GPRbuild User's Guide

Document identifier: $Rev:: 171835 $
Date: $Date:: 2011-03-30#$

AdaCore

1 GNAT Project Manager

1.1 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:

- 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 for any of the project-enabled tools; you can apply these settings either globally or to individual compilation units.
- The source files containing the main subprogram(s) to be built
- The source programming language(s)
- Source file naming conventions; you can specify these either globally or for individual compilation units (see Section 1.2.8 [Naming Schemes], page 13).
- Change any of the above settings depending on external values, thus enabling the reuse of the projects in various **scenarios** (see Section 1.4 [Scenarios in Projects], page 21).
- Automatically build libraries as part of the build process (see Section 1.5 [Library Projects], page 23).

Project files are written in a syntax close to that of Ada, using familiar notions such as packages, context clauses, declarations, default values, assignments, and inheritance (see Section 1.8 [Project File Reference], page 42).

Project files can be built hierarchically from other project files, simplifying complex system integration and project reuse (see Section 1.3 [Organizing Projects into Subsystems], page 16).

- One project can import other projects containing needed source files. More generally, the Project Manager lets you structure large development efforts into hierarchical subsystems, where build decisions are delegated to the subsystem level, and thus different compilation environments (switch settings) used for different subsystems.
- You can organize GNAT projects in a hierarchy: a child project can extend a parent project, inheriting the parent's source files and optionally overriding any of them with alternative versions (see Section 1.6 [Project Extension], page 31).

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*' and '`-X`*vbl*`=`*value*'). See Section 2.1.1 [Switches Related to Project Files], page 59.

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. It can be used for instance, for generating 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. It can be used for instance, for grouping and hiding

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 various sections below introduce the different concepts related to projects. Each section starts with examples and use cases, and then goes into the details of related project file capabilities.

1.2 Building With Projects

In its simplest form, a unique project is used to build a single executable. This section concentrates on such a simple setup. Later sections will extend this basic model to more complex setups.

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 using an Ada-like syntax, generally using 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 Section 1.8.9 [Attributes], page 51.

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 Section 1.8.4 [Packages], page 45.

Project variables:
In addition to attributes, a project can use variables to store intermediate values and avoid duplication in complex expressions. It can be initialized with a value coming from the environment. A frequent use of variables is to define scenarios. See Section 1.8.6 [External Values], page 49, See Section 1.4 [Scenarios in Projects], page 21, and See Section 1.8.8 [Variables], page 50.

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 compilation unit 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 Section 1.2.8 [Naming Schemes], page 13, 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 with their own project file.

The following subsections introduce gradually all the attributes of interest for simple build needs. Here is the simple 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 withs package Pack. We want to compile these source files with the switch '-O2', and put the resulting files in the directory 'obj/'.

```
common/
  pack.ads
  pack.adb
  proc.adb
common/release/
  proc.ali, proc.o pack.ali, pack.o
```

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 enough for it. We will thus create a new file, that for now should contain the following code:

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

1.2.1 Source Files and Directories

When you create a new project, the first thing to describe is how to find the corresponding source files. This is 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,...).

First step is to declare the source directories, which are the directories to be searched to find source files. In the case of the example, the 'common' directory is the only source directory.

There are several ways of defining source directories:

- When the attribute **Source_Dirs** is not used, 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 has 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 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 path 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 ".".

  When using that construct, it can sometimes be 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 .svn/**. To do that, attribute **Ignore_Source_Sub_Dirs** can be used. It specifies the list of simple file names for the root of these undesirable directory subtrees.

When applied to the simple example, and because we generally prefer to have the project file at the toplevel directory rather than mixed with the sources, we will create the following file

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

Once source directories have been specified, one may need to indicate 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 specify the list of sources to consider explicitly. 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 search 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 to say, 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. Tools like `gnatmake` only know about Ada, while other tools like `gprbuild` know about many more languages such as C, C++, Fortran, assembly and others can be added dynamically.

Even when using only Ada, the default naming might not be suitable. Indeed, how does the project manager recognizes an "Ada file" from any other file? Project files can describe the naming scheme used for source files, and override the default (see Section 1.2.8 [Naming Schemes], page 13). The default is the standard GNAT extension ('`.adb`' for bodies and '`.ads`' for specs), which is what is used in our example, explaining why no naming scheme is explicitly specified. See Section 1.2.8 [Naming Schemes], page 13.

- `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 reports errors if they cannot be found in the source directories or does not correspond to the naming scheme.
- For various reasons, 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 see Section 1.3 [Organizing Projects into Subsystems], page 16. 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 great 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.It might be more convenient to use the default search rules with specific exceptions. This can be done thanks to 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 Section 1.6 [Project Extension], page 31. 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 it is considered an error for a project file to have no sources attached to it unless explicitly declared as mentionend above.

If the order of the source directories is known statically, that is if `"**"` is not used in the string list `Source_Dirs`, then there may be several files with the same source file name sitting 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 not used in the string list `Source_Dirs`, it is an error to have several files with the same source file name in the same directory `"**"` subtree, since there would be an ambiguity as to which one should be used. However, two files with the same source file name may in two single directories or directory subtrees. In this case, the one in the first directory or directory subtree is a source of the project.

1.2.2 Object and Exec Directory

The next step when writing a project is to indicate where the compiler should put the object files. In fact, the compiler and other tools might create several different kind of files (for GNAT, there is the object file and the ALI file for instance). One of the important concepts in projects is that most tools may consider source directories as read-only and do not attempt to create new or temporary files there. Instead, all files are created in the object directory. It is of course not true for project-aware IDEs, whose purpose it 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 `gnatmake` and `gprbuild`).

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

For our example, we can specify the object dir in this way:

```
project Build is
   for Source_Dirs use ("common");
   for Object_Dir use "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 in several directories, you can either move all of them into the same directory if you want to build it with a single project file, or study the section on subsystems

(see Section 1.3 [Organizing Projects into Subsystems], page 16) to see how each separate object directory can be associated with one of the subsystem constituting the application.

When the `linker` is called, it usually creates an executable. By default, this executable is placed in the object directory of the project. It might be convenient to store it in its own directory.

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. When not specified, It defaults to the object directory and therefore to the project file's directory if neither *Object_Dir* nor *Exec_Dir* was specified.

In the case of the example, let's place the executable in the root of the hierarchy, ie the same directory as '`build.gpr`'. Hence the project file is now

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

1.2.3 Main Subprograms

In the previous section, executables were mentioned. The project manager needs to be taught what they are. In a project file, an executable is indicated by pointing to source file of the main subprogram. In C this is the file that contains the `main` function, and in Ada the file that contains the main unit.

There can be any number of such main files within a given project, and thus several executables can be built in the context of a single project file. Of course, one given executable might not (and in fact will not) need all the source files referenced by the project. As opposed to other build environments such as `makefile`, one does not need to specify the list of dependencies of each executable, the project-aware builders knows enough of the semantics of the languages to build ands 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

```
gnatmake -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.

1.2.4 Tools Options in Project Files

We now have a project file that fully describes our environment, and can be used to build the application with a simple `gnatmake` command as seen in the previous section. In fact, the empty project we showed immediately at the beginning (with no attribute at all) could already fullfill that need if it was put in the '`common`' directory.

Of course, we always want more control. This section will show 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 into the project file, it is not necessary to use switches on the command line for this purpose (switches 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 response files for this and thus should not be subject to command line overflows.

Several tools are participating to the creation of an executable: the compiler produces object files from the source files; the binder (in the Ada case) creates an source file that takes care, among other things, of elaboration issues and global variables initialization; and the linker gathers everything into a single executable that users can execute. All these tools are known by the project manager and will be called with user defined switches from the project files. However, we need to introduce a new project file concept to express which switches to be used for any of the tools involved in the build.

A project file is subdivided into zero or more **packages**, each of which contains 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 Section 1.8.4 [Packages], page 45), and the contents of packages are limited to a small set of constructs and attributes (see Section 1.8.9 [Attributes], page 51).

Our example project file can be extended with the following 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");
end Build;

package Builder is  --<<<  for gnatmake and 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;
```

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 usual, on the command line, to pass it to the builder which then passes it to the compiler. It is recommended to use directly the right package, which will make the setup easier to understand for other people.

Several attributes can be used to specify the switches:

Default_Switches:

This is the first mention in this manual of an **indexed attribute**. When this attribute is defined, one 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 '`-O2`' switch, and the resulting project file is as follows (only the `Compiler` package is shown):

```
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 possible at high level of optimization because of a compiler issue. In such a case, the *Switches* attribute (indexed on the file name) can be used and will override the switches defined by *Default_Switches*. 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 `gnatmake` and `gprbuild`), the *Binder* package (binding Ada executables) and the *Linker* package (for linking executables).

1.2.5 Compiling with Project Files

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

```
gnatmake -Pbuild
```

This will automatically build the executables specified through 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.

`gnatmake` only knows how to handle Ada files. By using `gprbuild` as a builder, you could automatically manage C files the same way: create the file '`utils.c`' in the '`common`' directory, set the attribute *Languages* to `"(Ada, C)"`, and 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's execution is kept reasonably simple and easy to understand. In particular, some of the less frequently used commands are not shown, and some parameters are abbreviated. So it is not possible to rerun the effect of the `gprbuild` command by cut-and-pasting its output. GPRbuild's option `-v` provides a much more verbose output which includes, among other information, more complete compilation, post-compilation and link commands.

1.2.6 Executable File Names

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

For instance, instead of building `proc` (or `proc.exe` on Windows), we could configure our project file to build `"proc1"` (resp proc1.exe) with the following addition:

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

Attribute **Executable_Suffix**, when specified, may change the suffix of the executable files, when no attribute `Executable` applies: its value replace the platform-specific executable suffix. The default executable suffix is 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`'. When several mains are defined in the project, it is not possible to use the '`-o`' switch and the only way to change the names of the executable is provided by Attributes `Executable` and `Executable_Suffix`.

1.2.7 Avoid Duplication With Variables

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. The attribute *Exec_Dir* is now omitted, thus the resulting executable will be put in the 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 revisit the use of variables in the context of scenarios (see Section 1.4 [Scenarios in Projects], page 21).

In this example, we see how the file '`main.c`' can 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 (') 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.

1.2.8 Naming Schemes

Sometimes an Ada software system is ported from one compilation environment to another (say GNAT), and the file are not 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 finding of source files when searching in the source directories, and given a source file name it makes it possible to guess the associated language, and thus the compiler to use.

Note that the use by the Ada compiler of pragmas Source_File_Name is not supported when using project files. You must use the features described in this paragraph. You can however specify other configuration pragmas (see Section 2.1.3 [Specifying Configuration Pragmas], page 63).

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 regards to the Ada unit name. Given an Ada unit My_Unit, the file name will respectively be '`my_unit.adb`' (lowercase), '`MY_UNIT.ADB`' (uppercase) or '`My_Unit.adb`' (mixedcase). 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 `'.'` except if the entire string is `"."`

Spec_Suffix and **Specification_Suffix**:
For Ada, these attributes give the suffix used in file names that contain specifications. For other languages, they give the extension for files that contain declaration (header files in C for instance). The attribute is indexed on the language. The two attributes are equivalent, but the latter is obsolescent. If `Spec_Suffix ("Ada")` is

not specified, then the default is `".ads"`. The value must satisfy the following requirements:

- It must not be empty
- It cannot start with an alphanumeric character
- It cannot start with an underscore followed by an alphanumeric character
- It must include at least one dot

Body_Suffix and **Implementation_Suffix**:

These attributes give the extension used for file names that contain code (bodies in Ada). They are indexed on the language. The second version is obsolescent and fully replaced by the first attribute.

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`, a file called '`Makefile`' will be part of the project. This capability is usually not interesting 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 that contain separate bodies. If it is not specified, then it defaults to same value as `Body_Suffix ("Ada")`. The same rules apply as for the `Body_Suffix` attribute. The only accepted index is `"Ada"`.

Spec or **Specification**:

This attribute `Spec` 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* for Ada bodies.

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

For example, 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;
```

1.3 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 one single object directory. A system can be composed of a single subsystem when it is simple as we have seen in the first section. 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, the previous example is being extended. 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.

1.3.1 Project Dependencies

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 easily solved 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, `gnatmake` will automatically check the other 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, the attribute **Externally_Built** to `"true"` can be set, 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, although 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 Section 1.7 [Aggregate Projects], page 34).

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 with the following algorithm, in that order, as soon as a matching file is found, the search stops:

- 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** and **ADA_PROJECT_PATH** (in

that order) if they exist. The former is recommended, the latter is kept for backward compatibility.

- Finally, it is searched relative to the default project directories. Such directories depends on the tool used. The different locations searched in the specified order are:
  - '`<prefix>/<target>/lib/gnat`' (for `gnatmake` in all cases, and for `gprbuild` if option '`--target`' is specified)
  - '`<prefix>/share/gpr/`' (for `gnatmake` and `gprbuild`)
  - '`<prefix>/lib/gnat/`' (for `gnatmake` and `gprbuild`)

  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 `gnatls -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.

One very important aspect of a project hierarchy is that **a given source can only belong to one project** (otherwise the project manager would not know which settings apply to it and when to recompile it). It means that different project files do not usually share source directories or when they do, they need to specify precisely which project owns which sources using attribute `Source_Files` or equivalent. By contrast, 2 projects can each own a source with the same base file name as long as they live in different directories. The latter is not true for Ada Sources because of the correlation betwen source files and Ada units.

1.3.2 Cyclic Project Dependencies

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

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

```
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;
```

1.3.3 Sharing Between Projects

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

As seen before (see Section 1.2.4 [Tools Options in Project Files], page 9), 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 files with the same value. Of course, that means duplication of data, and both places need to be changed in order to recompile the whole application with different switches. It can become a real problem 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 it 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 second version 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 third project. That project is setup without any sources (so that, as opposed to the first example, none of the project plays a special role), and will only be used to define the attributes. Such a project is typically called '`shared.gpr`'.

```
abstract project Shared is
   for Source_Files use ();   --  no project
   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 for 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 of course 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 sources (see Section 1.8.2 [Qualified Projects], page 44 for more qualifiers).

1.3.4 Global Attributes

We have already seen many examples of attributes used to specify a special option of one of the tools involved in the build process. Most of those attributes are project specific. That it 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 apply to all projects in a hierarchy as long as they are defined on the "main" project. The main project is the project explicitly mentioned on the command-line. The project hierarchy is the "with"-closure of the main project.

Here is a list of commonly used global attributes:

Builder.Global_Configuration_Pragmas:
: This attribute points to a file that contains configuration pragmas to use when building executables. These pragmas apply for all executables build from this project hierarchy. As we have seen before, 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 to use when compiling any source file in the project hierarchy. 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 on the name of the language.

Using such global capabilities is convenient. It can also lead to unexpected behavior. Especially when several subsystems are shared among different main projects and the different global attributes are not compatible. Note that using aggregate projects can be a safer and more powerful replacement to global attributes.

1.4 Scenarios in Projects

Various aspects of the projects can be modified based on **scenarios**. These are user-defined modes that change the behavior of a project. 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, when the second will focus on improving code optimization).

Let's enhance our example to support a debug and a release modes.The issue is to let the user choose what kind of system he is building: use '`-g`' as compiler switches in debug mode and '`-O2`' in release mode. We will also setup 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 2 modes.

One naive 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 presence of multiple projects depending on each other, you will also have to duplicate the complete hierarchy and adapt the project files to point to the right copies.

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

Command line:
When launching `gnatmake` or `gprbuild`, the user can pass extra '`-X`' switches to define the external value. In our case, the command line might look like

```
    gnatmake -Pbuild.gpr -Xmode=debug
or  gnatmake -Pbuild.gpr -Xmode=release
```

Environment variables:
When the external value does not come from the command line, it can come from the value of environment variables of the appropriate name. In our case, if an environment variable called `"mode"` exist, its value will be taken into account.

External function second parameter

We now need to get that value in the project. The general form is to use the predefined function **external** which returns the current value of the external. For instance, we could setup 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.

In order to set the switches according to the different scenarios, other constructs have to be introduced such as typed variables and case statements.

A **typed variable** is a variable that can take only a limited number of values, similar to an enumeration in Ada. Such a variable can then be used in a **case statement** and create 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;
```

The project has suddenly grown in size, but has become much more flexible. `Mode_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 statement to set the switches depending on the scenario the user has chosen.

Most aspects of the projects can depend on scenarios. The notable exception are project dependencies (`with` clauses), which may not depend on a scenario.

Scenarios work the same way with **project hierarchies**: you can either duplicate a variable similar to `Mode` in each of the project (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 Section 1.3.3 [Sharing Between Projects], page 19).

1.5 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 **stand-**

alone libraries (SAL) which offers 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.

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).

1.5.1 Building Libraries

Let's enhance our example and transform the `logging` subsystem into a library.In orer to do so, a few changes need to be made to '`logging.gpr`'. A number of specific attributes needs to be defined: at least `Library_Name` and `Library_Dir`; in addition, a number of other attributes can be used to specify specific aspects of the library. For readablility, 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; however, there may be system specific restrictions on the name. In general, it is recommended to stick to alphanumeric characters (and possibly 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, the object files end up in the explicit or implicit `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 exists 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 enough 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"`, `"dynamic"` or `"relocatable"` (the latter is a synonym for dynamic). It indicates which kind of library should be build (the default is to build a static library, that is an archive of object files that can potentially be linked into a static executable). When the library is set to be dynamic, a separate image is created that will be loaded independnently, 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 independant way so that the same project file can be used to build a library on different Oses.

If you need to build both a static and a dynamic library, it is recommended use 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 which uses scenarios to indicate at the various kinds of library to be build 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 can be hundreds of them. The same restrictions as for the `Library_Dir` attribute apply.

Library_Version:

This attribute is platform dependent, and has no effect on VMS and 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 'libdummy.so.1' library and a symbolic link to it called 'libdummy.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 is calling 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.

Leading_Library_Options:
This attribute, that is taken into account only by gprbuild, may be used to specified leading options (first switches) when linking a shared library.

Linker.Linker_Options:
This attribute specifies 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.

1.5.2 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 rebuild 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 shard 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.

It is also possible to build **multi-language libraries**. When using `gprbuild` as a builder, multi-language library projects allow naturally the creation of multi-language libraries . `gnatmake`, does n ot try to compile non Ada sources. However, when the project is multi-language, it will automatically link all object files found in the object directory, whether or not they were compiled from an Ada source file. This specific behavior does not apply to Ada-only projects which only take into account the objects corresponding to the sources of the project.

A non-library project can import a library project. When the builder is invoked on the former, the library of the latter is only rebuilt when absolutely necessary. For instance, if a unit of the library is not up-to-date but non of the executables need this unit, then the unit is not recompiled and the library is not reassembled. 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` only if '`proc.ads`', '`pack.ads`' or '`pack.adb`' 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 needs to be used:

```
gnatmake -Plogging.gpr
gnatmake -Pbuild.gpr
```

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

Library projects can also be useful to describe a library that need to be used but, for some reason, cannot be rebuilt. For instance, it is the case when some of the library sources are not available. Such library projects need simply 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 sub-

systems 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 thanks to a project `with` clause, which in turn will trigger the builder to find the proper order of libraries in the final link command.

1.5.3 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 & redeployement of complex systems when localized changes are made.

The most proeminent 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 characterised by a third attribute, **Library_Interface**, in addition to the two attributes that make a project a Library Project (`Library_Name` and `Library_Dir`).

Library_Interface:

This attribute defines an explicit subset of the units of the project. 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 "loggin";
for Library_Interface use ("lib1", "lib2");  --  unit names
```

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". A static Stand-alone Library is

never automatically initialized. Specifying "false" for this attribute prevent 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 is a generic units in the spec. 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 and, on some platforms (like VMS) that have the notions of major and minor IDs built in the library files, it controls the setting of these IDs. It is not supported on all platforms (where it will just have no effect). It may have one of the following values:

- "autonomous" or "default": exported symbols are not controlled
- "compliant": if attribute **Library_Reference_Symbol_File** is not defined, then it is equivalent to policy "autonomous". If there are exported symbols in the reference symbol file that are not in the object files of the interfaces, the major ID of the library is increased. If there are symbols in the object files

of the interfaces that are not in the reference symbol file, these symbols are put at the end of the list in the newly created symbol file and the minor ID is increased.

- `"controlled"`: the attribute **Library_Reference_Symbol_File** must be defined. The library will fail to build if the exported symbols in the object files of the interfaces do not match exactly the symbol in the symbol file.
- `"restricted"`: The attribute **Library_Symbol_File** must be defined. The library will fail to build if there are symbols in the symbol file that are not in the exported symbols of the object files of the interfaces. Additional symbols in the object files are not added to the symbol file.
- `"direct"`: The attribute **Library_Symbol_File** must be defined and must designate an existing file in the object directory. This symbol file is passed directly to the underlying linker without any symbol processing.

Library_Reference_Symbol_File
: This attribute may define the path name of a reference symbol file that is read when the symbol policy is either `"compliant"` or `"controlled"`, on platforms that support symbol control, such as VMS, when building a stand-alone library. The path may be an absolute path or a path relative to the project directory.

Library_Symbol_File
: This attribute may define the name of the symbol file to be created when building a stand-alone library when the symbol policy is either `"compliant"`, `"controlled"` or `"restricted"`, on platforms that support symbol control, such as VMS. When symbol policy is `"direct"`, then a file with this name must exist in the object directory.

1.5.4 Installing a library with project files

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

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. Section 1.5.2 [Using Library Projects], page 26

1.6 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 for instance that our example `Build` project is build every night for the whole team, in some shared directory. A developer usually need 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 (mostly because that would require too much disk space, time to recompile everything). He prefers to be able to override some of the source files in his directory, while taking advantage of all the object files generated at night.

Another example can be taken from large software systems, where it is common to have multiple implementations of a common interface; in Ada terms, multiple versions of a package body for the same spec. For example, one implementation might be safe for use in tasking programs, while another might only be used in sequential applications. This can be modeled in GNAT using the concept of *project extension*. If one project (the "child") *extends* another project (the "parent") then by default all source files of the parent project are inherited by the child, but the child project can override any of the parent's source files with new versions, and can also add new files or remove unnecessary ones. This facility is the project analog of a type extension in object-oriented programming. Project hierarchies are permitted (an extending project may itself be extended), and a project that extends a project can also import other projects.

A third example is that of using project extensions to provide 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, the other development branch still needs evolution of `Common`. Project extensions provide a flexible solution to create a new version of a subsystem while sharing and reusing as much as possible from the original one.

A project extension inherits implicitly all the sources and objects from the project it extends. It is possible to create a new version of some of the sources in one of the additional source dirs of the extending project. Those new versions hide the original versions. Adding new sources or removing existing ones is also possible. Here is an example on how to extend the project `Build` from previous examples:

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

The project after **extends** is the one 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 (see Section 1.3.1 [Project Dependencies], page 16). This project does not

specify source or object directories, so the default value for these attribute will be used that is to say the current directory (where project `Work` is placed). We can already compile that project with

```
gnatmake -Pwork
```

If no sources have been placed in the current directory, this command won't do anything, since this project does not change the sources it inherited from `Build`, therefore all the object files in `Build` and its dependencies are still valid and are reused automatically.

Suppose we now want to supply an alternate version of '`pack.adb`' but use the existing versions of '`pack.ads`' and '`proc.adb`'. We can create the new file Work's current directory (likely 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, `gnatmake` 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. A `base` project would contain the sources for '`pack.ads`' and '`proc.adb`', and `Work` would import `base` and add '`pack.adb`'. In this scenario, `base` cannot contain the original version of '`pack.adb`' otherwise there would be 2 versions of the same unit in the closure of the project and this is not allowed. Generally speaking, it is not recommended to put the spec and the 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 sometimes when a package spec has been overridden and no longer requires a body: 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 when compiling the spec.

For that purpose, the attribute **Excluded_Source_Files** is used. Its value is a list of file names. It is also possible to use attribute `Excluded_Source_List_File`. Its 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;
```

An extending project retains all the switches specified in the extended project.

1.6.1 Project Hierarchy Extension

One of the fundamental restrictions in project extension is the following: **A project is not allowed to import directly or indirectly at the same time an extending project and one of its ancestors**.

By means of example, consider the following hierarchy of projects.

```
a.gpr  contains package A1
b.gpr, imports a.gpr and contains B1, which depends on A1
c.gpr, imports b.gpr and contains C1, which depends on B1
```

If we want to locally extend the packages `A1` and `C1`, we need to create several extending projects:

```
a_ext.gpr which extends a.gpr, and overrides A1
b_ext.gpr which extends b.gpr and imports a_ext.gpr
c_ext.gpr which extends c.gpr, imports b_ext.gpr and overrides C1
```

```
project A_Ext extends "a.gpr" is
   for Source_Files use ("a1.adb", "a1.ads");
end A_Ext;

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

with "b_ext.gpr";
project C_Ext extends "c.gpr" is
   for Source_Files use ("c1.adb");
end C_Ext;
```

The extension '`b_ext.gpr`' is required, even though we are not overriding any of the sources of '`b.gpr`' because otherwise '`c_expr.gpr`' would import '`b.gpr`' which itself knows nothing about '`a_ext.gpr`'.

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

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

It is possible to explicitly extend one or more projects in the hierarchy in order to modify the sources. These extending projects must be imported by the "extending all" project, which will replace the corresponding virtual projects with the explicit ones.

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

Thus, in our example we could create the following projects instead:

```
a_ext.gpr, extends a.gpr and overrides A1
c_ext.gpr, "extends all" c.gpr, imports a_ext.gpr and overrides C1

project A_Ext extends "a.gpr" is
   for Source_Files use ("a1.adb", "a1.ads");
end A_Ext;

with "a_ext.gpr";
project C_Ext extends all "c.gpr" is
  for Source_Files use ("c1.adb");
end C_Ext;
```

When building project '`c_ext.gpr`', the entire modified project space is considered for recompilation, including the sources of '`b.gpr`' that are impacted by the changes in `A1` and `C1`.

1.7 Aggregate Projects

Aggregate projects are an extension of the project paradigm, and are meant to solve a few specific use cases that cannot be solved directly using standard projects. This section will go over a few of these use cases to try and explain what you can use aggregate projects for.

1.7.1 Building all main units from a single project tree

Most often, an application is organized into modules and submodules, which are very conveniently represented as a project tree or graph (the root project A `with`s the projects for each modules (say 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 gnatmake or gprbuild, using a syntax similar to

```
gprbuild -PA.gpr
```

this will only rebuild the main units of project A, not those of the imported projects B and C. Therefore you have to spawn several gnatmake commands, one per project, to build all executables. This is a little inconvenient, but more importantly is inefficient (since gnatmake 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 rebuild when building a project.

You could therefore define an aggregate project Agg that groups A, B and C. Then, when you build with

```
gprbuild -PAgg.gpr
```

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

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

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

If you add a main unit 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 unit will not be built when building the aggregate project.

1.7.2 Building a set of projects with a single command

One other case is when you have multiple applications and libraries that are build independently from each other (but they can be build in parallel). For instance, you have a project tree 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 trees that have common sources.

```
Aggregate projects can group projects with duplicate file names
```

This scenario is particularly useful in environment like VxWork 653 where the applications running in the multiple partitions can be build in parallel through a single gprbuild command. This also works nicely with Annex E.

```
Aggregate projects can be used to build multiple partitions
```

1.7.3 Define a build environment

The environment variables at the time you launch gprbuild or gprbuild will influence the view these tools have of the project (PATH to find the compiler,

ADA_PROJECT_PATH or GPR_PROJECT_PATH to find the projects, environment variables that are referenced in project files through the "external" statement,...). 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 build through that aggregate. One of the nice aspects is that you can put the aggregate project under configuration management, and make sure all your user have a consistent environment when building. The syntax looks like

```
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 Switches ("Ada") use ("-q");
   end Builder;
end Agg;
```

One of the often requested features in projects is to be able to reference external variables in `with` statements, as in

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

For various reasons, this isn't authorized. But using aggregate projects provide an elegant solution. For instance, you could use a project file like:

```
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;
```

1.7.4 Performance improvements in builder

The loading of aggregate projects is optimized in gprbuild and gnatmake, so that all files are searched for only once on the disk (thus reducing the number of system calls and contributing to faster compilation times especially on systems with sources on remote servers). As part of the loading, gprbuild and gnatmake compute how and where a source file should be compiled, and even if it is found 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 and gnatmake commands in parallel).

1.7.5 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 be put 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. 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 the (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_Fies
- Excluded_Source_List_File
- Interfaces

The only package that is authorized (albeit optional) is Builder. Other packages (in particular Compiler, Binder and Linker) are forbidden. It is an error to have any of these (and such an error prevents the proper loading of the aggregate project).

Three new attributes have been created, which can only be used in the context of aggregate projects:

Project_Files:

This attribute is compulsory (or else we are not aggregating any project, and thus not doing anything). It specifies a list of '.gpr' files that are grouped in the aggregate. The list may be empty. The project files can be either other aggregate projects, or standard projects. When grouping standard projects, you can have both the root of a project tree (and you do not need to specify all its imported projects), and any project within the tree.

Basically, the idea is to specify all those projects that have main units you want to build and link, or libraries you want to build. You can even specify projects that do not use the Main attribute nor 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, we expect to find the .gpr file in the same directory as the aggregate project file). 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 latter ("**") can only occur at the last position in the directory part (ie "a/**/*.gpr" is supported, but not "**/a/*.gpr"). Starting the pattern with "**" is equivalent to starting with "./**".

For now, 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 valid 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 look for project files in with statements.

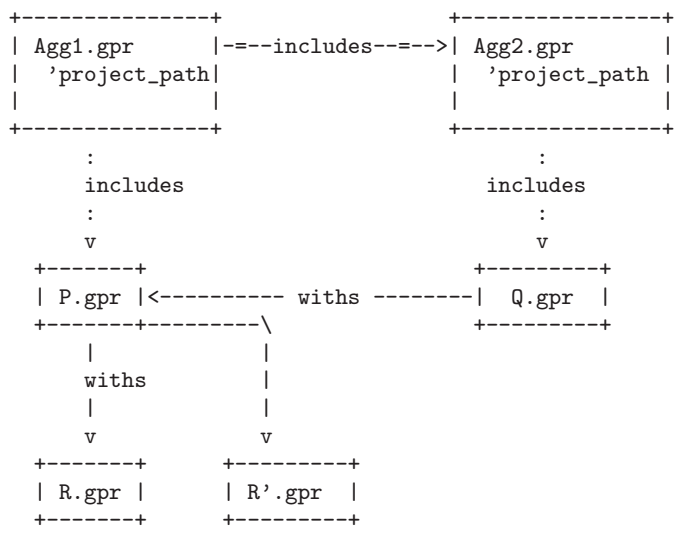
When you specify a project in Project_Files say "x/y/a.gpr"), and this projects imports a project "b.gpr", only b.gpr is searched in the project path. a.gpr must be exactly at <dir of the aggregate>/x/y/a.gpr.

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

Each aggregate project has its own (that is if agg1.gpr includes agg2.gpr, they can potentially both have a different project path). This project path is defined as the concatenation, in that order, of the current directory, followed by the command line -aP switches, then the directories from the Project_Path attribute, then the directories from the GPR_PROJECT_PATH and ADA_PROJECT_PATH env. variables, and finally the predefined directories.

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

This can potentially lead to errors. In the following example:



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) "with"ing 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.

Here are a few valid examples:

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

External:

This attribute can be used to set the value of environment variables as retrieved through the `external` statement 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 aggreate projects, and in the aggregated projects.

The exact value of external a 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 in his team) aggregate project.
- The -X command line switch to gprbuild and gnatmake

  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 or natmake), 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 build through the aggregate, which would not be the case in the diamond case: 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.

1.7.6 package Builder in aggregate projects

As we mentioned before, 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 the builder (gprbuild or gnatmake), depending on the language of the main file. For instance,

```
for Switches ("Ada") use ("-d", "-p");
for Switches ("C")   use ("-p");
```

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 have 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, that both depend on C. Here is an extra 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;
```

then 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 "-O0", since the 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 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.

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

1.8 Project File Reference

This section describes the syntactic structure of project files, the various constructs that can be used. Finally, it ends with a summary of all available attributes.

1.8.1 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 semi-colon.

Identifiers (ie 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 same as for standard Ada 95, and cannot be used for identifiers. In particular, the following words are currently used in project files, but others could be added later on. In bold are the extra reserved words in project files: `all`, `at`, `case`, `end`, `for`, `is`, `limited`, `null`, `others`, `package`, `renames`, `type`, `use`, `when`, `with`, **extends**, **external**, **project**.

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. But a project may also depend on other projects, either by importing them through **with clauses**, or by **extending** at most one other project. Both types of dependency can be used in the same project.

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 identifies uniquely 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 variable 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.

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 clause in the cycle is a **limited with**.

```
with "other_project.gpr";
project My_Project extends "extended.gpr" is
end My_Project;
```

These dependencies form a **directed graph**, potentially cyclic when using **limited with**. The subprogram reflecting the **extends** relations is a tree.

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 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.

A **project hierarchy** can be created, where projects are children of other projects. The name of such a child project must be `Parent.Child`, where `Parent` is the name of the parent project. In particular, this makes all `with` clauses of the parent project automatically visible in the child project.

```
project        ::= context_clause project_declaration

context_clause ::= {with_clause}
with_clause    ::= with path_name { , path_name } ;
path_name      ::= string_literal

project_declaration ::= simple_project_declaration | project_extension
simple_project_declaration ::=
  project <project_>name is
    {declarative_item}
  end <project_>simple_name;
```

1.8.2 Qualified Projects

Before the reserved `project`, there may be one or two **qualifiers**, that is identifiers or reserved words, to qualify the project. The current list of qualifiers is:

abstract: qualifies a project with no sources. Such a project must either have no declaration of 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 project it extends must also be a qualified abstract project.

standard: a standard project is a non library project with sources. This is the default (implicit) qualifier.

aggregate: for future extension

aggregate library: for future extension

library: a library project must declare both attributes `Library_Name` and `Library_Dir`.

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

1.8.3 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.

1.8.4 Packages

A project file may contain **packages**, that 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 see Section 1.8.9 [Attributes], page 51 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 a builder such as `gnatmake` or `gprbuild`. See Section 1.2.3 [Main Subprograms], page 8.

`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 Section 1.2.3 [Main Subprograms], page 8, and see Section 1.2.6 [Executable File Names], page 12 in particular.

`Check` This package specifies the options used when calling the checking tool `gnatcheck` via the `gnat` driver. Its attribute **Default_Switches** has 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`.

`Compiler` This package specifies the compilation options used by the compiler for each languages. See Section 1.2.4 [Tools Options in Project Files], page 9.

`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`.

`Eliminate`
: This package specifies the options used when calling the tool `gnatelim` via the `gnat` driver. 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 the options to use when invoking `gnatls` via the `gnat` driver.

`Gnatstub`
: This package specifies the options used when calling the tool `gnatstub` via the `gnat` driver. 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`. See Section 2.3 [The Development Environments], page 69.

`Linker`
: This package specifies the options used by the linker. See Section 1.2.3 [Main Subprograms], page 8.

`Makefile`
: This package is used by the GPS plugin Makefile.py. See the documentation in that plugin (from GPS: /Tools/Plug-ins).

`Metrics`
: This package specifies the options used when calling the tool `gnatmetric` via the `gnat` driver. 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 Section 1.2.8 [Naming Schemes], page 13.

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

`Stack`
: This package specifies the options used when calling the tool `gnatstack` via the `gnat` driver. 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`.

A package can also 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 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.

Finally, 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 ;
```

1.8.5 Expressions

An expression is any value that can be assigned to an attribute or a variable. It is either a litteral value, or a construct requiring runtime 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 Section 1.8.8 [Variables], page 50)
- The name of an attribute that evaluates to a string (see Section 1.8.9 [Attributes], page 51)
- An external reference (see Section 1.8.6 [External Values], page 49)
- 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
```

1.8.6 External Values

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

There are two kinds of external values, one that returns a single string, and one that returns a string list.

The syntax of a single string external value is:

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

The first string_literal is the string to be used on the command line or in the environment to specify the external value. 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.

Typically, the external value will either exist in the environment variables or be specified on the command line through the '`-X`*vbl*`=`*value*' switch. If both are specified, then the command line value is used, so that a user can more easily override the value.

The function `external` always returns a string. It is an error if the value was not found in the environment and no default was specified in the call to `external`.

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.

Most of the time, this construct is used to initialize typed variables, which are then used in **case** statements to control the value assigned to attributes in various scenarios. Thus such variables are often called **scenario variables**.

The syntax for a string list external value is:

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

The first string_literal is the string to be used on the command line or in the environment to specify the external value. The second string_literal is the separator between each component of the string list.

If the external value does not exist in the environment or on the command line, the result is an empty list. This is also the case, 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 vaue, the result is a string list with only one string. Otherwise, any string between the biginning and

the first separator, between two consecutive separators and between the last separator and the end are components of the string list.

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

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

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

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

1.8.7 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 string type may only be declared 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 ("NT", "nt", "Unix", "GNU/Linux", "other OS");
```

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 Section 1.8.10 [Case Statements], page 57).

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.

1.8.8 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 statements 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 an ancestor 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.

1.8.9 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;
```

Attributes references may be 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.

```
attribute_reference ::= attribute_prefix ' <simple_attribute>_simple_name [ (string_literal) ]
attribute_prefix ::= project
  | <project_>simple_name
  | package_identifier
  | <project_>simple_name . package_identifier
```

Examples are:

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

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/parent project name, and whose selector is a package name

Legal attribute names are listed below, including the package in which they must be declared. These names are case-insensitive. The semantics for the attributes is explained in great details in other sections.

The column *index* indicates whether the attribute is an indexed attribute, and when it is whether its index is case sensitive (sensitive) or not (insensitive), or if case sensitivity depends is the same as file names sensitivity on the system (file). The text is between brackets ([]) if the index is optional.

Attribute Name	**Value**	**Package**	**Index**
General attributes			**see Section 1.2 [Building With Projects], page 2**
Name	string	-	(Read-only, name of project)
Project_Dir	string	-	(Read-only, directory of project)
Source_Files	list	-	-
Source_Dirs	list	-	-
Source_List_File	string	-	-
Locally_Removed_Files	list	-	-
Excluded_Source_Files	list	-	-
Object_Dir	string	-	-
Exec_Dir	string	-	-
Excluded_Source_Dirs	list	-	-
Excluded_Source_Files	list	-	-
Excluded_Source_List_File	list	-	-
Inherit_Source_Path	list	-	insensitive
Languages	list	-	-
Main	list	-	-
Main_Language	string	-	-
Externally_Built	string	-	-
Roots	list	-	file
Library-related attributes			**see Section 1.5 [Library Projects], page 23**
Library_Dir	string	-	-
Library_Name	string	-	-

Library_Kind	string	-	-
Library_Version	string	-	-
Library_Interface	string	-	-
Library_Auto_Init	string	-	-
Library_Options	list	-	-
Leading_Library_Options	list	-	-
Library_Src_Dir	string	-	-
Library_ALI_Dir	string	-	-
Library_GCC	string	-	-
Library_Symbol_File	string	-	-
Library_Symbol_Policy	string	-	-
Library_Reference_Symbol_File	string	-	-
Interfaces	list	-	-
Naming			**see Section 1.2.8 [Naming Schemes], page 13**
Spec_Suffix	string	Naming	insensitive (language)
Body_Suffix	string	Naming	insensitive (language)
Separate_Suffix	string	Naming	-
Casing	string	Naming	-
Dot_Replacement	string	Naming	-
Spec	string	Naming	insensitive (Ada unit)
Body	string	Naming	insensitive (Ada unit)
Specification_Exceptions	list	Naming	insensitive (language)
Implementation_Exceptions	list	Naming	insensitive (language)
Building			**see Section 2.1.2 [Switches and Project Files], page 60**
Default_Switches	list	Builder, Compiler, Binder, Linker, Cross_Reference, Finder, Pretty_Printer, gnatstub, Check, Synchronize, Eliminate, Metrics, IDE	insensitive (language name)

Switches	list	Builder, Compiler, Binder, Linker, Cross_Reference, Finder, gnatls, Pretty_Printer, gnatstub, Check, Synchronize, Eliminate, Metrics, Stack	[file] (file name)
Local_Configuration_Pragmas	string	Compiler	-
Local_Config_File	string	insensitive	-
Global_Configuration_Pragmas	string	Builder	-
Global_Compilation_Switches	list	Builder	language
Executable	string	Builder	[file]
Executable_Suffix	string	Builder	-
Global_Config_File	string	Builder	insensitive (language)
IDE (used and created by GPS)			
Remote_Host	string	IDE	-
Program_Host	string	IDE	-
Communication_Protocol	string	IDE	-
Compiler_Command	string	IDE	insensitive (language)
Debugger_Command	string	IDE	-
Gnatlist	string	IDE	-
Gnat	string	IDE	-
VCS_Kind	string	IDE	-
VCS_File_Check	string	IDE	-
VCS_Log_Check	string	IDE	-
Documentation_Dir	string	IDE	-
Configuration files			**See gprbuild manual**
Default_Language	string	-	-
Run_Path_Option	list	-	-
Run_Path_Origin	string	-	-
Separate_Run_Path_Options	string	-	-
Toolchain_Version	string	-	insensitive
Toolchain_Description	string	-	insensitive
Object_Generated	string	-	insensitive

Objects_Linked	string	-	insensitive
Target	string	-	-
Library_Builder	string	-	-
Library_Support	string	-	-
Archive_Builder	list	-	-
Archive_Builder_Append_Option	list	-	-
Archive_Indexer	list	-	-
Archive_Suffix	string	-	-
Library_Partial_Linker	list	-	-
Shared_Library_Prefix	string	-	-
Shared_Library_Suffix	string	-	-
Symbolic_Link_Supported	string	-	-
Library_Major_Minor_Id_Supported	string	-	-
Library_Auto_Init_Supported	string	-	-
Shared_Library_Minimum_Switches	list	-	-
Library_Version_Switches	list	-	-
Library_Install_Name_Option	string	-	-
Runtime_Library_Dir	string	-	insensitive
Runtime_Source_Dir	string	-	insensitive
Driver	string	Compiler,Binder,Linker	insensitive (language)
Required_Switches	list	Compiler,Binder,Linker	insensitive (language)
Leading_Required_Switches	list	Compiler	insensitive (language)
Trailing_Required_Switches	list	Compiler	insensitive (language)
Pic_Options	list	Compiler	insensitive (language)
Path_Syntax	string	Compiler	insensitive (language)
Object_File_Suffix	string	Compiler	insensitive (language)
Object_File_Switches	list	Compiler	insensitive (language)
Multi_Unit_Switches	list	Compiler	insensitive (language)
Multi_Unit_Object_Separator	string	Compiler	insensitve (language)
Mapping_File_Switches	list	Compiler	insensitive (language)
Mapping_Spec_Suffix	string	Compiler	insensitive (language)
Mapping_body_Suffix	string	Compiler	insensitive (language)
Config_File_Switches	list	Compiler	insensitive (language)
Config_Body_File_Name	string	Compiler	insensitive (language)
Config_Body_File_Name_Index	string	Compiler	insensitive (language)
Config_Body_File_Name_Pattern	string	Compiler	insensitive (language)
Config_Spec_File_Name	string	Compiler	insensitive (language)
Config_Spec_File_Name_Index	string	Compiler	insensitive (language)
Config_Spec_File_Name_Pattern	string	Compiler	insensitive (language)
Config_File_Unique	string	Compiler	insensitive (language)
Dependency_Switches	list	Compiler	insensitive (language)
Dependency_Driver	list	Compiler	insensitive (language)
Include_Switches	list	Compiler	insensitive (language)

Include_Path	string	Compiler	insensitive (language)
Include_Path_File	string	Compiler	insensitive (language)
Prefix	string	Binder	insensitive (language)
Objects_Path	string	Binder	insensitive (language)
Objects_Path_File	string	Binder	insensitive (language)
Linker_Options	list	Linker	-
Leading_Switches	list	Linker	-
Map_File_Options	string	Linker	-
Executable_Switches	list	Linker	-
Lib_Dir_Switch	string	Linker	-
Lib_Name_Switch	string	Linker	-
Max_Command_Line_Length	string	Linker	-
Response_File_Format	string	Linker	-
Response_File_Switches	list	Linker	-

1.8.10 Case Statements

A **case** statement is used in a project file to effect conditional behavior. Through this statement, 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 statement (although the `null` statement for empty alternatives is optional).

The case expression must be a typed string variable, whose value is often given by an external reference (see Section 1.8.6 [External Values], page 49).

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. Each literal string must belong to the string type that is the type of the case variable. After each `=>`, there are zero or more statements. The only statements allowed in a case construction are other case statements, 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_statement ::=
  case <typed_variable_>name is {case_item} end case ;

case_item ::=
  when discrete_choice_list =>
    {case_statement
      | attribute_declaration
      | variable_declaration
      | empty_declaration}
```

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

Here is a typical example:

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

   package Compiler is
     case OS is
       when "GNU/Linux" | "Unix" =>
         for Switches ("Ada") use ("-gnath");
       when "NT" =>
         for Switches ("Ada") use ("-gnatP");
       when others =>
         null;
     end case;
   end Compiler;
end MyProj;
```

2 Tools Supporting Project Files

2.1 gnatmake and Project Files

This section covers several topics related to `gnatmake` and project files: defining switches for `gnatmake` and for the tools that it invokes; specifying configuration pragmas; the use of the `Main` attribute; building and rebuilding library project files.

2.1.1 Switches Related to Project Files

The following switches are used by GNAT tools that support project files:

'`-P`*project*'
: Indicates the name of a project file. This project file will be parsed with the verbosity indicated by '`-vP`*x*', if any, and using the external references indicated by '`-X`' switches, if any. There may zero, one or more spaces between '`-P`' and *project*.

  There must be only one '`-P`' switch on the command line.

  Since the Project Manager parses the project file only after all the switches on the command line are checked, the order of the switches '`-P`', '`-vP`*x*' or '`-X`' is not significant.

'`-X`*name*`=`*value*'
: Indicates that external variable *name* has the value *value*. The Project Manager will use this value for occurrences of `external(name)` when parsing the project file.

  If *name* or *value* includes a space, then *name=value* should be put between quotes.

  ```
  -XOS=NT
  -X"user=John Doe"
  ```

  Several '`-X`' switches can be used simultaneously. If several '`-X`' switches specify the same *name*, only the last one is used.

  An external variable specified with a '`-X`' switch takes precedence over the value of the same name in the environment.

'`-vP`*x*'
: Indicates the verbosity of the parsing of GNAT project files.

  '`-vP0`' means Default; '`-vP1`' means Medium; '`-vP2`' means High.

  The default is Default: no output for syntactically correct project files. If several '`-vP`*x*' switches are present, only the last one is used.

'`-aP<dir>`'
: Add directory <dir> at the beginning of the project search path, in order, after the current working directory.

'-eL' Follow all symbolic links when processing project files.

'--subdirs=<subdir>'
This switch is recognized by gnatmake and gnatclean. It indicate that the real directories (except the source directories) are the subdirectories <subdir> of the directories specified in the project files. This applies in particular to object directories, library directories and exec directories. If the subdirectories do not exist, they are created automatically.

2.1.2 Switches and Project Files

For each of the packages Builder, Compiler, Binder, and Linker, you can specify a Default_Switches attribute, a Switches attribute, or both; as their names imply, these switch-related attributes affect the switches that are used for each of these GNAT components when gnatmake is invoked. As will be explained below, these component-specific switches precede the switches provided on the gnatmake command line.

The Default_Switches attribute is an attribute indexed by language name (case insensitive) whose value is a string list. For example:

```
package Compiler is
  for Default_Switches ("Ada")
      use ("-gnaty",
           "-v");
end Compiler;
```

The Switches attribute is indexed on a file name (which may or may not be case sensitive, depending on the operating system) whose value is a string list. For example:

```
package Builder is
   for Switches ("main1.adb")
       use ("-O2");
   for Switches ("main2.adb")
       use ("-g");
end Builder;
```

For the Builder package, the file names must designate source files for main subprograms. For the Binder and Linker packages, the file names must designate 'ALI' or source files for main subprograms. In each case just the file name without an explicit extension is acceptable.

For each tool used in a program build (gnatmake, the compiler, the binder, and the linker), the corresponding package *contributes* a set of switches for each file on which the tool is invoked, based on the switch-related attributes defined in the package. In particular, the switches that each of these packages contributes for a given file *f* comprise:

- the value of attribute `Switches (f)`, if it is specified in the package for the given file,
- otherwise, the value of `Default_Switches ("Ada")`, if it is specified in the package.

If neither of these attributes is defined in the package, then the package does not contribute any switches for the given file.

When `gnatmake` is invoked on a file, the switches comprise two sets, in the following order: those contributed for the file by the `Builder` package; and the switches passed on the command line.

When `gnatmake` invokes a tool (compiler, binder, linker) on a file, the switches passed to the tool comprise three sets, in the following order:

1. the applicable switches contributed for the file by the `Builder` package in the project file supplied on the command line;
2. those contributed for the file by the package (in the relevant project file – see below) corresponding to the tool; and
3. the applicable switches passed on the command line.

The term *applicable switches* reflects the fact that `gnatmake` switches may or may not be passed to individual tools, depending on the individual switch.

`gnatmake` may invoke the compiler on source files from different projects. The Project Manager will use the appropriate project file to determine the `Compiler` package for each source file being compiled. Likewise for the `Binder` and `Linker` packages.

As an example, consider the following package in a project file:

```
project Proj1 is
   package Compiler is
      for Default_Switches ("Ada")
          use ("-g");
      for Switches ("a.adb")
          use ("-O1");
      for Switches ("b.adb")
          use ("-O2",
               "-gnaty");
   end Compiler;
end Proj1;
```

If `gnatmake` is invoked with this project file, and it needs to compile, say, the files '`a.adb`', '`b.adb`', and '`c.adb`', then '`a.adb`' will be compiled with the switch '`-O1`', '`b.adb`' with switches '`-O2`' and '`-gnaty`', and '`c.adb`' with '`-g`'.

The following example illustrates the ordering of the switches contributed by different packages:

```
project Proj2 is
   package Builder is
      for Switches ("main.adb")
          use ("-g",
               "-O1",
               "-f");
   end Builder;

   package Compiler is
      for Switches ("main.adb")
          use ("-O2");
   end Compiler;
end Proj2;
```

If you issue the command:

```
gnatmake -Pproj2 -O0 main
```

then the compiler will be invoked on '`main.adb`' with the following sequence of switches

```
-g -O1 -O2 -O0
```

with the last '`-O`' switch having precedence over the earlier ones; several other switches (such as '`-c`') are added implicitly.

The switches '`-g`' and '`-O1`' are contributed by package `Builder`, '`-O2`' is contributed by the package `Compiler` and '`-O0`' comes from the command line.

The '`-g`' switch will also be passed in the invocation of `Gnatlink`.

A final example illustrates switch contributions from packages in different project files:

```
project Proj3 is
   for Source_Files use ("pack.ads", "pack.adb");
   package Compiler is
      for Default_Switches ("Ada")
          use ("-gnata");
   end Compiler;
end Proj3;

with "Proj3";
project Proj4 is
   for Source_Files use ("foo_main.adb", "bar_main.adb");
   package Builder is
      for Switches ("foo_main.adb")
          use ("-s",
               "-g");
   end Builder;
end Proj4;
```

```
-- Ada source file:
with Pack;
procedure Foo_Main is
   ...
end Foo_Main;
```

If the command is

```
gnatmake -PProj4 foo_main.adb -cargs -gnato
```

then the switches passed to the compiler for '`foo_main.adb`' are '`-g`' (contributed by the package `Proj4.Builder`) and '`-gnato`' (passed on the command line). When the imported package `Pack` is compiled, the switches used are '`-g`' from `Proj4.Builder`, '`-gnata`' (contributed from package `Proj3.Compiler`, and '`-gnato`' from the command line.

When using `gnatmake` with project files, some switches or arguments may be expressed as relative paths. As the working directory where compilation occurs may change, these relative paths are converted to absolute paths. For the switches found in a project file, the relative paths are relative to the project file directory, for the switches on the command line, they are relative to the directory where `gnatmake` is invoked. The switches for which this occurs are: -I, -A, -L, -aO, -aL, -aI, as well as all arguments that are not switches (arguments to switch -o, object files specified in package `Linker` or after -largs on the command line). The exception to this rule is the switch –RTS= for which a relative path argument is never converted.

2.1.3 Specifying Configuration Pragmas

When using `gnatmake` with project files, if there exists a file '`gnat.adc`' that contains configuration pragmas, this file will be ignored.

Configuration pragmas can be defined by means of the following attributes in project files: `Global_Configuration_Pragmas` in package `Builder` and `Local_Configuration_Pragmas` in package `Compiler`.

Both these attributes are single string attributes. Their values is the path name of a file containing configuration pragmas. If a path name is relative, then it is relative to the project directory of the project file where the attribute is defined.

When compiling a source, the configuration pragmas used are, in order, those listed in the file designated by attribute `Global_Configuration_Pragmas` in package `Builder` of the main project file, if it is specified, and those listed in the file designated by attribute `Local_Configuration_Pragmas` in package `Compiler` of the project file of the source, if it exists.

2.1.4 Project Files and Main Subprograms

When using a project file, you can invoke `gnatmake` with one or several main subprograms, by specifying their source files on the command line.

```
gnatmake -Pprj main1 main2 main3
```

Each of these needs to be a source file of the same project, except when the switch -u is used.

When -u is not used, all the mains need to be sources of the same project, one of the project in the tree rooted at the project specified on the command line. The package `Builder` of this common project, the "main project" is the one that is considered by `gnatmake`.

When -u is used, the specified source files may be in projects imported directly or indirectly by the project specified on the command line. Note that if such a source file is not part of the project specified on the command line, the switches found in package `Builder` of the project specified on the command line, if any, that are transmitted to the compiler will still be used, not those found in the project file of the source file.

When using a project file, you can also invoke `gnatmake` without explicitly specifying any main, and the effect depends on whether you have defined the `Main` attribute. This attribute has a string list value, where each element in the list is the name of a source file (the file extension is optional) that contains a unit that can be a main subprogram.

If the `Main` attribute is defined in a project file as a non-empty string list and the switch '`-u`' is not used on the command line, then invoking `gnatmake` with this project file but without any main on the command line is equivalent to invoking `gnatmake` with all the file names in the `Main` attribute on the command line.

Example:

```
project Prj is
   for Main use ("main1", "main2", "main3");
end Prj;
```

With this project file, "`gnatmake -Pprj`" is equivalent to "`gnatmake -Pprj main1 main2 main3`".

When the project attribute `Main` is not specified, or is specified as an empty string list, or when the switch '`-u`' is used on the command line, then invoking `gnatmake` with no main on the command line will result in all immediate sources of the project file being checked, and potentially recompiled. Depending on the presence of the switch '`-u`', sources from other project files on which the immediate sources of the main project file depend are also checked and potentially recompiled. In other words, the '`-u`' switch is applied to all of the immediate sources of the main project file.

When no main is specified on the command line and attribute `Main` exists and includes several mains, or when several mains are specified on the command line, the default switches in package `Builder` will be used for all mains, even if there are specific switches specified for one or several mains.

But the switches from package `Binder` or `Linker` will be the specific switches for each main, if they are specified.

2.1.5 Library Project Files

When `gnatmake` is invoked with a main project file that is a library project file, it is not allowed to specify one or more mains on the command line.

When a library project file is specified, switches -b and -l have special meanings.

- -b is only allowed for stand-alone libraries. It indicates to `gnatmake` that `gnatbind` should be invoked for the library.
- -l may be used for all library projects. It indicates to `gnatmake` that the binder generated file should be compiled (in the case of a stand-alone library) and that the library should be built.

2.2 The GNAT Driver and Project Files

A number of GNAT tools, other than `gnatmake` can benefit from project files: (`gnatbind`, `gnatcheck`, `gnatclean`, `gnatelim`, `gnatfind`, `gnatlink`, `gnatls`, `gnatmetric`, `gnatpp`, `gnatstub`, and `gnatxref`). However, none of these tools can be invoked directly with a project file switch ('`-P`'). They must be invoked through the `gnat` driver.

The `gnat` driver is a wrapper that accepts a number of commands and calls the corresponding tool. It was designed initially for VMS platforms (to convert VMS qualifiers to Unix-style switches), but it is now available on all GNAT platforms.

On non-VMS platforms, the `gnat` driver accepts the following commands (case insensitive):

- BIND to invoke `gnatbind`
- CHOP to invoke `gnatchop`
- CLEAN to invoke `gnatclean`
- COMP or COMPILE to invoke the compiler
- ELIM to invoke `gnatelim`
- FIND to invoke `gnatfind`
- KR or KRUNCH to invoke `gnatkr`
- LINK to invoke `gnatlink`
- LS or LIST to invoke `gnatls`
- MAKE to invoke `gnatmake`
- NAME to invoke `gnatname`
- PREP or PREPROCESS to invoke `gnatprep`

- PP or PRETTY to invoke `gnatpp`
- METRIC to invoke `gnatmetric`
- STUB to invoke `gnatstub`
- XREF to invoke `gnatxref`

(note that the compiler is invoked using the command `gnatmake -f -u -c`).

On non-VMS platforms, between `gnat` and the command, two special switches may be used:

- `-v` to display the invocation of the tool.
- `-dn` to prevent the `gnat` driver from removing the temporary files it has created. These temporary files are configuration files and temporary file list files.

The command may be followed by switches and arguments for the invoked tool.

```
gnat bind -C main.ali
gnat ls -a main
gnat chop foo.txt
```

Switches may also be put in text files, one switch per line, and the text files may be specified with their path name preceded by '`@`'.

```
gnat bind @args.txt main.ali
```

In addition, for commands BIND, COMP or COMPILE, FIND, ELIM, LS or LIST, LINK, METRIC, PP or PRETTY, STUB and XREF, the project file related switches ('`-P`', '`-X`' and '`-vPx`') may be used in addition to the switches of the invoking tool.

When GNAT PP or GNAT PRETTY is used with a project file, but with no source specified on the command line, it invokes `gnatpp` with all the immediate sources of the specified project file.

When GNAT METRIC is used with a project file, but with no source specified on the command line, it invokes `gnatmetric` with all the immediate sources of the specified project file and with '`-d`' with the parameter pointing to the object directory of the project.

In addition, when GNAT PP, GNAT PRETTY or GNAT METRIC is used with a project file, no source is specified on the command line and switch -U is specified on the command line, then the underlying tool (gnatpp or gnatmetric) is invoked for all sources of all projects, not only for the immediate sources of the main project. (-U stands for Universal or Union of the project files of the project tree)

For each of the following commands, there is optionally a corresponding package in the main project.

- package `Binder` for command BIND (invoking `gnatbind`)
- package `Check` for command CHECK (invoking `gnatcheck`)

- package `Compiler` for command COMP or COMPILE (invoking the compiler)
- package `Cross_Reference` for command XREF (invoking `gnatxref`)
- package `Eliminate` for command ELIM (invoking `gnatelim`)
- package `Finder` for command FIND (invoking `gnatfind`)
- package `Gnatls` for command LS or LIST (invoking `gnatls`)
- package `Gnatstub` for command STUB (invoking `gnatstub`)
- package `Linker` for command LINK (invoking `gnatlink`)
- package `Check` for command CHECK (invoking `gnatcheck`)
- package `Metrics` for command METRIC (invoking `gnatmetric`)
- package `Pretty_Printer` for command PP or PRETTY (invoking `gnatpp`)

Package `Gnatls` has a unique attribute `Switches`, a simple variable with a string list value. It contains switches for the invocation of `gnatls`.

```
project Proj1 is
   package gnatls is
      for Switches
          use ("-a",
               "-v");
   end gnatls;
end Proj1;
```

All other packages have two attribute `Switches` and `Default_Switches`.

`Switches` is an indexed attribute, indexed by the source file name, that has a string list value: the switches to be used when the tool corresponding to the package is invoked for the specific source file.

`Default_Switches` is an attribute, indexed by the programming language that has a string list value. `Default_Switches ("Ada")` contains the switches for the invocation of the tool corresponding to the package, except if a specific `Switches` attribute is specified for the source file.

```
project Proj is

   for Source_Dirs use ("**");

   package gnatls is
      for Switches use
          ("-a",
           "-v");
   end gnatls;

   package Compiler is
      for Default_Switches ("Ada")
          use ("-gnatv",
               "-gnatwa");
   end Binder;
```

```
   package Binder is
      for Default_Switches ("Ada")
          use ("-C",
               "-e");
   end Binder;

   package Linker is
      for Default_Switches ("Ada")
          use ("-C");
      for Switches ("main.adb")
          use ("-C",
               "-v",
               "-v");
   end Linker;

   package Finder is
      for Default_Switches ("Ada")
           use ("-a",
                "-f");
   end Finder;

   package Cross_Reference is
      for Default_Switches ("Ada")
          use ("-a",
               "-f",
               "-d",
               "-u");
   end Cross_Reference;
end Proj;
```

With the above project file, commands such as

```
gnat comp -Pproj main
gnat ls -Pproj main
gnat xref -Pproj main
gnat bind -Pproj main.ali
gnat link -Pproj main.ali
```

will set up the environment properly and invoke the tool with the switches found in the package corresponding to the tool: `Default_Switches ("Ada")` for all tools, except `Switches ("main.adb")` for `gnatlink`. It is also possible to invoke some of the tools, (`gnatcheck`, `gnatmetric`, and `gnatpp`) on a set of project units thanks to the combination of the switches '`-P`', '`-U`' and possibly the main unit when one is interested in its closure. For instance,

```
gnat metric -Pproj
```

will compute the metrics for all the immediate units of project `proj`.

```
gnat metric -Pproj -U
```

will compute the metrics for all the units of the closure of projects rooted at `proj`.

```
gnat metric -Pproj -U main_unit
```

will compute the metrics for the closure of units rooted at `main_unit`. This last possibility relies implicitly on `gnatbind`'s option '`-R`'. But if the argument files for the tool invoked by the the `gnat` driver are explicitly specified either directly or through the tool '`-files`' option, then the tool is called only for these explicitly specified files.

2.3 The Development Environments

See the appropriate manuals for more details. These environments will store a number of settings in the project itself, when they are meant to be shared by the whole team working on the project. Here are the attributes defined in the package **IDE** in projects.

`Remote_Host`
: This is a simple attribute. Its value is a string that designates the remote host in a cross-compilation environment, to be used for remote compilation and debugging. This field should not be specified when running on the local machine.

`Program_Host`
: This is a simple attribute. Its value is a string that specifies the name of IP address of the embedded target in a cross-compilation environment, on which the program should execute.

`Communication_Protocol`
: This is a simple string attribute. Its value is the name of the protocol to use to communicate with the target in a cross-compilation environment, e.g. `"wtx"` or `"vxworks"`.

`Compiler_Command`
: This is an associative array attribute, whose domain is a language name. Its value is string that denotes the command to be used to invoke the compiler. The value of `Compiler_Command ("Ada")` is expected to be compatible with gnatmake, in particular in the handling of switches.

`Debugger_Command`
: This is simple attribute, Its value is a string that specifies the name of the debugger to be used, such as gdb, powerpc-wrs-vxworks-gdb or gdb-4.

`Default_Switches`
: This is an associative array attribute. Its indexes are the name of the external tools that the GNAT Programming System (GPS) is supporting. Its value is a list of switches to use when invoking that tool.

`Gnatlist` This is a simple attribute. Its value is a string that specifies the name of the `gnatls` utility to be used to retrieve information about the predefined path; e.g., `"gnatls"`, `"powerpc-wrs-vxworks-gnatls"`.

`VCS_Kind` This is a simple attribute. Its value is a string used to specify the Version Control System (VCS) to be used for this project, e.g. CVS, RCS ClearCase or Perforce.

`Gnat` This is a simple attribute. Its value is a string that specifies the name of the `gnat` utility to be used when executing various tools from GPS, in particular `"gnat pp"`, `"gnat stub"`,...

`VCS_File_Check`
This is a simple attribute. Its value is a string that specifies the command used by the VCS to check the validity of a file, either when the user explicitly asks for a check, or as a sanity check before doing the check-in.

`VCS_Log_Check`
This is a simple attribute. Its value is a string that specifies the command used by the VCS to check the validity of a log file.

`VCS_Repository_Root`
The VCS repository root path. This is used to create tags or branches of the repository. For subversion the value should be the `URL` as specified to check-out the working copy of the repository.

`VCS_Patch_Root`
The local root directory to use for building patch file. All patch chunks will be relative to this path. The root project directory is used if this value is not defined.

2.4 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
```

2.4.1 Switches for GPRclean

The switches for GPRclean 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.
- '`-c`' : Only delete compiler-generated files. Do not delete executables and libraries.
- '`-f`' : Force deletions of unwritable files
- '`-F`' : Display full project path name in brief error messages
- '`-h`' : Display this message
- '`-n`' : Do not delete files, only list files to delete
- '`-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
- '`-vPx`' : Specify verbosity when parsing Project Files. x = 0 (default), 1 or 2.
- '`-Xnm=val`' : Specify an external reference for Project Files.

3 Gprbuild

`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.

3.1 Building with GPRbuild

3.1.1 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. Finally, it is possible to specify Ada unit names (and gprbuild automatically looks up the corresponding file name in the project).

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 '`default.cgpr`', or '`<targetname>.cgpr`' if option '`--target=<targetname>`' is used. 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.
- '`-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`.

3.1.2 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:

- '`--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.

- '`--no-object-check`'

  Do not check if an object has been created after compilation.

- '`--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.

- '`--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.

- '`-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.

- '`-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.

- '`-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`').

- '`-Inn`' (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.
- '`-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.
- '`-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.

- '`-vPx`' (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"`.

Then, the switches that may be specified on the command line as well as in package Builder of the main project (attribute Switches):

- '`--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-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.

- '-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.

- '-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.

- '-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)

  Display full paths, all options used in spawned processes, and reasons why these processes are spawned. Cancel any previous switch '`-q`'.
- '`-vl`' (Verbose output, low level)

  Verbose output. Some verbose messages are not displayed.
- '`-vm`' (Verbose output, medium level)

  Verbose output. Some verbose messages may not be displayed.
- '`-vh`' (Verbose output, high level)

  Equivalent to '`-v`'.
- '`-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.

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.

3.1.3 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 configuration project file named '`auto.cgpr`' that is located in the object directory of the main project file.

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.

3.1.4 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 Section 3.1.5 [Compilation Phase], page 83.

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.

3.1.5 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.

3.1.6 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.

3.1.7 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.

3.1.8 Incompatibilities with gnatmake

Here is a list of incompatibilities between gnatmake invoked with a project file and gprbuild:

- gprbuild never recompiles the runtime sources.
- gnatmake switches that are not recognized by gprbuild:
  - -a (Consider all files, even readonly ali files)
  - -M (List object file dependences for Makefile)
  - -n (Check objects up to date, output next file to compile if not)
  - -x (Allow compilation of needed units external to the projects)
  - -z No main subprogram (zero main)
  - –GCC=command
  - –GNATBIND=command
  - –GNATLINK=command
  - -aLdir (Skip missing library sources if ali in dir)
  - -Adir (like -aLdir -aIdir)
  - -aOdir (Specify library/object files search path)
  - -aIdir (Specify source files search path)
  - -Idir (Like -aIdir -aOdir)
  - -I- (Don't look for sources & library files in the default directory)
  - -Ldir (Look for program libraries also in dir)
- The switches that are not directly recognized by gprbuild and passed to the Ada compiler are only:
  - -nostdlib
  - -nostdinc
  - -fstack-check
  - -fno-inline
  - -Oxxx (any switch starting with -O)
  - -gxxx (any switch starting with -g)

3.2 Configuring with GPRconfig

3.2.1 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 Section 3.2 [Configuring with GPRconfig], page 86), 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, `<gprbuild_install_root>/share/gpr/default.cgpr`)

  ```
  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
```

3.2.2 Using GPRconfig

3.2.3 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.

3.2.4 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. 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,. . .

3.2.5 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 Ⓢ, and GPRconfig will generate the configuration file.

3.2.6 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.

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.

3.2.6.1 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.

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.

3.2.6.2 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 (see Section 3.2.6.5 [Configurations], page 99), 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 Section 3.2.6.3 [GPRconfig external values], page 94 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 Section 3.2.6.3 [GPRconfig external values], page 94 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 Section 3.2.6.3 [GPRconfig external values], page 94 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 Section 3.2.6.3 [GPRconfig external values], page 94 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 Section 3.2.6.3 [GPRconfig external values], page 94 below for valid children. The returned value is split into words as for `<languages>`.

3.2.6.3 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 reg-

ular 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">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 (ie '/'), 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.

  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 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.

3.2.6.4 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 (see Section 3.2.1 [Configuration], page 86).

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"> <tar-
  get 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 lowercase

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.

3.2.6.5 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 (see Section 3.2.6.4 [GPRconfig variable substitution], page 97) 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 name and language attributes, when specified, match the corresponding attributes 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 a single mandatory attribute *`name`*, which is a regexp matched against the architecture on which GPRconfig is running. 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 active 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>
```

3.3 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 associative array indexed by the 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.

3.3.1 Project Level Attributes

3.3.1.1 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 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,");
  ```

- 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";
```

3.3.1.2 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";
```

3.3.1.3 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");
```

3.3.1.4 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.

3.3.2 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 "-";
```

3.3.3 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";
```

3.3.4 Package Compiler

3.3.4.1 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");
```

3.3.4.2 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";
```

3.3.4.3 Config File Related Attributes

In the value of config file attributes defined below, there are some placeholders that GPRbuild will replace. These placeholders are:

- %u : the unit name
- %f : the file name of the source
- %s : the spec suffix
- %b : the body suffix
- %c : the casing
- %d : the 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";
```

3.3.4.4 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.

3.3.4.5 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";
  ```

3.3.5 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";
  ```

3.3.6 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,");
```

Appendix A GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written 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 that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

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. (For example, 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.

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 "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and 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 has been designed to thwart or discourage subsequent modification by readers is not Transparent. 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 designed for human modification. Opaque formats include PostScript, PDF, 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 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.

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 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 publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. 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:

A. 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.

B. 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 less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. 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.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and 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.

J. 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.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. 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.

M. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

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.

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."

Heading 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, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they 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 quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around 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 provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify,

sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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.

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 (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

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

M

N

O

P

S

T

Table of Contents