ASIS-for-GNAT User's Guide

Document revision level 1.22
Date: 2004/04/13 08:37:18

GNAT GPL 2005 Edition

Ada Core Technologies, Inc.

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 15, gives an overview of ASIS, allowing an ASIS newcomer to navigate through the ASIS definition (readers already familiar with ASIS can skip this section).
- Chapter 4 [ASIS Context], page 23, defines the ASIS `Context` concept in ASIS-for-GNAT and explains how to prepare a set of Ada components to be processed by an ASIS application.
- Chapter 6 [ASIS Application Templates], page 39, describes a set of Ada source components provided by the ASIS-for-GNAT distribution that may be used as a basis for developing ASIS applications.
- Chapter 7 [ASIS Tutorials], page 41, describes some examples included in the ASIS-for-GNAT distribution.
- Chapter 8 [How to Build Efficient ASIS Applications], page 43, describes how to deal with “tree swapping”, a potential performance issue with ASIS applications.
- Chapter 9 [Processing an Ada Library by an ASIS-Based Tool], page 47, shows how to use an ASIS tool on pre-compiled Ada libraries.
- Chapter 10 [Compiling Binding and Linking Applications with ASIS-for-GNAT], page 49, explains how to compile an ASIS application with ASIS-for-GNAT and how to create the resulting executable.
- Chapter 11 [ASIS-for-GNAT Warnings], page 51, describes the warnings generated by the ASIS implementation.
- Chapter 12 [Exception Handling and Reporting Internal Bugs], page 53, explains what happens if an ASIS implementation internal problem is detected during the processing of an ASIS or ASIS Extensions query

- Chapter 13 [File Naming Conventions and Application Name Space], page 55, explains which names can and cannot be used as names of ASIS application components.

What You Should Know Before Reading This Guide

This User's Guide assumes that you are familiar with Ada 95 language, as described in the International Standard ANSI/ISO/IEC-8652:1995 (hereafter referred to as the *Ada Reference Manual*), 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 ASIS README file).

This guide does not require previous knowledge of or experience with ASIS itself.

Related Information

The following sources contain useful supplemental information:

- *GNAT User's Guide*, for information about the GNAT environment
- *ASIS-for-GNAT Installation Guide*
- The *ASIS-for-GNAT Reference Manual*
- The *ASIS 95 definition*, available as ISO/IEC International Standard 15291.
- The Web site for the ASIS Working Group: `http://www.acm.org/sigada/wg/asiswg`

Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- `Functions`, `utility program names`, `standard names`, and `classes`.
- '`Option flags`'
- '`File Names`', '`button names`', and '`field names`'.
- *`Variables`*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text

```
and then shown this way.
```

Commands that are entered by the user are preceded in this manual by the characters "$ " (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the $ replaced by whatever prompt character you are using.

Full file names are shown with the "/" character as the directory separator; e.g., '`parent-dir/subdir/myfile.adb`'. If you are using GNAT on a Windows platform, please note that the "\" character should be used instead.

1 Introduction

1.1 What Is ASIS?

The *Ada Semantic Interface Specification* (ASIS) is an open and published callable interface that allows a tool to access syntactic and semantic information about an Ada program, independent of the compilation environment that compiled the program.

Technically, ASIS comprises a hierarchy of Ada packages rooted at the package `Asis`. These packages define a set of Ada private types that model the components of an Ada program (e.g., declarations, statements, expressions) and their interrelationships. 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 – Which Kinds of Tools Can Be Built with ASIS?

The following ASIS properties define the ASIS scope:

- ASIS is a read-only interface.
- ASIS provides only statically-determinable information about Ada programs.
- ASIS provides access to the syntactic and basic semantic properties of compiled Ada units. 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.
- ASIS provides information from/about Ada units in high-level terms that conform with the *Ada Reference Manual* and that are Ada/ASIS-implementation-independent in nature.

Examples of tools that benefit from the ASIS interface include, but are not limited to: 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

This section outlines the ASIS application development and usage cycle. We first take a sample problem and present an ASIS application that offers a solution; then we show how to build the executable 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

We wish to process some set of Ada compilation units as follows: for every unit, print its full expanded Ada name, whether this unit is a spec[1], a body or a subunit, and whether this unit is a user-defined unit, an Ada predefined unit or an implementation-specific unit (such as a part of a Run-Time Library).

2.2 An ASIS Application that 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 compilation environment,
   --  it defines a set of ASIS Compilation Units available through
   --  ASIS queries
```

[1] It may seem that an Ada unit such as

```
package Pack is
  type T is array(Positive range <>) of Float;
  procedure Proc(X : in out T);
end Pack;
```

is a package *specification*, but in fact the "specification" (as defined in the *Ada Reference Manual*) comprises all but the final semicolon. The form with the final semicolon is known as a "package declaration". Since this official term is not familiar to most Ada users, the GNAT documentation uses the term "spec" (for a unit) to mean that unit's *declaration* − thus a package spec includes the final semicolon.

```
begin
   --  first, by initializing an ASIS implementation, we make it
   --  ready for work
   Asis.Implementation.Initialize ("-ws");
   --  The "-ws" parameter of the Initialize procedure means
   --  "turn off all the ASIS warnings"

   --  then we define our Context by making an association with
   --  the "physical" environment:
   Asis.Ada_Environments.Associate
    (My_Context, "My Asis Context", "-CA");
   --  "-CA" as a Context parameter means "consider all the tree
   --  files in the current directory"
   --  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 lists 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);
      --  The Compilation_Units query returns 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,
         --  that is why we are using Ada.Wide_Text_IO

         Put (Asis.Compilation_Units.Unit_Full_Name (Next_Unit));

         --  to get more info about a unit, we ask about unit class
```

```
            --  and about unit 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 the Context, break its
      --  association with the external environment and 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 initializes the ASIS implementation's internal data structures. In general, calling an ASIS query is erroneous unless the `Initialize` procedure has been invoked.

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`. The value represents 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 to all ASIS queries.

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

   ASIS relies on the fact that the content of a `Context` remains "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. Now all the ASIS queries can be used. It is possible to access `Compilation_Units` from the `Context`, to decompose units into syntactic `Elements`, to query syntactic and semantic properties of these `Elements` and so on.

5. `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.

6. `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.

7. `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 Section 2.2 [An ASIS Application that Solves the Problem], page 7 (assuming that it is located in your current directory as the Ada source file named '`example1.adb`'), invoke `gnatmake` as follows[2]:

```
$ gnatmake example1.adb -largs -lasis
```

For more details concerning compiling ASIS applications and building executables for them with ASIS-for-GNAT see Chapter 10 [Compiling Binding and Linking Applications with ASIS-for-GNAT], page 49.

[2] The '`.adb`' is optional

2.5 Preparing Data for an ASIS Application – Generating Tree Files

The general ASIS implementation technique is to use some information generated by the underlying Ada compiler as the basis for retrieving information from the Ada environment. As a consequence, an ASIS application can process only legal (compilable) Ada code, and in most of the cases to make a compilation unit "visible" for ASIS means to compile this unit (probably with some ASIS-specific options)

ASIS-for-GNAT uses *tree output files* (or, in short, *tree files*) to capture information about an Ada unit from an Ada environment. A tree file is generated by GNAT, and it contains a snapshot of the compiler's internal data structures at the end of the successful compilation of the corresponding source file.

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

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

and as a result, GNAT will generate the tree file named '`example1.adt`' in the current directory.

For more information on how to generate and deal with tree files, see Chapter 4 [ASIS Context], page 23, and Chapter 7 [ASIS Tutorials], page 41.

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 this chapter, your current directory should contain the executable '`example1`' ('`example1.exe`' on a Windows platform) and the tree file '`example1.adt`'. 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 4 [ASIS Context], page 23, and the *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 (spec) - Ada predefined unit
   Ada.Wide_Text_IO (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) - implementation-specific unit
Interfaces (spec) - Ada predefined unit
Interfaces.C_Streams (spec) - implementation-specific unit
System.Parameters (spec) - implementation-specific unit
System.WCh_Con (spec) - implementation-specific unit
Ada.Characters (spec) - Ada predefined unit
Ada.Characters.Handling (spec) - Ada predefined unit
Asis (spec) - user-defined unit
A4G (spec) - user-defined unit
A4G.A_Types (spec) - user-defined unit
Ada.Characters.Latin_1 (spec) - Ada predefined unit
GNAT (spec) - implementation-specific unit
GNAT.OS_Lib (spec) - implementation-specific unit
GNAT.Strings (spec) - implementation-specific unit
Unchecked_Deallocation (spec) - Ada predefined unit
Sinfo (spec) - user-defined unit
Types (spec) - user-defined unit
Uintp (spec) - user-defined unit
Alloc (spec) - user-defined unit
Table (spec) - user-defined unit
Urealp (spec) - user-defined unit
A4G.Int_Knds (spec) - user-defined 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
System.Unsigned_Types (spec) - implementation-specific unit
```

Note that the tree file contains the full syntactic and semantic information not only about the unit compiled by the given call to gcc, but also about all the units upon which this unit depends semantically; that is why you can see in the output list a number of units which are not mentioned in our example.

In the current version of ASIS-for-GNAT, ASIS implementation components are considered user-defined, rather than implementation-specific, units.

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 find needed information in the ASIS definition.

For more details, please refer to the ASIS definition itself. To gain some initial experience with ASIS, try the examples in Chapter 7 [ASIS Tutorials], page 41.

3.1 Main ASIS Abstractions

ASIS is based on three main abstractions used to describe Ada programs; these abstractions are implemented as Ada private types:

`Context` An ASIS `Context` is a logical handle to an Ada environment, as defined in the *Ada Reference Manual*, Chapter 10. 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 all the properties of compilation units defined by the *Ada Reference Manual*, and it also reflects some properties of "physical objects" used by an underlying Ada implementation to model compilation units. Examples of such properties are the time of the last update, and the name of the object containing the unit's source text. An ASIS `Compilation_Unit` provides the "black-box" view of a compilation unit, considering the unit as a whole. It may be decomposed into ASIS `Elements` and then analyzed in "white-box" fashion.

`Element` An ASIS `Element` is a logical handle to a syntactic component of an ASIS `Compilation_Unit` (either explicit or implicit).

Some ASIS components use additional abstractions (private types) needed for specific pieces of functionality:

`Container`
An ASIS `Container` (defined by the `Asis.Ada_Environments.Containers` package) provides a means for structuring the content of an ASIS `Context`; i.e., ASIS `Compilation_Units` are grouped into `Containers`.

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

`Span` An ASIS `Span` (defined 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 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` and store it in a file. Subsequently the same or another application may read this `Id` value and convert it back into the corresponding `Element` value.

3.2 ASIS Package Hierarchy

ASIS is defined as a hierarchy of Ada packages. Below is a short description of this hierarchy.

`Asis` The root 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 the *Ada Reference Manual*) and the 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 diagnosis information. Its child package `Asis.Implementation.Permissions` contains boolean queries that reflect how ASIS implementation-specific features are implemented.

`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 for processing pragmas.

Packages working on specific `Elements`
: This group contains the following packages: `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 status values.

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` might not correspond to what could be expected from the Ada syntax and semantics; 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` or the list of `Elements` representing some semantic property – e.g., a type declaration

for an expression as the 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 retrieve the structural components of this assignment statement by applying the appropriate structural queries:

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

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 a 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 difference between classifying queries working on `Elements` as structural versus semantic is that all the structural queries must be within one ASIS `Compilation_Unit`, but for semantic queries it is typical for the argument of a query to be in one ASIS `Compilation_Unit`, while 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) propagate out from 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 the `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 the `Diagnosis` string by invoking the `Asis.Implementation.Set_Status` procedure.

Caution: The ASIS way of providing error information is not tasking safe. The `Diagnosis` string and Error Kind are global to an entire partition, and are not "per task". If ASIS exceptions are raised in more then one task of a multi-tasking ASIS application, the result of querying the error information in a particular task may be incorrect.

3.5 Dynamic Typing of ASIS Queries

The ASIS type `Element` covers all Ada syntactic constructs, and `Compilation_Unit` covers all Ada compilation units. ASIS defines an `Element` classification hierarchy (which reflects very closely the hierarchy of Ada syntactic categories defined in the *Ada Reference Manual*, and ASIS similarly defines a classification scheme for ASIS `Compilation_Units`. For any `Element` you can get its position in the `Element` classification hierarchy by means of classification queries defined in the package `Asis.Elements`. The classification queries for `Compilation_Units` are defined in the package `Asis.Compilation_Units`.

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 to query `Assignment_Variable_Name` for an `Element` of `An_Ordinary_Type_Declaration` kind. An attempt to perform such an operation will be detected at run-time, and an exception will be raised as explained in the next paragraph.

ASIS may be viewed as a dynamically typed interface. For any `Element` structural or semantic query (that is, for a query having an `Element` as an argument and returning either an `Element` or `Element` list as a result) a list of appropriate `Element` kinds is explicitly defined in the query documentation which immediately follows the declaration of the corresponding subprogram in the code of the ASIS package. This means that the query can be applied only to argument `Elements` being of the kinds from this list. If the kind of the argument `Element` does not belong to this list, the corresponding call to this query raises the `Asis.Exceptions.ASIS_Inappropriate_Element` exception with `Asis.Errors.Value_Error` error status set.

The situation for the queries working on `Compilation_Units` is similar. If a query lists appropriate unit kinds in its documentation, then this query can work only on `Compilation_Units` of the kinds from this list. The 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 mean-

ingful 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.

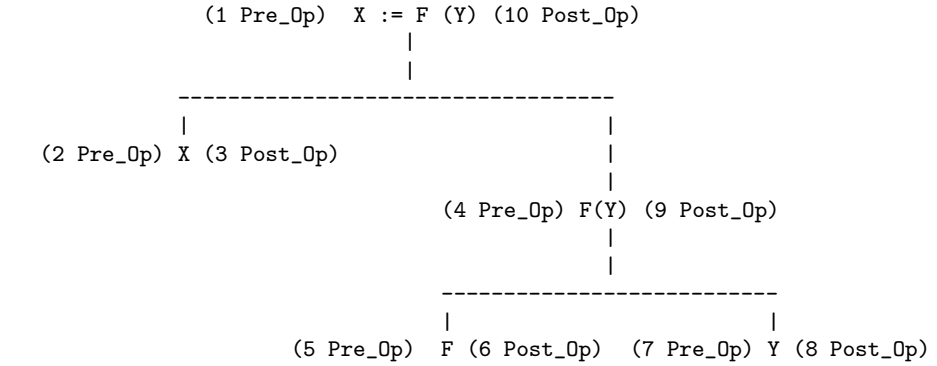
3.6 ASIS Iterator

ASIS provides a powerful mechanism to traverse an Ada unit, the generic procedure `Asis.Iterator.Traverse_Element`. This procedure makes a top-down left-to-right (or depth-first) traversal of the ASIS tree (that is, of the syntax structure of the Ada code viewed as 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 specific procedures for `Pre_Operation` and `Post_Operation` when instantiating the generic unit, you can 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):



To see in more detail how `Traverse_Element` may be used for rapid development of a number of useful ASIS applications, try the examples in Chapter 7 [ASIS Tutorials], page 41.

3.7 How to Navigate through the `Asis` Package Hierarchy

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

- Use the short overview of the ASIS packages given in Section 3.2 [ASIS Package Hierarchy], page 16, 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` then look only in `Asis.Compilation_Units`; if you are looking for queries that can be used to decompose and analyze declarations, limit your search to `Asis.Declarations`).
- 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 the *Ada Reference Manual* (e.g., package `Asis.Statements` starts from a query retrieving labels and ends with the query decomposing a code statement).
- The names of all the semantic queries (and only ones) start from `Corresponding_...` or `Implicit_...`
- 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

From an ASIS application viewpoint we may view an ASIS Context as a set of ASIS Compilation_Units accessible through 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.

4.1 ASIS Context and Tree Files

The ASIS-for-GNAT 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 if no error was detected during the compilation. The tree file is a kind of snapshot of the compiler internal data structures (basically, of the Abstract Syntax Tree (AST)) at the end of the successful compilation. ASIS then inputs tree files and recreates in its internal data structures exactly the same picture the compiler had at the end of the corresponding successful compilation.

An important consequence of the GNAT source-based compilation model is that the 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, a tool can provide information 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, this occurs when an application switches between units which do not semantically depend on each other, for example, two package bodies). 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 gcc nor gnatmake will create tree files automatically when you are working with your Ada program. It is your responsibility as a user of an ASIS application to create a set of tree files that correctly reflect the set of the Ada components to be processed by the ASIS application,

as well as to maintain the consistency of the trees and the related source files.

To create a tree file for a given source file, you need to compile the corresponding source file with the '`-gnatc`' and '`-gnatt`' options (these may be combined into the '`-gnatct`' option. Thus

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

will produce '`foo.adt`', provided that '`foo.adb`' contains the source of a legal Ada compilation unit. The '`-gnatt`' option generates a tree file, and '`-gnatc`' turns off AST expansion. ASIS needs tree files created without AST expansion, whereas to create an object file, GNAT needs an expanded AST. Therefore it is impossible for one compilation command to to produce both a tree file and an object file for a given source file.

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

- 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.
- A tree file is not created if an error has been detected during the compilation.
- In contrast with object files, a tree file may be generated for any legal Ada compilation unit, including a library package declaration requiring a body or a subunit.
- A set of tree files processed by an ASIS application may be inconsistent; for example, two tree files may have been created with different versions of the source of the same unit. This will lead to inconsistencies in the corresponding ASIS `Context`. See Section 4.4 [Consistency Problems], page 28, for more details.
- Do not move tree, object or source files among directories in the underlying file system! ASIS might assume an inconsistency between tree and source files when opening a `Context`, or you may get wrong results when querying the source or object file for a given ASIS `Compilation_Unit`.
- When invoking `gcc` or `gnatmake` to create tree files, make sure that all file and directory names containing relative path information start from '`./`' or '`../`' ('`.\`' and '`..\`' respectively in MS Windows). That is, to create a tree file for the source file '`foo.adb`' located in the inner directory named '`inner`', you should invoke gcc (assuming an MS Windows platform) as:

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

  but not as

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

  Otherwise ASIS will not perform correctly.

- When reading in a tree file, ASIS checks that this tree file was created with the '-gnatc' option, and it does not accept trees created without this option.
- 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 place some information from the existing 'ALI' file into the tree file. If you would like to have both object and tree files for your program, first create the object files, and then the tree files.
- There is only one extension for tree files, namely '.adt', whereas the standard GNAT name convention for the Ada source files uses different extensions for a spec ('.ads') and for a body ('.adb'). This means that if you first generate a tree for a unit's body:

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

  and then generate the tree for the corresponding spec:

  ```
  $ 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, try to use gnatmake whenever possible (see Section 8.4 [Using gnatmake to Create Tree Files], page 45 for more details). Otherwise, first create trees for specs and then for bodies:

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

- Reading tree files is a time-consuming operation. Try to minimize the number of tree files to be processed by your application, and try to avoid unnecessary tree swappings. (See Chapter 8 [How to Build Efficient ASIS Applications], page 43, for some tips).
- It is possible to create tree files "on the fly", as part of the processing of the ASIS queries that obtain units from a Context. In this case there is no need to create tree files before running an ASIS application using the corresponding Context mode. Note that this possibility goes beyond the ASIS Standard, and there are some limitations imposed on some ASIS queries, but this functionality may be useful for ASIS tools that process only one Compilation_Unit at a time. See the *ASIS-for-GNAT Reference Manual* for more details.

Note that between opening and closing a `Context`, an ASIS application should not change its working directory; otherwise execution of the application is 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, with one exception. The implementation of the ASIS DDA is based on some special annotations added by the compiler to the trees used by ASIS. An ASIS user should be aware of the fact that trees created for subunits do not have this special annotation. Therefore ASIS DDA queries do not work correctly on trees created for subunits (and these queries might not work correctly if a set of tree files making up a `Context` contains a tree created for a subunit).

Thus, when working with the ASIS DDA, you should avoid creating separate trees for subunits. Actually, this is not a limitation: to create a tree for a subunit, you should also have the source of the parent body available. If in this situation you create the tree for the parent body, it will contain the full information (including DDA-specific annotation) for all the subunits that are present. From the other side, a tree created for a single subunit has to contain information about the parent body, so it has about 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 that defines a `Context` has the following spec:

```
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 the properties of the `Context` are set by "options" specified in the `Parameters` string:

- How to define a set of tree files making up the `Context` ('`-C`' options);
- How to deal with tree files when opening a `Context` and when processing ASIS queries ('`-F`' options);
- How to process 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);

The association parameters may (and in some cases must) also 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*.

4.3.1 Defining a set of tree files making up a Context

The following options are available:

'-C1' "One tree" Context, defining a Context comprising a single tree file; this tree file name should be given explicitly in the Parameters string.

'-CN' "N-trees" Context, defining a Context comprising a set of tree files; the names of the tree files making up the Context should be given explicitly in the Parameters string.

'-CA' "All trees" Context, defining a Context comprising 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 default option is '-CA'.

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

4.3.2 Dealing with tree files when opening a Context and processing ASIS queries

The following options are available:

'-FT' Only pre-created trees are used, no tree file can be created by ASIS.

'-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.

'-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 default option is '-FT'.

Note that the '-FT' and '-FM' options go beyond the scope of the official ASIS standard. They may be useful for some ASIS applications with specific requirements for defining and processing an ASIS Context, but in each case the ramifications of using such non-standard options should be carefully considered. See the *ASIS-for-GNAT Reference Manual* for a detailed description of these option.

4.3.3 Processing source files during the consistency check when opening a Context

The following options are available:

'-SA' Source files for all the Compilation_Units belonging to the Context (except the predefined Standard package) have to be available, and all of them are taken into account for consistency checks when opening the Context.

'-SE' Only existing source files for all the Compilation_Units belonging to the Context are taken into account for consistency checks when opening the Context.

'-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 option is '-SA'. See Section 4.4 [Consistency Problems], page 28, concerning consistency issues in ASIS-for-GNAT.

4.3.4 Setting search paths

Using the '-I', '-gnatec' and '-gnatA' options for defining an ASIS Context is similar to using the same optionsfor gcc. The '-T' option is used in the same way, for tree files. For full details about the '-T' and '-I' options, refer to the *ASIS-for-GNAT Reference Manual*. Note that the '-T' option is used only to locate existing tree files, and it has no effect for '-FS' Contexts. On the other hand, the '-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 Problems

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, a tree file may have been created by another version of GNAT (see the 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, the tree files may be 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 an ASIS-for-GNAT installation problem and raises `Program_Error` with a corresponding exception message. In this case, `Program_Error` is not caught by any ASIS query, and it propagates outside ASIS.[1] Note that the real cause may be an old tree file you have forgotten to remove when reinstalling ASIS-for-GNAT. This is also considered an installation error.

ASIS uses the tree files created by the GNAT compiler installed on your machine, and the ASIS implementation includes some compiler components to define and to get access to the corresponding data structures. Therefore, the version of the GNAT compiler installed on your machine and the version of the GNAT compiler whose sources are used as a part of the ASIS implementation should be close enough to define the same data structures. We do not require these versions to be exactly the same, and, by default, when ASIS reads a tree file it only checks for significant differences. That is, it will accept tree files from previous versions of GNAT as long as it is possible for such files to be read. In theory, this check is not 100% safe; that is, a tree created by one version of GNAT might not be correctly processed by ASIS built with GNAT sources taken from another version. But in practice this situation is extremely unlikely.

An ASIS application may set a strong GNAT version check by providing the '`-vs`' parameter for the ASIS `Initialize` procedure, see *ASIS-for-GNAT Reference Manual* for more details. If the strong version check is set, then only a tree created by exactly the same version of GNAT whose sources are used as a part of the ASIS implementation can be successfully read in, and `Program_Error` will be raised otherwise.

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

[1] 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.

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` (via the `Asis.Ada_Environments.Open` query), ASIS does the following checks for all the tree files making up the `Context`:

- If the '`-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 references to all the source files used to create this tree file).
- If the '`-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 the '`-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, the `Asis_Failed` exception is raised as a result of opening a `Context`. If the `Context` has been successfully opened, you are guaranteed that ASIS will process only consistent sets of tree and source files until the `Context` is closed (provided that this set is not 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 an '`-FS`' or '`-FM`' option, be aware 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 an '`-FS`' or '`-FM`' option. If the application has such a `Context`, all the other `Contexts` should not use tree files located in the current directory.

4.6 Using ASIS with a cross-compiler

If you would like to use ASIS with a cross-compiler, you should use this cross-compiler to create the tree files to be used for the ASIS Context defined with '-FS' option. If you would like to use trees created on the fly (that is, to use a Context defined with the '-FS' or '-FM' option), you have to tell ASIS which compiler should be called to perform this function. There are two ways to do this.

- You can use the '--GCC' option in the Context definition to specify explicitly the name of the command to be called to create the trees on the fly
- You may use the prefix of the name of your ASIS tool to indicate the name of the command to be used to call the compiler. If the name of your tool contains a hyphen character "-", for example some_specific-foo, then ASIS will try to call the command with the name created as a concatenation of the tool name prefix preceding the rightmost hyphen, the hyphen character itself, and gcc. For example, for some_specific-foo, ASIS will try to call some_specific-gcc to create the tree file.

The algorithm for defining the name of the command to be used to create trees on the fly is as follows. If the '--GCC' option is used in the Context definition and if the name that is the parameter of this option denotes some executable existing in the path, this executable is used. Otherwise ASIS tries to define the name of the executable from the name of the ASIS application. If the corresponding executable exists on the path, it is used. Otherwise the standard gcc installation is used.

5 ASIS Interpreter asistant

This chapter describes asistant, an interactive interface to ASIS queries.

5.1 asistant Introduction

The asistant tool allows you to use ASIS without building your own ASIS applications. It provides a simple command language that allows you to define variables of ASIS types and to assign them values by calling ASIS queries.

This tool may be very useful while you are learning ASIS: it lets you try different ASIS queries and see the results immediately. It does not crash when there is an error in calling an ASIS query (such as passing an inappropriate Element); instead asistant reports an error and lets you try again.

You can also use asistant as a debug and "ASIS visualization" tool in an ASIS application project. If you have problems 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, you may use asistant to reconstruct the situation that causes the problems, and then experiment with ASIS queries.

Though primarily an interactive tool, asistant also can interpret sequences of commands written to a file (called a "script file" below). The asistant tool can also store in a file the log of an interactive session that 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 is a brief overview of asistant usage.

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 (or it is placed in the 'GNATPRO/bin' directory when installing ASIS from the binary distribution). Put this executable somewhere on your path[1], and then type "asistant" to call asistant in an interactive mode. As a result, the program will output brief information about itself and then the asistant prompt ">" will appear:

```
ASIStant - ASIS Tester And iNTerpreter, v1.2
(C) 1997-2002, Free Software Foundation, Inc.
  Asis Version: ASIS 2.0.R

>
```

[1] You do not have to do this if you have installed ASIS from the binary distribution, because the executable for asistant has been added to other GNAT executables

Now you can input `asistant` commands (`asistant` supports in its command language the same form of comments as Ada, and 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 as
  --  the result of the call to the ASIS query Compilation_Unit_Body.
  --  As a result, C_U will represent a 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 as the result of calling
  --  the ASIS Unit_Declaration query

>print (Unit) --  as a result of this command, some information about
              --  the current value of Unit will be printed (a user can set
              --  the desired level of detail of this information):

A_PROCEDURE_BODY_DECLARATION at ( 1 : 1 )-( 9 : 9 )

--  suppose now, that we do make an error - we call an ASIS query for
--  an inappropriate 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 asistant commands given in this section is incomplete; its purpose is only to give a general idea of asistant's capabilities. Standard metalanguage is assumed (i.e., "[*construct*]" denotes an optional instance of "*construct*").

`Help [(name)]`
: Outputs the profile of the ASIS query "name"; when called with no argument, generates general asistant help information.

`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)`
: Evaluates the expression "expr" and outputs its value (some information may be omitted depending on the level specified by the PrintDetail command).

`Run ('filename')`
: Launches the script from a file 'filename', reading further commands from it.

`Pause`
: Pauses 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 the file 'filename' for session logging.

`Log`
: Closes the current log file.

`PrintDetail`
: Toggles whether the Print command outputs additional information.

`Quit [(exit-status)]`
: Quits asistant.

5.3 asistant variables

The asistant tool lets you define variables with 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 Set command; there are 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

You perform ASIS tree browsing by invoking the asistant service function Browse. This will disable the asistant command interpreter and activate the Browser command interpreter. The Browser Q command switches back into the asistant environment by enabling the asistant command interpreter and disabling the Browser interpreter.

Browse has a single parameter of Element type, which establishes where the ASIS tree browsing will begin. Browse returns a result of type Element, namely the Element at which the tree browsing was stopped. Thus, if you type:

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

you will start ASIS tree browsing from e1; when you finish browsing, e0 will represent the last Element visited during the browsing.

If you type:

```
> Browse (e1)
```

you 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 commands:

U	Go one step up the ASIS tree (equivalent to calling the ASIS Enclosing_Element query);
D	Go 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)
\k1k2	where k1 is either D or d, and k2 is either T or t. Change the form of displaying the current Element: D turns ON display-

ing 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.

Q
: Go 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). These queries should accept one parameter of type Element and return Element as a result.

When you press <SPACE>, you are 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;
```

Suppose also that the tree for this source is created in the current directory. Below is a sequence of asistant commands which does process this unit. Explanation is provided via asistant comments.

```
initialize ("")
```

```
--  Create and open a Context comprising all the tree files
--  in the current directory:

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

-- Get a Compilation_Unit (body) named "Demo" from this Context:

Set (CU, Compilation_Unit_Body ("Demo", Cont))

--  Go into the unit structure and get 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))

-- Output 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_Called_Fun, Corresponding_Called_Function (Expr))

--  Print the debug and the text image of the declaration of the called
--  function:

Print (Corr_Called_Fun)
Print (Element_Image (Corr_Called_Fun))

-- Close the asistant session:

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 you can easily build an ASIS application by adding the code performing some specific ASIS analysis in well-defined places in these templates.

Refer to the ASIS tutorial’s solutions for 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 a quick start with ASIS. The tutorial contains a set of simple exercises 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 exercises, so the tutorial may also be considered as a set of ASIS examples.

For more information see the '`README`' file in the '`tutorial`' subdirectory.

8 How to Build Efficient ASIS Applications

This chapter identifies some potential performance issues with ASIS applications and offers some advice on how to address these issues.

8.1 Tree Swapping as a Performance Issue

If an ASIS `Context` comprises more then one tree, then ASIS may need to switch between different trees during an ASIS application run. Switching between trees may require ASIS to repeatedly read in the same set of trees, and this may slow down an application considerably.

Basically, there are two causes for tree swapping:

- *Processing of semantically independent units*. Suppose in `Context Cont` we have units `P` and `Q` that 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 obtain information about `P`, ASIS needs to access the tree '`p.adt`', and to get some information about `Q`, ASIS needs '`q.adt`'. Therefore, if an application retrieves some information from `P`, and then starts processing `Q`, ASIS has to read '`q.adt`'.
- *Processing of information from dependent units*. A unit `U` may be present not only in the tree created for `U`, but also in all the trees created for units which semantically depend upon `U`. 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 some `Element` representing a 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
```

but not for those queries in the above packages 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 guidelines may help:

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

  Minimizing the 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.

- Choose the `Context` definition appropriate to 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 syntactic and semantic information about this unit. Using the "one tree" `Context` definition, an application has only one tree file to read when opening a `Context`, and no other tree file will be read during the application run. An "N-trees" `Context` is a natural extension of "one tree" `Context` for applications that know in advance which units will be processed, but opening a `Context` takes 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 when an application has to process a large system consisting of a many units. When using an application based on an "all trees" `Context`, use the approach for creating tree files described above to minimize a set of tree files to be processed.

- In your ASIS application, try to avoid switching between processing

units or sets of units with no dependencies among them; such a switching will cause tree swapping.

- If you are going to analyze a library unit having both a spec and a body, start by 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 allow both the spec and the body of this unit to be processed without tree swapping.
- To see a "tree swapping profile" of your application use the '`-dt`' debug flag when initializing ASIS (`Asis.Implementation.Initialize ("-dt")`). The information returned may give you some hints on 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 an '`ALI`' file for every successful compilation, whether or not code has been generated. Therefore, it is possible to run `gnatmake` with the '`-gnatc`' and '`-gnatt`' options; this will create the set of tree files for all the compilation units needed in the resulting program. 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: it 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 being called with the '`-f`' option (which means "force recompilation"). Usng `gnatmake` without the '`-f`' option for creating tree files is not reliable if your tree files are in the same directory as the object files, because object and tree files "share" the same set of '`ALI`' files. If the object files exist and are consistent with the '`ALI`' and source files, the source will not be recompiled for creating a tree file unless the '`-f`' option is set.

A different approach is to combine 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. Thus, the object files and their associated 'ALI' files should be in another directory. In this case, by invoking gnatmake through:

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

or simply:

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

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

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

9 Processing an Ada Library by an ASIS-Based Tool

When an Ada unit to be processed by some ASIS-based tool makes use of an Ada library, you need to be aware of the following features of using Ada libraries with GNAT:

- An Ada library is a collection of precompiled Ada components. The sources of the Ada components belonging to the library are present, but if your program uses some components from a library, these components are not recompiled by `gnatmake` (except in circumstances described below). For example, `Ada.Text_IO` is not recompiled when you invoke `gnatmake` on a unit that `with`s `Ada.Text_IO`.
- According to the GNAT source-based compilation model, the spec of a library component is processed when an application unit depending on such a component is compiled, but the body of the library component is not processed. As a result, if you invoke `gnatmake` to create a set of tree files covering a given program, and if this program references an entity from an 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 on a Windows GNAT platform).
- In ASIS-for-GNAT, there is no standard way to define whether a given `Compilation_Unit` belongs to some precompiled Ada library other than the GNAT Run-Time Library (some heuristics may be added to `Asis.Extensions`). ASIS-for-GNAT classifies (by means of the `Asis.Compilation_Units.Unit_Origin` query) a unit as `A_Predefined_Unit`, if it is from the Run-Time Library and if it is mentioned in the *Ada Reference Manual*, Annex A, Paragraph 2 as an Ada 95 predefined unit; a unit is classified as `An_Implementation_Unit` if is belongs to Run-Time Library but is not mentioned in the paragraph just cited. Components of Ada libraries other than the Run-Time Library are always classified as `An_Application_Unit`;
- It is possible to recompile the components of the Ada libraries used by a given program. To do this, you have to invoke `gnatmake` for this program with the '`-a`' option. If you create a set of tree files for your program by invoking `gnatmake` with the '`-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 an ASIS-based tool if processing (or avoiding processing) of Ada libraries is important for the functionality of the tool:

- If the tool is not to process components of Ada libraries, then a set of tree files for this tool may be created by invoking `gnatmake` without the '`-a`' option (this is the usual way of using `gnatmake`). When the tool encounters a `Compilation_Unit` which represents a spec of some library unit, and for which `Asis.Compilation_Units.Is_Body_Required` returns `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.
- If a tool needs to process all the Ada compilation units making up a program, then a set of tree files for this program should be created by invoking `gnatmake` with the '`-a`' option.

You can use `Asis.Compilation_units.Unit_Origin` to filter out Run-Time Library 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 to supply any ASIS-specific options for gcc or 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 with 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 this either by the '-I' option to gcc or by adding these directories to 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 the top-level ASIS 'README' file) in the search path for gnatbind. You can do this either with the '-aO' option to gnatbind or by adding this directory to 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 certain ASIS-defined exceptions should be raised, and ASIS-for-GNAT conforms to 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 raising an exception. Usually this occurs with actual or potential problems in an implementation-specific part of ASIS, such as providing implementation-specific parameters to the queries `Initialize`, `Finalize` and `Associate` or opening a `Context`.

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

default Warning messages are output to `Standard_Error`.

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 `Standard_Error`, 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, the '`-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 completed.

12 Exception Handling and Reporting Internal Bugs

According to the ASIS Standard, only ASIS-defined exceptions can be propagated from ASIS queries. The same holds for the ASIS Extensions queries supported by ASIS-for-GNAT.

If a non-ASIS exception is raised during the processing of an ASIS or ASIS extension query, this symptom reflects an internal implementation problem. Under such a circumstance, by default the ASIS query will output some diagnostic information to `Standard_Error` and then exit to the OS; that is, the execution of the ASIS application is aborted.

In order to allow the execution of an ASIS-based program to continue even in case of such internal ASIS implementation errors, you can change the default behavior by supplying appropriate parameters to `Asis.Implementation.Initialize`. See *ASIS-for-GNAT Reference Manual* for more details.

13 File Naming Conventions and Application Name Space

Any ASIS application depends on the ASIS interface components; an ASIS application programmer thus needs to be alert to (and to avoid) clashes with the names of these components.

ASIS-for-GNAT includes the full specification of the ASIS Standard, and also adds the following children and grandchildren of the root `Asis` package:

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

All other ASIS-for-GNAT Ada implementation components belong to the hierarchy rooted at the package `A4G` (which comes from "ASIS-for-GNAT").

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 should not add children at any level of the `Asis` or `A4G` hierarchies, and you should avoid using any name from the list of the GNAT component names 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 "krunch"ing.

Index

-

A

B

C

D

E

G

H

I

L

N

O

P

Q

R

S

T

W

Table of Contents