GPR Tools User's Guide

Release 2018

May 25, 2018

This page is intentionally left blank.

CONTENTS

This page is intentionally left blank.

Version 2018
Date: May 25, 2018

AdaCore

This page is intentionally left blank.

CHAPTER

ONE

INTRODUCTION

This User's Guide describes several software tools that use the GNAT project facility to drive their behavior. GNAT projects are stored in text files with the extension `.gpr`, commonly called *GPR files*.

These GPR tools use a common facility, the GNAT Project Manager, that is fully described in *GNAT Project Manager*.

The main GPR tool is `GPRbuild`, a multi-language builder for systems organized into subsystems and libraries. This tool is described in *Building with GPRbuild*.

The other GPR tools are described in *GPRbuild Companion Tools*:

- `GPRconfig`

  A configuration project file generator (see *Configuring with GPRconfig*).

- `GPRclean`

  A tool to remove compilation artifacts created by GPRbuild (see *Cleaning up with GPRclean*).

- `GPRinstall`

  Executable and library installer using GPR files (see *Installing with GPRinstall*).

- `GPRname`

  Naming scheme generator (see *Specifying a Naming Scheme with GPRname*).

- `GPRls`

  Library browser (see *The Library Browser GPRls*).

This page is intentionally left blank.

CHAPTER

TWO

GNAT PROJECT MANAGER

2.1 Introduction

This chapter describes GNAT's *Project Manager*, a facility that allows you to manage complex builds involving a number of source files, directories, and options for different system configurations. In particular, project files allow you to specify properties including:

- The directory or set of directories containing the source files, and/or the names of the specific source files themselves;
- The directory in which the compiler's output (`ALI` files, object files, tree files, etc.) is to be placed;
- The directory in which the executable programs are to be placed;
- Switch settings, which can be applied either globally or to individual compilation units, for any of the project-enabled tools;
- The source files containing the main subprograms to be built;
- The source programming language(s); and
- Source file naming conventions, which can be specified either globally or for individual compilation units (see *Naming Schemes*).

Project files also allow you to:

- Change any of the above settings depending on external values, thus enabling the reuse of the projects in various **scenarios** (see *Scenarios in Projects*); and
- Automatically build libraries as part of the build process (see *Library Projects*).

Project files are written in an Ada-like syntax, using familiar notions such as packages, context clauses, declarations, default values, assignments, and inheritance (see *Project File Reference*).

Project files can depend upon other project files in a modular fashion, simplifying complex system integration and project reuse.

- One project can **import** other projects containing needed source files. More generally, the Project Manager lets you structure large development efforts into possibly interrelated subsystems, where build decisions are delegated to the subsystem level, and thus different compilation environments (switch settings) are used for different subsystems. See *Organizing Projects into Subsystems*.
- You can organize GNAT projects in a hierarchy: a project can **extend** a base project, inheriting its source files and optionally overriding any of them with alternative versions. See *Project Extension*.

Several tools support project files, generally in addition to specifying the information on the command line itself. They share common switches to control the loading of the project (in particular `-P`*`projectfile`* to define the applicable project file and `-X`*`vbl`*`=`*`value`* to set the value of an external variable).

The Project Manager supports a wide range of development strategies, for systems of all sizes. Here are some typical practices that are easily handled:

- Using a common set of source files and generating object files in different directories via different switch settings. This can be used for instance to generate separate sets of object files for debugging and for production.
- Using a mostly shared set of source files with different versions of some units or subunits. This can be used for instance to group and hide all OS dependencies in a small number of implementation units.

Project files can be used to achieve some of the effects of a source versioning system (for example, defining separate projects for the different sets of sources that comprise different releases) but the Project Manager is independent of any source configuration management tool that might be used by the developers.

The sections below use an example-driven approach to present and illustrate the various concepts related to projects.

2.2 Building with Projects

In its simplest form a project may be used in a stand-alone fashion to build a single executable, and this section will focus on such a setup in order to introduce the main ideas. Later sections will extend this basic model to more complex and realistic configurations.

The following concepts are the foundation of project files, and will be further detailed later in this documentation. They are summarized here as a reference.

Project file: A text file expressed in an Ada-like syntax, generally with the `.gpr` extension. It defines build-related characteristics of an application. The characteristics include the list of sources, the location of those sources, the location for the generated object files, the name of the main program, and the options for the various tools involved in the build process.

Project attribute: A specific project characteristic is defined by an *attribute clause*. Its value is a string or a sequence of strings. All settings in a project are defined through a list of predefined attributes with precise semantics. See *Attributes*.

Package in a project: Global attributes are defined at the top level of a project. Attributes affecting specific tools are grouped in a package whose name is related to tool's function. The most common packages are *Builder*, *Compiler*, *Binder*, and *Linker*. See *Packages*.

Project variables: In addition to attributes, a project can use variables to store intermediate values and avoid duplication in complex expressions. Variables can be initialized with external values coming from the environment. A frequent use of variables is to define *scenarios*. See *External Values*, *Scenarios in Projects*, and *Variables*.

Source files and source directories: A source file is associated with a language through a naming convention. For instance, `foo.c` is typically the name of a C source file; `bar.ads` or `bar.1.ada` are two common naming conventions for a file containing an Ada spec. A compilable entity is often composed of a main source file and potentially several auxiliary ones, such as header files in C. The naming conventions can be user-defined (see *Naming Schemes*), and will drive the builder to call the appropriate compiler for the given source file.

Source files are searched for in the source directories associated with the project through the **Source_Dirs** attribute. By default, all the files (in these source directories) following the naming conventions associated with the declared languages are considered to be part of the project. It is also possible to limit the list of source files using the **Source_Files** or **Source_List_File** attributes. Note that those last two attributes only accept basenames with no directory information.

Object files and object directory: An object file is an intermediate file produced by the compiler from a compilation unit. It is used by post-compilation tools to produce final executables or libraries. Object files produced in the context of a given project are stored in a single directory that can be specified by the **Object_Dir** attribute. In

order to store objects in two or more object directories, the system must be split into distinct subsystems, each with its own project file.

The following subsections introduce the attributes of interest for simple build needs. Here is the basic setup that will be used in the following examples:

The Ada source files `pack.ads`, `pack.adb`, and `proc.adb` are in the `common/` directory. The file `proc.adb` contains an Ada main subprogram `Proc` that `with`s package `Pack`. We want to compile these source files with the switch `-O2`, and place the resulting files in the `common/obj/` directory. Here is the directory structure:

```
common/
  pack.ads
  pack.adb
  proc.adb
common/obj/
  proc.ali, proc.o pack.ali, pack.o, proc.exe
```

Our project is to be called *Build*. The name of the file is the name of the project (case-insensitive) with the `.gpr` extension, therefore the project file name is `build.gpr`. This is not mandatory, but a warning is issued when this convention is not followed.

This is a very simple example, and as stated above, a single project file is sufficient. We will thus create a new file, `build.gpr`, that initially contains an empty project declaration:

```
project Build is
end Build;
```

Note that repeating the project name after `end` is mandatory.

2.2.1 Source Files and Directories

When you create a new project, the first task is to specify where the corresponding source files are located. These are the only settings that are needed by all the tools that will use this project (builder, compiler, binder and linker for the compilation, IDEs to edit the source files, etc.).

The first step is thus to declare the source directories, which are the directories to be searched to find source files. In the current example, the `common` directory is the only source directory.

There are several ways to specify the source directories:

- When the attribute **Source_Dirs** is not defined, a project contains a single source directory which is the one where the project file itself resides. In our example, if `build.gpr` is placed in the `common` directory, the project will have the needed implicit source directory.
- The attribute **Source_Dirs** can be set to a list of path names, one for each of the source directories. Such paths can either be absolute names (for instance `"/usr/local/common/"` on Unix), or relative to the directory in which the project file resides (for instance `"."` if `build.gpr` is inside `common/`, or `"common"` if it is one level up). Each of the source directories must exist and be readable.

  The syntax for directories is platform specific. For portability, however, the project manager will always properly translate Unix-like path names to the native format of the specific platform. For instance, when the same project file is to be used both on Unix and Windows, `"/"` should be used as the directory separator rather than `"\"`.
- The attribute **Source_Dirs** can automatically include subdirectories using a special syntax inspired by some Unix shells. If any of the paths in the list ends with "`**`", then that path and all its subdirectories (recursively) are included in the list of source directories. For instance, "`**`" and "`./**`" represent the complete directory tree rooted at the directory in which the project file resides.

When using the `Source_Dirs` construct, you may sometimes find it convenient to also use the attribute `Excluded_Source_Dirs`, which is also a list of paths. Each entry specifies a directory whose immediate content, not including subdirs, is to be excluded. It is also possible to exclude a complete directory subtree using the `**` notation.

It is often desirable to remove, from the source directories, directory subtrees rooted at some subdirectories. An example is the subdirectories created by a Version Control System such as Subversion that creates directory subtrees rooted at a subdirectory named `.svn`. To do that, attribute **Ignore_Source_Sub_Dirs** can be used. It specifies the list of simple file names or patterns for the roots of these undesirable directory subtrees.

```
for Source_Dirs use ("./**");
for Ignore_Source_Sub_Dirs use (".svn", "@*");
```

With the declaration of attribute Ignore_Source_Sub_Dirs above, .svn subtrees as weel as subtrees rooted at subdirectories with a name starting with '@' are not part of the source directories of the project.

When applied to the simple example, and because we generally prefer to have the project file at the top-level directory rather than mixed with the sources, we will add the relevant definition for the `Source_Dirs` attribute to our `build.gpr` project file:

```
project Build is
   for Source_Dirs use ("common");  --  <<<<
end Build;
```

Once the source directories have been specified, you may need to indicate specific source files of interest. By default, all source files present in the source directories are considered by the Project Manager. When this is not desired, it is possible to explicitly specify the list of sources to consider. In such a case, only source file base names are indicated and not their absolute or relative path names. The project manager is in charge of locating the specified source files in the specified source directories.

- By default, the project manager searches for all source files of all specified languages in all the source directories.

  Since the project manager was initially developed for Ada environments, the default language is usually Ada and the above project file is complete: it defines without ambiguity the sources composing the project: that is, all the sources in subdirectory `common` for the default language (Ada) using the default naming convention.

  However, when compiling a multi-language application, or a pure C application, the project manager must be told which languages are of interest, which is done by setting the **Languages** attribute to a list of strings, each of which is the name of a language.

  Even when only Ada is used, the default naming might not be suitable. Indeed, how does the project manager distinguish an Ada source file from any other file? Project files can describe the naming scheme used for source files, and override the default (see *Naming Schemes*). The default is the standard GNAT extension (`.adb` for bodies and `.ads` for specs), which is what is used in our example, and thus no naming scheme is explicitly specified. See *Naming Schemes*.

- *Source_Files*. In some cases, source directories might contain files that should not be included in a project. One can specify the explicit list of file names to be considered through the **Source_Files** attribute. When this attribute is defined, instead of looking at every file in the source directories, the project manager takes only those names into consideration and reports errors if they cannot be found in the source directories or do not correspond to the naming scheme.

- It is sometimes useful to have a project with no sources (most of the time because the attributes defined in the project file will be reused in other projects, as explained in *Organizing Projects into Subsystems*. To do this, the attribute `Source_Files` is set to the empty list, i.e. `()`. Alternatively, `Source_Dirs` can be set to the empty list, with the same result.

- *Source_List_File*. If there is a large number of files, it might be more convenient to use the attribute **Source_List_File**, which specifies the full path of a file. This file must contain a list of source file names (one per line, no directory information) that are searched as if they had been defined through `Source_Files`. Such a file can easily be created through external tools.

  A warning is issued if both attributes `Source_Files` and `Source_List_File` are given explicit values. In this case, the attribute `Source_Files` prevails.

- *Excluded_Source_Files*. Specifying an explicit list of files is not always convenient. Instead it might be preferable to use the default search rules with specific exceptions. This can be done through the attribute **Excluded_Source_Files** (or its synonym **Locally_Removed_Files**). Its value is the list of file names that should not be taken into account. This attribute is often used when extending a project, see *Project Extension*. A similar attribute **Excluded_Source_List_File** plays the same role but takes the name of file containing file names similarly to `Source_List_File`.

In most simple cases, such as the above example, the default source file search behavior provides the expected result, and we do not need to add anything after setting `Source_Dirs`. The Project Manager automatically finds `pack.ads`, `pack.adb`, and `proc.adb` as source files of the project.

Note that by default a warning is issued when a project has no sources attached to it and this is not explicitly indicated in the project file.

2.2.2 Duplicate Sources in Projects

If the order of the source directories is known statically, that is if `"/**"` is not used in the string list for `Source_Dirs`, then there may be several files with the same name situated in different directories of the project. In this case, only the file in the first directory is considered as a source of the project and the others are hidden. If `"/**"` is used in the string list for `Source_Dirs`, it is an error to have several files with the same name in the same directory `"/**"` subtree, since there would be an ambiguity as to which one should be used.

If there are two sources with the same name in different directories of the same `"/**"` subtree, one way to resolve the problem is to exclude the directory of the file that should not be used as a source of the project.

2.2.3 Object and Exec Directory

Another consideration when designing a project is to decide where the compiler should place the object files. In fact, the compiler and other tools might create several different kinds of files (for GNAT, there is the object file and the ALI file). One of the important concepts in projects is that most tools may consider source directories as read-only and thus do not attempt to create new or temporary files there. Instead, all such files are created in the object directory. (This is not true for project-aware IDEs, one of whose purposes is to create the source files.)

The object directory is specified through the **Object_Dir** attribute. Its value is the path to the object directory, either absolute or relative to the directory containing the project file. This directory must already exist and be readable and writable, although some tools have a switch to create the directory if needed (See the switch `-p` for *gprbuild*).

If the attribute `Object_Dir` is not specified, it defaults to the directory containing the project file.

For our example, we can specify the object directory in this way (assuming that the project file will reside in the parent directory of `common`):

```
project Build is
   for Source_Dirs use ("common");
   for Object_Dir use "common/obj";    --  <<<<
end Build;
```

As mentioned earlier, *there is a single object directory per project*. As a result, if you have an existing system where the object files are spread across several directories, one option is to move all of them into the same directory if you want to build it with a single project file. An alternative approach is described below (see *Organizing Projects into Subsystems*), allowing each separate object directory to be associated with a corresponding subsystem of the application.

When the *linker* is called, it usually creates an executable. By default, this executable is placed in the project's object directory. However in some situations it may be convenient to store it in elsewhere. This can be done through the **Exec_Dir** attribute, which, like `Object_Dir` contains a single absolute or relative path and must point to an existing and writable directory, unless you ask the tool to create it on your behalf. If neither `Object_Dir` nor `Exec_Dir` is specified then the executable is placed in the directory containing the project file.

In our example, let's specify that the executable is to be placed in the same directory as the project file `build.gpr`. The project file is now:

```
project Build is
   for Source_Dirs use ("common");
   for Object_Dir use "obj";
   for Exec_Dir use ".";  --   <<<<
end Build;
```

2.2.4 Main Subprograms

An important role of a project file is to identify the executable(s) that will be built. It does this by specifying the source file for the main subprogram (for Ada) or the file that contains the `main` function (for C).

There can be any number of such main files within a given project, and thus several executables can be built from a single project file. Of course, a given executable might not (and in general will not) need all the source files referenced by the project. As opposed to other build mechanisms such as through a *Makefile*, you do not need to specify the list of dependencies of each executable. The project-aware builder knows enough of the semantics of the languages to build and link only the necessary elements.

The list of main files is specified via the **Main** attribute. It contains a list of file names (no directories). If a project defines this attribute, it is not necessary to identify main files on the command line when invoking a builder, and editors like *GPS* will be able to create extra menus to spawn or debug the corresponding executables.

```
project Build is
   for Source_Dirs use ("common");
   for Object_Dir use "obj";
   for Exec_Dir use ".";
   for Main use ("proc.adb");  --   <<<<
end Build;
```

If this attribute is defined in the project, then spawning the builder with a command such as

```
gprbuild -Pbuild
```

automatically builds all the executables corresponding to the files listed in the *Main* attribute. It is possible to specify one or more executables on the command line to build a subset of them.

One or more spaces may be placed between the `-P` and the project name, and the project name may be a simple name (no file extension) or a path for the project file. Thus each of the following is equivalent to the command above:

```
gprbuild -P build
gprbuild -P build.gpr
gprbuild -P ./build.gpr
```

2.2.5 Tools Options in Project Files

We now have a project file that fully describes our environment, and it can be used to build the application with a simple *GPRbuild* command as shown above. In fact, the empty project that we saw at the beginning (with no attribute definitions) could already achieve this effect if it was placed in the `common` directory.

Of course, we might want more control. This section shows you how to specify the compilation switches that the various tools involved in the building of the executable should use.

Since source names and locations are described in the project file, it is not necessary to use switches on the command line for this purpose (such as `-I` for gcc). This removes a major source of command line length overflow. Clearly, the builders will have to communicate this information one way or another to the underlying compilers and tools they call, but they usually use various text files, such as response files, for this purpose and thus are not subject to command line overflow.

Several tools are used to create an executable: the compiler produces object files from the source files; the binder (when the language is Ada) creates a "source" file that, among other things, takes care of elaboration issues and global variable initialization; and the linker gathers everything into a single executable. All these tools are known to the project manager and will be invoked with user-defined switches from the project files. To obtain this effect, a project file feature known as a *package* is used.

A project file contains zero or more **packages**, each of which defines the attributes specific to one tool (or one set of tools). Project files use an Ada-like syntax for packages. Package names permitted in project files are restricted to a predefined set (see *Packages*), and the contents of packages are limited to a small set of constructs and attributes (see *Attributes*).

Our example project file below includes several empty packages. At this stage, they could all be omitted since they are empty, but they show which packages would be involved in the build process.

```
project Build is
   for Source_Dirs use ("common");
   for Object_Dir use "obj";
   for Exec_Dir use ".";
   for Main use ("proc.adb");

   package Builder is  --<<<  for gprbuild
   end Builder;

   package Compiler is --<<<  for the compiler
   end Compiler;

   package Binder is   --<<<  for the binder
   end Binder;

   package Linker is   --<<<  for the linker
   end Linker;
end Build;
```

Let's first examine the compiler switches. As stated in the initial description of the example, we want to compile all files with `-O2`. This is a compiler switch, although it is typical, on the command line, to pass it to the builder which

then passes it to the compiler. We recommend directly using the correct package, which will make the setup easier to understand.

Several attributes can be used to specify the switches:

Default_Switches:

This illustrates the concept of an **indexed attribute**. When such an attribute is defined, you must supply an *index* in the form of a literal string. In the case of *Default_Switches*, the index is the name of the language to which the switches apply (since a different compiler will likely be used for each language, and each compiler has its own set of switches). The value of the attribute is a list of switches.

In this example, we want to compile all Ada source files with the switch `-O2`; the resulting *Compiler* package is as follows:

```
package Compiler is
  for Default_Switches ("Ada") use ("-O2");
end Compiler;
```

Switches:

In some cases, we might want to use specific switches for one or more files. For instance, compiling `proc.adb` might not be desirable at a high level of optimization. In such a case, the *Switches* attribute (indexed by the file name) can be used and will override the switches defined by *Default_Switches*. The *Compiler* package in our project file would become:

```
package Compiler is
   for Default_Switches ("Ada")
       use ("-O2");
   for Switches ("proc.adb")
       use ("-O0");
end Compiler;
```

Switches may take a pattern as an index, such as in:

```
package Compiler is
  for Default_Switches ("Ada")
      use ("-O2");
  for Switches ("pkg*")
      use ("-O0");
end Compiler;
```

Sources `pkg.adb` and `pkg-child.adb` would be compiled with `-O0`, not `-O2`.

Switches can also be given a language name as index instead of a file name in which case it has the same semantics as *Default_Switches*. However, indexes with wild cards are never valid for language name.

Local_Configuration_Pragmas:

This attribute may specify the path of a file containing configuration pragmas for use by the Ada compiler, such as *pragma Restrictions (No_Tasking)*. These pragmas will be used for all the sources of the project.

The switches for the other tools are defined in a similar manner through the **Default_Switches** and **Switches** attributes, respectively in the *Builder* package (for *GPRbuild*), the *Binder* package (binding Ada executables) and the *Linker* package (for linking executables).

2.2.6 Compiling with Project Files

Now that our project file is written, let's build our executable. Here is the command we would use from the command line:

```
gprbuild -Pbuild
```

This will automatically build the executables specified in the *Main* attribute: for each, it will compile or recompile the sources for which the object file does not exist or is not up-to-date; it will then run the binder; and finally run the linker to create the executable itself.

The *GPRbuild* builder can automatically manage C files the same way: create the file `utils.c` in the `common` directory, set the attribute *Languages* to *"(Ada, C)"*, and re-run

```
gprbuild -Pbuild
```

GPRbuild knows how to recompile the C files and will recompile them only if one of their dependencies has changed. No direct indication on how to build the various elements is given in the project file, which describes the project properties rather than a set of actions to be executed. Here is the invocation of *GPRbuild* when building a multi-language program:

```
$ gprbuild -Pbuild
gcc -c proc.adb
gcc -c pack.adb
gcc -c utils.c
gprbind proc
...
gcc proc.o -o proc
```

Notice the three steps described earlier:

- The first three gcc commands correspond to the compilation phase.
- The gprbind command corresponds to the post-compilation phase.
- The last gcc command corresponds to the final link.

The default output of *GPRbuild* is reasonably simple and easy to understand. In particular, some of the less frequently used commands are not shown, and some parameters are abbreviated. Thus it is not possible to rerun the effect of the *GPRbuild* command by cut-and-pasting its output. The `-v` option to *GPRbuild* provides a much more verbose output which includes, among other information, more complete compilation, post-compilation and link commands.

2.2.7 Executable File Names

By default, the executable name corresponding to a main file is computed from the main source file name. Through the attribute **Executable** in package `Builder`, it is possible to change this default.

For instance, instead of building an executable named `"proc"` (or `"proc.exe"` on Windows), we could configure our project file to build `proc1` (respectively `proc1.exe`) as follows:

```
project Build is
   ...  --  same as before
   package Builder is
      for Executable ("proc.adb") use "proc1";
   end Builder
end Build;
```

Attribute **Executable_Suffix**, when specified, changes the suffix of the executable files when no attribute `Executable` applies: its value replaces the platform-specific executable suffix. The default executable suffix is the empty string empty on Unix and `".exe"` on Windows.

It is also possible to change the name of the produced executable by using the command line switch `-o`. However, when several main programs are defined in the project, it is not possible to use the `-o` switch; then the only way to change the names of the executable is through the attributes `Executable` and `Executable_Suffix`.

2.2.8 Using Variables to Avoid Duplication

To illustrate some other project capabilities, here is a slightly more complex project using similar sources and a main program in C:

```
project C_Main is
   for Languages    use ("Ada", "C");
   for Source_Dirs  use ("common");
   for Object_Dir   use  "obj";
   for Main         use ("main.c");
   package Compiler is
      C_Switches := ("-pedantic");
      for Default_Switches ("C")   use C_Switches;
      for Default_Switches ("Ada") use ("-gnaty");
      for Switches ("main.c") use C_Switches & ("-g");
   end Compiler;
end C_Main;
```

This project has many similarities with the previous one. As expected, its `Main` attribute now refers to a C source file. The attribute `Exec_Dir` is now omitted, thus the resulting executable will be put in the object directory `obj`.

The most noticeable difference is the use of a variable in the `Compiler` package to store settings used in several attributes. This avoids text duplication and eases maintenance (a single place to modify if we want to add new switches for C files). We will later revisit the use of variables in the context of scenarios (see *Scenarios in Projects*).

In this example, we see that the file `main.c` will be compiled with the switches used for all the other C files, plus `-g`. In this specific situation the use of a variable could have been replaced by a reference to the `Default_Switches` attribute:

```
for Switches ("c_main.c") use Compiler'Default_Switches ("C") & ("-g");
```

Note the tick character "'", which is used to refer to attributes defined in a package.

Here is the output of the *GPRbuild* command using this project:

```
$ gprbuild -Pc_main
gcc -c -pedantic -g main.c
gcc -c -gnaty proc.adb
gcc -c -gnaty pack.adb
gcc -c -pedantic utils.c
gprbind main.bexch
...
gcc main.o -o main
```

The default switches for Ada sources, the default switches for C sources (in the compilation of `lib.c`), and the specific switches for `main.c` have all been taken into account.

2.2.9 Naming Schemes

Sometimes an Ada software system needs to be ported from one compilation environment to another (such as GNAT), but the files might not be named using the default GNAT conventions. Instead of changing all the file names, which for a variety of reasons might not be possible, you can define the relevant file naming scheme in the **Naming** package of your project file.

The naming scheme has two distinct goals for the Project Manager: it allows source files to be located when searching in the source directories, and given a source file name it makes it possible to infer the associated language, and thus which compiler to use.

Note that the Ada compiler's use of pragma *Source_File_Name* is not supported when using project files. You must use the features described here. You can, however, specify other configuration pragmas.

The following attributes can be defined in package *Naming*:

Casing:

Its value must be one of `"lowercase"` (the default if unspecified), `"uppercase"` or `"mixedcase"`. It describes the casing of file names with regard to the Ada unit name.

Given an Ada package body My_Unit, the base file name (i.e. minus the extension, which is controlled by other attributes described below) will respectively be:

- for "lowercase": "my_unit"
- for "uppercase": "MY_UNIT"
- for "mixedcase": any spelling with indifferent casing such as "My_Unit", "MY_Unit", "My_UnIT" etc... The case insensitive name must be unique, otherwise an error will be reported. For example, there cannot be two source file names such as "My_Unit.adb" and "MY_UnIT.adb".

On Windows, file names are case insensitive, so this attribute is irrelevant.

Dot_Replacement:

This attribute specifies the string that should replace the `"."` in unit names. Its default value is `"-"` so that a unit `Parent.Child` is expected to be found in the file `parent-child.adb`. The replacement string must satisfy the following requirements to avoid ambiguities in the naming scheme:

- It must not be empty
- It cannot start or end with an alphanumeric character
- It cannot be a single underscore
- It cannot start with an underscore followed by an alphanumeric
- It cannot contain a dot `'.'` unless the entire string is `"."`
- It cannot include a space or a character that is not printable ASCII

Spec_Suffix and **Specification_Suffix**:

For Ada, these attributes specify the suffix used in file names that contain specifications. For other languages, they give the extension for files that contain declarations (header files in C for instance). The attribute is indexed by the language name. The two attributes are equivalent, but `Specification_Suffix` is obsolescent.

If the value of the attribute is the empty string, it indicates to the Project Manager that the only specifications/header files for the language are those specified with attributes `Spec` or `Specification_Exceptions`.

If `Spec_Suffix ("Ada")` is not specified, then the default is `".ads"`.

A non empty value must satisfy the following requirements:

- It must include at least one dot
- If `Dot_Replacement` is a single dot, then it cannot include more than one dot.

Body_Suffix and **Implementation_Suffix**:

These attributes are equivalent and specify the extension used for file names that contain code (bodies in Ada). They are indexed by the language name. `Implementation_Suffix` is obsolescent and fully replaced by the first attribute.

For each language of a project, one of these two attributes needs to be specified, either in the project itself or in the configuration project file.

If the value of the attribute is the empty string, it indicates to the Project Manager that the only source files for the language are those specified with attributes `Body` or `Implementation_Exceptions`.

These attributes must satisfy the same requirements as `Spec_Suffix`. In addition, they must be different from any of the values in `Spec_Suffix`. If `Body_Suffix ("Ada")` is not specified, then the default is `".adb"`.

If `Body_Suffix ("Ada")` and `Spec_Suffix ("Ada")` end with the same string, then a file name that ends with the longest of these two suffixes will be a body if the longest suffix is `Body_Suffix ("Ada")`, or a spec if the longest suffix is `Spec_Suffix ("Ada")`.

If the suffix does not start with a `'.'`, a file with a name exactly equal to the suffix will also be part of the project (for instance if you define the suffix as `Makefile.in`, a file called `Makefile.in` will be part of the project. This capability is usually not of interest when building. However, it might become useful when a project is also used to find the list of source files in an editor, like the GNAT Programming System (GPS).

Separate_Suffix:

This attribute is specific to Ada. It denotes the suffix used in file names for files that contain subunits (separate bodies). If it is not specified, then it defaults to same value as `Body_Suffix ("Ada")`.

The value of this attribute cannot be the empty string.

Otherwise, the same rules apply as for the `Body_Suffix` attribute.

Spec or **Specification**:

These attributes are equivalent. The `Spec` attribute can be used to define the source file name for a given Ada compilation unit's spec. The index is the literal name of the Ada unit (case insensitive). The value is the literal base name of the file that contains this unit's spec (case sensitive or insensitive depending on the operating system). This attribute allows the definition of exceptions to the general naming scheme, in case some files do not follow the usual convention.

When a source file contains several units, the relative position of the unit can be indicated. The first unit in the file is at position 1.

```
for Spec ("MyPack.MyChild") use "mypack.mychild.spec";
for Spec ("top") use "foo.a" at 1;
for Spec ("foo") use "foo.a" at 2;
```

Body or **Implementation**:

These attribute play the same role as `Spec`, but for Ada bodies.

Specification_Exceptions and **Implementation_Exceptions**:

These attributes define exceptions to the naming scheme for languages other than Ada. They are indexed by the language name, and contain a list of file names respectively for headers and source code.

As an example of several of these attributes, the following package models the Apex file naming rules:

```
package Naming is
  for Casing               use "lowercase";
  for Dot_Replacement      use ".";
  for Spec_Suffix ("Ada")  use ".1.ada";
  for Body_Suffix ("Ada")  use ".2.ada";
end Naming;
```

2.3 Organizing Projects into Subsystems

A **subsystem** is a coherent part of the complete system to be built. It is represented by a set of sources and a single object directory. A system can consist of a single subsystem when it is simple as we have seen in the earlier examples. Complex systems are usually composed of several interdependent subsystems. A subsystem is dependent on another subsystem if knowledge of the other one is required to build it, and in particular if visibility on some of the sources of this other subsystem is required. Each subsystem is usually represented by its own project file.

In this section, we'll enhance the previous example. Let's assume some sources of our `Build` project depend on other sources. For instance, when building a graphical interface, it is usual to depend upon a graphical library toolkit such as GtkAda. Furthermore, we also need sources from a logging module we had previously written.

2.3.1 Importing Projects

GtkAda comes with its own project file (appropriately called `gtkada.gpr`), and we will assume we have already built a project called `logging.gpr` for the logging module. With the information provided so far in `build.gpr`, building the application would fail with an error indicating that the gtkada and logging units that are relied upon by the sources of this project cannot be found.

This is solved by defining `build.gpr` to *import* the gtkada and logging projects: this is done by adding the following `with` clauses at the beginning of our project:

```
with "gtkada.gpr";
with "a/b/logging.gpr";
project Build is
  ...  --  as before
end Build;
```

When such a project is compiled, *gprbuild* will automatically check the imported projects and recompile their sources when needed. It will also recompile the sources from *Build* when needed, and finally create the executable.

In some cases, the implementation units needed to recompile a project are not available, or come from some third party and you do not want to recompile it yourself. In this case, set the attribute **Externally_Built** to `"true"`, indicating to the builder that this project can be assumed to be up-to-date, and should not be considered for recompilation. In Ada, if the sources of this externally built project were compiled with another version of the compiler or with incompatible options, the binder will issue an error.

The project's `with` clause has several effects. It provides source visibility between projects during the compilation process. It also guarantees that the necessary object files from `Logging` and `GtkAda` are available when linking `Build`.

As can be seen in this example, the syntax for importing projects is similar to the syntax for importing compilation units in Ada. However, project files use literal strings instead of names, and the `with` clause identifies project files rather than packages.

Each literal string after `with` is the path (absolute or relative) to a project file. The `.gpr` extension is optional, but we recommend adding it. If no extension is specified, and no project file with the `.gpr` extension is found, then the file is searched for exactly as written in the `with` clause, that is with no extension.

As mentioned above, the path after a `with` has to be a literal string, and you cannot use concatenation, or lookup the value of external variables to change the directories from which a project is loaded. A solution if you need something like this is to use aggregate projects (see *Aggregate Projects*).

When a relative path or a base name is used, the project files are searched relative to each of the directories in the **project path**. This path includes all the directories found by the following procedure, in decreasing order of priority; the first matching file is used:

- First, the file is searched relative to the directory that contains the current project file.
- Then it is searched relative to all the directories specified in the environment variables `GPR_PROJECT_PATH_FILE`, `GPR_PROJECT_PATH` and `ADA_PROJECT_PATH` (in that order) if they exist. The value of `GPR_PROJECT_PATH_FILE`, when defined, is the path name of a text file that contains project directory path names, one per line. `GPR_PROJECT_PATH` and `ADA_PROJECT_PATH`, when defined, contain project directory path names separated by directory separators. `ADA_PROJECT_PATH` is used for compatibility, it is recommended to use `GPR_PROJECT_PATH_FILE` or `GPR_PROJECT_PATH`.
- Finally, it is searched relative to the default project directories. The following locations are searched, in the specified order:
  - `<compiler_prefix>/<target>/<runtime>/share/gpr`
  - `<compiler_prefix>/<target>/<runtime>/lib/gnat`
  - `<compiler_prefix>/<target>/share/gpr`
  - `<compiler_prefix>/<target>/lib/gnat`
  - `<compiler_prefix>/share/gpr/`
  - `<compiler_prefix>/lib/gnat/`

The first two paths are only added if the explicit runtime is specified either via `--RTS` switch or via `Runtime` attribute. <target> can be communicated via `--target` switch or `Target` attribute, otherwise default target will be used. <compiler_prefix> is typically discovered automatically based on target, runtime and language information.

In our example, `gtkada.gpr` is found in the predefined directory if it was installed at the same root as GNAT.

Some tools also support extending the project path from the command line, generally through the `-aP`. You can see the value of the project path by using the `gprls -v` command.

Any symbolic link will be fully resolved in the directory of the importing project file before the imported project file is examined.

Any source file in the imported project can be used by the sources of the importing project, transitively. Thus if *A* imports *B*, which imports *C*, the sources of *A* may depend on the sources of *C*, even if *A* does not import *C* explicitly. However, this is not recommended, because if and when *B* ceases to import *C*, some sources in *A* will no longer compile. *GPRbuild* has a switch `--no-indirect-imports` that will report such indirect dependencies.

Project import closure

The *project import closure* for a given project *proj* is the set of projects consisting of *proj* itself, together with each project that is directly or indirectly imported by *proj*. The import may be from either a `with` or, as will be explained below, a `limited with`.

Note: One very important aspect of a project import closure is that **a given source can only belong to one project** in this set (otherwise the project manager would not know which settings apply to it and when to recompile it). Thus different project files do not usually share source directories, or, when they do, they need to specify precisely which project owns which sources using the attribute *Source_Files* or equivalent. By contrast, two projects can each own a source with the same base file name as long as they reside in different directories. The latter is not true for Ada sources because of the correlation between source files and Ada units.

2.3.2 Cyclic Project Dependencies

In general, cyclic import dependencies are forbidden: if project *A* `with`s project *B* (directly or indirectly) then *B* is not allowed to `with` *A*. However, there are cases when cyclic dependencies would be beneficial. For these cases, another form of import between projects is supplied: the **limited with**. A project *A* that imports a project *B* with a simple `with` may also be imported, directly or indirectly, by *B* through a `limited with`.

The difference between a simple `with` and `limited with` is that the name of a project imported with a `limited with` cannot be used in the importing project. In particular, its packages cannot be renamed and its variables cannot be referenced.

```
with "b.gpr";
with "c.gpr";
project A is
    for Exec_Dir use B'Exec_Dir; -- OK
end A;

limited with "a.gpr";   --  Cyclic dependency: A -> B -> A
project B is
   for Exec_Dir use A'Exec_Dir; -- not OK
end B;

with "d.gpr";
project C is
end C;

limited with "a.gpr";  --  Cyclic dependency: A -> C -> D -> A
project D is
   for Exec_Dir use A'Exec_Dir; -- not OK
end D;
```

2.3.3 Sharing between Projects

When building an application, it is common to have similar needs in several of the projects corresponding to the subsystems under construction. For instance, they might all have the same compilation switches.

As seen above (see *Tools Options in Project Files*), setting compilation switches for all sources of a subsystem is simple: it is just a matter of adding a `Compiler'Default_Switches` attribute to each project file with the same value. However, that would entail duplication of data, and both places would need to be changed in order to recompile the whole application with different switches. This may be a serious issue if there are many subsystems and thus many project files to edit.

There are two main approaches to avoiding this duplication:

- Since `build.gpr` imports `logging.gpr`, we could change the former to reference the attribute in Logging, either through a package renaming, or by referencing the attribute. The following example shows both cases:

```
project Logging is
   package Compiler is
      for Switches ("Ada")
          use ("-O2");
   end Compiler;
   package Binder is
      for Switches ("Ada")
          use ("-E");
   end Binder;
end Logging;

with "logging.gpr";
project Build is
   package Compiler renames Logging.Compiler;
   package Binder is
      for Switches ("Ada") use Logging.Binder'Switches ("Ada");
   end Binder;
end Build;
```

The solution used for *Compiler* gets the same value for all attributes of the package, but you cannot modify anything from the package (adding extra switches or some exceptions). The solution for the *Binder* package is more flexible, but more verbose.

If you need to refer to the value of a variable in an imported project, rather than an attribute, the syntax is similar but uses a `"."` rather than an apostrophe. For instance:

```
with "imported";
project Main is
   Var1 := Imported.Var;
end Main;
```

- The second approach is to define the switches in a separate project. That project does not contain any source files (thus, as opposed to the first example, none of the projects plays a special role), and will only be used to define the attributes. Such a project is typically named `shared.gpr`.

```
abstract project Shared is
   for Source_Files use ();   --  no sources
   package Compiler is
      for Switches ("Ada")
          use ("-O2");
   end Compiler;
end Shared;

with "shared.gpr";
project Logging is
   package Compiler renames Shared.Compiler;
end Logging;

with "shared.gpr";
project Build is
   package Compiler renames Shared.Compiler;
end Build;
```

As with the first example, we could have chosen to set the attributes one by one rather than to rename a package. The reason we explicitly indicate that *Shared* has no sources is so that it can be created in any directory, and we are sure it shares no sources with *Build* or *Logging*, which would be invalid.

Note the additional use of the **abstract** qualifier in `shared.gpr`. This qualifier is optional, but helps convey the message that we do not intend this project to have source files (see *Qualified Projects* for additional information about project qualifiers).

2.3.4 Global Attributes

We have already seen many examples of attributes used to specify a particular option for one of the tools involved in the build process. Most of those attributes are project specific. That is to say, they only affect the invocation of tools on the sources of the project where they are defined.

There are a few additional attributes that, when defined for a "main" project *proj*, also apply to all other projects in the project import closure of *proj*. A *main project* is a project explicitly specified on the command line.

Such attributes are known as *global attributes*; here are several that are commonly used:

Builder'Global_Configuration_Pragmas:

This attribute specifies a file that contains configuration pragmas to use when building executables. These pragmas apply to all executables built from this project import closure. As noted earlier, additional pragmas can be specified on a per-project basis by setting the `Compiler'Local_Configuration_Pragmas` attribute.

Builder'Global_Compilation_Switches:

This attribute is a list of compiler switches that apply when compiling any source file in the project import closure. These switches are used in addition to the ones defined in the `Compiler` package, which only apply to the sources of the corresponding project. This attribute is indexed by the name of the language.

Using such global capabilities is convenient, but care is needed since it can also lead to unexpected behavior. An example is when several subsystems are shared among different main projects but the different global attributes are not compatible. Note that using aggregate projects can be a safer and more powerful alternative to global attributes.

2.4 Scenarios in Projects

Various project properties can be modified based on **scenarios**. These are user-defined modes (the values of project variables and attributes) that determine the behavior of a project, based on the values of externally defined variables. Typical examples are the setup of platform-specific compiler options, or the use of a debug and a release mode (the former would activate the generation of debug information, while the latter would request an increased level of code optimization).

Let's enhance our example to support debug and release modes. The issue is to let the user choose which kind of system to build: use `-g` as a compiler switch in debug mode and `-O2` in release mode. We will also set up the projects so that we do not share the same object directory in both modes; otherwise switching from one to the other might trigger more recompilations than needed or mix objects from the two modes.

One approach is to create two different project files, say `build_debug.gpr` and `build_release.gpr`, that set the appropriate attributes as explained in previous sections. This solution does not scale well, because in the presence of multiple projects depending on each other, you will also have to duplicate the complete set of projects and adapt the project files accordingly.

Instead, project files support the notion of scenarios controlled by the values of externally defined variables. Such values can come from several sources (in decreasing order of priority):

Command line: When launching *gprbuild*, the user can pass `-X` switches to define the external variables. In our case, the command line might look like

```
gprbuild -Pbuild.gpr -Xmode=release
```

which defines the external variable named `mode` and sets its value to `"release"`.

Environment variables: When the external value does not come from the command line, it can come from the value of an environment variable of the appropriate name. In our case, if an environment variable named `mode` exists, its value will be used.

External function second parameter. Once an external variable is defined, its value needs to be obtained by the project. The general form is to use the predefined function `external`, which returns the current value of the external variable. For instance, we could set up the object directory to point to either `obj/debug` or `obj/release` by changing our project to

```
project Build is
    for Object_Dir use "obj/" & external ("mode", "debug");
    ... --  as before
end Build;
```

The second parameter to `external` is optional, and is the default value to use if `mode` is not set from the command line or the environment. If the second parameter is not supplied, and there is no external or environment variable named by the first parameter, then an error is reported.

In order to set the switches according to the different scenarios, other constructs are needed, such as typed variables and case constructions.

A **typed variable** is a variable that can take only a limited number of values, similar to variable from an enumeration type in Ada. Such a variable can then be used in a **case construction**, resulting in conditional sections in the project. The following example shows how this can be done:

```
project Build is
   type Mode_Type is ("debug", "release");          -- all possible values
   Mode : Mode_Type := external ("mode", "debug"); -- a typed variable

   package Compiler is
      case Mode is
         when "debug" =>
            for Switches ("Ada")
                use ("-g");
         when "release" =>
            for Switches ("Ada")
                use ("-O2");
      end case;
   end Compiler;
end Build;
```

This project is larger than the ones we have seen previously, but it has become much more flexible. The `Mode_Type` type defines the only valid values for the `Mode` variable. If any other value is read from the environment, an error is reported and the project is considered as invalid.

The `Mode` variable is initialized with an external value defaulting to `"debug"`. This default could be omitted and that would force the user to define the value. Finally, we can use a case construction to set the switches depending on the scenario the user has chosen.

Most aspects of a project can depend on scenarios. The notable exception is the identity of an imported project (via a `with` or `limited with` clause), which cannot depend on a scenario.

Scenarios work analogously across projects in a project import closure. You can either duplicate a variable similar to

`Mode` in each of the projects (as long as the first argument to `external` is always the same and the type is the same), or simply set the variable in the `shared.gpr` project (see *Sharing between Projects*).

2.5 Library Projects

So far, we have seen examples of projects that create executables. However, it is also possible to create libraries instead. A **library** is a specific type of subsystem where, for convenience, objects are grouped together using system-specific means such as archives or Windows DLLs.

Library projects provide a system- and language-independent way of building both **static** and **dynamic** libraries. They also support the concept of **standalone libraries** (SAL) which offer two significant properties: the elaboration (e.g. initialization) of the library is either automatic or very simple; a change in the implementation part of the library implies minimal post-compilation actions on the complete system and potentially no action at all for the rest of the system in the case of dynamic SALs.

There is a restriction on shared library projects: by default, they are only allowed to import other shared library projects. They are not allowed to import non-library projects or static library projects.

The GNAT Project Manager takes complete care of the library build, rebuild and installation tasks, including recompilation of the source files for which objects do not exist or are not up to date, assembly of the library archive, and installation of the library (i.e., copying associated source, object and `ALI` files to the specified location).

2.5.1 Building Libraries

Let's enhance our example and transform the *logging* subsystem into a library. In order to do so, a few changes need to be made to `logging.gpr`. Some attributes need to be defined: at least *Library_Name* and *Library_Dir*; in addition, some other attributes can be used to specify specific aspects of the library. For readability, it is also recommended (although not mandatory), to use the qualifier *library* in front of the *project* keyword.

Library_Name:

> This attribute is the name of the library to be built. There is no restriction on the name of a library imposed by the project manager, except for stand-alone libraries whose names must follow the syntax of Ada identifiers; however, there may be system-specific restrictions on the name. In general, we recommend using only alphanumeric characters (and possibly single underscores), to help portability.

Library_Dir:

> This attribute is the path (absolute or relative) of the directory where the library is to be installed. In the process of building a library, the sources are compiled and the object files are placed in the explicitly- or implicitly specified `Object_Dir` directory. When all sources of a library are compiled, some of the compilation artifacts, including the library itself, are copied to the library_dir directory. This directory must exist and be writable. It must also be different from the object directory so that cleanup activities in the Library_Dir do not affect recompilation needs.

Here is the new version of `logging.gpr` that makes it a library:

```
library project Logging is          --  "library" is optional
   for Library_Name use "logging";  --  will create "liblogging.a" on Unix
   for Object_Dir   use "obj";
   for Library_Dir  use "lib";      --  different from object_dir
end Logging;
```

Once the above two attributes are defined, the library project is valid and is sufficient for building a library with default characteristics. Other library-related attributes can be used to change the defaults:

Library_Kind:

The value of this attribute must be either `"static"`, `"static-pic"`, `"dynamic"` or `"relocatable"` (the last is a synonym for `"dynamic"`). It indicates which kind of library should be built (the default is to build a static library, that is an archive of object files that can potentially be linked into a static executable). A static-pic library is also an archive, but the code is Position Independent Code, usually compiled with the switch -fPIC. When the library is set to be dynamic, a separate image is created that will be loaded independently, usually at the start of the main program execution. Support for dynamic libraries is very platform specific, for instance on Windows it takes the form of a DLL while on GNU/Linux, it is a dynamic *elf* image whose suffix is usually `.so`. Library project files, on the other hand, can be written in a platform independent way so that the same project file can be used to build a library on different operating systems.

If you need to build both a static and a dynamic library, we recommend using two different object directories, since in some cases some extra code needs to be generated for the latter. For such cases, one can either define two different project files, or a single one that uses scenarios to indicate the various kinds of library to be built and their corresponding object_dir.

Library_ALI_Dir:

This attribute may be specified to indicate the directory where the ALI files of the library are installed. By default, they are copied into the `Library_Dir` directory, but as for the executables where we have a separate *Exec_Dir* attribute, you might want to put them in a separate directory since there may be hundreds of such files. The same restrictions as for the `Library_Dir` attribute apply.

Library_Version:

This attribute is platform dependent, and has no effect on Windows. On Unix, it is used only for dynamic libraries as the internal name of the library (the "soname"). If the library file name (built from the `Library_Name`) is different from the `Library_Version`, then the library file will be a symbolic link to the actual file whose name will be `Library_Version`. This follows the usual installation schemes for dynamic libraries on many Unix systems.

```
project Logging is
   Version := "1";
   for Library_Dir use "lib";
   for Library_Name use "logging";
   for Library_Kind use "dynamic";
   for Library_Version use "liblogging.so." & Version;
end Logging;
```

After the compilation, the directory `lib` will contain both a `liblogging.so.1` library and a symbolic link to it called `liblogging.so`.

Library_GCC:

This attribute is the name of the tool to use instead of `gcc` to link shared libraries. A common use of this attribute is to define a wrapper script that accomplishes specific actions before calling `gcc` (which itself calls the linker to build the library image).

Library_Options:

This attribute may be used to specify additional switches ("last switches") when linking a shared library.

It may also be used to add foreign object files to a static library. Each string in `Library_Options` is an absolute or relative path of an object file. When a relative path, it is relative to the object directory.

Leading_Library_Options:

This attribute, which is taken into account only by *GPRbuild*, may be used to specify leading options ("first switches") when linking a shared library.

2.5.2 Using Library Projects

When the builder detects that a project file is a library project file, it recompiles all sources of the project that need recompilation and rebuilds the library if any of the sources have been recompiled. It then groups all object files into a single file, which is a shared or a static library. This library can later on be linked with multiple executables. Note that the use of shared libraries reduces the size of the final executable and can also reduce the memory footprint at execution time when the library is shared among several executables.

GPRbuild also allows building **multi-language libraries** when specifying sources from multiple languages.

A non-library project *NLP* can import a library project *LP*. When the builder is invoked on *NLP*, it always rebuilds *LP* even if all of the latter's files are up to date. For instance, let's assume in our example that `logging` has the following sources: `log1.ads`, `log1.adb`, `log2.ads` and `log2.adb`. If `log1.adb` has been modified, then the library `liblogging` will be rebuilt when compiling all the sources of `Build` even if `proc.ads`, `pack.ads` and `pack.adb` do not include a `"with Log1"`.

To ensure that all the sources in the `Logging` library are up to date, and that all the sources of `Build` are also up to date, the following two commands need to be used:

```
gprbuild -Plogging.gpr
gprbuild -Pbuild.gpr
```

All `ALI` files will also be copied from the object directory to the library directory. To build executables, *GPRbuild* will use the library rather than the individual object files.

Library projects can also be useful to specify a library that needs to be used but, for some reason, cannot be rebuilt. Such a situation may arise when some of the library sources are not available. Such library projects need to use the `Externally_Built` attribute as in the example below:

```
library project Extern_Lib is
   for Languages    use ("Ada", "C");
   for Source_Dirs  use ("lib_src");
   for Library_Dir  use "lib2";
   for Library_Kind use "dynamic";
   for Library_Name use "l2";
   for Externally_Built use "true";  --  <<<<
end Extern_Lib;
```

In the case of externally built libraries, the `Object_Dir` attribute does not need to be specified because it will never be used.

The main effect of using such an externally built library project is mostly to affect the linker command in order to reference the desired library. It can also be achieved by using `Linker'Linker_Options` or `Linker'Switches` in the project corresponding to the subsystem needing this external library. This latter method is more straightforward in simple cases but when several subsystems depend upon the same external library, finding the proper place for the `Linker'Linker_Options` might not be easy and if it is not placed properly, the final link command is likely to present ordering issues. In such a situation, it is better to use the externally built library project so that all other subsystems depending on it can declare this dependency through a project `with` clause, which in turn will trigger the builder to find the proper order of libraries in the final link command.

2.5.3 Stand-alone Library Projects

A **stand-alone library** is a library that contains the necessary code to elaborate the Ada units that are included in the library. A stand-alone library is a convenient way to add an Ada subsystem to a more global system whose main is not in Ada since it makes the elaboration of the Ada part mostly transparent. However, stand-alone libraries are also

useful when the main is in Ada: they provide a means for minimizing relinking and redeployment of complex systems when localized changes are made.

The name of a stand-alone library, specified with attribute `Library_Name`, must have the syntax of an Ada identifier.

The most prominent characteristic of a stand-alone library is that it offers a distinction between interface units and implementation units. Only the former are visible to units outside the library. A stand-alone library project is thus characterized by a third attribute, usually `Library_Interface`, in addition to the two attributes that make a project a Library Project (*Library_Name* and *Library_Dir*). This third attribute may also be `Interfaces`. `Library_Interface` only works when the interface is in Ada and takes a list of units as parameter. `Interfaces` works for any supported language and takes a list of sources as parameter.

Library_Interface:

This attribute defines an explicit subset of the units of the project. Units from projects importing this library project may only "with" units whose sources are listed in the *Library_Interface*. Other sources are considered implementation units.

```
for Library_Dir use "lib";
for Library_Name use "logging";
for Library_Interface use ("lib1", "lib2");  --  unit names
```

Interfaces

This attribute defines an explicit subset of the source files of a project. Sources from projects importing this project, can only depend on sources from this subset. This attribute can be used on non library projects. It can also be used as a replacement for attribute `Library_Interface`, in which case, units have to be replaced by source files. For multi-language library projects, it is the only way to make the project a Stand-Alone Library project whose interface is not purely Ada.

Library_Standalone:

This attribute defines the kind of stand-alone library to build. Values are either `standard` (the default), `no` or `encapsulated`. When `standard` is used the code to elaborate and finalize the library is embedded, when `encapsulated` is used the library can furthermore depend only on static libraries (including the GNAT runtime). This attribute can be set to `no` to make it clear that the library should not be stand-alone in which case the `Library_Interface` should not defined. Note that this attribute only applies to shared libraries, so `Library_Kind` must be set to *dynamic* or *relocatable*.

```
for Library_Dir use "lib";
for Library_Name use "logging";
for Library_Kind use "dynamic";
for Library_Interface use ("lib1", "lib2");  --  unit names
for Library_Standalone use "encapsulated";
```

In order to include the elaboration code in the stand-alone library, the binder is invoked on the closure of the library units creating a package whose name depends on the library name (`b~logging.ads/b` in the example). This binder-generated package includes **initialization** and **finalization** procedures whose names depend on the library name (`logginginit` and `loggingfinal` in the example). The object corresponding to this package is included in the library.

Library_Auto_Init:

A dynamic stand-alone Library is automatically initialized if automatic initialization of stand-alone Libraries is supported on the platform and if attribute `Library_Auto_Init` is not specified or is specified with the value `"true"`. Whether a static stand-alone Library is automatically initialized is platform dependent. Specifying `"false"` for the `Library_Auto_Init` attribute prevents automatic initialization.

When a non-automatically initialized stand-alone library is used in an executable, its initialization procedure must be called before any service of the library is used. When the main subprogram is in Ada, it may mean that the initialization procedure has to be called during elaboration of another package.

Library_Dir:

For a stand-alone library, only the `ALI` files of the interface units (those that are listed in attribute *Library_Interface*) are copied to the library directory. As a consequence, only the interface units may be imported from Ada units outside of the library. If other units are imported, the binding phase will fail.

Binder'Default_Switches:

When a stand-alone library is bound, the switches that are specified in the attribute `Binder'Default_Switches ("Ada")` are used in the call to *gnatbind*.

Library_Src_Dir:

This attribute defines the location (absolute or relative to the project directory) where the sources of the interface units are copied at installation time. These sources includes the specs of the interface units along with the closure of sources necessary to compile them successfully. That may include bodies and subunits, when pragmas *Inline* are used, or when there are generic units in specs. This directory cannot point to the object directory or one of the source directories, but it can point to the library directory, which is the default value for this attribute.

Library_Symbol_Policy:

This attribute controls the export of symbols on some platforms (like Windows, GNU/Linux). It is not supported on all platforms (where it will just have no effect). It may have one of the following values:

- `"restricted"`: The exported symbols will be restricted to the one from the interface of the stand-alone library. This is either computed automatically or using the `Library_Symbol_File` if specified.
- `"unrestricted"`: All symbols from the stand-alone library are exported.

Library_Symbol_File

This attribute may define the name of the symbol file to be used when building a stand-alone library when the symbol policy is `"restricted"`, on platforms that support symbol control. This file must contain one symbol per line and only those symbols will be exported from the stand-alone library.

2.5.4 Installing a Library with Project Files

When using project files, a usable version of the library is created in the directory specified by the `Library_Dir` attribute of the library project file. Thus no further action is needed in order to make use of the libraries that are built as part of the general application build.

You may want to install a library in a context different from where the library is built. This situation arises with third party suppliers, who may want to distribute a library in binary form where the user is not expected to be able to recompile the library. The simplest option in this case is to provide a project file slightly different from the one used to build the library, by using the `Externally_Built` attribute. See *Using Library Projects*.

Another option is to use *gprinstall* to install the library in a different context than the build location. The *gprinstall* tool automatically generates a project to use this library, and also copies the minimum set of sources needed to use the library to the install location. See *Package Install Attributes*.

2.6 Project Extension

During development of a large system, it is sometimes necessary to use modified versions of some of the source files, without changing the original sources. This can be achieved through the *project extension* facility.

Suppose that our example `Build` project is built every night for the whole team, in some shared directory. A developer usually needs to work on a small part of the system, and might not want to have a copy of all the sources and all the object files since that could require too much disk space and too much time to recompile everything. A better approach is to override some of the source files in a separate directory, while still using the object files generated at night for the non-overridden shared sources.

Another use case is a large software system with multiple implementations of a common interface; in Ada terms, multiple versions of a package body for the same spec, or perhaps different versions of a package spec that have the same visible part but different private parts. For example, one package might be safe for use in tasking programs, while another might be used only in sequential applications.

A third example is different versions of the same system. For instance, assume that a `Common` project is used by two development branches. One of the branches has now been frozen, and no further change can be done to it or to `Common`. However, on the other development branch the sources in `Common` are still evolving. A new version of the subsystem is needed, which reuses as much as possible from the original.

Each of these can be implemented in GNAT using **project extension**:

> If one project *extends* another project (the *base project*) then by default all source files of the base project are inherited by the extending project, but the latter can override any of the base project's source files with a new version, and can also add new files or remove unnecessary ones. A project can extend at most one base project.

This facility is somewhat analogous to class extension (with single inheritance) in object-oriented programming. Project extension hierarchies are permitted (an extending project may itself serve as a base project and be extended), and a project that extends a project can also import other projects.

An extending project implicitly inherits all the sources and objects from its base project. It is possible to create a new version of some of the sources in one of the additional source directories of the extending project. Those new versions hide the original versions. As noted above, adding new sources or removing existing ones is also possible. Here is an example of how to extend the project *Build* from previous examples:

```
project Work extends "../bld/build.gpr" is
end Work;
```

The project after the `extends` keyword is the base project being extended. As usual, it can be specified using an absolute path, or a path relative to any of the directories in the project path. The `Work` project does not specify source or object directories, so the default values for these attributes will be used; that is, the current directory (where project `Work` is placed). We can compile that project with

```
gprbuild -Pwork
```

If no sources have been placed in the current directory, this command has no effect, since this project does not change the sources it inherited from `Build` and thus all the object files in `Build` and its dependencies are still valid and are reused automatically.

Suppose we now want to supply an alternative version of `pack.adb` but use the existing versions of `pack.ads` and `proc.adb`. We can create the new file in the `Work` project's directory (for example by copying the one from the `Build` project and making changes to it). If new packages are needed at the same time, we simply create new files in the source directory of the extending project.

When we recompile, *GPRbuild* will now automatically recompile this file (thus creating `pack.o` in the current directory) and any file that depends on it (thus creating `proc.o`). Finally, the executable is also linked locally.

Note that we could have obtained the desired behavior using project import rather than project inheritance. Some project `proj` would contain the sources for `pack.ads` and `proc.adb`, and `Work` would import `proj` and add `pack.adb`. In this situation `proj` cannot contain the original version of `pack.adb` since otherwise two versions of the same unit would be in project import closure of `proj`, which is not allowed. In general we do not recommended placing the spec and body of a unit in different projects, since this affects their autonomy and reusability.

In a project file that extends another project, it is possible to indicate that an inherited source is **not part** of the sources of the extending project. This is necessary, for example, when a package spec has been overridden in such a way that a body is forbidden. In this case, it is necessary to indicate that the inherited body is not part of the sources of the project, otherwise there will be a compilation error.

Two attributes are available for this purpose:

- **Excluded_Source_Files**, whose value is a list of file names, and
- **Excluded_Source_List_File**, whose value is the path of a text file containing one file name per line.

```
project Work extends "../bld/build.gpr" is
   for Source_Files use ("pack.ads");
   --  New spec of Pkg does not need a completion
   for Excluded_Source_Files use ("pack.adb");
end Work;
```

All tool packages that are not declared in the extending project are inherited from the base project, with their attributes, with the exception of `Linker'Linker_Options` which is never inherited. In particular, an extending project retains all the switches specified in its base project.

At the project level, if they are not declared in the extending project, some attributes are inherited from the base project. They are: `Languages`, `Main` (for a root non library project) and `Library_Name` (for a project extending a library project).

2.6.1 Importing and Project Extension

One of the fundamental restrictions for project extension is the following:

> **A project is not allowed to import, directly or indirectly, both an extending project P and also some project that P extends either directly or indirectly**

In the absence of this rule, two imports might access different versions of the same source file, or different sets of tool switches for the same source file (one from the base project and the other from an extending project).

As an example of this problem, consider the following set of project files:

- `a.gpr` which contains the source files `foo.ads` and `foo.adb`, among others
- `b.gpr` which imports `a.gpr` (one of its source files `withs foo`)
- `c.gpr` which imports `b.gpr`

Suppose we want to extend the projects as follows:

- `a_ext.gpr` extends `a.gpr` and overrides `foo.adb`
- `c_ext.gpr` extends `c.gpr`, overriding one of its source files

Since `c_ext.gpr` needs to access sources in `b.gpr`, it will import `b.gpr`

Finally, `main.gpr` needs to access the overridden source files in `a_ext.gpr` and `c_ext.gpr` and thus will import these two projects.

This project structure is shown in figure 2.1.

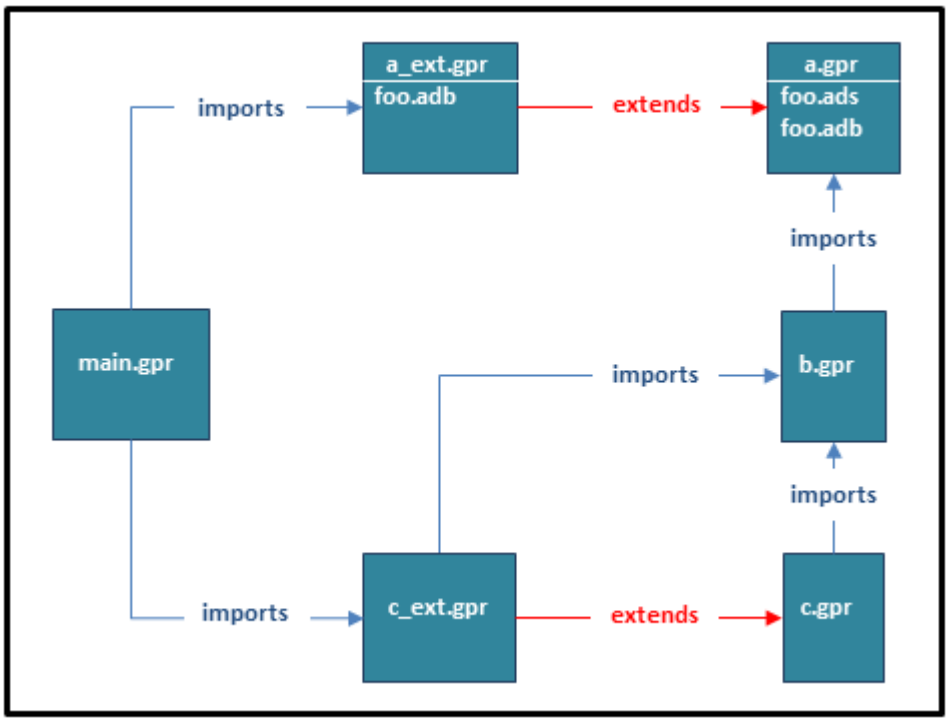

Figure 2.1: Example of Source File Ambiguity from *imports/extends* Violation

This violates the restriction above, since `main.gpr` imports the extending project `a_ext.gpr` and also (indirectly through `c_ext.gpr` and `b.gpr`) the project `a.gpr` that `a_ext.gpr` extends. The problem is that the import path through `c_ext.gpr` and `b.gpr` would build with the version of `foo.adb` from `a.gpr`, whereas the import path through `a_ext.gpr` would use that project's version of `foo.adb`. The error will be detected and reported by `gprbuild`.

A solution is to introduce an "empty" extension of `b.gpr`, which is imported by `c_ext.gpr` and imports `a_ext.gpr`:

```
with "a_ext.gpr";
project B_Ext extends "b.gpr" is
end B_Ext;
```

This project structure is shown in figure 2.2.

There is now no ambiguity over which version of `foo.adb` to use; it will be the one from `a_ext.gpr`.

When extending a large system spanning multiple projects, it is often inconvenient to extend every project in the project import closure that is impacted by a small change introduced in a low layer. In such cases, it is possible to create an **implicit extension** of an entire hierarchy using the **extends all** relationship.

When a project `P` is extended using *extends all* inheritance, all projects that are imported by `P`, both directly and indirectly, are considered virtually extended. That is, the project manager creates implicit projects that extend every project in the project import closure; all these implicit projects do not control sources on their own and use the object directory of the *extends all* project.

It is possible to explicitly extend one or more projects in the import closure in order to adapt the sources. These

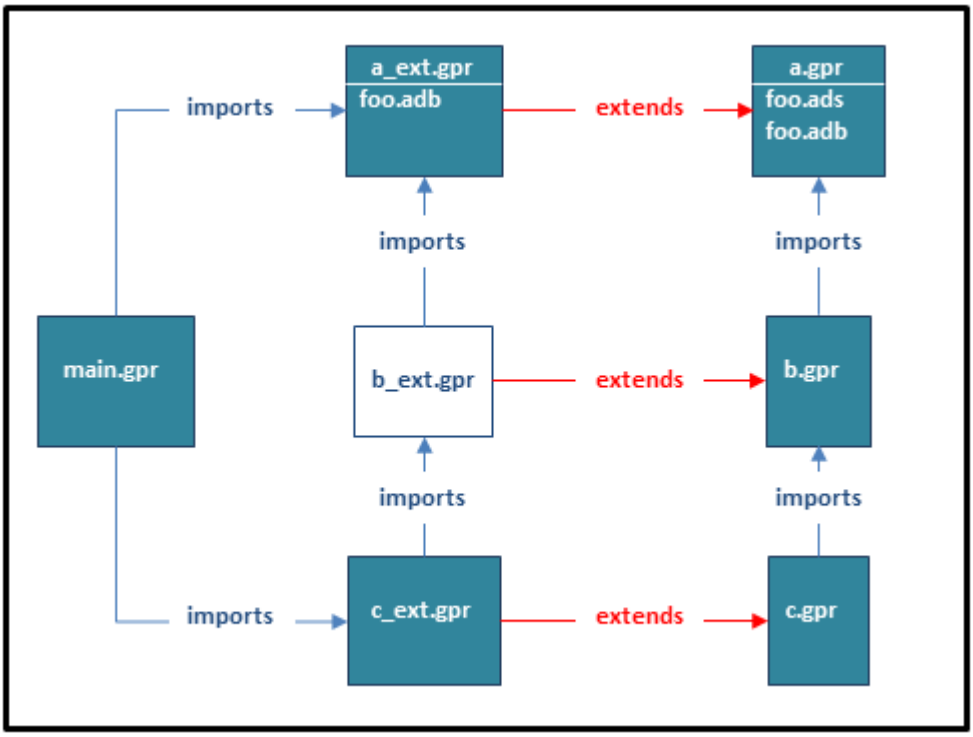

Figure 2.2: Using "Empty" Project Extension to Avoid *imports*/*extends* Violation

extending projects must be imported by the `extends all` project, which will replace the corresponding virtual projects with the explicit ones.

When building such a project closure extension, the project manager will ensure recompilation of both the modified sources and the sources in implicit extending projects that depend on them.

To illustrate the `extends all` feature, here's a slight variation on the earlier examples. We have a `Main` project that imports project `C`, which imports `B`, which imports `A`. The source files in `Main` refer to compilation units whose sources are in `C` and `A`. (Recall that `imports` is transitive, so `A` is implicitly accessible in `Main`.)

This project structure is shown in figure 2.3.

Suppose that we want to extend `a.gpr`, overriding one of its source files, and create a new version of `main.gpr` that can access the overridden file in the extending project `a_ext.gpr` and otherwise use the sources in `b.gpr` and `c.gpr`.

Instead of explicitly defining empty projects to extend `b.gpr` and `c.gpr`, we can create a new project `main_ext.gpr` that does an `extends all` of `main.gpr` and imports `a_ext.gpr`. The `extends_all` will implicitly create the empty projects `b_ext.gpr` and `c_ext.gpr` as well as the relevant import relationships:

- `c_ext.gpr` will import `b_ext.gpr`, which will import `a_ext.gpr`
- `main_ext.gpr` will implicitly import `c_ext.gpr` since `main.gpr` imports `c.gpr`.

The resulting project structure is shown in figure 2.4, where the italicized labels, dashed arrows, and dashed boxes indicate what was added implicitly as an effect of the `extends_all`.

When project `main_ext.gpr` is built, the entire modified project space is considered for recompilation, including the sources from `b.gpr` and `c.gpr` that are affected by the changes to `a.gpr`.

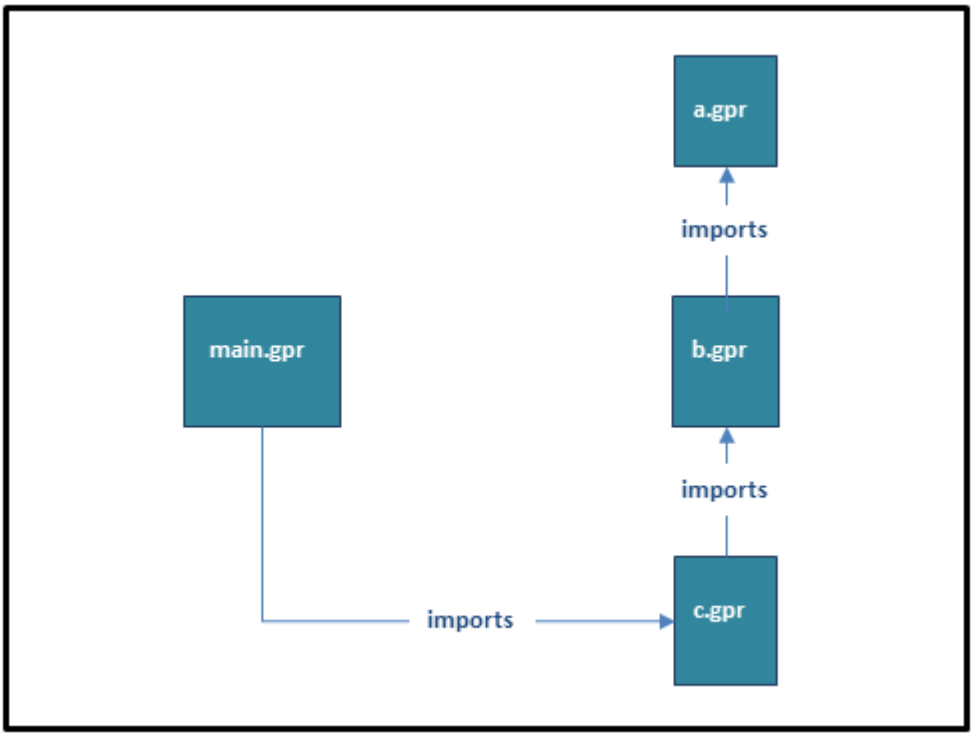

Figure 2.3: Simple Project Structure before Extension

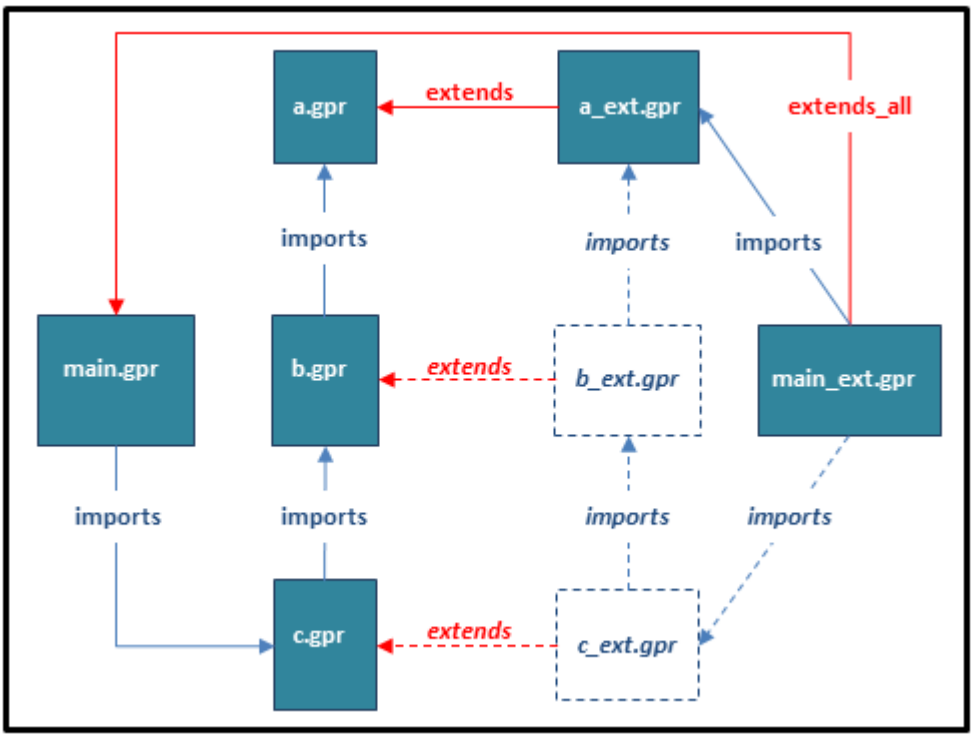

Figure 2.4: Project Structure with `extends_all`

2.7 Child Projects

In order to more clearly express the relationship between a project `Q` and some other project `P` that `Q` either imports or extends, you can use the notation `P.Q` to declare `Q` as a **child** of `P`. The project `P` is then referred to as the **parent** of `Q`. This is useful, for example, when the purpose of the child is to serve as a testing subsystem for the parent.

The visibility of the child on the sources and other properties of the parent is determined by whether the child imports or extends the parent. No additional visibility is obtained by declaring the project as a child; the *parent.child* notation serves solely as a naming convention to convey to the reader the closeness of the relationship between the projects.

For example:

```
-- math_proj.gpr
project Math_Proj is
   ...
end Math_Proj;

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

with "math_proj.gpr";
project Math_Proj.Tests is      -- Legal; child imports parent
   ...
end Math_Proj.Tests;

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

project Math_Proj.High_Performance
   extends "math_proj.gpr" is   -- Legal; child extends parent
   ...
end Math_Proj.High_Performance;

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

project GUI_Proj.Tests is       -- Illegal
   ...
end GUI_Proj.Tests;
```

Child projects may in turn be the parents of other projects, so in general a project hierarchy can be created. A project may be the parent of many child projects, but a child project can only have one parent.

Note that child projects have slightly different semantics from their Ada language analog (child units). An Ada child unit implicitly `with`s its parent, whereas a child project must have an explicit `with` clause (or else extend its parent). The need to explicitly `with` or extend the parent project helps avoid the error of unintentionally creating a child of some project that happens to be on the project path.

2.8 Aggregate Projects

Aggregate projects are an extension of the project paradigm, and are designed to handle a few specific situations that cannot be solved directly using standard projects. This section will present several such use cases.

2.8.1 Building all main programs from a single project closure

A large application is typically organized into modules and submodules, which are conveniently represented as a project graph (the project import closure): a "root" project *A* `with`s the projects for modules *B* and *C*, which in turn

`with` projects for submodules.

Very often, modules will build their own executables (for testing purposes for instance) or libraries (for easier reuse in various contexts).

However, if you build your project through *GPRbuild*, using a syntax similar to

```
gprbuild -PA.gpr
```

this will only rebuild the main programs of project A, not those of the imported projects B and C. Therefore you have to spawn several *GPRbuild* commands, one per project, to build all executables. This is somewhat inconvenient, but more importantly is inefficient because *GPRbuild* needs to do duplicate work to ensure that sources are up-to-date, and cannot easily compile things in parallel when using the `-j` switch.

Also, libraries are always rebuilt when building a project.

To solve this problem you can define an *aggregate project* `Agg` that groups `A`, `B` and `C`:

```
aggregate project Agg is
   for Project_Files use ("a.gpr", "b.gpr", "c.gpr");
end Agg;
```

Then, when you build with

```
gprbuild -PAgg.gpr
```

this will build all main programs from `A`, `B` and `C`.

If *B* or *C* do not define any main program (through their *Main* attribute), all their sources are built. When you do not group them in an aggregate project, only those sources that are needed by *A* will be built.

If you add a main to a project `P` not already explicitly referenced in the aggregate project, you will need to add `p.gpr` in the list of project files for the aggregate project, or the main will not be built when building the aggregate project.

2.8.2 Building a set of projects with a single command

Another application of aggregate projects is when you have multiple applications and libraries that are built independently (but can be built in parallel). For instance, you might have a project graph rooted at `A`, and another one (which might share some subprojects) rooted at `B`.

Using only *GPRbuild*, you could do

```
gprbuild -PA.gpr
gprbuild -PB.gpr
```

to build both. But again, *GPRbuild* has to do some duplicate work for those files that are shared between the two, and cannot truly build things in parallel efficiently.

If the two projects are really independent, share no sources other than through a common subproject, and have no source files with a common basename, you could create a project `C` that imports `A` and `B`. But these restrictions are often too strong, and one has to build them independently. An aggregate project does not have these limitations and can aggregate two project graphs that have common sources:

```
aggregate project Agg is
   for Project_Files use ("a.gpr", "b.gpr");
end Agg;
```

This scenario is particularly useful in environments like VxWorks 653 where the applications running in the multiple partitions can be built in parallel through a single *GPRbuild* command. This also works well with Annex E of the Ada Language Reference Manual.

2.8.3 Defining a build environment

The environment variables at the time you launch *GPRbuild* will influence the view these tools have of the project (for example `PATH` to find the compiler, `ADA_PROJECT_PATH` or `GPR_PROJECT_PATH` to find the projects, and environment variables that are referenced in project files through the `external` built-in function). Several command line switches can be used to override those (`-X` or `-aP`), but on some systems and with some projects, this might make the command line too long, and on all systems often make it hard to read.

An aggregate project can be used to set the environment for all projects built through that aggregate. One of the benefits is that you can put the aggregate project under configuration management, and make sure all your users have a consistent environment when building. For example:

```
aggregate project Agg is
   for Project_Files use ("A.gpr", "B.gpr");
   for Project_Path use ("../dir1", "../dir1/dir2");
   for External ("BUILD") use "PRODUCTION";

   package Builder is
      for Global_Compilation_Switches ("Ada") use ("-g");
   end Builder;
end Agg;
```

Another use of aggregate projects is to simulate the referencing of external variables in `with` clauses, For technical reasons the following project file is not allowed:

```
with external("SETUP") & "path/prj.gpr";    --  ILLEGAL
project MyProject is
   ...
end MyProject;
```

However, you can use aggregate projects to obtain an equivalent effect:

```
aggregate project Agg is
    for Project_Path use (external("SETUP") & "path");
    for Project_Files use ("myproject.gpr");
end Agg;
```

```
with "prj.gpr";  --  searched on Agg'Project_Path
project MyProject is
   ...
end MyProject;
```

2.8.4 Improving builder performance

The loading of aggregate projects is optimized in *GPRbuild*, so that all files are searched for only once on the disk (thus reducing the number of system calls and yielding faster compilation times, especially on systems with sources on remote servers). As part of the loading, *GPRbuild* computes how and where a source file should be compiled, and even if it is located several times in the aggregated projects it will be compiled only once.

Since there is no ambiguity as to which switches should be used, files can be compiled in parallel (through the usual `-j` switch) and this can be done while maximizing the use of CPUs (compared to launching multiple *GPRbuild* commands in parallel).

2.8.5 Syntax of aggregate projects

An aggregate project follows the general syntax of project files. The recommended extension is still `.gpr`. However, a special `aggregate` qualifier must appear before the keyword `project`.

An aggregate project cannot `with` any other project (standard or aggregate), except an abstract project (which can be used to share attribute values). Also, aggregate projects cannot be extended or imported though a `with` clause by any other project. Building other aggregate projects from an aggregate project is done through the `Project_Files` attribute (see below).

An aggregate project does not have any source files directly (only through other standard projects). Therefore a number of the standard attributes and packages are forbidden in an aggregate project. Here is a (non exhaustive) list:

- `Languages`
- `Source_Files`, `Source_List_File` and other attributes dealing with list of sources.
- `Source_Dirs`, `Exec_Dir` and `Object_Dir`
- `Library_Dir`, `Library_Name` and other library-related attributes
- `Main`
- `Roots`
- `Externally_Built`
- `Inherit_Source_Path`
- `Excluded_Source_Dirs`
- `Locally_Removed_Files`
- `Excluded_Source_Files`
- `Excluded_Source_List_File`
- `Interfaces`

The only package that is allowed (and optional) is `Builder`. Other packages (in particular `Compiler`, `Binder` and `Linker`) are forbidden.

The following three attributes can be used only in an aggregate project:

Project_Files:

> This attribute is compulsory. It specifies a list of constituent `.gpr` files that are grouped in the aggregate. The list may be empty. The project files can be any projects except configuration or abstract projects; they can be other aggregate projects. When grouping standard projects, you can have both the root of a project import closure (and you do not need to specify all its imported projects), and any project within the closure.
>
> The basic idea is to specify all those projects that have main programs you want to build and link, or libraries you want to build. You can specify projects that do not use the `Main` attribute or the `Library_*` attributes, and the result will be to build all their source files (not just the ones needed by other projects).
>
> The file can include paths (absolute or relative). Paths are relative to the location of the aggregate project file itself (if you use a base name, the `.gpr` file is expected in the same directory as the aggregate project file). The environment variables `ADA_PROJECT_PATH`, `GPR_PROJECT_PATH` and

GPR_PROJECT_PATH_FILE are not used to find the project files. The extension .gpr is mandatory, since this attribute contains file names, not project names.

Paths can also include the "*" and "**" globbing patterns. The latter indicates that any subdirectory (recursively) will be searched for matching files. The "**" pattern can only occur at the last position in the directory part (i.e. "a/**/*.gpr" is supported, but not "**/a/*.gpr"). Starting the pattern with "**" is equivalent to starting with "./**".

At present the pattern "*" is only allowed in the filename part, not in the directory part. This is mostly for efficiency reasons to limit the number of system calls that are needed.

Here are a few examples:

```
for Project_Files use ("a.gpr", "subdir/b.gpr");
--  two specific projects relative to the directory of agg.gpr

for Project_Files use ("/.gpr");
--  all projects recursively
```

Project_Path:

This attribute can be used to specify a list of directories in which to search for project files in with clauses.

When you specify a project in Project_Files (say x/y/a.gpr), and a.gpr imports a project b.gpr, only b.gpr is searched in the project path. The file a.gpr must be exactly at *dir of the aggregate*/x/y/a.gpr.

This attribute, however, does not affect the search for the aggregated project files specified with Project_Files.

Each aggregate project has its own Project_Path (thus if agg1.gpr includes agg2.gpr, they can potentially both have a different *Project_Path*).

This project path is defined as the concatenation, in this order, of:

- the current directory;
- followed by the command line -aP switches;
- then the directories from the GPR_PROJECT_PATH and ADA_PROJECT_PATH environment variables;
- then the directories from the Project_Path attribute;
- and finally the predefined directories.

In the example above, the project path for agg2.gpr is not influenced by the attribute *agg1'Project_Path*, nor is *agg1* influenced by *agg2'Project_Path*.

This can potentially lead to errors. Consider the example in figure 2.5.

When looking for p.gpr, both aggregates find the same physical file on the disk. However, it might happen that with their different project paths, both aggregate projects would in fact find a different r.gpr. Since we have a common project p.gpr withing two different r.gpr, this will be reported as an error by the builder.

Directories are relative to the location of the aggregate project file.

Example:

```
for Project_Path use ("/usr/local/gpr", "gpr/");
```

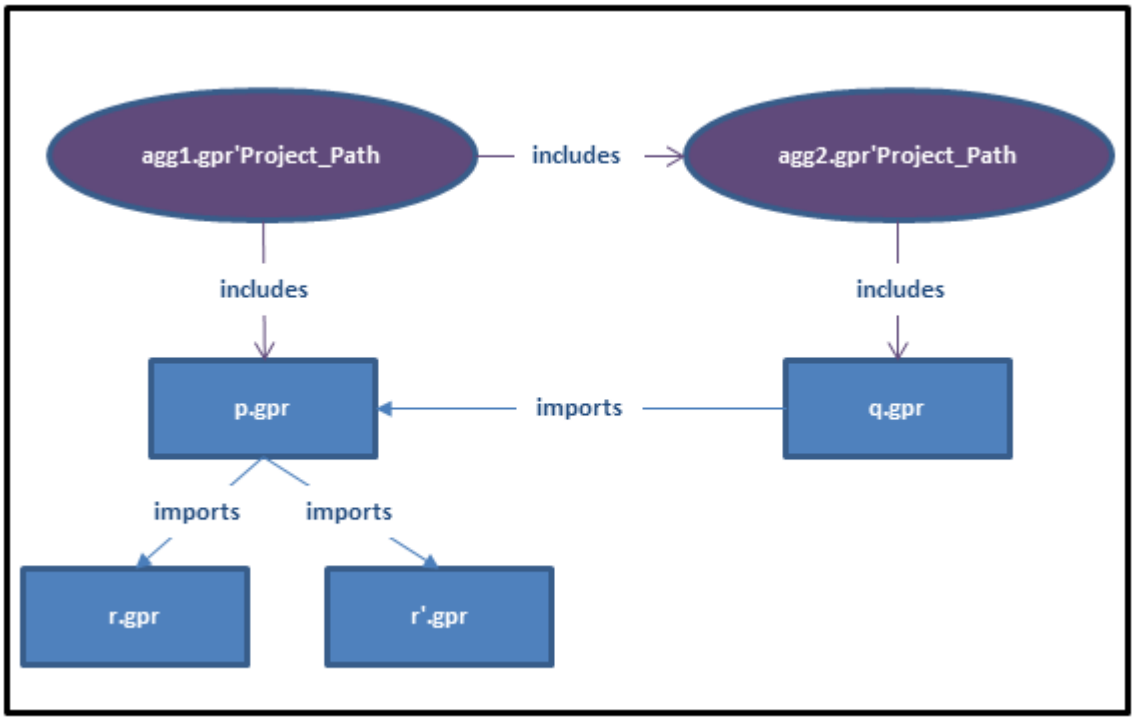

Figure 2.5: Example of `Project_Path` Error

External:

This attribute can be used to set the value of environment variables as retrieved through the `external` function in projects. It does not affect the environment variables themselves (so for instance you cannot use it to change the value of your `PATH` as seen from the spawned compiler).

This attribute affects the external values as seen in the rest of the aggregate project, and in the aggregated projects.

The exact value of an external variable comes from one of three sources (each level overrides the previous levels):

- An External attribute in aggregate project, for instance *for External ("BUILD_MODE") use "DEBUG"*;
- Environment variables. These override the value given by the attribute, so that users can override the value set in the (presumably shared with others team members) aggregate project.
- The `-X` command line switch to *gprbuild*. This always takes precedence.

This attribute is only taken into account in the main aggregate project (i.e. the one specified on the command line to *GPRbuild*), and ignored in other aggregate projects. It is invalid in standard projects. The goal is to have a consistent value in all projects that are built through the aggregate, which would not be the case in a "diamond" situation: `A` groups the aggregate projects `B` and `C`, which both (either directly or indirectly) build the project `P`. If `B` and `C` could set different values for the environment variables, we would have two different views of `P`, which in particular might impact the list of source files in `P`.

2.8.6 package Builder in aggregate projects

As mentioned above, only the package `Builder` can be specified in an aggregate project. In this package, only the following attributes are valid:

Switches:

This attribute gives the list of switches to use for *GPRbuild*. Because no mains can be specified for aggregate projects, the only possible index for attribute `Switches` is `others`. All other indexes will be ignored.

Example:

```
for Switches (others) use ("-v", "-k", "-j8");
```

These switches are only read from the main aggregate project (the one passed on the command line), and ignored in all other aggregate projects or projects.

It can only contain builder switches, not compiler switches.

Global_Compilation_Switches

This attribute gives the list of compiler switches for the various languages. For instance,

```
for Global_Compilation_Switches ("Ada") use ("O1", "-g");
for Global_Compilation_Switches ("C")   use ("-O2");
```

This attribute is only taken into account in the aggregate project specified on the command line, not in other aggregate projects.

In the projects grouped by that aggregate, the attribute `Builder'Global_Compilation_Switches` is also ignored. However, the attribute `Compiler'Default_Switches` will be taken into account (but that of the aggregate has higher priority). The attribute `Compiler'Switches` is also taken into account and can be used to override the switches for a specific file. As a result, it always has priority.

The rules are meant to avoid ambiguities when compiling. For instance, aggregate project `Agg` groups the projects `A` and `B`, which both depend on `C`. Here is an example for all of these projects:

```
aggregate project Agg is
    for Project_Files use ("a.gpr", "b.gpr");
    package Builder is
       for Global_Compilation_Switches ("Ada") use ("-O2");
    end Builder;
end Agg;
```

```
with "c.gpr";
project A is
    package Builder is
       for Global_Compilation_Switches ("Ada") use ("-O1");
       --  ignored
    end Builder;

    package Compiler is
       for Default_Switches ("Ada")
           use ("-O1", "-g");
       for Switches ("a_file1.adb")
           use ("-O0");
```

```
    end Compiler;
end A;
```

```
with "c.gpr";
project B is
    package Compiler is
       for Default_Switches ("Ada") use ("-O0");
    end Compiler;
end B;
```

```
project C is
    package Compiler is
       for Default_Switches ("Ada")
           use ("-O3",
                "-gnatn");
       for Switches ("c_file1.adb")
           use ("-O0", "-g");
    end Compiler;
end C;
```

The following switches are used:

- all files from project `A` except `a_file1.adb` are compiled with `-O2 -g`, since the aggregate project has priority.
- the file `a_file1.adb` is compiled with :option"-*O0*, since `Compiler'Switches` has priority
- all files from project `B` are compiled with `-O2`, since the aggregate project has priority
- all files from `C` are compiled with `-O2 -gnatn`, except for `c_file1.adb` which is compiled with `-O0 -g`

Even though `C` is seen through two paths (through `A` and through `B`), the switches used by the compiler are unambiguous.

Global_Configuration_Pragmas

This attribute can be used to specify a file containing configuration pragmas, to be passed to the Ada compiler. Since we ignore the package `Builder` in other aggregate projects and projects, only those pragmas defined in the main aggregate project will be taken into account.

Projects can locally add to those by using the `Compiler'Local_Configuration_Pragmas` attribute if they need.

Global_Config_File

This attribute, indexed with a language name, can be used to specify a config when compiling sources of the language. For Ada, these files are configuration pragmas files.

For projects that are built through the aggregate mechanism, the package `Builder` is ignored, except for the `Executable` attribute which specifies the name of the executables resulting from the link of the main programs, and for the `Executable_Suffix`.

2.9 Aggregate Library Projects

Aggregate library projects make it possible to build a single library using object files built using other standard or library projects. This gives the flexibility to describe an application as having multiple modules (for example a GUI,

database access, and other) using different project files (so possibly built with different compiler options) and yet create a single library (static or relocatable) out of the corresponding object files.

2.9.1 Building aggregate library projects

For example, we can define an aggregate project `Agg` that groups `A`, `B` and `C`:

```
aggregate library project Agg is
   for Project_Files use ("a.gpr", "b.gpr", "c.gpr");
   for Library_Name  use "agg";
   for Library_Dir   use "lagg";
end Agg;
```

Then, when you build with:

```
gprbuild agg.gpr
```

this will build all units from projects `A`, `B` and `C` and will create a static library named `libagg.a` in the `lagg` directory. An aggregate library project has the same set of restrictions as a standard library project.

Note that a shared aggregate library project cannot aggregate a static library project. In platforms where a compiler option is required to create relocatable object files, a `Builder package` in the aggregate library project may be used:

```
aggregate library project Agg is
   for Project_Files use ("a.gpr", "b.gpr", "c.gpr");
   for Library_Name use ("agg");
   for Library_Dir use ("lagg");
   for Library_Kind use "relocatable";

   package Builder is
      for Global_Compilation_Switches ("Ada") use ("-fPIC");
   end Builder;
end Agg;
```

With the above aggregate library Builder package, the `-fPIC` option will be passed to the compiler when building any source code from projects `a.gpr`, `b.gpr` and `c.gpr`.

2.9.2 Syntax of aggregate library projects

An aggregate library project follows the general syntax of project files. The recommended extension is still `.gpr`. However, a special `aggregate library` qualifier must appear before the keyword `project`.

An aggregate library project cannot `with` any other project (standard or aggregate), except an abstract project which can be used to share attribute values.

An aggregate library project does not have any source files directly (only through other standard projects). Therefore a number of the standard attributes and packages are forbidden in an aggregate library project. Here is a (non-exhaustive) list:

- `Languages`
- `Source_Files`, `Source_List_File` and other attributes dealing with a list of sources.
- `Source_Dirs`, `Exec_Dir` and `Object_Dir`

- Main
- Roots
- Externally_Built
- Inherit_Source_Path
- Excluded_Source_Dirs
- Locally_Removed_Files
- Excluded_Source_Files
- Excluded_Source_List_File

The only package that is allowed (and optional) is Builder.

The Project_Files attribute is used to describe the aggregated projects whose object files have to be included into the aggregate library. The environment variables ADA_PROJECT_PATH, GPR_PROJECT_PATH and GPR_PROJECT_PATH_FILE are not used to find the project files.

2.10 Project File Reference

This section describes the syntactic structure of project files, explains the various constructs that can be used, and summarizes the available attributes.

The syntax is presented in a notation similar to what is used in the Ada Language Reference Manual. Curly braces '{' and '}' indicate 0 or more occurrences of the enclosed construct, and square brackets '[' and ']' indicate 0 or 1 occurrence of the enclosed construct. Reserved words are enclosed between apostrophes.

2.10.1 Project Declaration

Project files have an Ada-like syntax. The minimal project file is:

```
project Empty is
end Empty;
```

The identifier Empty is the name of the project. This project name must be present after the reserved word end at the end of the project file, followed by a semicolon.

Identifiers (i.e., the user-defined names such as project or variable names) have the same syntax as Ada identifiers: they must start with a letter, and be followed by zero or more letters, digits or underscore characters; it is also illegal to have two underscores next to each other. Identifiers are always case-insensitive ("Name" is the same as "name").

```
simple_name ::= identifier
name        ::= simple_name { . simple_name }
```

Strings are used for values of attributes or as indexes for these attributes. They are in general case sensitive, except when noted otherwise (in particular, strings representing file names will be case insensitive on some systems, so that "file.adb" and "File.adb" both represent the same file).

Reserved words are the standard Ada 95 reserved words, plus several others listed below, and cannot be used for identifiers. In particular, the following Ada 95 reserved words are currently used in project files:

```
abstract  all     at       case
end       for     is       limited
null      others  package  renames
type      use     when     with
```

The additional project file reserved words are:

```
extends external external_as_list project
```

Note that `aggregate` and `library` are qualifiers that may appear before the keyword `project`, but they are not themselves keywords.

To avoid possible compatibility issues in the future, we recommend that the reserved words introduced by Ada 2005 and Ada 2012 not be used as identifiers in project files. Note also that new reserved words may be added to the project file syntax in a later release.

Comments in project files have the same syntax as in Ada, two consecutive hyphens through the end of the line.

A project may be an **independent project**, entirely defined by a single project file. Any source file in an independent project depends only on the predefined library and other source files in the same project. Alternatively, a project may depend on other projects in various ways:

- by **importing** them through context clauses (`with` clauses), or
- by **extending** at most one other project (its base project).

A given project may exhibit either or both of these dependencies; for example:

```
with "imported_proj.gpr";
project My_Project extends "base_proj.gpr" is
end My_Project;
```

The import dependencies form a **directed graph**, potentially cyclic when using **limited with**. The subgraph reflecting the **extends** relationship is a tree (hierarchy).

A path name denotes a project file. It can be absolute or relative. An absolute path name includes a sequence of directories, in the syntax of the host operating system, that uniquely identifies the project file in the file system. A relative path name identifies the project file, relative to the directory that contains the current project, or relative to a directory listed in the environment variables `ADA_PROJECT_PATH` and `GPR_PROJECT_PATH`. Path names are case sensitive if file names in the host operating system are case sensitive. As a special case, the directory separator can always be `'/'` even on Windows systems, so that project files can be made portable across architectures. The syntax of the environment variables `ADA_PROJECT_PATH` and `GPR_PROJECT_PATH` is a list of directory names separated by colons on Unix and semicolons on Windows.

A given project name can appear only once in a context clause, and may not appear in different context clauses for the same project.

It is illegal for a project imported by a context clause to refer, directly or indirectly, to the project in which this context clause appears (the dependency graph cannot contain cycles), except when one of the `with` clauses in the cycle is a `limited with`.

A project's **immediate sources** are the source files directly defined by that project, either implicitly by residing in the project source directories, or explicitly through any of the source-related attributes. More generally, a project's **sources** are the immediate sources of the project together with the immediate sources (unless overridden) of any project on which it depends directly or indirectly.

```
project        ::= context_clause project_declaration
```

```
 context_clause ::= {with_clause}
 with_clause    ::= [ 'limited' ] 'with' path_name { , path_name } ;
 path_name      ::= string_literal

 project_declaration ::= simple_project_declaration | project_extension

 simple_project_declaration ::=
   [ qualifier ] 'project' <project_>name 'is'
     {declarative_item}
   'end' <project_>name ;

 project_extension ::=
  [ qualifier ] 'project' <project_>name 'extends' [ 'all' ] <base_project_>name 'is'
    {declarative_item}
  'end' <project_>name ;

qualifier ::=
  'abstract' | identifier [ identifier ]
```

2.10.2 Qualified Projects

Immediately preceding the reserved `project`, a **qualifier** may be specified which identifies the nature of the project. The following qualifiers are allowed:

standard: A standard project is a non-library project with source files. This is the default (implicit) qualifier.

abstract: A project with no source files. Such a project must either have no declaration for attributes `Source_Dirs`, `Source_Files`, `Languages` or `Source_List_File`, or one of `Source_Dirs`, `Source_Files`, or `Languages` must be declared as empty. If it extends another project, the base project must also be an abstract project.

aggregate: A project whose sources are aggregated from other project files.

aggregate library: A library whose sources are aggregated from other project or library project files.

library: A library project must define both of the attributes *Library_Name* and *Library_Dir*.

configuration: A configuration project cannot be in a project tree. It describes compilers and other tools to *gprbuild*.

2.10.3 Declarations

Declarations introduce new entities that denote types, variables, attributes, and packages. Some declarations can only appear immediately within a project declaration. Others can appear within a project or within a package.

```
declarative_item ::= simple_declarative_item
  | typed_string_declaration
  | package_declaration

simple_declarative_item ::= variable_declaration
  | typed_variable_declaration
  | attribute_declaration
  | case_construction
  | empty_declaration

empty_declaration ::= 'null' ;
```

An empty declaration is allowed anywhere a declaration is allowed. It has no effect.

2.10.4 Packages

A project file may contain **packages**, which group attributes (typically all the attributes that are used by one of the GNAT tools).

A package with a given name may only appear once in a project file. The following packages are currently supported in project files (See *Attributes* for the list of attributes that each can contain).

Binder This package specifies characteristics useful when invoking the binder either directly via the *gnat* driver or when using *GPRbuild*. See *Main Subprograms*.

Builder This package specifies the compilation options used when building an executable or a library for a project. Most of the options should be set in one of `Compiler`, `Binder` or `Linker` packages, but there are some general options that should be defined in this package. See *Main Subprograms*, and *Executable File Names* in particular.

Check This package specifies the options used when calling the coding standard verification tool *gnatcheck*. Its attributes `Default_Switches` and `Switches` have the same semantics as for the package `Builder`. The first string should always be `-rules` to specify that all the other options belong to the `-rules` section of the parameters to *gnatcheck*.

Clean This package specifies the options used when cleaning a project or a project tree using the tools *gnatclean* or *gprclean*.

Compiler This package specifies the compilation options used by the compiler for each language. See *Tools Options in Project Files*.

Cross_Reference This package specifies the options used when calling the library tool *gnatxref* via the *gnat* driver. Its attributes `Default_Switches` and `Switches` have the same semantics as for the package `Builder`.

Documentation This package specifies the options used when calling the tool *gnatdoc*.

Eliminate This package specifies the options used when calling the tool *gnatelim*. Its attributes `Default_Switches` and `Switches` have the same semantics as for the package `Builder`.

Finder This package specifies the options used when calling the search tool *gnatfind* via the *gnat* driver. Its attributes `Default_Switches` and `Switches` have the same semantics as for the package `Builder`.

Gnatls This package specifies the options to use when invoking *gnatls* via the *gnat* driver.

Gnatstub This package specifies the options used when calling the tool *gnatstub*. Its attributes `Default_Switches` and `Switches` have the same semantics as for the package `Builder`.

IDE This package specifies the options used when starting an integrated development environment, for instance *GPS* or *GNATbench*.

Install This package specifies the options used when installing a project with *gprinstall*. See *Package Install Attributes*.

Linker This package specifies the options used by the linker. See *Main Subprograms*.

Metrics This package specifies the options used when calling the tool *gnatmetric*. Its attributes `Default_Switches` and `Switches` have the same semantics as for the package `Builder`.

Naming This package specifies the naming conventions that apply to the source files in a project. In particular, these conventions are used to automatically find all source files in the source directories, or given a file name to find out its language for proper processing. See *Naming Schemes*.

Pretty_Printer This package specifies the options used when calling the formatting tool *gnatpp*. Its attributes `Default_Switches` and `Switches` have the same semantics as for the package `Builder`.

Remote This package is used by *GPRbuild* to describe how distributed compilation should be done.

Stack This package specifies the options used when calling the tool *gnatstack*. Its attributes **Default_Switches** and **Switches** have the same semantics as for the package *Builder*.

Synchronize This package specifies the options used when calling the tool *gnatsync* via the *gnat* driver.

In its simplest form, a package may be empty:

```
project Simple is
  package Builder is
  end Builder;
end Simple;
```

A package may contain **attribute declarations**, **variable declarations** and **case constructions**, as will be described below.

When there is ambiguity between a project name and a package name, the name always designates the project. To avoid possible confusion, it is always a good idea to avoid naming a project with one of the names allowed for packages or any name that starts with *gnat*.

Package renaming

A package may be defined by a **renaming declaration**. The new package renames a package declared in a different project file, and has the same attributes as the package it renames. The name of the renamed package must be the same as the name of the renaming package. The project must contain a package declaration with this name, and the project must appear in the context clause of the current project, or be its base or parent project. It is not possible to add or override attributes to the renaming project. If you need to do so, you should use an **extending declaration** (see below).

Packages that are renamed in other project files often come from project files that have no sources: they are just used as templates. Any modification in the template will be reflected automatically in all the project files that rename a package from the template. This is a very common way to share settings between projects.

Package extension

A package can also be defined by an **extending declaration**. This is similar to a **renaming declaration**, except that it is possible to add or override attributes.

```
package_declaration ::= package_spec | package_renaming | package_extension

package_spec ::=
  'package' <package_>simple_name 'is'
     { simple_declarative_item }
  'end' package_identifier ;

package_renaming ::=
  'package' <package_>simple_name 'renames'
        <project_>simple_name.package_identifier ;

package_extension ::=
  'package' <package_>simple_name 'extends'
        <project_>simple_name.package_identifier 'is'
     { simple_declarative_item }
  'end' package_identifier ;
```

2.10.5 Expressions

An expression is any value that can be assigned to an attribute or a variable. It is either a literal value, or a construct requiring run-time computation by the Project Manager. In a project file, the computed value of an expression is either a string or a list of strings.

A string value is one of:

- A literal string, for instance `"comm/my_proj.gpr"`
- The name of a variable that evaluates to a string (see *Variables*)
- The name of an attribute that evaluates to a string (see *Attributes*)
- An external reference (see *External_Values*)
- A concatenation of the above, as in `"prefix_" & Var`.

A list of strings is one of the following:

- A parenthesized comma-separated list of zero or more string expressions, for instance `(File_Name, "gnat.adc", File_Name & ".orig")` or `()`.
- The name of a variable that evaluates to a list of strings
- The name of an attribute that evaluates to a list of strings
- A concatenation of a list of strings and a string (as defined above), for instance `("A", "B") & "C"`
- A concatenation of two lists of strings

The following is the grammar for expressions

```
string_literal ::= "{string_element}"  --  Same as Ada

string_expression ::= string_literal
    | <variable_>name
    | external_value
    | attribute_reference
    | ( string_expression { & string_expression } )

string_list  ::= ( string_expression { , string_expression } )
   | <string_variable>_name
   | <string_>attribute_reference

term ::= string_expression | string_list

expression ::= term { & term }     --  Concatenation
```

Concatenation involves strings and list of strings. As soon as a list of strings is involved, the result of the concatenation is a list of strings. The following Ada declarations show the existing operators:

```
function "&" (X : String;      Y : String)      return String;
function "&" (X : String_List; Y : String)      return String_List;
function "&" (X : String_List; Y : String_List) return String_List;
```

Here are some specific examples:

```
List := () & File_Name; --  One string in this list
List2 := List & (File_Name & ".orig"); -- Two strings
```

```
Big_List := List & Lists2;  --  Three strings
Illegal := "gnat.adc" & List2;  --  Illegal, must start with list
```

2.10.6 Built-in Functions

Built-in functions may be used in expression. The names of built-in functions are not reserved words and may also be used as variable names. In an expression, a built-in function is recognized if its name is immediately followed by an open parenthesis ('(').

The function `external`

An external value is an expression whose value is obtained from the command that invoked the processing of the current project file (typically a *gprbuild* command).

The syntax of a single string external value is:

```
external_value ::= 'external' ( string_literal [, string_literal] )
```

The first string_literal is the name of the external variable, whose value (a string) may be specified by an environment variable with this name, or on the command line via the `-Xname=value` option. The command line takes precedence if the name is defined in both contexts, thus allowing the user to locally override an environment variable. The second string_literal, if present, is the default to use if there is no specification for this external value either on the command line or in the environment. If the value of the external variable is not obtained from an environment variable or the command line, and the invocation of the `external` function does not supply a second parameter, then an error is reported.

An external reference may be part of a string expression or of a string list expression, and can therefore appear in a variable declaration or an attribute declaration. This construct is typically used to initialize *typed variables*, which are then used in *case* constructions to control the value assigned to attributes in various scenarios. Thus such variables are often called *scenario variables*.

The function `external_as_list`

An external value is an expression whose value is obtained from the command that invoked the processing of the current project file (typically a *gprbuild* command).

The syntax for a string list external value is:

```
external_value ::= 'external_as_list' ( string_literal , string_literal )
```

The first string_literal is the name of the external variable, with the same interpretation as for the `external` function; it is looked up first on the command line (as the name in a `-Xname=value` option) and, if not so specified, then as an environment variable. If it is not defined by either of these, then the function returns an empty list. The second string_literal is the separator between each component of the string list. An empty list is returned if the separator is an empty string or if the external value is only one separator.

Any separator at the beginning or at the end of the external value is discarded. Then, if there is no separator in the external value, the result is a string list with only one string. Otherwise, any string between the beginning and the first separator, between two consecutive separators and between the last separator and the end are components of the string list.

Note the following differences between `external` and `external_as_list`:

- The `external_as_list` function has no default value for the external variable
- The `external_as_list` function returns an empty list, and does not report an error, when the value of the external variable is undefined.

These differences reflect the different use cases for the two functions. External variables evaluated by the `external` function are often used for configuration control, and misspellings should be detected as errors rather than silently returning the empty string. If the user intended an empty string as the result when the external variable was undefined, then this could easily be obtained:

```
external ("SOME_VAR", "")
```

In contrast, the `external_as_list` function more typically is used for external variables that may or may not have definitions (for example, lists of options or paths) and then the desired result in the undefined case is an empty list, not a reported error.

Here is an example of the `external_as_list` function:

```
external_as_list ("SWITCHES", ",")
```

If the external value of `SWITCHES` is `"-O2,-g"`, the result is `("-O2", "-g")`.

If the external value is `",-O2,-g,"`, the result is also `("-O2", "-g")`.

if the external value is `"-gnatv"`, the result is `("-gnatv")`.

If the external value is `",,"`, the result is `("")`.

If the external value is `","`, the result is `()`, the empty string list.

Split

Function Split takes two single string parameters and return a string list.

Example:

```
Split ("-gnatf,-gnatv", ",")

=> ("-gnatf", "gnatv")
```

The first string argument is the string to be split. The second argument is the separator. Each occurrence of the separator in the first argument is a place where it is split. If the first argument is an empty string or contains only occurrences of the separator, then the result is an empty string list. If the argument does not contains any occurrence of the separator, then the result is a list with only one string: the first argument. Empty strings are not included in the result.

```
Split ("-gnatf   -gnatv", " ")

=> ("-gnatf", "gnatv")
```

2.10.7 Typed String Declaration

A **type declaration** introduces a discrete set of string literals. If a string variable is declared to have this type, its value is restricted to the given set of literals. These are the only named types in project files. A type declaration may only appear at the project level, not inside a package.

```
typed_string_declaration ::=
  'type' <typed_string_>simple_name 'is' ( string_literal {, string_literal} );
```

The string literals in the list are case sensitive and must all be different. They may include any graphic characters allowed in Ada, including spaces. Here is an example of a string type declaration:

```
type OS is ("GNU/Linux", "Unix", "Windows", "VMS");
```

Variables of a string type are called **typed variables**; all other variables are called **untyped variables**. Typed variables are particularly useful in *case* constructions, to support conditional attribute declarations. (See *Case Constructions*).

A string type may be referenced by its name if it has been declared in the same project file, or by an expanded name whose prefix is the name of the project in which it is declared.

2.10.8 Variables

Variables store values (strings or list of strings) and can appear as part of an expression. The declaration of a variable creates the variable and assigns the value of the expression to it. The name of the variable is available immediately after the assignment symbol, if you need to reuse its old value to compute the new value. Before the completion of its first declaration, the value of a variable defaults to the empty string (`""`).

A **typed** variable can be used as part of a **case** expression to compute the value, but it can only be declared once in the project file, so that all case constructions see the same value for the variable. This provides more consistency and makes the project easier to understand. The syntax for its declaration is identical to the Ada syntax for an object declaration. In effect, a typed variable acts as a constant.

An **untyped** variable can be declared and overridden multiple times within the same project. It is declared implicitly through an Ada assignment. The first declaration establishes the kind of the variable (string or list of strings) and successive declarations must respect the initial kind. Assignments are executed in the order in which they appear, so the new value replaces the old one and any subsequent reference to the variable uses the new value.

A variable may be declared at the project file level, or within a package.

```
typed_variable_declaration ::=
  <typed_variable_>simple_name : <typed_string_>name := string_expression;

variable_declaration ::= <variable_>simple_name := expression;
```

Here are some examples of variable declarations:

```
This_OS : OS := external ("OS"); --  a typed variable declaration
That_OS := "GNU/Linux";          --  an untyped variable declaration

Name      := "readme.txt";
Save_Name := Name & ".saved";

Empty_List := ();
List_With_One_Element := ("-gnaty");
List_With_Two_Elements := List_With_One_Element & "-gnatg";
Long_List := ("main.ada", "pack1_.ada", "pack1.ada", "pack2_.ada");
```

A **variable reference** may take several forms:

- The simple variable name, for a variable in the current package (if any) or in the current project

- An expanded name, whose prefix is a context name.

A **context** may be one of the following:

- The name of an existing package in the current project
- The name of an imported project of the current project
- The name of a direct or indirect base project (i.e., a project extended by the current project, either directly or indirectly)
- An expanded name whose prefix is an imported/parent project name, and whose selector is a package name in that project.

2.10.9 Case Constructions

A **case** construction is used in a project file to effect conditional behavior. Through this construction, you can set the value of attributes and variables depending on the value previously assigned to a typed variable.

All choices in a choice list must be distinct. Unlike Ada, the choice lists of all alternatives do not need to include all values of the type. An *others* choice must appear last in the list of alternatives.

The syntax of a `case` construction is based on the Ada case construction (although the `null` declaration for empty alternatives is optional).

The case expression must be a string variable, either typed or not, whose value is often given by an external reference (see *External_Values*).

Each alternative starts with the reserved word `when`, either a list of literal strings separated by the `"|"` character or the reserved word `others`, and the `"=>"` token. When the case expression is a typed string variable, each literal string must belong to the string type that is the type of the case variable. After each =>, there are zero or more declarations. The only declarations allowed in a case construction are other case constructions, attribute declarations, and variable declarations. String type declarations and package declarations are not allowed. Variable declarations are restricted to variables that have already been declared before the case construction.

```
case_construction ::=
  'case' <variable_>name 'is' {case_item} 'end' 'case' ;

case_item ::=
  'when' discrete_choice_list =>
    {case_declaration
      | attribute_declaration
      | variable_declaration
      | empty_declaration}

discrete_choice_list ::= string_literal {| string_literal} | 'others'
```

Here is a typical example, with a typed string variable:

```
project MyProj is
   type OS_Type is ("GNU/Linux", "Unix", "Windows", "VMS");
   OS : OS_Type := external ("OS", "GNU/Linux");

   package Compiler is
     case OS is
       when "GNU/Linux" | "Unix" =>
         for Switches ("Ada")
             use ("-gnath");
       when "Windows" =>
```

```
         for Switches ("Ada")
             use ("-gnatP");
       when others =>
         null;
     end case;
   end Compiler;
end MyProj;
```

2.10.10 Attributes

A project (and its packages) may have **attributes** that define the project's properties. Some attributes have values that are strings; others have values that are string lists.

```
attribute_declaration ::=
   simple_attribute_declaration | indexed_attribute_declaration

simple_attribute_declaration ::= 'for' attribute_designator 'use' expression ;

indexed_attribute_declaration ::=
  'for' *<indexed_attribute_>*simple_name ( string_literal) 'use' expression ;

attribute_designator ::=
  <simple_attribute_>simple_name
  | <indexed_attribute_>simple_name ( string_literal )
```

There are two categories of attributes: **simple attributes** and **indexed attributes**. Each simple attribute has a default value: the empty string (for string attributes) and the empty list (for string list attributes). An attribute declaration defines a new value for an attribute, and overrides the previous value. The syntax of a simple attribute declaration is similar to that of an attribute definition clause in Ada.

Some attributes are indexed. These attributes are mappings whose domain is a set of strings. They are declared one association at a time, by specifying a point in the domain and the corresponding image of the attribute. Like untyped variables and simple attributes, indexed attributes may be declared several times. Each declaration supplies a new value for the attribute, and replaces the previous setting.

Here are some examples of attribute declarations:

```
--  simple attributes
for Object_Dir use "objects";
for Source_Dirs use ("units", "test/drivers");

--  indexed attributes
for Body ("main") use "Main.ada";
for Switches ("main.ada")
    use ("-v", "-gnatv");
for Switches ("main.ada") use Builder'Switches ("main.ada") & "-g";

--  indexed attributes copy (from package Builder in project Default)
--  The package name must always be specified, even if it is the current
--  package.
for Default_Switches use Default.Builder'Default_Switches;
```

When an attribute is defined in the configuration project but not in the user project, it is inherited in the user project.

When a single string attribute is defined in both the configuration project and the user project, its value in the user project is as declared; the value in the configuration project does not matter.

For string list attributes, there are two cases. Some of these attributes are **configuration concatenable**. For these attributes, when they are declared in both the configuration project and the user project, the final value is the concatenation of the value in the configuration project with the value in the user project. The configuration concatenable attributes are indicated in the list below.

Attributes references may appear anywhere in expressions, and are used to retrieve the value previously assigned to the attribute. If an attribute has not been set in a given package or project, its value defaults to the empty string or the empty list, with some exceptions.

```
attribute_reference ::=
  attribute_prefix ' <simple_attribute>_simple_name [ (string_literal) ]

attribute_prefix ::= 'project'
  | <project_>simple_name
  | package_identifier
  | <project_>simple_name . package_identifier
```

Here are some examples:

```
project'Object_Dir
Naming'Dot_Replacement
Imported_Project'Source_Dirs
Imported_Project.Naming'Casing
Builder'Default_Switches ("Ada")
```

The exceptions to the empty defaults are:

- `Object_Dir`: default is `"."`
- `Exec_Dir`: default is `'Object_Dir`, that is, the value of attribute `Object_Dir` in the same project, declared or defaulted
- `Source_Dirs`: default is `("."`)

The prefix of an attribute may be:

- `project` for an attribute of the current project
- The name of an existing package of the current project
- The name of an imported project
- The name of a parent project that is extended by the current project
- An expanded name whose prefix is imported/base/parent project name, and whose selector is a package name

In the following sections, all predefined attributes are succinctly described, first the project level attributes (that is, those attributes that are not in a package), then the attributes in the different packages.

It is possible for different tools to dynamically create new packages with attributes, or new attributes in predefined packages. These attributes are not documented here.

The attributes under Configuration headings are usually found only in configuration project files.

The characteristics of each attribute are indicated as follows:

- **Type of value**

  The value of an attribute may be a single string, indicated by the word "single", or a string list, indicated by the word "list".

- **Read-only**

  When the attribute is read-only – that is when a declaration for the attribute is forbidden – this is indicated by the "read-only".

- **Optional index**

  If an optional index is allowed in the value of the attribute (both single and list), this is indicated by the words "optional index".

- **Indexed attribute**

  An indexed attribute is indicated by the word "indexed".

- **Case-sensitivity of the index**

  For an indexed attribute, if the index is case-insensitive, this is indicated by the words "case-insensitive index".

- **File name index**

  For an indexed attribute, when the index is a file name, this is indicated by the words "file name index". The index may or may not be case-sensitive, depending on the platform.

- **others allowed in index**

  For an indexed attribute, if it is allowed to use **others** as the index, this is indicated by the words "others allowed".

  When **others** is used as the index of an indexed attribute, the value of the attribute indexed by **others** is used when no other index would apply.

- **configuration concatenable**

  For a string list attribute, the final value if the attribute is declared in both the configuration project and the user project is the concatenation of the two value, configuration then user.

Project Level Attributes

- **General**
  - **Name**: single, read-only

    The name of the project.

  - **Project_Dir**: single, read-only

    The path name of the project directory.

  - **Main**: list, optional index

    The list of main sources for the executables.

  - **Languages**: list

    The list of languages of the sources of the project.

  - **Roots**: list, indexed, file name index

    The index is the file name of an executable source. Indicates the list of units from the main project that need to be bound and linked with their closures with the executable. The index is either a file name, a language name or "*". The roots for an executable source are those in **Roots** with an index that is the executable source file name, if declared. Otherwise, they are those in **Roots** with an index that is the language name of the executable source, if present. Otherwise, they are those in **Roots ("*")**, if declared. If none of these three possibilities are declared, then there are no roots for the executable source.

  - **Externally_Built**: single

    Indicates if the project is externally built. Only case-insensitive values allowed are "true" and "false", the default.

- **Directories**
  - **Object_Dir**: single

    Indicates the object directory for the project.

  - **Exec_Dir**: single

    Indicates the exec directory for the project, that is the directory where the executables are.

  - **Source_Dirs**: list

    The list of source directories of the project.

  - **Inherit_Source_Path**: list, indexed, case-insensitive index

    Index is a language name. Value is a list of language names. Indicates that in the source search path of the index language the source directories of the languages in the list should be included.

    Example:

```
for Inherit_Source_Path ("C++") use ("C");
```

  - **Exclude_Source_Dirs**: list

    The list of directories that are included in Source_Dirs but are not source directories of the project.

  - **Ignore_Source_Sub_Dirs**: list

    Value is a list of simple names or patterns for subdirectories that are removed from the list of source directories, including their subdirectories.

- **Source Files**
  - **Source_Files**: list

    Value is a list of source file simple names.

  - **Locally_Removed_Files**: list

    Obsolescent. Equivalent to Excluded_Source_Files.

  - **Excluded_Source_Files**: list

    Value is a list of simple file names that are not sources of the project. Allows to remove sources that are inherited or found in the source directories and that match the naming scheme.

  - **Source_List_File**: single

    Value is a text file name that contains a list of source file simple names, one on each line.

  - **Excluded_Source_List_File**: single

    Value is a text file name that contains a list of file simple names that are not sources of the project.

  - **Interfaces**: list

    Value is a list of file names that constitutes the interfaces of the project.

- **Aggregate Projects**
  - **Project_Files**: list

    Value is the list of aggregated projects.

  - **Project_Path**: list

    Value is a list of directories that are added to the project search path when looking for the aggregated projects.

  - **External**: single, indexed

    Index is the name of an external reference. Value is the value of the external reference to be used when parsing the aggregated projects.

- **Libraries**
  - **Library_Dir**: single

    Value is the name of the library directory. This attribute needs to be declared for each library project.

  - **Library_Name**: single

    Value is the name of the library. This attribute needs to be declared or inherited for each library project.

  - **Library_Kind**: single

    Specifies the kind of library: static library (archive) or shared library. Case-insensitive values must be one of "static" for archives (the default), "static-pic" for archives of Position Independent Code, or "dynamic" or "relocatable" for shared libraries.

  - **Library_Version**: single

    Value is the name of the library file.

  - **Library_Interface**: list

    Value is the list of unit names that constitutes the interfaces of a Stand-Alone Library project.

  - **Library_Standalone**: single

    Specifies if a Stand-Alone Library (SAL) is encapsulated or not. Only authorized case-insensitive values are "standard" for non encapsulated SALs, "encapsulated" for encapsulated SALs or "no" for non SAL library project.

  - **Library_Encapsulated_Options**: list, configuration concatenable

    Value is a list of options that need to be used when linking an encapsulated Stand-Alone Library.

  - **Library_Encapsulated_Supported**: single

    Indicates if encapsulated Stand-Alone Libraries are supported. Only authorized case-insensitive values are "true" and "false" (the default).

  - **Library_Auto_Init**: single

    Indicates if a Stand-Alone Library is auto-initialized. Only authorized case-insensitive values are "true" and "false".

  - **Leading_Library_Options**: list, configuration concatenable

    Value is a list of options that are to be used at the beginning of the command line when linking a shared library.

  - **Library_Options**: list, configuration concatenable

    Value is a list of options that are to be used when linking a shared library.

  - **Library_Rpath_Options**: list, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is a list of options for an invocation of the compiler of the language. This invocation is done for a shared library project with sources of the language. The output of the invocation is

the path name of a shared library file. The directory name is to be put in the run path option switch when linking the shared library for the project.

- **Library_Src_Dir**: single

  Value is the name of the directory where copies of the sources of the interfaces of a Stand-Alone Library are to be copied.

- **Library_ALI_Dir**: single

  Value is the name of the directory where the ALI files of the interfaces of a Stand-Alone Library are to be copied. When this attribute is not declared, the directory is the library directory.

- **Library_gcc**: single

  Obsolescent attribute. Specify the linker driver used to link a shared library. Use instead attribute Linker'Driver.

- **Library_Symbol_File**: single

  Value is the name of the library symbol file.

- **Library_Symbol_Policy**: single

  Indicates the symbol policy kind. Only authorized case-insensitive values are "restricted", "unrestricted".

- **Library_Reference_Symbol_File**: single

  Value is the name of the reference symbol file.

- **Configuration - General**

  - **Default_Language**: single

    Value is the case-insensitive name of the language of a project when attribute Languages is not specified.

  - **Run_Path_Option**: list

    Value is the list of switches to be used when specifying the run path option in an executable.

  - **Run_Path_Origin**: single

    Value is the the string that may replace the path name of the executable directory in the run path options.

  - **Separate_Run_Path_Options**: single

    Indicates if there may be several run path options specified when linking an executable. Only authorized case-insensitive values are "true" or "false" (the default).

  - **Toolchain_Version**: single, indexed, case-insensitive index

    Index is a language name. Specify the version of a toolchain for a language.

  - **Toolchain_Description**: single, indexed, case-insensitive index

    Obsolescent. No longer used.

  - **Object_Generated**: single, indexed, case-insensitive index

    Index is a language name. Indicates if invoking the compiler for a language produces an object file. Only authorized case-insensitive values are "false" and "true" (the default).

  - **Objects_Linked**: single, indexed, case-insensitive index

    Index is a language name. Indicates if the object files created by the compiler for a language need to be linked in the executable. Only authorized case-insensitive values are "false" and "true" (the default).

  - **Target**: single

    Value is the name of the target platform. Taken into account only in the main project.

    Note that when the target is specified on the command line (usually with a switch –target=), the value of attribute reference 'Target is the one specified on the command line.

  - **Runtime**: single, indexed, case-insensitive index

    Index is a language name. Indicates the runtime directory that is to be used when using the compiler of the language. Taken into account only in the main project.

    Note that when the runtime is specified for a language on the command line (usually with a switch –RTS), the value of attribute reference 'Runtime for this language is the one specified on the command line.

  - **Runtime_Dir**: single, indexed, case-insensitive index

    Index is a language name. Value is the path name of the runtime directory for the language.

  - **Runtime_Library_Dirs**: list, indexed, case-insensitive index

    Index is a language name. Value is the path names of the directories where the runtime libraries are located. This attribute is not normally declared.

  - **Runtime_Library_Dir**: single, indexed, case-insensitive index

    Index is a language name. Value is the path name of the directory where the runtime libraries are located. This attribute is obsolete.

  - **Runtime_Source_Dirs**: list, indexed, case-insensitive index

    Index is a language name. Value is the path names of the directories where the sources of runtime libraries are located. This attribute is not normally declared.

  - **Runtime_Source_Dir**: single, indexed, case-insensitive index

    Index is a language name. Value is the path name of the directory where the sources of runtime libraries are located. This attribute is obsolete.

  - **Runtime_Library_Version**: single, indexed, case-insensitive index

    Index is a language name. Value is library version for the language. This attribute is not normally declared.

- **Configuration - Libraries**
  - **Library_Builder**: single

    Value is the path name of the application that is to be used to build libraries. Usually the path name of "gprlib".

  - **Library_Support**: single

    Indicates the level of support of libraries. Only authorized case-insensitive values are "static_only", "full" or "none" (the default).

- **Configuration - Archives**
  - **Archive_Builder**: list

    Value is the name of the application to be used to create a static library (archive), followed by the options to be used.

  - **Archive_Builder_Append_Option**: list

    Value is the list of options to be used when invoking the archive builder to add project files into an archive.

  - **Archive_Indexer**: list

    Value is the name of the archive indexer, followed by the required options.

  - **Archive_Suffix**: single

    Value is the extension of archives. When not declared, the extension is ".a".

  - **Library_Partial_Linker**: list

    Value is the name of the partial linker executable, followed by the required options.

- **Configuration - Shared Libraries**
  - **Shared_Library_Prefix**: single

    Value is the prefix in the name of shared library files. When not declared, the prefix is "lib".

  - **Shared_Library_Suffix**: single

    Value is the the extension of the name of shared library files. When not declared, the extension is ".so".

  - **Symbolic_Link_Supported**: single

    Indicates if symbolic links are supported on the platform. Only authorized case-insensitive values are "true" and "false" (the default).

  - **Library_Major_Minor_Id_Supported**: single

    Indicates if major and minor ids for shared library names are supported on the platform. Only authorized case-insensitive values are "true" and "false" (the default).

  - **Library_Auto_Init_Supported**: single

    Indicates if auto-initialization of Stand-Alone Libraries is supported. Only authorized case-insensitive values are "true" and "false" (the default).

  - **Shared_Library_Minimum_Switches**: list, configuration concatenable

    Value is the list of required switches when linking a shared library.

  - **Library_Version_Switches**: list, configuration concatenable

    Value is the list of switches to specify a internal name for a shared library.

  - **Library_Install_Name_Option**: single

    Value is the name of the option that needs to be used, concatenated with the path name of the library file, when linking a shared library.

Package Binder Attributes

- **General**
  - **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is the list of switches to be used when binding code of the language, if there is no applicable attribute Switches.

  - **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

    Index is either a language name or a source file name. Value is the list of switches to be used when binding code. Index is either the source file name of the executable to be bound or the language name of the code to be bound.

- **Configuration - Binding**

- **Driver**: single, indexed, case-insensitive index

  Index is a language name. Value is the name of the application to be used when binding code of the language.

- **Required_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is the list of the required switches to be used when binding code of the language.

- **Prefix**: single, indexed, case-insensitive index

  Index is a language name. Value is a prefix to be used for the binder exchange file name for the language. Used to have different binder exchange file names when binding different languages.

- **Objects_Path**: single,indexed, case-insensitive index

  Index is a language name. Value is the name of the environment variable that contains the path for the object directories.

- **Object_Path_File**: single,indexed, case-insensitive index

  Index is a language name. Value is the name of the environment variable. The value of the environment variable is the path name of a text file that contains the list of object directories.

Package Builder Attributes

- **Default_Switches: list, indexed, case-insensitive index,** configuration concatenable

  Index is a language name. Value is the list of builder switches to be used when building an executable of the language, if there is no applicable attribute Switches.

- **Switches: list, optional index, indexed, case-insensitive index,** others allowed, configuration concatenable

  Index is either a language name or a source file name. Value is the list of builder switches to be used when building an executable. Index is either the source file name of the executable to be built or its language name.

- **Global_Compilation_Switches: list, optional index, indexed,** case-insensitive index, configuration concatenable

  Index is a language name. Value is the list of compilation switches to be used when building an executable. Index is either the source file name of the executable to be built or its language name.

- **Executable**: single, indexed, case-insensitive index

  Index is an executable source file name. Value is the simple file name of the executable to be built.

- **Executable_Suffix**: single

  Value is the extension of the file names of executable. When not specified, the extension is the default extension of executables on the platform.

- **Global_Configuration_Pragmas**: single

  Value is the file name of a configuration pragmas file that is specified to the Ada compiler when compiling any Ada source in the project tree.

- **Global_Config_File**: single, indexed, case-insensitive index

  Index is a language name. Value is the file name of a configuration file that is specified to the compiler when compiling any source of the language in the project tree.

Package Check Attributes

- **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is a list of switches to be used when invoking *gnatcheck* for a source of the language, if there is no applicable attribute Switches.

- **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

  Index is a source file name. Value is the list of switches to be used when invoking *gnatcheck* for the source.

Package Clean Attributes

- **Switches**: list, configuration concatenable

  Value is a list of switches to be used by the cleaning application.

- **Source_Artifact_Extensions**: list, indexed, case-insensitive index

  Index is a language names. Value is the list of extensions for file names derived from object file names that need to be cleaned in the object directory of the project.

- **Object_Artifact_Extensions**: list, indexed, case-insensitive index

  Index is a language names. Value is the list of extensions for file names derived from source file names that need to be cleaned in the object directory of the project.

- **Artifacts_In_Object_Dir**: single

  Value is a list of file names expressed as regular expressions that are to be deleted by gprclean in the object directory of the project.

- **Artifacts_In_Exec_Dir**: single

  Value is list of file names expressed as regular expressions that are to be deleted by gprclean in the exec directory of the main project.

Package Compiler Attributes

- **General**
  - **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is a list of switches to be used when invoking the compiler for the language for a source of the project, if there is no applicable attribute Switches.

  - **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

    Index is a source file name or a language name. Value is the list of switches to be used when invoking the compiler for the source or for its language.

  - **Local_Configuration_Pragmas**: single

    Value is the file name of a configuration pragmas file that is specified to the Ada compiler when compiling any Ada source in the project.

  - **Local_Config_File**: single, indexed, case-insensitive index

    Index is a language name. Value is the file name of a configuration file that is specified to the compiler when compiling any source of the language in the project.

- **Configuration - Compiling**

- **Driver**: single, indexed, case-insensitive index

  Index is a language name. Value is the name of the executable for the compiler of the language.

- **Language_Kind**: single, indexed, case-insensitive index

  Index is a language name. Indicates the kind of the language, either file based or unit based. Only authorized case-insensitive values are "unit_based" and "file_based" (the default).

- **Dependency_Kind**: single, indexed, case-insensitive index

  Index is a language name. Indicates how the dependencies are handled for the language. Only authorized case-insensitive values are "makefile", "ali_file", "ali_closure" or "none" (the default).

- **Required_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Equivalent to attribute Leading_Required_Switches.

- **Leading_Required_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is the list of the minimum switches to be used at the beginning of the command line when invoking the compiler for the language.

- **Trailing_Required_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is the list of the minimum switches to be used at the end of the command line when invoking the compiler for the language.

- **PIC_Option**: list, indexed, case-insensitive index

  Index is a language name. Value is the list of switches to be used when compiling a source of the language when the project is a shared library project.

- **Path_Syntax**: single, indexed, case-insensitive index

  Index is a language name. Value is the kind of path syntax to be used when invoking the compiler for the language. Only authorized case-insensitive values are "canonical" and "host" (the default).

- **Source_File_Switches**: single, indexed, case-insensitive index configuration concatenable

  Index is a language name. Value is a list of switches to be used just before the path name of the source to compile when invoking the compiler for a source of the language.

- **Object_File_Suffix**: single, indexed, case-insensitive index

  Index is a language name. Value is the extension of the object files created by the compiler of the language. When not specified, the extension is the default one for the platform.

- **Object_File_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is the list of switches to be used by the compiler of the language to specify the path name of the object file. When not specified, the switch used is "-o".

- **Multi_Unit_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is the list of switches to be used to compile a unit in a multi unit source of the language. The index of the unit in the source is concatenated with the last switches in the list.

- **Multi_Unit_Object_Separator**: single, indexed, case-insensitive index

  Index is a language name. Value is the string to be used in the object file name before the index of the unit, when compiling a unit in a multi unit source of the language.

- **Configuration - Mapping Files**

  - **Mapping_File_Switches**: list, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is the list of switches to be used to specify a mapping file when invoking the compiler for a source of the language.

  - **Mapping_Spec_Suffix**: single, indexed, case-insensitive index

    Index is a language name. Value is the suffix to be used in a mapping file to indicate that the source is a spec.

  - **Mapping_Body_Suffix**: single, indexed, case-insensitive index

    Index is a language name. Value is the suffix to be used in a mapping file to indicate that the source is a body.

- **Configuration - Config Files**
  - **Config_File_Switches**: list: single, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is the list of switches to specify to the compiler of the language a configuration file.

  - **Config_Body_File_Name**: single, indexed, case-insensitive index

    Index is a language name. Value is the template to be used to indicate a configuration specific to a body of the language in a configuration file.

  - **Config_Body_File_Name_Index**: single, indexed, case-insensitive index

    Index is a language name. Value is the template to be used to indicate a configuration specific to the body a unit in a multi unit source of the language in a configuration file.

  - **Config_Body_File_Name_Pattern**: single, indexed, case-insensitive index

    Index is a language name. Value is the template to be used to indicate a configuration for all bodies of the languages in a configuration file.

  - **Config_Spec_File_Name**: single, indexed, case-insensitive index

    Index is a language name. Value is the template to be used to indicate a configuration specific to a spec of the language in a configuration file.

  - **Config_Spec_File_Name_Index**: single, indexed, case-insensitive index

    Index is a language name. Value is the template to be used to indicate a configuration specific to the spec a unit in a multi unit source of the language in a configuration file.

  - **Config_Spec_File_Name_Pattern**: single, indexed, case-insensitive index

    Index is a language name. Value is the template to be used to indicate a configuration for all specs of the languages in a configuration file.

  - **Config_File_Unique**: single, indexed, case-insensitive index

    Index is a language name. Indicates if there should be only one configuration file specified to the compiler of the language. Only authorized case-insensitive values are "true" and "false" (the default).

- **Configuration - Dependencies**
  - **Dependency_Switches**: list, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is the list of switches to be used to specify to the compiler the dependency file when the dependency kind of the language is file based, and when Dependency_Driver is not specified for the language.

  - **Dependency_Driver**: list, indexed, case-insensitive index

    Index is a language name. Value is the name of the executable to be used to create the dependency file for a source of the language, followed by the required switches.

- **Configuration - Search Paths**
  - **Include_Switches**: list, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is the list of switches to specify to the compiler of the language to indicate a directory to look for sources.

  - **Include_Path**: single, indexed, case-insensitive index

    Index is a language name. Value is the name of an environment variable that contains the path of all the directories that the compiler of the language may search for sources.

  - **Include_Path_File**: single, indexed, case-insensitive index

    Index is a language name. Value is the name of an environment variable the value of which is the path name of a text file that contains the directories that the compiler of the language may search for sources.

  - **Object_Path_Switches**: list, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is the list of switches to specify to the compiler of the language the name of a text file that contains the list of object directories. When this attribute is not declared, the text file is not created.

- **Configuration - Response Files**
  - **Max_Command_Line_Length**: single

    Value is the maximum number of character in the command line when invoking a compiler that supports response files.

  - **Response_File_Format**: single, indexed, case-insensitive index

    Indicates the kind of response file to create when the length of the compiling command line is too large. The index is the name of the language for the compiler. Only authorized case-insensitive values are "none", "gnu", "object_list", "gcc_gnu", "gcc_option_list" and "gcc_object_list".

  - **Response_File_Switches**: list, indexed, case-insensitive index, configuration concatenable

    Value is the list of switches to specify a response file for a compiler. The index is the name of the language for the compiler.

Package Cross_Reference Attributes

- **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is a list of switches to be used when invoking *gnatxref* for a source of the language, if there is no applicable attribute Switches.

- **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

  Index is a source file name. Value is the list of switches to be used when invoking *gnatxref* for the source.

Package Documentation Attributes

Please refer to GNATdoc documentation for the list of supported attributes and their meaning.

Package Eliminate Attributes

- **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is a list of switches to be used when invoking *gnatelim* for a source of the language, if there is no applicable attribute Switches.

- **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

  Index is a source file name. Value is the list of switches to be used when invoking *gnatelim* for the source.

Package Finder Attributes

- **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is a list of switches to be used when invoking *gnatfind* for a source of the language, if there is no applicable attribute Switches.

- **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

  Index is a source file name. Value is the list of switches to be used when invoking *gnatfind* for the source.

Package Gnatls Attributes

- **Switches**: list

  Value is a list of switches to be used when invoking *gnatls*.

Package gnatstub Attributes

- **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is a list of switches to be used when invoking *gnatstub* for a source of the language, if there is no applicable attribute Switches.

- **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

  Index is a source file name. Value is the list of switches to be used when invoking *gnatstub* for the source.

Package IDE Attributes

Please refer to your IDE documentation for the list of supported attributes and their meaning.

Package Install Attributes

- **Artifacts**: list, indexed

  An indexed attribute to declare a set of files not part of the sources to be installed. The array index is the directory where the file is to be installed. If a relative directory then Prefix (see below) is prepended. Note also that if the same file name occurs multiple time in the attribute list, the last one will be the one installed. If an artifact is not found a warning is displayed.

- **Required_Artifacts**: list, indexed

  As above, but artifacts must be present or an error is reported.

- **Prefix**: single

  Value is the install destination directory. If the value is a relative path, it is taken as relative to the global prefix directory. That is, either the value passed to *–prefix* option or the default installation prefix.

- **Sources_Subdir**: single

  Value is the sources directory or subdirectory of Prefix.

- **Exec_Subdir**: single

  Value is the executables directory or subdirectory of Prefix.

- **ALI_Subdir**: single

  Value is ALI directory or subdirectory of Prefix.

- **Lib_Subdir**: single

  Value is library directory or subdirectory of Prefix.

- **Project_Subdir**: single

  Value is the project directory or subdirectory of Prefix.

- **Active**: single

  Indicates that the project is to be installed or not. Case-insensitive value "false" means that the project is not to be installed, all other values mean that the project is to be installed.

- **Mode**: single

  Value is the installation mode, it is either **dev** (default) or **usage**.

- **Install_Name**: single

  Specify the name to use for recording the installation. The default is the project name without the extension.

- **Side_Debug**: single

  Indicates that the project's executable and shared libraries are to be stripped of the debug symbols. Those debug symbols are written into a side file named after the original file with the ".debug" extension added. Case-insensitive value "false" (default) disables this feature. Set it to "true" to activate.

- **Install_Project**: single

  Indicates that a project is to be generated and installed. The value is either "true" to "false". Default is "true".

Package Linker Attributes

- **General**
  - **Required_Switches**: list, configuration concatenable

    Value is a list of switches that are required when invoking the linker to link an executable.

  - **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

    Index is a language name. Value is a list of switches for the linker when linking an executable for a main source of the language, when there is no applicable Switches.

  - **Leading_Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

    Index is a source file name or a language name. Value is the list of switches to be used at the beginning of the command line when invoking the linker to build an executable for the source or for its language.

  - **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

    Index is a source file name or a language name. Value is the list of switches to be used when invoking the linker to build an executable for the source or for its language.

  - **Trailing_Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

    Index is a source file name or a language name. Value is the list of switches to be used at the end of the command line when invoking the linker to build an executable for the source or for its language. These switches may override the Required_Switches.

  - **Linker_Options**: list, configuration concatenable

    This attribute specifies a list of additional switches to be given to the linker when linking an executable. It is ignored when defined in the main project and taken into account in all other projects that are imported directly or indirectly. These switches complement the `Linker'Switches` defined in the main project. This is useful when a particular subsystem depends on an external library: adding this dependency as a `Linker_Options` in the project of the subsystem is more convenient than adding it to all the `Linker'Switches` of the main projects that depend upon this subsystem.

  - **Map_File_Option**: single

    Value is the switch to specify the map file name that the linker needs to create.

- **Configuration - Linking**

  - **Driver**: single

    Value is the name of the linker executable.

- **Configuration - Response Files**

  - **Max_Command_Line_Length**: single

    Value is the maximum number of character in the command line when invoking the linker to link an executable.

  - **Response_File_Format**: single

    Indicates the kind of response file to create when the length of the linking command line is too large. Only authorized case-insensitive values are "none", "gnu", "object_list", "gcc_gnu", "gcc_option_list" and "gcc_object_list".

  - **Response_File_Switches**: list, configuration concatenable

    Value is the list of switches to specify a response file to the linker.

Package Metrics Attribute

- **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is a list of switches to be used when invoking *gnatmetric* for a source of the language, if there is no applicable attribute Switches.

- **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

  Index is a source file name. Value is the list of switches to be used when invoking *gnatmetric* for the source.

Package Naming Attributes

- **Specification_Suffix**: single, indexed, case-insensitive index

  Equivalent to attribute Spec_Suffix.

- **Spec_Suffix**: single, indexed, case-insensitive index

  Index is a language name. Value is the extension of file names for specs of the language.

- **Implementation_Suffix**: single, indexed, case-insensitive index

  Equivalent to attribute Body_Suffix.

- **Body_Suffix**: single, indexed, case-insensitive index

  Index is a language name. Value is the extension of file names for bodies of the language.

- **Separate_Suffix**: single

  Value is the extension of file names for subunits of Ada.

- **Casing**: single

  Indicates the casing of sources of the Ada language. Only authorized case-insensitive values are "lowercase", "uppercase" and "mixedcase".

- **Dot_Replacement**: single

  Value is the string that replace the dot of unit names in the source file names of the Ada language.

- **Specification**: single, optional index, indexed, case-insensitive index

  Equivalent to attribute Spec.

- **Spec**: single, optional index, indexed, case-insensitive index

  Index is a unit name. Value is the file name of the spec of the unit.

- **Implementation**: single, optional index, indexed, case-insensitive index

  Equivalent to attribute Body.

- **Body**: single, optional index, indexed, case-insensitive index

  Index is a unit name. Value is the file name of the body of the unit.

- **Specification_Exceptions**: list, indexed, case-insensitive index

  Index is a language name. Value is a list of specs for the language that do not necessarily follow the naming scheme for the language and that may or may not be found in the source directories of the project.

- **Implementation_Exceptions**: list, indexed, case-insensitive index

  Index is a language name. Value is a list of bodies for the language that do not necessarily follow the naming scheme for the language and that may or may not be found in the source directories of the project.

Package Pretty_Printer Attributes

- **Default_Switches**: list, indexed, case-insensitive index, configuration concatenable

  Index is a language name. Value is a list of switches to be used when invoking *gnatpp* for a source of the language, if there is no applicable attribute Switches.

- **Switches**: list, optional index, indexed, case-insensitive index, others allowed, configuration concatenable

  Index is a source file name. Value is the list of switches to be used when invoking *gnatpp* for the source.

Package Remote Attributes

- **Included_Patterns**: list

  If this attribute is defined it sets the patterns to synchronized from the master to the slaves. It is exclusive with Excluded_Patterns, that is it is an error to define both.

- **Included_Artifact_Patterns**: list

  If this attribute is defined it sets the patterns of compilation artifacts to synchronized from the slaves to the build master. This attribute replace the default hard-coded patterns.

- **Excluded_Patterns**: list

  Set of patterns to ignore when synchronizing sources from the build master to the slaves. A set of predefined patterns are supported (e.g. *.o, *.ali, *.exe, etc.), this attributes make it possible to add some more patterns.

- **Root_Dir**: single

  Value is the root directory used by the slave machines.

Package Stack Attributes

- **Switches**: list, configuration concatenable

  Value is the list of switches to be used when invoking *gnatstack*.

Package Synchronize Attributes

- **Default_Switches**: list, indexed, case-insensitive index

  Index is a language name. Value is a list of switches to be used when invoking *gnatsync* for a source of the language, if there is no applicable attribute Switches.

- **Switches**: list, optional index, indexed, case-insensitive index, others allowed

  Index is a source file name. Value is the list of switches to be used when invoking *gnatsync* for the source.

2.11 Glossary

Abstract project A project with no source files, typically used to define common attributes that are shared by other project files. See *Sharing between Projects*.

Aggregate project A project that in effect combines several projects in order to efficiently support concurrent builds or builds of all main programs from the constituent projects, or the convenient definition of a common environment for the constituent projects. See *Aggregate Projects*.

Attribute A named property of a project or one of its packages. See *Attributes*.

Base project A project that is extended by some other project. See *Project Extension*.

Child project A project that is defined by a name `Parent_proj.Child_proj` where `Child_proj` either imports or extends `Parent_Proj`. This feature is typically used to show a close relationship between the two projects, for example where the child project serves as a testbed for the parent. See *Child Projects*.

Configuration project A project that describes compilers and other tools, for use by *GPRbuild*. See *Configuration Project*.

Extending a project The reuse and possible adaption by one project of the source files from another project (the base project). Somewhat analogous to (single) class inheritance in object-oriented programming. See *Project Extension*.

External variable A variable that is defined on the command line (by the `-X` switch), as the value of an environment variable, or, by default, as the second parameter to the `external` function. See *Scenarios in Projects*.

Global attribute An attribute that applies to all projects in the project import closure of a main project. See *Global Attributes*.

Importing a project The usage of a `with` or `limited with` clause on a project file in order to reuse properties of some other project file. See *Importing Projects*.

Independent project A project defined by a single project file and thus not dependent on any other projects. See *Independent Project*.

Library project A project that is used to define a library rather than an executable program. See *Library Projects*.

Main project A project that is specified on the command line. See *Global Attributes*.

Package A grouping of attribute definitions related to a particular GNAT tool. See *Packages*.

Parent project A project that has one or more child projects. See *Child Projects*.

Project A set of named properties and their values, associated with the GNAT tools that are used during the development of software in Ada and other languages. Properties include directories for source files, object files, and executables; the switch settings for the various tools; and the naming scheme for source files.

Project extension See glossary item *Extending a project*

Project file A textual representation of a project, which uses an Ada-like notation. The syntax is presented in *Project File Reference*.

Project import closure The *project import closure* for a given project *proj* is the set of projects consisting of *proj* itself, together with each project that is directly or indirectly imported by *proj*. The import may be from either a `with` or a `limited with`. See *Project Import Closure*.

Scenario The values of a project's variables and attributes, as determined by the settings of external variables referenced by a project. A scenario typically defines a particular mode of usage for the project. See *Scenarios in Projects*.

Scenario variable An external variable, typically assigned to a typed variable and queried in a *case construction*. See *Scenario variable*.

Standard project A non-library project with source files. See *Standard project*

Typed variable A project variable that can take any of a specified set of values, analogous to a variable of an Ada enumeration type but where the values are string literals. See *Scenarios in Projects*.

CHAPTER

THREE

BUILDING WITH GPRBUILD

3.1 Introduction

GPRbuild is a generic build tool designed for the construction of large multi-language systems organized into subsystems and libraries. It is well-suited for compiled languages supporting separate compilation, such as Ada, C, C++ and Fortran.

GPRbuild manages a three step build process.

- compilation phase:

  Each compilation unit of each subsystem is examined in turn, checked for consistency, and compiled or recompiled when necessary by the appropriate compiler. The recompilation decision is based on dependency information that is typically produced by a previous compilation.

- post-compilation phase (or binding):

  Compiled units from a given language are passed to a language-specific post-compilation tool if any. Also during this phase objects are grouped into static or dynamic libraries as specified.

- linking phase:

  All units or libraries from all subsystems are passed to a linker tool specific to the set of toolchains being used.

The tool is generic in that it provides, when possible, equivalent build capabilities for all supported languages. For this, it uses a configuration file `<file>.cgpr` that has a syntax and structure very similar to a project file, but which defines the characteristics of the supported languages and toolchains. The configuration file contains information such as:

- the default source naming conventions for each language,
- the compiler name, location and required options,
- how to compute inter-unit dependencies,
- how to build static or dynamic libraries,
- which post-compilation actions are needed,
- how to link together units from different languages.

On the other hand, *GPRbuild* is not a replacement for general-purpose build tools such as *make* or *ant* which give the user a high level of control over the build process itself. When building a system requires complex actions that do not fit well in the three-phase process described above, *GPRbuild* might not be sufficient. In such situations, *GPRbuild* can still be used to manage the appropriate part of the build. For instance it can be called from within a Makefile.

3.2 Command Line

Three elements can optionally be specified on GPRbuild's command line:

- the main project file,
- the switches for GPRbuild itself or for the tools it drives, and
- the main source files.

The general syntax is thus:

```
gprbuild [<proj>.gpr] [switches] [names]
 {[-cargs opts] [-cargs:lang opts] [-largs opts] [-gargs opts]}
```

GPRbuild requires a project file, which may be specified on the command line either directly or through the `-P` switch. If not specified, GPRbuild uses the project file `default.gpr` if there is one in the current working directory. Otherwise, if there is only one project file in the current working directory, GPRbuild uses this project file.

Main source files represent the sources to be used as the main programs. If they are not specified on the command line, GPRbuild uses the source files specified with the *Main* attribute in the project file. If none exists, then no executable will be built. It is also possible to specify absolute file names, or file names relative to the current directory.

When source files are specified along with the option `-c`, then recompilation will be considered only for those source files. In all other cases, GPRbuild compiles or recompiles all sources in the project tree that are not up to date, and builds or rebuilds libraries that are not up to date.

If invoked without the `--config=` or `--autoconf=` options, then GPRbuild will look for a configuration project file. The file name or path name of this configuration project file depends on the target, the runtime and environment variable *GPR_CONFIG* See *Configuring with GPRconfig*. If there is no such file in the default locations expected by GPRbuild (<install>/share/gpr and the current directory) then GPRbuild will invoke GPRconfig with the languages from the project files, and create a configuration project file `auto.cgpr` in the object directory of the main project. The project `auto.cgpr` will be rebuilt at each GPRbuild invocation unless you use the switch `--autoconf=path/auto.cgpr`, which will use the configuration project file if it exists and create it otherwise.

Options given on the GPRbuild command line may be passed along to individual tools by preceding them with one of the "command line separators" shown below. Options following the separator, up to the next separator (or end of the command line), are passed along. The different command line separators are:

- `-cargs`

  The arguments that follow up to the next command line separator are options for all compilers for all languages. Example: `-cargs -g`

- `-cargs:`*`language name`*

  The arguments that follow up to the next command line separator are options for the compiler of the specific language.

  Examples:

  - `-cargs:Ada -gnatf`
  - `-cargs:C -E`

- `-bargs`

  The arguments that follow up to the next command line separator are options for all binder drivers.

- `-bargs:`*`language name`*

  The arguments that follow up to the next command line separators are options for the binder driver of the specific language.

Examples:

- `-bargs:Ada binder_prefix=ppc-elf`
- `-bargs:C++ c_compiler_name=ccppc`

- `-largs`

  The arguments that follow up to the next command line separator are options for the linker, when linking an executable.

- `-gargs`

  The arguments that follow up to the next command line separator are options for GPRbuild itself. Usually `-gargs` is specified after one or several other command line separators.

- `-margs`

  Equivalent to `-gargs`, provided for compatibility with *gnatmake*.

3.3 Switches

GPRbuild takes into account switches that may be specified on the command line or in attributes Switches(<main or language>) or Default_Switches (<language>) in package Builder of the main project.

When there are a single main (specified on the command line or in attribute Main in the main project), the switches that are taken into account in package Builder of the main project are Switches (<main>), if declared, or Switches (<language of main>), if declared.

When there are several mains, if there are sources of the same language, then Switches (<language of main>) is taken into account, if specified.

When there are no main specified, if there is only one compiled language (that is a language with a non empty Compiler Driver), then Switches (<single language>) is taken into account, if specified.

The switches that are interpreted directly by GPRbuild are listed below.

First, the switches that may be specified only on the command line, but not in package Builder of the main project:

- `--build-script=<script_file>`

  This switch is not compatible with `--distributed=`.

  When this switch is specified, a shell script <script_file> is created. Provided that the temporary files created by gprbuild are not deleted, running this script should perform the same build as the invocation of gprbuild, with the same sources.

- `--no-project`

  This switch cannot be used if a project file is specified on the command line.

  When this switch is specified, it indicates to gprbuild that the project files in the current directory should not be considered and that the default project file in <prefix>/share/gpr is to be used.

  It is usually used with one or several mains specified on the command line.

- `--complete-output`

  This switch is not compatible with `--distributed=`.

  When this switch is specified, the standard output and the standard error of the compilations are redirected to different text files. When a source is up to date, if such text files exist, their contents are send to standard output and standard error. This allows to redisplay any warning or info from the last invocation of gprbuild –complete-output.

- `--distributed[=slave1[,slave2]]`

  This switch is not compatible with `--complete-output`, or with `--build-script=`.

  Activate the distributed compilation on the listed slaves nodes (IP or name). Or if no slave are specified they are search in *GPR_SLAVES* or *GPR_SLAVES_FILE* environment variables. see *Distributed compilation*.

- `--hash=string`

  Specify an hash string. This is just a value which is checked against the GPRslave hash value. If GPRslave has a hash value specified this string must match, otherwise it is ignored. For example:

```
$ gprbuild --hash=$(echo $ADA_PROJECT_PATH | shasum) --distributed=...
```

- `--slave-env=name`

  Use name as the slave's environment directory instead of the default one. This options is only used in distributed mode.

- `--version`

  Display information about GPRbuild: version, origin and legal status, then exit successfully, ignoring other options.

- `--help`

  Display GPRbuild usage, then exit successfully, ignoring other options.

- `--display-paths`

  Display two lines: the configuration project file search path and the user project file search path, then exit successfully, ignoring other options.

- `--config=config project file name`

  This specifies the configuration project file name. By default, the configuration project file name is `default.cgpr`. Option `--config=` cannot be specified more than once. The configuration project file specified with `--config=` must exist.

- `--autoconf=config project file name`

  This specifies a configuration project file name that already exists or will be created automatically. Option `--autoconf=` cannot be specified more than once. If the configuration project file specified with `--autoconf=` exists, then it is used. Otherwise, GPRconfig is invoked to create it automatically.

- `--target=targetname`

  This specifies that the default configuration project file is `<targetname>.cgpr`. If no configuration project file with this name is found, then GPRconfig is invoked with option `--target=targetname` to create a configuration project file `auto.cgpr`.

  Note: only one of `--config`, `--autoconf` or `--target=` can be specified.

- `--subdirs=subdir`

  This indicates that the real directories (except the source directories) are subdirectories of the directories specified in the project files. This applies in particular to object directories, library directories and exec directories. If the directories do not exist, they are created automatically. For externally built projects, the directories are never created. If such a subdirectory exists, it is used, otherwise the directory without –subdirs= is used in externally built projects.

- `--relocate-build-tree[=dir]`

  With this option it is possible to achieve out-of-tree build. That is, real object, library or exec directories are relocated to the current working directory or dir if specified.

- `--root-dir=`*`dir`*

  This option is to be used with –relocate-build-tree above and cannot be specified alone. This option specifies the root directory for artifacts for proper relocation. The default value is the main project directory. This may not be suitable for relocation if for example some artifact directories are in parent directory of the main project. The specified directory must be a parent of all artifact directories.

- `--unchecked-shared-lib-imports`

  Allow shared library projects to import projects that are not shared library projects.

- `--source-info=`*`source info file`*

  Specify a source info file. If the source info file is specified as a relative path, then it is relative to the object directory of the main project. If the source info file does not exist, then after the Project Manager has successfully parsed and processed the project files and found the sources, it creates the source info file. If the source info file already exists and can be read successfully, then the Project Manager will get all the needed information about the sources from the source info file and will not look for them. This reduces the time to process the project files, especially when looking for sources that take a long time. If the source info file exists but cannot be parsed successfully, the Project Manager will attempt to recreate it. If the Project Manager fails to create the source info file, a message is issued, but GPRbuild does not fail.

- `--restricted-to-languages=`*`list of language names`*

  Restrict the sources to be compiled to one or several languages. Each language name in the list is separated from the next by a comma, without any space.

  Example: `--restricted-to-languages=Ada,C`

  When this switch is used, switches `-c`, `-b` and `-l` are ignored. Only the compilation phase is performed and the sources that are not in the list of restricted languages are not compiled, including mains specified in package Builder of the main project.

- `--no-sal-binding`

  Specify to GPRbuild to not rebind a Stand-Alone Library (SAL), but instead to reuse the files created during a previous build of the SAL. GPRbuild will fail if there are missing files. This option is unsafe and not recommended, as it may result in incorrect binding of the SAL, for example if sources have been added, removed or modified in a significant way related to binding. It is only provided to improve performance, when it is known that the resulting binding files will be the same as the previous ones.

- `-aP` *`dir`* (Add directory `dir` to project search path)

  Specify to GPRbuild to add directory `dir` to the user project file search path, before the default directory.

- `-d` (Display progress)

  Display progress for each source, up to date or not, as a single line *completed x out of y (zz%)...* If the file needs to be compiled this is displayed after the invocation of the compiler. These lines are displayed even in quiet output mode (switch `-q`).

- `-I`*`nn`* (Index of main unit in multi-unit source file) Indicate the index of the main unit in a multi-unit source file. The index must be a positive number and there should be one and only one main source file name on the command line.

- `-eL` (Follow symbolic links when processing project files)

  By default, symbolic links on project files are not taken into account when processing project files. Switch `-eL` changes this default behavior.

- `-eS` (no effect)

This switch is only accepted for compatibility with gnatmake, but it has no effect. For gnatmake, it means: echo commands to standard output instead of standard error, but for gprbuild, commands are always echoed to standard output.

- `-F` (Full project path name in brief error messages)

  By default, in non verbose mode, when an error occurs while processing a project file, only the simple name of the project file is displayed in the error message. When switch `-F` is used, the full path of the project file is used. This switch has no effect when switch `-v` is used.

- `-o` *`name`* (Choose an alternate executable name)

  Specify the file name of the executable. Switch `-o` can be used only if there is exactly one executable being built; that is, there is exactly one main on the command line, or there are no mains on the command line and exactly one main in attribute *Main* of the main project.

- `-P` *`proj`* (use Project file *proj*)

  Specify the path name of the main project file. The space between `-P` and the project file name is optional. Specifying a project file name (with suffix `.gpr`) may be used in place of option `-P`. Exactly one main project file can be specified.

- `-r` (Recursive)

  This switch has an effect only when `-c` or `-u` is also specified and there are no mains: it means that all sources of all projects need to be compiled or recompiled.

- `-u` (Unique compilation, only compile the given files)

  If there are sources specified on the command line, only compile these sources. If there are no sources specified on the command line, compile all the sources of the main project.

  In both cases, do not attempt the binding and the linking phases.

- `-U` (Compile all sources of all projects)

  If there are sources specified on the command line, only compile these sources. If there are no sources specified on the command line, compile all the sources of all the projects in the project tree.

  In both cases, do not attempt the binding and the linking phases.

- `-vP`*`x`* (Specify verbosity when parsing Project Files)

  By default, GPRbuild does not display anything when processing project files, except when there are errors. This default behavior is obtained with switch `-vP0`. Switches `-vP1` and `-vP2` yield increasingly detailed output.

- `-Xnm=`*`val`* (Specify an external reference for Project Files)

  Specify an external reference that may be queried inside the project files using built-in function *external*. For example, with `-XBUILD=DEBUG`, *external("BUILD")* inside a project file will have the value *"DEBUG"*.

- `--compiler-subst=`*`lang`*`,`*`tool`* (Specify alternative compiler)

  Use *tool* for compiling files in language *lang*, instead of the normal compiler. For example, if `--compiler-subst=ada,my-compiler` is given, then Ada files will be compiled with *my-compiler* instead of the usual *gcc*. This and `--compiler-pkg-subst` are intended primarily for use by ASIS tools using `--incremental` mode.

- `--compiler-pkg-subst=`*`pkg`* (Specify alternative package)

  Use the switches in project-file package *pkg* when running the compiler, instead of the ones in package Compiler.

Then, the switches that may be specified on the command line as well as in package Builder of the main project (attribute Switches):

- `--keep-temp-files`

  Normally, GPRbuild delete the temporary files that it creates. When this switch is used, the temporary files that GPRbuild creates are not deleted.

- `--create-map-file`

  When linking an executable, if supported by the platform, create a map file with the same name as the executable, but with suffix `.map`.

- `--create-map-file=`*`map file`*

  When linking an executable, if supported by the platform, create a map file with file name `map file`.

- `--no-indirect-imports`

  This indicates that sources of a project should import only sources or header files from directly imported projects, that is those projects mentioned in a with clause and the projects they extend directly or indirectly. A check is done in the compilation phase, after a successful compilation, that the sources follow these restrictions. For Ada sources, the check is fully enforced. For non Ada sources, the check is partial, as in the dependency file there is no distinction between header files directly included and those indirectly included. The check will fail if there is no possibility that a header file in a non directly imported project could have been indirectly imported. If the check fails, the compilation artifacts (dependency file, object file, switches file) are deleted.

- `--indirect-imports`

  This indicates that sources of a project can import sources or header files from directly or indirectly imported projects. This is the default behavior. This switch is provided to cancel a previous switch `--no-indirect-imports` on the command line.

- `--no-object-check`

  Do not check if an object has been created after compilation.

- `--no-split-units`

  Forbid the sources of the same Ada unit to be in different projects.

- `--single-compile-per-obj-dir`

  Disallow several simultaneous compilations for the same object directory.

- `-b` (Bind only)

  Specify to GPRbuild that the post-compilation (or binding) phase is to be performed, but not the other phases unless they are specified by appropriate switches.

- `-c` (Compile only)

  Specify to GPRbuild that the compilation phase is to be performed, but not the other phases unless they are specified by appropriate switches.

- `-f` (Force recompilations)

  Force the complete processing of all phases (or of those explicitly specified) even when up to date.

- `-j`*`num`* (use *num* simultaneous compilation jobs)

  By default, GPRbuild invokes one compiler at a time. With switch `-j`, it is possible to instruct GPRbuild to spawn several simultaneous compilation jobs if needed. For example, `-j2` for two simultaneous compilation jobs or `-j4` for four. On a multi-processor system, `-j`*`num`* can greatly speed up the build process. If `-j0` is used, then the maximum number of simultaneous compilation jobs is the number of core processors on the platform.

  Switch `-j`*`num`* is also used to spawned several simultaneous binding processes and several simultaneous linking processes when there are several mains to be bound and/or linked.

- `-k` (Keep going after compilation errors)

  By default, GPRbuild stops spawning new compilation jobs at the first compilation failure. Using switch `-k`, it is possible to attempt to compile/recompile all the sources that are not up to date, even when some compilations failed. The post-compilation phase and the linking phase are never attempted if there are compilation failures, even when switch `-k` is used.

- `-l` (Link only)

  Specify to GPRbuild that the linking phase is to be performed, but not the other phases unless they are specified by appropriate switches.

- `-m` (Minimum Ada recompilation)

  Do not recompile Ada code if timestamps are different but checksums are the same.

- `-p` or `--create-missing-dirs` (Create missing object, library and exec directories)

  By default, GPRbuild checks that the object, library and exec directories specified in project files exist. Switch `-p` instructs GPRbuild to attempt to create missing directories. Note that these switches may be specified in package Builder of the main project, but they are useless there as either the directories already exist or the processing of the project files has failed before the evaluation of the Builder switches, because there is at least one missing directory.

- `-q` (Quiet output)

  Do not display anything except errors and progress (switch `-d`). Cancel any previous switch `-v`.

- `-R` (no run path option)

  Do not use a run path option to link executables or shared libraries, even when attribute Run_Path_Option is specified.

- `-s` (recompile if compilation switches have changed)

  By default, GPRbuild will not recompile a source if all dependencies are satisfied. Switch `-s` instructs GPRbuild to recompile sources when a different set of compilation switches has been used in the previous compilation, even if all dependencies are satisfied. Each time GPRbuild invokes a compiler, it writes a text file that lists the switches used in the invocation of the compiler, so that it can retrieve these switches if `-s` is used later.

- `-v` (Verbose output)

  Same as switch `-vl`.

- `-vl` (Verbose output, low level)

  Display full paths, all options used in spawned processes, as well as creations of missing directories and changes of current working directories.

- `-vm` (Verbose output, medium level)

  Not significantly different from switch `-vh`.

- `-vh` (Verbose output, high level)

  In addition to what is displayed with switch `vl`, displayed internal behavior of gprbuild and reasons why the spawned processes are invoked.

- `-we` (Treat all warnings as errors)

  When `-we` is used, any warning during the processing of the project files becomes an error and GPRbuild does not attempt any of the phases.

- `-wn` (Treat warnings as warnings)

  Switch `-wn` may be used to restore the default after `-we` or `-ws`.

- `-ws` (Suppress all warnings)

  Do not generate any warnings while processing the project files.

- `-x` (Create include path file)

  Create the include path file for the Ada compiler. This switch is often necessary when Ada sources are compiled with switch `-gnatep=`.

Switches that are accepted for compatibility with gnatmake, either on the command line or in the Builder Ada switches in the main project file:

- `-nostdinc`
- `-nostdlib`
- `-fstack-check`
- `-fno-inline`
- `-g*` Any switch starting with `-g`
- `-O*` Any switch starting with `-O`

These switches are passed to the Ada compiler.

3.4 Initialization

Before performing one or several of its three phases, GPRbuild has to read the command line, obtain its configuration, and process the project files.

If GPRbuild is invoked with an invalid switch or without any project file on the command line, it will fail immediately.

Examples:

```
$ gprbuild -P
gprbuild: project file name missing after -P

$ gprbuild -P c_main.gpr -WW
gprbuild: illegal option "-WW"
```

GPRbuild looks for the configuration project file first in the current working directory, then in the default configuration project directory. If the GPRbuild executable is located in a subdirectory `<prefix>/bin`, then the default configuration project directory is `<prefix>/share/gpr`, otherwise there is no default configuration project directory.

When it has found its configuration project path, GPRbuild needs to obtain its configuration. By default, the file name of the main configuration project is `default.cgpr`. This default may be modified using the switch `--config=...`

Example:

```
$ gprbuild --config=my_standard.cgpr -P my_project.gpr
```

If GPRbuild cannot find the main configuration project on the configuration project path, then it will look for all the languages specified in the user project tree and invoke GPRconfig to create a temporary configuration project file. This file is located in the directory computed by the following sequence: * Look for a valid absolute path in the environment variables TMPDIR, TEMP, and TMP. * If this fails, check some predefined platform-specific temp dirs (e.g. /tmp for linux). * Finally if none is accessible we fall back onto the current working directory.

The invocation of GPRconfig will take into account the target, if specified either by switch –target= on the command line or by attribute Target in the main project. Also, if Ada is one of the languages, it will take into account the Ada runtime directory, specified either by switches –RTS= or –RTS:ada= on the command line or by attribute Runtime ("Ada") in the main project file. If the Ada runtime is specified as a relative path, gprbuild will try to locate the Ada runtime directory as a subdirectory of the main project directory, or if environment variable GPR_RUNTIME_PATH is defined in the path specified by GPR_RUNTIME_PATH.

Once it has found the configuration project, GPRbuild will process its configuration: if a single string attribute is specified in the configuration project and is not specified in a user project, then the attribute is added to the user project. If a string list attribute is specified in the configuration project then its value is prepended to the corresponding attribute in the user project.

After GPRbuild has processed its configuration, it will process the user project file or files. If these user project files are incorrect then GPRbuild will fail with the appropriate error messages:

```
$ gprbuild -P my_project.gpr
ada_main.gpr:3:26: "src" is not a valid directory
gprbuild: "my_project.gpr" processing failed
```

Once the user project files have been dealt with successfully, GPRbuild will start its processing.

3.5 Compilation of one or several sources

If GPRbuild is invoked with `-u` or `-U` and there are one or several source file names specified on the command line, GPRbuild will compile or recompile these sources, if they are not up to date or if `-f` is also specified. Then GPRbuild will stop its execution.

The options/switches used to compile these sources are described in section *Compilation Phase*.

If GPRbuild is invoked with `-u` and no source file name is specified on the command line, GPRbuild will compile or recompile all the sources of the *main* project and then stop.

In contrast, if GPRbuild is invoked with `-U`, and again no source file name is specified on the command line, GPRbuild will compile or recompile all the sources of *all the projects in the project tree* and then stop.

3.6 Compilation Phase

When switch `-c` is used or when switches `-b` or `-l` are not used, GPRbuild will first compile or recompile the sources that are not up to date in all the projects in the project tree. The sources considered are:

- all the sources in languages other than Ada
- if there are no main specified, all the Ada sources
- if there is a non Ada main, but no attribute *Roots* specified for this main, all the Ada sources
- if there is a main with an attribute *Roots* specified, all the Ada sources in the closures of these Roots.
- if there is an Ada main specified, all the Ada sources in the closure of the main

Attribute Roots takes as an index a main and a string list value. Each string in the list is the name of an Ada library unit.

Example:

```
for Roots ("main.c") use ("pkga", "pkgb");
```

Package PkgA and PkgB will be considered, and all the Ada units in their closure will also be considered.

GPRbuild will first consider each source and decide if it needs to be (re)compiled.

A source needs to be compiled in the following cases:

- Switch `-f` (force recompilations) is used
- The object file does not exist
- The source is more recent than the object file
- The dependency file does not exist
- The source is more recent than the dependency file
- When `-s` is used: the switch file does not exist
- When `-s` is used: the source is more recent than the switch file
- The dependency file cannot be read
- The dependency file is empty
- The dependency file has a wrong format
- A source listed in the dependency file does not exist
- A source listed in the dependency file has an incompatible time stamp
- A source listed in the dependency file has been replaced
- Switch `-s` is used and the source has been compiled with different switches or with the same switches in a different order

When a source is successfully compiled, the following files are normally created in the object directory of the project of the source:

- An object file
- A dependency file, except when the dependency kind for the language is *none*
- A switch file if switch `-s` is used

The compiler for the language corresponding to the source file name is invoked with the following switches/options:

- The required compilation switches for the language
- The compilation switches coming from package *Compiler* of the project of the source
- The compilation switches specified on the command line for all compilers, after `-cargs`
- The compilation switches for the language of the source, specified after `-cargs:`*`language`*
- Various other options including a switch to create the dependency file while compiling, a switch to specify a configuration file, a switch to specify a mapping file, and switches to indicate where to look for other source or header files that are needed to compile the source.

If compilation is needed, then all the options/switches, except those described as 'Various other options' are written to the switch file. The switch file is a text file. Its file name is obtained by replacing the suffix of the source with `.cswi`. For example, the switch file for source `main.adb` is `main.cswi` and for `toto.c` it is `toto.cswi`.

If the compilation is successful, then if the creation of the dependency file is not done during compilation but after (see configuration attribute *Compute_Dependency*), then the process to create the dependency file is invoked.

If GPRbuild is invoked with a switch `-j` specifying more than one compilation process, then several compilation processes for several sources of possibly different languages are spawned concurrently.

For each project file, attribute Interfaces may be declared. Its value is a list of sources or header files of the project file. For a project file extending another one, directly or indirectly, inherited sources may be in the list. When Interfaces is not declared, all sources or header files are part of the interface of the project. When Interfaces is declared, only those sources or header files are part of the interface of the project file. After a successful compilation, gprbuild checks that all imported or included sources or header files that are from an imported project are part of the interface of the imported project. If this check fails, the compilation is invalidated and the compilation artifacts (dependency, object and switches files) are deleted.

Example:

```
project Prj is
   for Languages use ("Ada", "C");
   for Interfaces use ("pkg.ads", "toto.h");
end Prj;
```

If a source from a project importing project Prj imports sources from Prj other than package Pkg or includes header files from Prj other than "toto.h", then its compilation will be invalidated.

3.7 Post-Compilation Phase

The post-compilation phase has two parts: library building and program binding.

If there are libraries that need to be built or rebuilt, *gprbuild* will call the library builder, specified by attribute *Library_Builder*. This is generally the tool *gprlib*, provided with GPRbuild. If gprbuild can determine that a library is already up to date, then the library builder will not be called.

If there are mains specified, and for these mains there are sources of languages with a binder driver (specified by attribute Binder'Driver (<language>), then the binder driver is called for each such main, but only if it needs to.

For Ada, the binder driver is normally *gprbind*, which will call the appropriate version of *gnatbind*, that either the one in the same directory as the Ada compiler or the fist one found on the path. When neither of those is appropriate, it is possible to specify to *gprbind* the full path of *gnatbind*, using the Binder switch *–gnatbind_path=*.

Example:

```
package Binder is
   for Switches ("Ada") use ("--gnatbind_path=/toto/gnatbind");
end Binder;
```

If GPRbuild can determine that the artifacts from a previous post-compilation phase are already up to date, the binder driver is not called.

If there are no libraries and no binder drivers, then the post-compilation phase is empty.

3.8 Linking Phase

When there are mains specified, either in attribute Main or on the command line, and these mains are not up to date, the linker is invoked for each main, with all the specified or implied options, including the object files generated during the post-compilation phase by the binder drivers.

If switch `-jnnn` is used, with *nnn* other than 1, gprbuild will attempt to link simultaneously up to *nnn* executables.

3.9 Distributed compilation

3.9.1 Introduction to distributed compilation

For large projects the compilation time can become a limitation in the development cycle. To cope with that, GPRbuild supports distributed compilation.

In the distributed mode, the local machine (called the build master) compiles locally but also sends compilation requests to remote machines (called the build slaves). The compilation process can use one or more build slaves. Once the compilation phase is done, the build master will conduct the binding and linking phases locally.

3.9.2 Setup build environments

The configuration process to be able to use the distributed compilation support is the following:

- Optionally add a Remote package in the main project file

  This Remote package is to be placed into the project file that is passed to GPRbuild to build the application.

  The Root_Dir default value is the project's directory. This attribute designates the sources root directory. That is, the directory from which all the sources are to be found to build the application. If the project passed to GPRbuild to build the application is not at the top-level directory but in a direct sub-directory the Remote package should be:

  ```
  package Remote is
     for Root_Dir use "..";
  end Remote;
  ```

- Launch a slave driver on each build slave

  The build master will communicate with each build slave with a specific driver in charge of running the compilation process and returning statuses. This driver is *gprslave*, *GPRslave*.

  The requirement for the slaves are:

  - The same build environment must be setup (same compiler version).
  - The same libraries must be installed. That is, if the GNAT project makes use of external libraries the corresponding C headers or Ada units must be installed on the remote slaves.

  When all the requirement are set, just launch the slave driver:

  ```
  $ gprslave
  ```

When all this is done, the remote compilation can be used simply by running GPRbuild in distributed mode from the build master:

```
$ gprbuild --distributed=comp1.xyz.com,comp2.xyz.com prj.gpr
```

Alternatively the slaves can be set using the *GPR_SLAVES* environment variable. So the following command is equivalent to the above:

```
$ export GPR_SLAVES=comp1.xyz.com,comp2.xyz.com
$ gprbuild --distributed prj.gpr
```

A third alternative is proposed using a list of slaves in a file (one per line). In this case the *GPR_SLAVES_FILE* environment variable must contain the path name to this file:

```
$ export GPR_SLAVES_FILE=$HOME/slave-list.txt
$ gprbuild --distributed prj.gpr
```

Finally note that the search for the slaves are in this specific order. First the command line values, then *GPR_SLAVES* if set and finally *GPR_SLAVES_FILES*.

The build slaves are specified with the following form:

```
<machine_name>[:port]
```

3.9.3 GPRslave

This is the slave driver in charge of running the compilation jobs as requested by the build master. One instance of this tool must be launched in each build slave referenced in the project file.

Compilations for a specific project are conducted under a sub-directory from where the slave is launched by default. This can be overridden with the *-d* option below.

The current options are:

- `-v, --verbose`

  Activate the verbose mode

- `-vv`, `--debug`

  Activate the debug mode (very verbose)

- `-h`, `--help`

  Display the usage

- `-d`, `--directory=`

  Set the work directory for the slave. This is where the sources will be copied and where the compilation will take place. A sub-directory will be created for each root project built.

- `-s`, `--hash=`*`string`*

  Specify an hash string. This is just a value which is checked against the GPRbuild hash value. If set, GPRbuild hash value must match, otherwise the connection with the slave is aborted. For example:

```
$ gprslave --hash=$(echo $ADA_PROJECT_PATH | shasum)
```

- `-j`*N*, `--jobs=`*N*

  Set the maximum simultaneous compilation. The default for *N* is the number of cores.

- `-p`, `--port=`*N*

  Set the port the slave will listen to. The default value is 8484. The same port must be specified for the build slaves on *GPRbuild* command line.

- `-r`, `--response-handler=`*N*

  Set maximum number of simultaneous responses. With this option it is possible to control the number of simultaneous responses (sending back object code and ALI files) supported. The value must be between 1 and the maximum number of simultaneous compilations.

Note that a slave can be pinged to see if it is running and in response a set of information are delivered. The ping command has the following format:

```
<lower-bound><upper-bound>PG
```

When <lower-bound> and <upper-bound> are 32bits binary values for the PG string command. As an example here is how to send a ping command from a UNIX shell using the echo command:

```
echo -e "\x01\x00\x00\x00\x02\x00\x00\x00PG" | nc <HOSTNAME> 8484
```

The answer from the ping command has the following format:

:: OK<GPR Version String>[ASCII.GS]<time-stamp>[ASCII.GS]<slave hash>

The ASCII.GS is the Group Separator character whose code is 29.

This page is intentionally left blank.

CHAPTER

FOUR

GPRBUILD COMPANION TOOLS

This chapter describes the various tools that can be used in conjunction with GPRbuild.

4.1 Configuring with GPRconfig

4.1.1 Configuration

GPRbuild requires one configuration file describing the languages and toolchains to be used, and project files describing the characteristics of the user project. Typically the configuration file can be created automatically by *GPRbuild* based on the languages defined in your projects and the compilers on your path. In more involved situations — such as cross compilation, or environments with several compilers for the same language — you may need to control more precisely the generation of the desired configuration of toolsets. A tool, GPRconfig, described in *Configuring with GPRconfig*), offers this capability. In this chapter most of the examples can use autoconfiguration.

GPRbuild will start its build process by trying to locate a configuration file. The following tests are performed in the specified order, and the first that matches provides the configuration file to use.

- If a file has a base names that matches <*target*>-<*rts*>.*cgpr*, <*target.cgpr*, <*rts*>.*cgpr* or *default.cgpr* is found in the default configuration files directory, this file is used. The target and rts parameters are specified via the *–target* and *–RTS* switches of *gprbuild*. The default directory is is `share/gpr` in the installation directory of *gprbuild*
- If not found, the environment variable *GPR_CONFIG* is tested to check whether it contains the name of a valid configuration file. This can either be an absolute path name or a base name that will be searched in the same default directory as above.
- If still not found and you used the *–autoconf* switch, then a new configuration file is automatically generated based on the specified target and on the list of languages specified in your projects.

  GPRbuild assumes that there are known compilers on your path for each of the necessary languages. It is preferable and often necessary to manually generate your own configuration file when:

  - using cross compilers (in which case you need to use gprconfig's `--target=`) option,
  - using a specific Ada runtime (e.g. `--RTS=sjlj`),
  - working with compilers not in the path or not first in the path, or
  - autoconfiguration does not give the expected results.

GPRconfig provides several ways of generating configuration files. By default, a simple interactive mode lists all the known compilers for all known languages. You can then select a compiler for each of the languages; once a compiler has been selected, only compatible compilers for other languages are proposed. Here are a few examples of GPRconfig invocation:

- The following command triggers interactive mode. The configuration will be generated in GPRbuild's default location, *./default.cgpr)*, unless `-o` is used.

```
gprconfig
```

- The first command below also triggers interactive mode, but the resulting configuration file has the name and path selected by the user. The second command shows how GPRbuild can make use of this specific configuration file instead of the default one.

```
gprconfig -o path/my_config.cgpr
gprbuild --config=path/my_config.cgpr
```

- The following command again triggers interactive mode, and only the relevant cross compilers for target ppc-elf will be proposed.

```
gprconfig --target=ppc-elf
```

- The next command triggers batch mode and generates at the default location a configuration file using the first native Ada and C compilers on the path.

```
gprconfig --config=Ada --config=C --batch
```

- The next command, a combination of the previous examples, creates in batch mode a configuration file named `x.cgpr` for cross-compiling Ada with a run-time called *hi* and using C for the LEON processor.

```
gprconfig --target=leon-elf --config=Ada,,hi --config=C --batch -o x.cgpr
```

4.1.2 Using GPRconfig

Description

The GPRconfig tool helps you generate the configuration files for GPRbuild. It automatically detects the available compilers on your system and, after you have selected the one needed for your application, it generates the proper configuration file.

Note: In general, you will not launch GPRconfig explicitly. Instead, it is used implicitly by GPRbuild through the use of *–config* and *–autoconf* switches.

Command line arguments

GPRconfig supports the following command line switches:

`--target=`*`platform`*

This switch indicates the target computer on which your application will be run. It is mostly useful for cross configurations. Examples include `ppc-elf`, `ppc-vx6-windows`. It can also be used in native configurations and is useful when the same machine can run different kind of compilers such as mingw32 and cygwin on Windows or x86-32 and x86-64 on GNU Linux. Since different compilers will often return a different name for those targets, GPRconfig has an extensive knowledge of which targets are compatible, and will for example accept `x86-linux` as an alias for `i686-pc-linux-gnu`. The default target is the machine on which GPRconfig is run.

If you enter the special target `all`, then all compilers found on the `PATH` will be displayed.

`--show-targets`

As mentioned above, GPRconfig knows which targets are compatible. You can use this switch to find the list of targets that are compatible with *–target*.

`--config=`*`language`*`[,`*`version`*`[,`*`runtime`*`[,`*`path`*`[,`*`name`*`]]]]`

The intent of this switch is to preselect one or more compilers directly from the command line. This switch takes several optional arguments, which you can omit simply by passing the empty string. When omitted, the arguments will be computed automatically by GPRconfig.

In general, only *language* needs to be specified, and the first compiler on the `PATH` that can compile this language will be selected. As an example, for a multi-language application programmed in C and Ada, the command line would be:

```
--config=Ada --config=C
```

path is the directory that contains the compiler executable, for instance `/usr/bin` (and not the installation prefix `/usr`).

name should be one of the compiler names defined in the GPRconfig knowledge base. The list of supported names includes `GNAT`, `GCC`,.... This name is generally not needed, but can be used to distinguish among several compilers that could match the other arguments of `--config`.

Another possible more frequent use of *name* is to specify the base name of an executable. For instance, if you prefer to use a diab C compiler (executable is called `dcc`) instead of `gcc`, even if the latter appears first in the path, you could specify `dcc` as the name parameter.

```
gprconfig --config Ada,,,/usr/bin       # automatic parameters
gprconfig --config C,,,/usr/bin,GCC     # automatic version
gprconfig --config C,,,/usr/bin,gcc     # same as above, with exec name
```

This switch is also the only possibility to include in your project some languages that are not associated with a compiler. This is sometimes useful especially when you are using environments like GPS that support project files. For instance, if you select "Project file" as a language, the files matching the `.gpr` extension will be shown in the editor, although they of course play no role for gprbuild itself.

`--batch`

If this switch is specified, GPRconfig automatically selects the first compiler matching each of the *–config* switches, and generates the configuration file immediately. It will not display an interactive menu.

`-o` *`file`*

This specifies the name of the configuration file that will be generated. If this switch is not specified, a default file is generated in the installation directory of GPRbuild (assuming you have write access to that directory), so that it is automatically picked up by GPRbuild later on. If you select a different output file, you will need to specify it to GPRbuild.

`--db` ***`directory`*****`, --db-`** Indicates another directory that should be parsed for GPRconfig's knowledge base. Most of the time this is only useful if you are creating your own XML description files locally. Additional directories are always processed after the default knowledge base. The second version of the switch prevents GPRconfig from reading its default knowledge base.

`-h` Generates a brief help message listing all GPRconfig switches and the default value for their arguments. This includes the location of the knowledge base, the default target, etc.

Interactive use

When you launch GPRconfig, it first searches for all compilers it can find on your `PATH`, that match the target specified by `--target`. It is recommended, although not required, that you place the compilers that you expect to use for your application in your `PATH` before you launch *gprconfig*, since that simplifies the setup.

GPRconfig then displays the list of all the compilers it has found, along with the language they can compile, the run-time they use (when applicable),.... It then waits for you to select one of the compilers. This list is sorted by language, then by order in the `PATH` environment variable (so that compilers that you are more likely to use appear first), then by run-time names and finally by version of the compiler. Thus the first compiler for any language is most likely the one you want to use.

You make a selection by entering the letter that appears on the line for each compiler (be aware that this letter is case sensitive). If the compiler was already selected, it is deselected.

A filtered list of compilers is then displayed: only compilers that target the same platform as the selected compiler are now shown. GPRconfig then checks whether it is possible to link sources compiled with the selected compiler and each of the remaining compilers; when linking is not possible, the compiler is not displayed. Likewise, all compilers for the same language are hidden, so that you can only select one compiler per language.

As an example, if you need to compile your application with several C compilers, you should create another language, for instance called C2, for that purpose. That will give you the flexibility to indicate in the project files which compiler should be used for which sources.

The goal of this filtering is to make it more obvious whether you have a good chance of being able to link. There is however no guarantee that GPRconfig will know for certain how to link any combination of the remaining compilers.

You can select as many compilers as are needed by your application. Once you have finished selecting the compilers, select `s`, and GPRconfig will generate the configuration file.

4.1.3 The GPRconfig knowledge base

GPRconfig itself has no hard-coded knowledge of compilers. Thus there is no need to recompile a new version of GPRconfig when a new compiler is distributed.

Note: The role and format of the knowledge base are irrelevant for most users of GPRconfig, and are only needed when you need to add support for new compilers. You can skip this section if you only want to learn how to use GPRconfig.

All knowledge of compilers is embedded in a set of XML files called the *knowledge base*. Users can easily contribute to this general knowledge base, and have GPRconfig immediately take advantage of any new data.

The knowledge base contains various kinds of information:

- Compiler description

  When it is run interactively, GPRconfig searches the user's `PATH` for known compilers, and tries to deduce their configuration (version, supported languages, supported targets, run-times, ...). From the knowledge base GPRconfig knows how to extract the relevant information about a compiler.

  This step is optional, since a user can also enter all the information manually. However, it is recommended that the knowledge base explicitly list its known compilers, to make configuration easier for end users.

- Specific compilation switches

  When a compiler is used, depending on its version, target, run-time,..., some specific command line switches might have to be supplied. The knowledge base is a good place to store such information.

For instance, with the GNAT compiler, using the soft-float runtime should force *gprbuild* to use the `-msoft-float` compilation switch.

- Linker options

  Linking a multi-language application often has some subtleties, and typically requires specific linker switches. These switches depend on the list of languages, the list of compilers,....

- Unsupported compiler mix

  It is sometimes not possible to link together code compiled with two particular compilers. The knowledge base should store this information, so that end users are informed immediately when attempting to use such a compiler combination.

The end of this section will describe in more detail the format of this knowledge base, so that you can add your own information and have GPRconfig advantage of it.

General file format

The knowledge base is implemented as a set of XML files. None of these files has a special name, nor a special role. Instead, the user can freely create new files, and put them in the knowledge base directory, to contribute new knowledge.

The location of the knowledge base is `$prefix/share/gprconfig`, where `$prefix` is the directory in which GPRconfig was installed. Any file with extension `.xml` in this directory will be parsed automatically by GPRconfig at startup after sorting them alphabetically.

All files must have the following format:

```
<?xml version="1.0" ?>
<gprconfig>
   ...
</gprconfig>
```

The root tag must be <*gprconfig*>.

The remaining sections in this chapter will list the valid XML tags that can be used to replace the '...' code above. These tags can either all be placed in a single XML file, or split across several files.

Compiler description

One of the XML tags that can be specified as a child of <*gprconfig*> is <*compiler_description*>. This node and its children describe one of the compilers known to GPRconfig. The tool uses them when it initially looks for all compilers known on the user's `PATH` environment variable.

This is optional information, but simplifies the use of GPRconfig, since the user is then able to omit some parameters from the `--config` command line argument, and have them automatically computed.

The <*compiler_description*> node doesn't accept any XML attribute. However, it accepts a number of child tags that explain how to query the various attributes of the compiler. The child tags are evaluated (if necessary) in the same order as they are documented below.

<name> This tag contains a simple string, which is the name of the compiler. This name must be unique across all the configuration files, and is used to identify that *compiler_description* node.

```
<compiler_description>
<name>GNAT</name>
</compiler_description>
```

<executable> This tag contains a string, which is the name of an executable to search for on the PATH. Examples are `gnatls`, `gcc`,...

In some cases, the tools have a common suffix, but a prefix that might depend on the target. For instance, GNAT uses `gnatmake` for native platforms, but `powerpc-wrs-vxworks-gnatmake` for cross-compilers to VxWorks. Most of the compiler description is the same, however. For such cases, the value of the *executable* node is considered as beginning a regular expression. The tag also accepts an optional attribute *prefix*, which is an integer indicating the parenthesis group that contains the prefix. In the following example, you obtain the version of the GNAT compiler by running either *gnatls* or *powerpc-wrs-vxworks-gnatls*, depending on the name of the executable that was found.

The regular expression needs to match the whole name of the file, i.e. it contains an implicit '^' at the start, and an implicit '$' at the end. Therefore if you specify `.*gnatmake` as the regexp, it will not match `gnatmake-debug`.

A special case is when this node is empty (but it must be specified!). In such a case, you must also specify the language (see <language> below) as a simple string. It is then assumed that the specified language does not require a compiler. In the configurations file (*Configurations*), you can test whether that language was specified on the command line by using a filter such as

```
<compilers>
 <compiler language="name"/>
</compilers>
```

```
<executable prefix="1">(powerpc-wrs-vxworks-)?gnatmake</executable>
<version><external>${PREFIX}gnatls -v</external></version>
```

GPRconfig searches in all directories listed on the PATH for such an executable. When one is found, the rest of the <*compiler_description*> children are checked to know whether the compiler is valid. The directory in which the executable was found becomes the 'current directory' for the remaining XML children.

<target> This node indicates how to query the target architecture for the compiler. See *GPRconfig external values* for valid children.

If this isn't specified, the compiler will always be considered as matching on the current target.

<version> This tag contains any of the nodes defined in *GPRconfig external values* below. It shows how to query the version number of the compiler. If the version cannot be found, the executable will not be listed in the list of compilers.

<variable name="varname"> This node will define a user variable which may be later referenced. The variables are evaluated just after the version but before the languages and the runtimes nodes. See *GPRconfig external values* below for valid children of this node. If the evaluation of this variable is empty then the compiler is considered as invalid.

<languages> This node indicates how to query the list of languages. See *GPRconfig external values* below for valid children of this node.

The value returned by the system will be split into words. As a result, if the returned value is 'ada,c,c++', there are three languages supported by the compiler (and three entries are added to the menu when using GPRconfig interactively).

If the value is a simple string, the words must be comma-separated, so that you can specify languages whose names include spaces. However, if the actual value is computed from the result of a command, the words can also be space-separated, to be compatible with more tools.

<runtimes> This node indicates how to query the list of supported runtimes for the compiler. See *GPRconfig external values* below for valid children. The returned value is split into words as for <*languages*>.

This node accepts one attribute, *"default"*, which contains a list of comma-separated names of runtimes. It is used to sort the runtimes when listing which compilers were found on the PATH.

As a special case, gprconfig will merge two runtimes if the XML nodes refer to the same directories after normalization and resolution of links. As such, on Unix systems, the "adalib" link to "rts-native/adalib" (or similar) will be ignored and only the "native" runtime will be displayed.

GPRconfig external values

A number of the XML nodes described above can contain one or more children, and specify how to query a value from an executable. Here is the list of valid contents for these nodes. The <*directory*> and <*external*> children can be repeated multiple times, and the <*filter*> and <*must_match*> nodes will be applied to each of these. The final value of the external value is the concatenation of the computation for each of the <*directory*> and <*external*> nodes.

- A simple string

  A simple string given in the node indicates a constant. For instance, the list of supported languages might be defined as:

```
<compiler_description>
<name>GNAT</name>
<executable>gnatmake</executable>
<languages>Ada</languages>
</compiler_description>
```

  for the GNAT compiler, since this is an Ada-only compiler.

  Variables can be referenced in simple strings.

- <*getenv name="variable" /*>

  If the contents of the node is a <*getenv*> child, the value of the environment variable *variable* is returned. If the variable is not defined, this is an error and the compiler is ignored.

```
<compiler_description>
<name>GCC-WRS</name>
<executable prefix="1">cc(arm|pentium)</executable>
<version>
<getenv name="WIND_BASE" />
</version>
</compile_description>
```

- <*external*>*command*<*/external*>

  If the contents of the node is an <*external*> child, this indicates that a command should be run on the system. When the command is run, the current directory (i.e., the one that contains the executable found through the <*executable*> node), is placed first on the `PATH`. The output of the command is returned and may be later filtered. The command is not executed through a shell; therefore you cannot use output redirection, pipes, or other advanced features.

  For instance, extracting the target processor from *gcc* can be done with:

```
<version>
<external>gcc -dumpmachine</external>
</version>
```

Since the `PATH` has been modified, we know that the *gcc* command that is executed is the one from the same directory as the <*external*> node.

Variables are substituted in *command*.

- *<grep regexp="regexp" group="0" />*

  This node must come after the previously described ones. It is used to further filter the output. The previous output is matched against the regular expression *regexp* and the parenthesis group specified by *group* is returned. By default, group is 0, which indicates the whole output of the command.

  For instance, extracting the version number from *gcc* can be done with:

```
<version>
<external>gcc -v</external>
<grep regexp="^gcc version (\S+)" group="1" />
</version>
```

- *<directory group="0" contents="">regexp</directory>*

  If the contents of the node is a <*directory*> child, this indicates that GPRconfig should find all the files matching the regular expression. Regexp is a path relative to the directory that contains the <*executable*> file, and should use Unix directory separators (i.e. '/'), since the actual directory will be converted into this format before the match, for system independence of the knowledge base.

  The group attribute indicates which parenthesis group should be returned. It defaults to 0 which indicates the whole matched path. If this attribute is a string rather than an integer, then it is the value returned.

  regexp can be any valid regular expression. This will only match a directory or file name, not a subdirectory. Remember to quote special characters, including '.', if you do not mean to use a regexp.

  The optional attribute *contents* can be used to indicate that the contents of the file should be read. The first line that matches the regular expression given by *contents* will be used as a file path instead of the file matched by *regexp*. This is in general used on platforms that do not have symbolic links, and a file is used instead of a symbolic link. In general, this will work better than *group* specifies a string rather than a parenthesis group, since the latter will match the path matched by *regexp*, not the one read in the file.

  For instance, finding the list of supported runtimes for the GNAT compiler is done with:

```
<runtimes>
<directory group="1">
\.\./lib/gcc/${TARGET/.*/rts-(.*)/adainclude
</directory>
<directory group="default">
\.\./lib/gcc/${TARGET}/.*/adainclude
</directory>
</runtimes>}
```

  Note the second node, which matches the default run-time, and displays it as such.

- *<filter>value1,value2,...</filter>*

  This node must come after one of the previously described ones. It is used to further filter the output. The previous output is split into words (it is considered as a comma-separated or space-separated list of words), and only those words in `value1`, `value2`,... are kept.

  For instance, the *gcc* compiler will return a variety of supported languages, including 'ada'. If we do not want to use it as an Ada compiler we can specify:

```
<languages>
<external regexp="languages=(\S+)" group="1">gcc -v</external>
<filter>c,c++,fortran</filter>
</languages>
```

- *<must_match>regexp</must_match>*

  If this node is present, then the filtered output is compared with the specified regular expression. If no match is found, then the executable is not stored in the list of known compilers.

  For instance, if you want to have a *<compiler_description>* tag specific to an older version of GCC, you could write:

```
<version>
<external regexp="gcc version (\S+)"
group="1">gcc -v </external>
<must_match>2.8.1</must_match>
</version>
```

  Other versions of gcc will not match this *<compiler_description>* node.

GPRconfig variable substitution

The various compiler attributes defined above are made available as variables in the rest of the XML files. Each of these variables can be used in the value of the various nodes (for instance in *<directory>*), and in the configurations (*Configuration*).

A variable is referenced by *${name}* where *name* is either a user variable or a predefined variable. An alternate reference is *$name* where *name* is a sequence of alpha numeric characters or underscores. Finally *$$* is replaced by a simple *$*.

User variables are defined by *<variable>* nodes and may override predefined variables. To avoid a possible override use lower case names.

The variables are used in two contexts: either in a *<compiler_description>* node, in which case the variable refers to the compiler we are describing, or within a *<configuration>* node. In the latter case, and since there might be several compilers selected, you need to further specify the variable by adding in parenthesis the language of the compiler you are interested in.

For instance, the following is invalid:

```
<configuration>
<compilers>
<compiler name="GNAT" />
</compilers>
<targets negate="true">
<target name="^powerpc-elf$"/>
</targets>
<config>
package Compiler is
  for Driver ("Ada") use "${PATH}gcc";   --  Invalid !
end Compiler;
</config>
</configuration>
```

The trouble with the above is that if you are using multiple languages like C and Ada, both compilers will match the "negate" part, and therefore there is an ambiguity for the value of *${PATH}*. To prevent such issues, you need to use the following syntax instead when inside a <*configuration*> node:

```
for Driver ("Ada") use "${PATH(ada)}gcc";   --  Correct
```

Predefined variables are always in upper case. Here is the list of predefined variables

- ***EXEC*** is the name of the executable that was found through <*executable*>. It only contains the basename, not the directory information.
- ***HOST*** is replaced by the architecture of the host on which GPRconfig is running. This name is hard-coded in GPRconfig itself, and is generated by *configure* when GPRconfig was built.
- ***TARGET*** is replaced by the target architecture of the compiler, as returned by the <*target*> node. This is of course not available when computing the target itself.

  This variable takes the language of the compiler as an optional index when in a <*configuration*> block: if the language is specified, the target returned by that specific compiler is used; otherwise, the normalized target common to all the selected compilers will be returned (target normalization is also described in the knowledge base's XML files).
- ***VERSION*** is replaced by the version of the compiler. This is not available when computing the target or, of course, the version itself.
- ***PREFIX*** is replaced by the prefix to the executable name, as defined by the <*executable*> node.
- ***PATH*** is the current directory, i.e. the one containing the executable found through <*executable*>. It always ends with a directory separator.
- ***LANGUAGE*** is the language supported by the compiler, always folded to lower-case
- ***RUNTIME, RUNTIME_DIR*** This string will always be substituted by the empty string when the value of the external value is computed. These are special strings used when substituting text in configuration chunks.

  RUNTIME_DIR always end with a directory separator.
- ***GPRCONFIG_PREFIX*** is the directory in which GPRconfig was installed (e.g `"/usr/local/"` if the executable is `"/usr/local/bin/gprconfig"`. This directory always ends with a directory separator. This variable never takes a language in parameter, even within a <*configuration*> node.

If a variable is not defined, an error message is issued and the variable is substituted by an empty string.

Configurations

The second type of information stored in the knowledge base are the chunks of *gprbuild* configuration files.

Each of these chunks is also placed in an XML node that provides optional filters. If all the filters match, then the chunk will be merged with other similar chunks and placed in the final configuration file that is generated by GPRconfig.

For instance, it is possible to indicate that a chunk should only be included if the GNAT compiler with the soft-float runtime is used. Such a chunk can for instance be used to ensure that Ada sources are always compiled with the *-msoft-float* command line switch.

GPRconfig does not perform sophisticated merging of chunks. It simply groups packages together. For example, if the two chunks are:

```
chunk1:
   package Language_Processing is
     for Attr1 use ("foo");
```

```
   end Language_Processing;
chunk2:
   package Language_Processing is
     for Attr1 use ("bar");
   end Language_Processing;
```

Then the final configuration file will look like:

```
package Language_Processing is
  for Attr1 use ("foo");
  for Attr1 use ("bar");
end Language_Processing;
```

As a result, to avoid conflicts, it is recommended that the chunks be written so that they easily collaborate together. For instance, to obtain something equivalent to

```
package Language_Processing is
  for Attr1 use ("foo", "bar");
end Language_Processing;
```

the two chunks above should be written as:

```
chunk1:
  package Language_Processing is
    for Attr1 use Language_Processing'Attr1 & ("foo");
  end Language_Processing;
chunk2:
  package Language_Processing is
    for Attr1 use Language_Processing'Attr1 & ("bar");
end Language_Processing;
```

The chunks are described in a <*configuration*> XML node. The most important child of such a node is <*config*>, which contains the chunk itself. For instance, you would write:

```
<configuration>
  ...  list of filters, see below
  <config>
  package Language_Processing is
     for Attr1 use Language_Processing'Attr1 & ("foo");
  end Language_Processing;
  </config>
</configuration>
```

If <*config*> is an empty node (i.e., `<config/>` or `<config></config>` was used), then the combination of selected compilers will be reported as invalid, in the sense that code compiled with these compilers cannot be linked together. As a result, GPRconfig will not create the configuration file.

The special variables (*GPRconfig variable substitution*) are also substituted in the chunk. That allows you to compute some attributes of the compiler (its path, the runtime,...), and use them when generating the chunks.

The filters themselves are of course defined through XML tags, and can be any of:

<compilers negate="false"> This filter contains a list of <*compiler*> children. The <*compilers*> filter matches if any of its children match. However, you can have several <*compilers*> filters, in which case they must all match. This can be used to include linker switches chunks. For instance, the following code would be used to describe the linker switches to use when GNAT 5.05 or 5.04 is used in addition to g++ 3.4.1:

```
<configuration>
  <compilers>
    <compiler name="GNAT" version="5.04" />
    <compiler name="GNAT" version="5.05" />
  </compilers>
  <compilers>
    <compiler name="G++" version="3.4.1" />
  </compilers>
  ...
</configuration>
```

If the attribute *negate* is `true`, then the meaning of this filter is inverted, and it will match if none of its children matches.

The format of the <*compiler*> is the following:

```
<compiler name="name" version="..."
runtime="..." language="..." />
```

The language attribute, when specified, matches the corresponding attribute used in the <*compiler_description*> children. All other attributes are regular expressions, which are matched against the corresponding selected compilers. When an attribute is not specified, it will always match. Matching is done in a case-insensitive manner.

For instance, to check a GNAT compiler in the 5.x family, use:

```
<compiler name="GNAT" version="5.\d+" />
```

<hosts negate="false"> This filter contains a list of <*host*> children. It matches when any of its children matches. You can specify only one <*hosts*> node. The format of <*host*> is a node with one mandatory attribute *name*, which is a regexp matched against the architecture on which GPRconfig is running, and one optional attribute *except*, which is also a regexp, but a negative one. If both *name* and *except* match the architecture, corresponding <*configuration*> node is ignored. The name of the architecture was computed by *configure* when GPRconfig was built. Note that the regexp might match a substring of the host name, so you might want to surround it with "^" and "$" so that it only matches the whole host name (for instance, "elf" would match "powerpc-elf", but "^elf$" would not).

If the *negate* attribute is `true`, then the meaning of this filter is inverted, and it will match when none of its children matches.

For instance, to activate a chunk only if the compiler is running on an Intel Linux machine, use:

```
<hosts>
  <host name="i.86-.*-linux(-gnu)?" />
</hosts>
```

<targets negate="false"> This filter contains a list of <*target*> children. It behaves exactly like <*hosts*>, but matches against the architecture targeted by the selected compilers. For instance, to activate a chunk only when the code is targeted for linux, use:

If the *negate* attribute is `true`, then the meaning of this filter is inverted, and it will match when none of its children matches.

```
<targets>
  <target name="i.86-.*-linux(-gnu)?" />
```

```
</targets>
```

4.2 Configuration File Reference

A text file using the project file syntax. It defines languages and their characteristics as well as toolchains for those languages and their characteristics.

GPRbuild needs to have a configuration file to know the different characteristics of the toolchains that can be used to compile sources and build libraries and executables.

A configuration file is a special kind of project file: it uses the same syntax as a standard project file. Attributes in the configuration file define the configuration. Some of these attributes have a special meaning in the configuration.

The default name of the configuration file, when not specified to GPRbuild by switches `--config=` or `--autoconf=` is `default.cgpr`. Although the name of the configuration file can be any valid file name, it is recommended that its suffix be `.cgpr` (for Configuration GNAT Project), so that it cannot be confused with a standard project file which has the suffix `.gpr`.

When `default.cgpr` cannot be found in the configuration project path, GPRbuild invokes GPRconfig to create a configuration file.

In the following description of the attributes, when an attribute is an indexed attribute and its index is a language name, for example *Spec_Suffix (<language>)*, then the name of the language is case insensitive. For example, both *C* and *c* are allowed.

Any attribute may appear in a configuration project file. All attributes in a configuration project file are inherited by each user project file in the project tree. However, usually only the attributes listed below make sense in the configuration project file.

4.2.1 Project Level Configuration Attributes

General Attributes

- Default_Language

  Specifies the name of the language of the immediate sources of a project when attribute *Languages* is not declared in the project. If attribute *Default_Language* is not declared in the configuration file, then each user project file in the project tree must have an attribute *Languages* declared, unless it extends another project. Example:

  ```
  for Default_Language use "ada";
  ```

- Run_Path_Option

  Specifies a 'run path option'; i.e., an option to use when linking an executable or a shared library to indicate the path (Rpath) where to look for other libraries. The value of this attribute is a string list. When linking an executable or a shared library, the search path is concatenated with the last string in the list, which may be an empty string.

  Example:

  ```
  for Run_Path_Option  use ("-Wl,-rpath,");
  ```

- Run_Path_Origin

  Specifies the string to be used in an Rpath to indicate the directory of the executable, allowing then to have Rpaths specified as relative paths.

  Example:

```
for Run_Path_Origin use "$ORIGIN";
```

- Toolchain_Version (<language>)

  Specifies a version for a toolchain, as a single string. This toolchain version is passed to the library builder. Example:

```
for Toolchain_Version ("Ada") use "GNAT 6.1";
```

  This attribute is used by GPRbind to decide on the names of the shared GNAT runtime libraries.

- Toolchain_Description (<language>)

  Specifies as a single string a description of a toolchain. This attribute is not directly used by GPRbuild or its auxiliary tools (GPRbind and GPRlib) but may be used by other tools, for example GPS. Example:

```
for Toolchain_Description ("C") use "gcc version 4.1.3 20070425";
```

General Library Related Attributes

- Library_Support

  Specifies the level of support for library project. If this attribute is not specified, then library projects are not supported. The only potential values for this attribute are *none*, *static_only* and *full*. Example:

```
for Library_Support use "full";
```

- Library_Builder

  Specifies the name of the executable for the library builder. Example:

```
for Library_Builder use "/.../gprlib";
```

Archive Related Attributes

- Archive_Builder

  Specifies the name of the executable of the archive builder with the minimum options, if any. Example:

```
for Archive_Builder use ("ar", "cr");
```

- Archive_Indexer

  Specifies the name of the executable of the archive indexer with the minimum options, if any. If this attribute is not specified, then there is no archive indexer. Example:

```
for Archive_Indexer use ("ranlib");
```

- Archive_Suffix

  Specifies the suffix of the archives. If this attribute is not specified, then the suffix of the archives is defaulted to `.a`. Example:

```
for Archive_Suffix use ".olb"; --  for VMS
```

- Library_Partial_Linker

  Specifies the name of the executable of the partial linker with the options to be used, if any. If this attribute is not specified, then there is no partial linking. Example:

```
for Library_Partial_Linker use ("gcc", "-nostdlib", "-Wl,-r", "-o");
```

Shared Library Related Attributes

- Shared_Library_Prefix

  Specifies the prefix of the file names of shared libraries. When this attribute is not specified, the prefix is *lib*. Example:

```
for Shared_Library_Prefix use ""; --  for Windows, if needed
```

- Shared_Library_Suffix

  Specifies the suffix of the file names of shared libraries. When this attribute is not specified, the suffix is `.so`. Example:

```
for Shared_Library_Suffix use ".dll"; --  for Windows
```

- Symbolic_Link_Supported

  Specifies if symbolic links are supported by the platforms. The possible values of this attribute are *"false"* (the default) and *"true"*. When this attribute is not specified, symbolic links are not supported.

```
for Symbolic_Link_Supported use "true";
```

- Library_Major_Minor_ID_Supported

  Specifies if major and minor IDs are supported for shared libraries. The possible values of this attribute are *"false"* (the default) and *"true"*. When this attribute is not specified, major and minor IDs are not supported.

```
for Library_Major_Minor_ID_Supported use "True";
```

- Library_Auto_Init_Supported

  Specifies if library auto initialization is supported. The possible values of this attribute are *"false"* (the default) and *"true"*. When this attribute is not specified, library auto initialization is not supported.

```
for Library_Auto_Init_Supported use "true";
```

- Shared_Library_Minimum_Switches

  Specifies the minimum options to be used when building a shared library. These options are put in the appropriate section in the library exchange file when the library builder is invoked. Example:

```
for Shared_Library_Minimum_Switches use  ("-shared");
```

- Library_Version_Switches

  Specifies the option or options to be used when a library version is used. These options are put in the appropriate section in the library exchange file when the library builder is invoked. Example:

```
for Library_Version_Switches use ("-Wl,-soname,");
```

- Runtime_Library_Dir (<language>)

  Specifies the directory for the runtime libraries for the language. Example:

```
for Runtime_Library_Dir ("Ada") use "/path/to/adalib";
```

  This attribute is used by GPRlib to link shared libraries with Ada code.

- Object_Lister

  Specifies the name of the executable of the object lister with the minimum options, if any. This tool is used to list symbols out of object code to create a list of the symbols to export. Example:

```
for Object_Lister use ("nm", "-g", "--demangle");
```

- Object_Lister_Matcher

  A regular expression pattern for matching symbols out of the output of Object_Lister tool. Example:

```
for Object_Lister_Matcher use " T (.*)";
```

- Export_File_Format

  The export file format to generate, this is either DEF (Windows), Flat or GNU. Example:

```
for Export_File_Format use "GNU";
```

- Export_File_Switch

  The required switch to pass the export file to the linker. Example:

```
for Export_File_Switch use "-Wl,--version-script=";
```

4.2.2 Package Naming

Attributes in package *Naming* of a configuration file specify defaults. These attributes may be used in user project files to replace these defaults.

The following attributes usually appear in package *Naming* of a configuration file:

- Spec_Suffix (<language>)

  Specifies the default suffix for a 'spec' or header file. Examples:

```
for Spec_Suffix ("Ada") use ".ads";
for Spec_Suffix ("C")   use ".h";
for Spec_Suffix ("C++") use ".hh";
```

- Body_Suffix (<language>)

  Specifies the default suffix for a 'body' or a source file. Examples:

```
for Body_Suffix ("Ada") use ".adb";
for Body_Suffix ("C")   use ".c";
for Body_Suffix ("C++") use ".cpp";
```

- Separate_Suffix

  Specifies the suffix for a subunit source file (separate) in Ada. If attribute *Separate_Suffix* is not specified, then the default suffix of subunit source files is the same as the default suffix for body source files. Example:

```
for Separate_Suffix use ".sep";
```

- Casing

  Specifies the casing of spec and body files in a unit based language (such as Ada) to know how to map a unit name to its file name. The values for this attribute may only be *"lowercase"*, *"UPPERCASE"* and *"Mixedcase"*. The default, when attribute *Casing* is not specified is lower case. This attribute rarely needs to be specified, since on platforms where file names are not case sensitive (such as Windows or VMS) the default (lower case) will suffice.

- Dot_Replacement

  Specifies the string to replace a dot ('.') in unit names of a unit based language (such as Ada) to obtain its file name. If there is any unit based language in the configuration, attribute *Dot_Replacement* must be declared. Example:

```
for Dot_Replacement use "-";
```

4.2.3 Package Builder

- Executable_Suffix

  Specifies the default executable suffix. If no attribute *Executable_Suffix* is declared, then the default executable suffix for the host platform is used. Example:

```
for Executable_Suffix use ".exe";
```

4.2.4 Package Compiler

General Compilation Attributes

- Driver (<language>)

Specifies the name of the executable for the compiler of a language. The single string value of this attribute may be an absolute path or a relative path. If relative, then the execution path is searched. Specifying the empty string for this attribute indicates that there is no compiler for the language.

Examples:

```
for Driver ("C++") use "g++";
for Driver ("Ada") use "/.../bin/gcc";
for Driver ("Project file") use "";
```

- Required_Switches (<language>)

  Specifies the minimum options that must be used when invoking the compiler of a language. Examples:

```
for Required_Switches ("C")   use ("-c", "-x", "c");
for Required_Switches ("Ada") use ("-c", "-x", "ada", "-gnatA");
```

- PIC_Option (<language>)

  Specifies the option or options that must be used when compiling a source of a language to be put in a shared library. Example:

```
for PIC_Option ("C") use ("-fPIC");
```

Mapping File Related Attributes

- Mapping_File_Switches (<language>)

  Specifies the switch or switches to be used to specify a mapping file to the compiler. When attribute *Mapping_File_Switches* is not declared, then no mapping file is specified to the compiler. The value of this attribute is a string list. The path name of the mapping file is concatenated with the last string in the string list, which may be empty. Example:

```
for Mapping_File_Switches ("Ada") use ("-gnatem=");
```

- Mapping_Spec_Suffix (<language>)

  Specifies, for unit based languages that support mapping files, the suffix in the mapping file that needs to be added to the unit name for specs. Example:

```
for Mapping_Spec_Suffix ("Ada") use "%s";
```

- Mapping_Body_Suffix (<language>)

  Specifies, for unit based languages that support mapping files, the suffix in the mapping file that needs to be added to the unit name for bodies. Example:

```
for Mapping_Spec_Suffix ("Ada") use "%b";
```

Config File Related Attributes

In the value of config file attributes defined below, there are some placeholders that GPRbuild will replace. These placeholders are:

Placeholder	Interpretation
%u	unit name
%f	source file name
%s	spec suffix
%b	body suffix
%c	casing
%d	dot replacement string

Attributes:

- Config_File_Switches (<language>)

  Specifies the switch or switches to be used to specify a configuration file to the compiler. When attribute *Config_File_Switches* is not declared, then no config file is specified to the compiler. The value of this attribute is a string list. The path name of the config file is concatenated with the last string in the string list, which may be empty. Example:

```
for Config_File_Switches ("Ada") use ("-gnatec=");
```

- Config_Body_File_Name (<language>)

  Specifies the line to be put in a config file to indicate the file name of a body. Example:

```
for Config_Body_File_Name ("Ada") use
    "pragma Source_File_Name_Project (%u, Body_File_Name => ""%f"");";
```

- Config_Spec_File_Name (<language>)

  Specifies the line to be put in a config file to indicate the file name of a spec. Example:

```
for Config_Spec_File_Name ("Ada") use
    "pragma Source_File_Name_Project (%u, Spec_File_Name => ""%f"");";
```

- Config_Body_File_Name_Pattern (<language>)

  Specifies the line to be put in a config file to indicate a body file name pattern. Example:

```
for Config_Body_File_Name_Pattern ("Ada") use
    "pragma Source_File_Name_Project " &
    "   (Body_File_Name  => ""*%b"","  &
    "    Casing          => %c,"  &
    "    Dot_Replacement => ""%d"");";
```

- Config_Spec_File_Name_Pattern (<language>)

  Specifies the line to be put in a config file to indicate a spec file name pattern. Example:

```
for Config_Spec_File_Name_Pattern ("Ada") use
    "pragma Source_File_Name_Project " &
    "   (Spec_File_Name  => ""*%s"","  &
    "    Casing          => %c,"  &
    "    Dot_Replacement => ""%d"");";
```

- Config_File_Unique (<language>)

  Specifies, for languages that support config files, if several config files may be indicated to the compiler, or not. This attribute may have only two values: *"true"* or *"false"* (case insensitive). The default, when this attribute

is not specified, is *"false"*. When the value *"true"* is specified for this attribute, GPRbuild will concatenate the config files, if there are more than one. Example:

```
for Config_File_Unique ("Ada") use "True";
```

Dependency Related Attributes

There are two dependency-related attributes: *Dependency_Switches* and *Dependency_Driver*. If neither of these two attributes are specified for a language other than Ada, then the source needs to be (re)compiled if the object file does not exist or the source file is more recent than the object file or the switch file.

- Dependency_Switches (<language>)

  For languages other than Ada, attribute *Dependency_Switches* specifies the option or options to add to the compiler invocation so that it creates the dependency file at the same time. The value of attribute *Dependency_Option* is a string list. The name of the dependency file is added to the last string in the list, which may be empty. Example:

  ```
  for Dependency_Switches ("C") use ("-Wp,-MD,");
  ```

  With these *Dependency_Switches*, when compiling `file.c` the compiler will be invoked with the option `-Wp,-MD,file.d`.

- Dependency_Driver (<language>)

  Specifies the command and options to create a dependency file for a source. The full path name of the source is appended to the last string of the string list value. Example:

  ```
  for Dependency_Driver ("C") use ("gcc", "-E", "-Wp,-M", "");
  ```

  Usually, attributes *Dependency_Switches* and *Dependency_Driver* are not both specified.

Search Path Related Attributes

- Include_Switches (<language>)

  Specifies the option or options to use when invoking the compiler to indicate that a directory is part of the source search path. The value of this attribute is a string list. The full path name of the directory is concatenated with the last string in the string list, which may be empty. Example:

  ```
  for Include_Switches ("C") use ("-I");
  ```

  Attribute *Include_Switches* is ignored if either one of the attributes *Include_Path* or *Include_Path_File* are specified.

- Include_Path (<language>)

  Specifies the name of an environment variable that is used by the compiler to get the source search path. The value of the environment variable is the source search path to be used by the compiler. Example:

  ```
  for Include_Path ("C")   use "CPATH";
  for Include_Path ("Ada") use "ADA_INCLUDE_PATH";
  ```

  Attribute *Include_Path* is ignored if attribute *Include_Path_File* is declared for the language.

- Include_Path_File (<language>)

  Specifies the name of an environment variable that is used by the compiler to get the source search path. The value of the environment variable is the path name of a text file that contains the path names of the directories of the source search path. Example:

```
for Include_Path_File ("Ada") use "ADA_PRJ_INCLUDE_FILE";
```

4.2.5 Package Binder

- Driver (<language>)

  Specifies the name of the executable of the binder driver. When this attribute is not specified, there is no binder for the language. Example:

```
for Driver ("Ada") use "/.../gprbind";
```

- Required_Switches (<language>)

  Specifies the minimum options to be used when invoking the binder driver. These options are put in the appropriate section in the binder exchange file, one option per line. Example:

```
for Required_Switches ("Ada") use ("--prefix=<prefix>");
```

- Prefix (<language>)

  Specifies the prefix to be used in the name of the binder exchange file. Example:

```
for Prefix ("C++") use ("c__");
```

- Objects_Path (<language>)

  Specifies the name of an environment variable that is used by the compiler to get the object search path. The value of the environment variable is the object search path to be used by the compiler. Example:

```
for Objects_Path ("Ada") use "ADA_OBJECTS_PATH";
```

- Objects_Path_File (<language>)

  Specifies the name of an environment variable that is used by the compiler to get the object search path. The value of the environment variable is the path name of a text file that contains the path names of the directories of the object search path. Example:

```
for Objects_Path_File ("Ada") use "ADA_PRJ_OBJECTS_FILE";
```

4.2.6 Package Linker

- Driver

  Specifies the name of the executable of the linker. Example:

```
for Driver use "g++";
```

- Required_Switches

  Specifies the minimum options to be used when invoking the linker. Those options are happened at the end of the link command so that potentially conflicting user options take precedence.

- Map_File_Option

  Specifies the option to be used when the linker is asked to produce a map file.

```
for Map_File_Option use "-Wl,-Map,";
```

- Max_Command_Line_Length

  Specifies the maximum length of the command line to invoke the linker. If this maximum length is reached, a response file will be used to shorten the length of the command line. This is only taken into account when attribute Response_File_Format is specified.

```
for Max_Command_Line_Length use "8000";
```

- Response_File_Format

  Specifies the format of the response file to be generated when the maximum length of the command line to invoke the linker is reached. This is only taken into account when attribute Max_Command_Line_Length is specified.

  The allowed case-insensitive values are:

  - **"GNU"** Used when the underlying linker is gnu ld.
  - **"Object_List"** Used when the response file is a list of object files, one per line.
  - **"GCC_GNU"** Used with recent version of gcc when the underlined linker is gnu ld.
  - **"GCC_Object_List"** Used with recent version of gcc when the underlying linker is not gnu ld.

```
for Response_File_Format use "GCC_GNU";
```

- Response_File_Switches

  Specifies the option(s) that must precede the response file name when when invoking the linker. This is only taken into account when both attributes Max_Command_Line_Length and Response_File_Format are specified.

```
for Response_File_Switches  use ("-Wl,-f,");
```

4.3 Cleaning up with GPRclean

The GPRclean tool removes the files created by GPRbuild. At a minimum, to invoke GPRclean you must specify a main project file in a command such as *gprclean proj.gpr* or *gprclean -P proj.gpr*.

Examples of invocation of GPRclean:

```
gprclean -r prj1.gpr
gprclean -c -P prj2.gpr
```

4.3.1 Switches for GPRclean

The switches for GPRclean are:

- `--no-project`

  This switch cannot be used if a project file is specified on the command line.

  When this switch is specified, it indicates to gprclean that the project files in the current directory should not be considered and that the default project file in <prefix>/share/gpr is to be used.

  It is usually used with one or several mains specified on the command line.

- `--distributed`

  Also clean-up the sources on build slaves, see *Distributed compilation*.

- `--slave-env=`*`name`*

  Use *name* as the slave's environment directory instead of the default one. This options is only used in distributed mode.

- `--config=`*`config project file name`*

  Specify the configuration project file name.

- `--autoconf=`*`config project file name`*

  This specifies a configuration project file name that already exists or will be created automatically. Option `--autoconf=` cannot be specified more than once. If the configuration project file specified with `--autoconf=` exists, then it is used. Otherwise, GPRconfig is invoked to create it automatically.

- `--target=`*`targetname`*

  Specify a target for cross platforms.

- `--db` *`dir`*

  Parse *dir* as an additional knowledge base.

- `--db-`

  Do not parse the standard knowledge base.

- `--RTS=`*`runtime`*

  Use runtime *runtime* for language Ada.

- `--RTS:`*`lang=runtime`*

  Use runtime *runtime* for language *lang*.

- `--subdirs=`*`dir`*

  Real object, library or exec directories are subdirectories *dir* of the specified ones.

- `--relocate-build-tree[=`*`dir`*`]`

  With this option it is possible to achieve out-of-tree build. That is, real object, library or exec directories are relocated to the current working directory or dir if specified.

- `--root-dir=`*`dir`*

  This option is to be used with –relocate-build-tree above and cannot be specified alone. This option specifies the root directory for artifacts for proper relocation. The default value is the main project directory. This may not be suitable for relocation if for example some artifact directories are in parent directory of the main project. The specified directory must be a parent of all artifact directories.

- `--unchecked-shared-lib-imports`

  Shared library projects may import any project.

- `-aP`*`dir`*

  Add directory *dir* to the project search path.

- `-c`

  Only delete compiler-generated files. Do not delete executables and libraries.

- `-eL`

  Follow symbolic links when processing project files.

- `-f`

  Force deletions of unwritable files.

- `-F`

  Display full project path name in brief error messages.

- `-h`

  Display the usage.

- `-n`

  Do not delete files, only list files that would be deleted.

- `-P`*`proj`*

  Use Project File *proj*.

- `-q`

  Be quiet/terse. There is no output, except to report problems.

- `-r`

  Recursive. Clean all projects referenced by the main project directly or indirectly. Without this switch, GPRclean only cleans the main project.

- `-v`

  Verbose mode.

- `-vP`*`x`*

  Specify verbosity when parsing Project Files. *x* = 0 (default), 1 or 2.

- `-Xnm=`*`val`*

  Specify an external reference for Project Files.

4.4 Installing with GPRinstall

The GPRinstall tool installs projects. With GPRinstall it is not needed to create complex *makefiles* to install the components. This also removes the need for OS specific commands (like *cp*, *mkdir* on UNIXs) and so makes the installation process easier on all supported platforms.

After building a project it is often necessary to install the project to make it accessible to other projects. GPRinstall installs only what is necessary and nothing more. That is, for a library project the library itself is installed with the corresponding ALI files for Ada sources, but the object code is not installed as it not needed. Also if the Ada specs are

installed the bodies are not, because they are not needed in most cases. The cases where the bodies are required (if the spec has inline routines or is a generic) are properly detected by GPRinstall.

Furthermore, we can note that GPRinstall handles the preprocessed sources. So it installs the correct variant of the source after resolving the preprocessing directives.

The parts of a project that can be installed are:

- sources of a project
- a static or shared library built from a library project
- objects built from a standard project
- executables built from a standard project

Moreover, GPRinstall will create, when needed, a project to use the installed sources, objects or library. By default, this project file is installed in the GPRbuild's default path location so that it can be "with"ed easily without further configuration. The installation process keeps record of every file installed for easy and safe removal.

GPRinstall supports all kind of project:

- standard projects

  The object files, executable and source files are considered for installation.

- library and aggregate library projects

  The library itself and the source files are considered for installation.

- aggregate projects

  All aggregated projects are considered for installation.

Projects that won't be installed are:

- Project explicitly disabled for installation

  A project with the Active attribute set to False in the project's Install package.

- Projects with no sources

  Both abstract projects and standard projects without any sources

At a minimum, to invoke GPRinstall you must specify a main project file in a command such as *gprinstall proj.gpr* or *gprinstall -P proj.gpr* (in installing mode) or the install name (in uninstalling mode) *gprinstall –uninstall proj*.

Examples of invocation of GPRinstall:

```
gprinstall prj1.gpr
gprinstall -r --prefix=/my/root/install -P prj2.gpr
```

GPRinstall will record the installation under the *install name* which is by default the name of the project without the extension. That is above the project install names are *prj1* and *prj2*.

The installation name can be specified with the option *–install-name*. This makes it possible to record the installation of multiple projects under the same name. This is handy if an application comes with a library and a set of tools built with multiple projects. In this case we may want to record the installation under the same name. The install name is also used as a suffix to group include and library directories.

Examples of installation under the same name:

```
gprinstall --install-name=myapp lib.gpr
gprinstall --install-name=myapp --mode=usage tools/tools.gpr
```

Note the *–mode=usage* option above. This tells GPRinstall to only install the executable built as part of the project.

It is possible to uninstall a project by using the *–uninstall* option. In this case we just pass the install name to GPRinstall:

```
gprinstall --uninstall prj1
gprinstall --uninstall prj2
```

And both *lib.gpr* and *tools.gpr* above will be uninstalled with:

```
gprinstall --uninstall myapp
```

Note that GPRinstall does not deal with dependencies between projects. Also GPRinstall in uninstall mode does not need nor use information in the installed project. This is because the project may not be present anymore and many different project scenario may have been installed. So when uninstalling GPRinstall just use the manifest file (whose name is the install name) information.

4.4.1 Switches for GPRinstall

The switches for GPRinstall are:

- `--config=`*`main config project file name`*

  Specify the configuration project file name

- `--autoconf=`*`config project file name`*

  This specifies a configuration project file name that already exists or will be created automatically. Option `--autoconf=` cannot be specified more than once. If the configuration project file specified with `--autoconf=` exists, then it is used. Otherwise, GPRconfig is invoked to create it automatically.

- `--build-name`

  Specify under which name the current project build must be installed. The default value is *default*. Using this option it is possible to install different builds (using different configuration, options, etc...) of the same project. The given name will be used by client to select which build they want to use (link against).

- `--build-var`

  Specify the name of the build variable in the installed project. If this options is not used, the default build variable used is *<PROJECT_NAME>_BUILD*.

  It is possible to specify multiple variables in –build-var option. In this case, if the first build variable is not found, the second one will be checked, and so on. This makes it possible to have a project specific variable to select the corresponding build and a more generic build variable shared by multiple projects.

  ::

  $ gprinstall -Pproject1

  –build-var=PROJECT1_BUILD,LIBRARY_TYPE ^ Scenario variable to control specifically this project

  ^ Scenario variable to control the default for a set of projects

  $ gprinstall -Pproject2 –build-var=PROJECT2_BUILD,LIBRARY_TYPE

- `--no-build-var`

  Specify that no build/scenario variable should be generated. This option can be use for a project where there is single configuration, so a single installation. This option cannot be used with `--build-var`.

- `--dry-run`

  Install nothing, just display the actions that would have been done.

- `-a`

  Install all the sources (default). Cannot be used with *-m* below.

- `-m`

  Install only the interface sources (minimal set of sources). Cannot be used with *-a* above.

- `-f`

  Force overwriting of existing files

- `-h`

  Display this message

- `--mode=[dev/usage]`

  Specify the installation mode.

  - dev This is the default mode. The installation is done in developer mode. All files to use the project are copied to to install prefix. For a library this means that the specs, the corresponding ALI files for Ada units and the library itself (static or relocatable) are installed. For a standard project the object files are installed instead of the library.
  - usage The installation is done in usage mode. This means that only the library or the executable is installed. In this installation mode there is no project generated, nor specs or ALI files installed.

Mode	Interpretation
dev	For this mode the binaries (built libraries and executable) are installed together with the sources to use them.
usage	For this mode only the binaries are installed and no project are created.

- `-p`, `--create-missing-dirs`

  Create missing directories in the installation location.

- `-P`*`proj`*

  Specify the project file to install.

- `--prefix=`*`path`*

  Specify the location of the installation. If not specified, the default location for the current compiler is used. That is, *path* corresponds to parent directory where *gprinstall* is found.

- `--install-name=`*`name`*

  Specify the name to use for recording the installation. The default is the project name without the extension. If set this option is also used as include or library directories' suffix to group all related installations under a common directory.

- `--sources-subdir=`*`path`*

  Specify the value for the sources installation directory if an absolute path. Otherwise it is appended to the prefix above. The default is *include/<project_name>[.<build-name>]*

- `--lib-subdir=`*`path`*

  Specify the value for the library and object installation directory if an absolute path. Otherwise it is appended to the prefix above. The default is *lib/<project_name>[.<build-name>]*

- `--link-lib-subdir=`*`path`*

  Specify the value for the library symlink directory if an absolute path. Otherwise it is appended to the prefix above.

- `---exec-subdir=`*`path`*

  Specify the value for the executables installation directory if an absolute path. Otherwise it is appended to the prefix above. The default is *bin*.

- `--project-subdir=`*`path`*

  Specify the value for the project installation directory if an absolute path. Otherwise it is appended to the prefix above. The default is *share/gpr*.

- `--no-project`

  Specify that no project is to be generated and installed.

- `--target=`*`targetname`*

  Specify a target for cross platforms.

- `--no-lib-link`

  Disable copy of shared libraries into the executable directory on Windows or creation of symlink in the lib directory on UNIX. This is done by default to place the shared libraries into a directory where application will look for them.

- `--sources-only`

  Copy only sources part of the project, the object, library or executable files are never copied. When this switch is used the installed project is not set as externally built.

- `--side-debug`

  Write debug symbols out of executables and libraries into a separate file. The separate file is named after the main file with an added *.debug* extension. That is, if the executable to be installed is named *main*, then a file *main.debug* is also created in the same location, containing only the debug information. The debug information is then removed from the *main* executable.

- `--subdirs=`*`subdir`*

  This indicates that the real directories (except the source directories) are subdirectories of the directories specified in the project files. This applies in particular to object directories, library directories and exec directories. If the directories do not exist, they are created automatically. It is expected that the sub-dir option value here is the one used with gprbuild.

- `--relocate-build-tree[=`*`dir`*`]`

  With this option it is possible to achieve out-of-tree build. That is, real object, library or exec directories are relocated to the current working directory or dir if specified.

- `--root-dir=`*`dir`*

  This option is to be used with –relocate-build-tree above and cannot be specified alone. This option specifies the root directory for artifacts for proper relocation. The default value is the main project directory. This may not be suitable for relocation if for example some artifact directories are in parent directory of the main project. The specified directory must be a parent of all artifact directories.

- `-q`

  Be quiet/terse. There is no output, except to report problems.

- `-r`

  (Recursive.) Install all projects referenced by the main project directly or indirectly. Without this switch, GPRinstall only installs the main project.

- `--uninstall`

  Uninstall mode, files installed for a given project or install name will be removed. A check is done that no manual changes have been applied to the files before removing. Deletion of the files can be forced in this case by using the `-f` option. Note that the paramater in this case is not the project name but the install name which corresponds to the manifest file.

- `--list`

  List mode, displays all the installed packaged.

- `--stat`

  Apply to list mode above, displays also some statistics about the installed packages : number of files, total size used on disk, and whether there is some files missing.

- `-v`

  Verbose mode

- `-Xnm=`*`val`*

  Specify an external reference for Project Files.

4.5 Specifying a Naming Scheme with GPRname

When the Ada source file names do not follow a regular naming scheme, the mapping of Ada units to source file names must be indicated in package Naming with attributes Spec and Body.

To help maintain the correspondence between compilation unit names and source file names within the compiler, the tool *gprname* may be used to generate automatically these attributes.

4.5.1 Running *gprname*

The usual form of the *gprname* command is:

```
$ gprname ['switches'] 'naming_pattern' ['naming_patterns']
    [--and ['switches'] 'naming_pattern' ['naming_patterns']]
```

Most of the arguments are optional: switch *-P* must be specified to indicate the project file and at least one Naming Pattern.

gprname will attempt to find all the compilation units in files that follow at least one of the naming patterns. To find Ada compilation units, *gprname* will use the GNAT compiler in syntax-check-only mode on all regular files.

One or several Naming Patterns may be given as arguments to *gprname*. Each Naming Pattern is enclosed between double quotes (or single quotes on Windows). A Naming Pattern is a regular expression similar to the wildcard patterns used in file names by the Unix shells or the DOS prompt.

gprname may be called with several sections of directories/patterns. Sections are separated by switch *–and*. In each section, there must be at least one pattern. If no directory is specified in a section, the project directory is implied. The options other that the directory switches and the patterns apply globally even if they are in different sections.

Examples of Naming Patterns are:

```
"*.[12].ada"
"*.ad[sb]*"
"body_*"    "spec_*"
```

For a more complete description of the syntax of Naming Patterns, see the second kind of regular expressions described in `g-regexp.ads` (the 'Glob' regular expressions).

4.5.2 Switches for GPRname

Switches for *gprname* must precede any specified Naming Pattern.

You may specify any of the following switches to *gprname*:

- `--version`

  Display Copyright and version, then exit disregarding all other options.

- `--target=<targ>`

  Indicates the target of the GNAT compiler. This may be needed if there is no native compiler available.

- `--help`

  If *–version* was not used, display usage, then exit disregarding all other options.

- `--subdirs=dir`

  Real object, library or exec directories are subdirectories <dir> of the specified ones.

- `--no-backup`

  Do not create a backup copy of the project file if it already exists.

- `--ignore-duplicate-files`

  Ignore files with the same basename, and take the first one found into account only. By default when encountering a duplicate file, a warning is emitted, and duplicate entries in the *Naming* package will be generated, needing manual editing to resolve the conflict. With this switch, gprname assumes that only the first file should be used and others should be ignored.

- `--ignore-predefined-units`

  Ignore predefined units (children of System, Interfaces and Ada packages).

- `--and`

  Start another section of directories/patterns.

- `-ddir`

  Look for source files in directory `dir`. There may be zero, one or more spaces between *-d* and `dir`. `dir` may end with /**, that is it may be of the form *root_dir*/**. In this case, the directory *root_dir* and all of its subdirectories, recursively, have to be searched for sources. When a switch *-d* is specified, the current working directory will not be searched for source files, unless it is explicitly specified with a *-d* or *-D* switch. Several switches *-d* may be specified. If `dir` is a relative path, it is relative to the directory of the project file specified with switch *-P*. The directory specified with switch *-d* must exist and be readable.

- `-Dfilename`

  Look for source files in all directories listed in text file `filename`. There may be zero, one or more spaces between *-D* and `filename`. `filename` must be an existing, readable text file. Each nonempty line in `filename` must be a directory. Specifying switch *-D* is equivalent to specifying as many switches *-d* as there are nonempty lines in `file`.

- `-eL`

  Follow symbolic links when processing project files.

- `-fpattern`

  Foreign C language patterns. Using this switch, it is possible to add sources of language C to the list of sources of a project file.

  For example,

```
gprname -P prj.gpr -f"*.c" "*.ada" -f "*.clang"
```

  will look for Ada units in all files with the `.ada` extension, and will add to the list of file for project `prj.gpr` the C files with extensions `.c` and `.clang`. Attribute Languages will be declared with the list of languages with sources. In the above example, it will be ("Ada", "C") if Ada and C sources have been found.

- `-f:<lang> pattern`

  Foreign language {<lang>} patterns. Using this switch, it is possible to add sources of language <lang> to the list of sources of a project file.

  For example,

```
gprname -P prj.gpr "*.ada" -f:C++ "*.cpp" -f:C++ "*.CPP"
```

  Files with extensions `.cpp` and `*.CPP` are C++ sources. Attribute Languages will have value ("Ada", "C++") if Ada and C++ sources are found.

- `-h`

  Output usage (help) information. The output is written to `stdout`.

- `-Pproj`

  Create or update project file `proj`. There may be zero, one or more space between *-P* and `proj`. `proj` may include directory information. `proj` must be writable. There must be only one switch *-P*. If switch *–no-backup* is not specified, a backup copy of the project file is created in the project directory with file name <proj>.gpr.saved_x. 'x' is the first non negative number that makes this backup copy a new file.

- `-v`

  Verbose mode. Output detailed explanation of behavior to `stdout`. This includes name of the file written, the name of the directories to search and, for each file in those directories whose name matches at least one of the Naming Patterns, an indication of whether the file contains a unit, and if so the name of the unit.

- `-v -v`

  Very Verbose mode. In addition to the output produced in verbose mode, for each file in the searched directories whose name matches none of the Naming Patterns, an indication is given that there is no match.

- `-xpattern`

  Excluded patterns. Using this switch, it is possible to exclude some files that would match the name patterns. For example,

```
gprname -P prj.gpr -x "*_nt.ada" "*.ada"
```

  will look for Ada units in all files with the `.ada` extension, except those whose names end with `_nt.ada`.

4.5.3 Example of *gprname* Usage

```
$ gprname -P/home/me/proj.gpr -x "*_nt_body.ada"
-dsources -dsources/plus -Dcommon_dirs.txt "body_*" "spec_*"
```

Note that several switches *-d* may be used, even in conjunction with one or several switches *-D*. Several Naming Patterns and one excluded pattern are used in this example.

4.6 The Library Browser GPRls

gprls is a tool that outputs information about compiled sources. It gives the relationship between objects, unit names and source files. It can also be used to check source dependencies as well as various characteristics.

4.6.1 Running *gprls*

The *gprls* command has the form

```
$ gprls switches `object_or_dependency_files`
```

The main argument is the list of object files or `ali` files for Ada sources for which information is requested.

gprls uses a project file, either specified through a single switch -P, or the default project file. If no *object_or_dependency_files* is specified then all the object files corresponding to the sources of the project are deemed to be specified.

In normal mode, without option other that -P <project file>, *gprls* produces information for each object/dependency file: the full path of the object, the name of the principal unit in this object if the source is in Ada, the status of the source and the full path of the source.

Here is a simple example of use:

```
$ gprls -P prj.gpr
/my_path/obj/pkg.o
   pkg
     DIF pkg.adb
/my_path/obj/main.o
   main
     MOK main.adb
```

The first three lines can be interpreted as follows: the main unit which is contained in object file `pkg.o` is pkg, whose main source is in `pkg.adb`. Furthermore, the version of the source used for the compilation of pkg has been modified (DIF). Each source file has a status qualifier which can be:

OK (unchanged) The version of the source file used for the compilation of the specified unit corresponds exactly to the actual source file.

MOK (slightly modified) The version of the source file used for the compilation of the specified unit differs from the actual source file but not enough to require recompilation. If you use *gprbuild* with the qualifier *-m (minimal recompilation)*, a file marked MOK will not be recompiled.

DIF (modified) The source used to build this object has been modified and need to be recompiled.

??? (dependency file not found) The object/dependency file cannot be found.

4.6.2 Switches for GPRls

gprls recognizes the following switches:

--version Display Copyright and version, then exit disregarding all other options.

--help If *–version* was not used, display usage, then exit disregarding all other options.

--closure Display the Ada closures of the mains specified on the command line or in attribute Main of the main project. The absolute paths of the units in the closures are listed, but no status is checked. If all the ALI files are found, then the list is preceded with the line "Closure:" or "Closures:". Otherwise, it is preceded with the line "Incomplete Closure:" or "Incomplete closures:".

-P <project file> Use this project file. This switch may only be specified once.

-a Consider all units, including those of the predefined Ada library. Especially useful with *-d*.

-d List sources from which specified units depend on.

-h Output the list of options.

-o Only output information about object files.

-s Only output information about source files.

-u Only output information about compilation units.

-U If no object/dependency file is specified, list information for the sources of all the projects in the project tree.

-files=*file* Take as arguments the files listed in text file *file*. Text file *file* may contain empty lines that are ignored. Each nonempty line should contain the name of an existing object/dependency file. Several such switches may be specified simultaneously.

-aP*dir* Add *dir* at the beginning of the project search dir.

--RTS=*rts-path*` Specifies the default location of the Ada runtime library. Same meaning as the equivalent *gprbuild* switch.

-v Verbose mode. Output the complete source, object and project paths. For each Ada source, include special characteristics such as:

- *Preelaborable*: The unit is preelaborable in the Ada sense.
- *No_Elab_Code*: No elaboration code has been produced by the compiler for this unit.
- *Pure*: The unit is pure in the Ada sense.
- *Elaborate_Body*: The unit contains a pragma Elaborate_Body.
- *Remote_Types*: The unit contains a pragma Remote_Types.
- *Shared_Passive*: The unit contains a pragma Shared_Passive.
- *Predefined*: This unit is part of the predefined environment and cannot be modified by the user.
- *Remote_Call_Interface*: The unit contains a pragma Remote_Call_Interface.

4.6.3 Examples of *gprls* Usage

```
$ gprls -v -P prj.gpr

 5 lines: No errors
gprconfig --batch -o /my_path/obj/auto.cgpr --target=x86_64-linux --config=ada,,
Creating configuration file: /my_path/obj/auto.cgpr
```

```
Checking configuration /my_path/obj/auto.cgpr

GPRLS Pro 17.0 (20161010) (x86_64-unknown-linux-gnu)
Copyright (C) 2015-2016, AdaCore

Source Search Path:
   <Current directory>
   /my_path/local/lib/gcc/x86_64-pc-linux-gnu/4.9.4//adainclude/

Object Search Path:
   <Current directory>
   /my_path/local/lib/gcc/x86_64-pc-linux-gnu/4.9.4//adalib/

Project Search Path:
   <Current_Directory>
   /my_path/local/x86_64-unknown-linux-gnu/lib/gnat
   /my_path/local/x86_64-unknown-linux-gnu/share/gpr
   /my_path/local/share/gpr
   /my_path/local/lib/gnat

/my_path/obj/pkg.o
   Unit =>
     Name   => pkg
     Kind   => package body
     Flags  => No_Elab_Code
   Source => pkg.adb unchanged
   Unit =>
     Name   => pkg
     Kind   => package spec
     Flags  => No_Elab_Code
   Source => pkg.ads unchanged
/my_path/obj/main.o
   Unit =>
     Name   => main
     Kind   => subprogram body
     Flags  => No_Elab_Code
   Source => main.adb slightly modified

$ gprls -d -P prj.gpr main.o
/my_path/obj/main.o
   main
       MOK main.adb

        OK pkg.ads

$ gprls -s -P prj.gpr main.o
   main
main.adb
```

APPENDIX

A

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **Document**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **Opaque**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
15. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

INDEX

Symbols

A

B

C

D

E

R

S

T

V

W