt`. Suppose that now the application wants to do something with the Element representing some component of Pack and obtained from `pack.adt`. Even though the unit Pack is represented by the currently accessed tree proc.adt, ASIS has to switch back to `pack.adt`, because all the references into the tree structure kept as a part of the value of this Element are valid only for `pack.adt`.

## 8.2 Queries That Can Cause Tree Swapping

In ASIS-for-GNAT, tree swapping can currently take place only when processing queries defined in:

```
Asis.Elements
Asis.Declarations
Asis.Definitions
Asis.Statements
Asis.Clauses
Asis.Expressions
Asis.Text
```

except the queries that return enumeration or boolean results. For any instantiation of Asis.Iterator.Traverse_Element, the traversal itself can cause at most one tree read to get the tree appropriate for processing the Element to be traversed, but procedures provided as actuals for Pre_Operation and Post_Operation may cause additional tree swappings.

## 8.3 How to Avoid Unnecessary Tree Swapping

To speed up your application, try to avoid unnecessary tree swapping. The following advices may help you in this:

1. Try to minimize a set of tree files processed by your application. In particular, try to avoid having separate trees created for subunits.

   Minimizing of a set of tree files processed by the application also cuts down the time needed for opening a Context. Try to use gnatmake to create a suitable set of tree files covering an Ada program for processing by an ASIS application.
2. Choose the right way of Context definition for your application. For example, use "one tree" Context (-C1) for applications that are limited to processing single units (such as a pretty printer or gnatstub). By processing the tree file created for this unit, ASIS can get all the syntax and semantic information about this unit. Using "one tree" Context definition, an application has only one tree file to read during opening a Context, and no other tree file will be read during the application run. A "N-trees" Context is a natural extension of "one tree" Context for applications which know in advance what units shall be processed, but opening a Context becomes longer, and ASIS may switch among different tree files during an application run. Use "all trees" Context only for applications which are not targeted at processing a specific unit or a specific set of units, but are supposed to process all the available units, or in case when an application has to process a big system consisting of a large number of units. When using an application based on "all trees" context, use the approach for creating tree files described above to minimize a set of tree files to be processed.
3. In your application, try to avoid switching between processing units or sets of units with no dependencies among them - such a switching will certainly cause tree swapping.
4. If you are going to analyze some library unit having both spec and body, start from obtaining an Element from the body of this unit. This will set the tree created for the body as the tree accessed by ASIS, and this tree will be enough for processing both the spec and the body of this unit without tree swapping.
5. To see "tree swapping profile" of your application use -dt debug flag when initializing ASIS ( Asis.Implementation.Initialize ("-dt") ). The information you will get from the application run may give you some hints how to avoid tree swapping.

## 8.4 Using `gnatmake` to Create Tree Files

To create a suitable set of tree files, you may use `gnatmake`. GNAT creates the ALI files for every successful compilation, whether or not the code has been generated. Therefore, it is possible to run `gnatmake` with -gnatc and -gnatt parameters, and this will create the set of tree files representing all the compilation units needed by a unit to which `gnatmake` is applied to be included in a partition. Below we will use `gnatmake` to create a set of tree files for a complete Ada program (partition). You may adapt this approach to an incomplete program or to a partition without a main subprogram, applying gnatmake to some of its components.

Using `gnatmake` for creating tree files has another advantage - this will keep tree files consistent among themselves and with the sources.

There are two different ways to use `gnatmake` to create a set of tree files.

First, suppose you have object, ALI and tree files for your program in the same directory, and `main_subprogram.adb` contains the body of the main subprogram. If you run `gnatmake` as

```
gnatmake -f -c ... main_subprogram.adb -cargs -gnatc -gnatt
```

or simply as

```
gnatmake -f -c -gnatc -gnatt ... main_subprogram.adb
```

this will create the trees representing the full program for which main_subprogram is the main procedure. The trees will be created from scratch, that is, if some tree files already exist, they will be recreated. This is because gnatmake is called with -f option (which means "force recompilation"). Usng gnatmake without -f option for creating tree files is not reliable if your tree files are in the same directory with object files, because object and tree files "share" the same set of ALI files, and in case of object file existing and being consistent with the ALI and source files, the source will not be recompiled for creating a tree file if -f option was not set.

A different approach is to keep the tree files and the associated ALI files in a separate directory, and to use this directory only for keeping the tree files and maintaining their consistency with source files (that is, object files and ALI files corresponding to them should be in another directory). In this case, by calling gnatmake as

```
gnatmake -c ... main_subprogram.adb -cargs -gnatc -gnatt
```

or simply as

```
gnatmake -c -gnatc -gnatt ... main_subprogram.adb
```

(that is, without forcing recompilation) you will still get the full and consistent set of tree files representing your programs, but in this case the existing tree files will be reused.

See the next section for specific details related to Ada compilation units belonging to precompiled Ada libraries.

# 9 Processing an Ada Library by an ASIS-Based Tool

In the cases when an Ada program to be processed by some ASIS-based tool makes use of some Ada library, it is necessary to be aware of the following features of using Ada libraries in case of GNAT:

- an Ada library is a collection of precompiled Ada components. The sources of the Ada components belonging to the library are also presented as a part of a library, but if a user program uses some components from a library, these components are not recompiled when calling gnatmake (in a usual way) for this program (for example, you never recompile Ada.Text_IO when you call gnatmake for any program which uses Ada.Text_IO;
- according to the GNAT source-based compilation model, specs of library components are processed when a user unit which uses these components is compiled, but bodies of library components are not compiled. As a result, if you call gnatmake to create a set of tree files covering a given program, and if this program uses something from some Ada library, then the set of tree files created by such a call will contain only specs, but not bodies for library components;
- any GNAT installation contains the GNAT Run-Time Library (RTL) as a precompiled Ada library. In some cases, a GNAT installation may contain some other libraries (such as Win32Ada Binding in case of Windows95/NT GNAT porting);
- in ASIS-for-GNAT, there is no reliable way to define whether or not a given Compilation Unit belongs to some precompiled Ada library other then GNAT RTL (some euristics may be added to Asis.Extensions). ASIS-for-GNAT classifies (by means of Asis.Compilation_Units.Unit_Origin query) a unit as A_Predefined_Unit, if it is from RTL and if it is mentioned in RM95 A(2) as an Ada 95 predefined unit, and a unit is classified as An_Implementation_Unit if is belongs to RTL, but is not mentioned in RM 95 A(2). Components of Ada libraries other then RTL are always classified as An_Application_Unit;
- there is a possibility to recompile the components of the Ada libraries used by a given program. To do this, you have to call gnatmake for this program with '-a' gnatmake option. Therefore, if you create a set of tree files for you program by calling gnatmake with '-a' option, the resulting set of tree files will contain all the units needed by this program to make up a complete partition.

Therefore, there are two possibilities for ASIS-based tools and their users in case if processing (or avoiding processing) of Ada libraries is important for the functionality of the tool:

1. If the tool does not want to process components of Ada libraries, then a set of tree files for this tool may be created by calling gnatmake without '-a' option (this is the usual way of using gnatmake). When the tool faces a Compilation_Which which represents a spec of some library unit, and for which Asis.Compilation_Units.Is_Body_Required gives True, but Asis.Compilation_Units. Corresponding_Body yields a result of A_Nonexistent_Body kind, then the tool may conclude that this library unit belongs to some precompiled Ada library;
2. If a tool wants to process all the Ada compilation units making up a program, then a set of tree files for this program should be created by calling gnatmake with '-a' option;

3. Asis.Compilation_units.Unit_Origin may be used to filter out RTL components.

# 10 Compiling, Binding and Linking Applications with ASIS-for-GNAT

If you have installed ASIS-for-GNAT as an Ada library and added the directory containing all source, ALI and library files of this library to the values of the ADA_INCLUDE_PATH and ADA_OBJECTS_PATH environment variables (which is a recommended way to install ASIS-for-GNAT), you do not need any ASIS-specific options for the GNAT compiler (that is, for gcc calls) and for gnatbind when working with your ASIS applications. However for gnatlink you have to provide an additional parameter "-lasis":

```
gnatlink my_application -lasis
```

When using `gnatmake`, you also have to provide this linker parameter whenever a call to `gnatmake` invokes `gnatlink`:

```
gnatmake ... my_application -largs -lasis
```

You do not need these linker parameters if a call to gnatmake is not creating the executable:

```
gnatmake -c ... my_application
```

If you have installed ASIS-for-GNAT without building an ASIS library, then you have to do the following when working your ASIS application code:

- when compiling, you have to put catalogs with ASIS-for-GNAT implementation sources (asis-[version#]-src/asis and asis-[version#]-src/gnat) in the search path for the source files; you may do it either by -I gcc options or by adding these directories in the ADA_INCLUDE_PATH environment variable;
- when binding, you have to put the directory where all the object and ALI files for the ASIS-for-GNAT components were created (asis-[version#]-src/obj, if you followed the manual installation procedure described in README/ASIS Installation Guide) in the search path for gnatbind, you can do it either by -aO gnatbind option or by adding this directory in the ADA_OBJECTS_PATH environment variable;

If you have added directories with ASIS-for-GNAT source, object and ALI files to the values of the GNAT-specific environment variables, you do not have to provide any ASIS-specific parameter when using gnatmake for your ASIS application.

# 11 ASIS-for-GNAT Warnings

The ASIS definition specifies the situations when a certain ASIS-defined exception should be raised, and ASIS-for-GNAT follows these rules.

ASIS-for-GNAT also generates warnings if it considers some situation arising during the ASIS query processing to be potentially wrong, and if the ASIS definition does not require to raise an exception in this case. Usually this is the case for actual or potential problems happening in an implementation-specific parts of the ASIS functionality, such as providing implementation-specific parameters to the queries Initialize, Finalizes and Associate or opening a Context.

There are three warning modes in ASIS-for-GNAT:

default
: warning messages are generated into stderr;

suppress
: warning messages are suppressed;

treat as error
: a warning is treated as an error by ASIS-for-GNAT: instead of sending a message to stderr, ASIS-for-GNAT raises Asis_Failed and converts the warning message into the ASIS Diagnosis string. ASIS Error Status depends on the cause of the warning.

The ASIS-for-GNAT warning mode may be set when initializing the ASIS implementation. The `"-ws"` parameter of Asis.Implementation.Initialize query suppresses warnings, `"-we"` parameter of this query sets treating all the warnings as errors. When set, the warning mode remains the same for all Contexts processed until ASIS-for-GNAT has finalized.

# 12 File Naming Conventions and Application's Name Space

Any ASIS application being developed with ASIS-for-GNAT depends on the ASIS interface components and, transitively on other ASIS-for-GNAT implementation components. Therefore, the name space available for application's compilation unit names in the very beginning of the application development already contains some names, which cannot be used as the names of application's components.

ASIS-for-GNAT includes the full specification of the ISO/IEC 15291:1999 ASIS Standard.

The following children and grandchildren of the top Asis package are added in ASIS-for-GNAT

- Asis.Extensions hierarchy (the source file names start from `asis-extensions` - defines some useful ASIS extensions, see ASIS Reference Manual for more details;
- Asis.Set_Get (the source files `asis-set_get.ad[bs]` respectively) - contains the access and update routines for the implementation of the main ASIS abstractions defined in Asis;
- Asis.Text.Set_Get (the source files `asis-text-set_get.ad[bs]` respectively - contains the access and update routines for the implementation of the ASIS abstractions defined in Asis.Text;

All other ASIS-for-GNAT Ada implementation components belong to the hierarchy headed by the package named A4G (which comes from ASIS-for-GNAT) and have names starting from "A4G.".

ASIS-for-GNAT also incorporates the following GNAT components as a part of the ASIS implementation:

```
Alloc
Atree
Casing
Csets
Debug
Einfo
Elists
Fname
Gnatvsn
Hostparm
Krunch
Lib
  Lib-List
  Lib-Sort
Namet
Nlists
Opt
Output
Repinfo
Scans
Sinfo
```

```
Sinput
Snames
Stand
Stringt
Table
Tree_In
Tree_Io
Types
Uintp
Uname
Urealp
Widechar
```

Therefore, in your ASIS application you can use for your Ada components any names except package names defined by ASIS as the names of the ASIS interface packages, Asis.Extensions, Asis.Set_Get, Asis.Text.Set_Get, and any name from the hierarchy headed by "A4G" and any name from the list of the GNAT component names given above.

All Ada source files making up the ASIS implementation for GNAT (including the GNAT components being a part of ASIS-for-GNAT) follow the GNAT file name conventions without any name krunching.