GNATcheck Reference Manual

Release 27.0w

AdaCore

Apr 28, 2026

This page is intentionally left blank.

CONTENTS

This page is intentionally left blank.

GNATcheck, The GNAT coding standard checker

GNATcheck
Version 27.0w

This page is intentionally left blank.

CHAPTER

ONE

GETTING STARTED

The `gnatcheck` tool is a utility that checks properties of Ada source files according to a given set of syntactic and semantic rules.

It can be used to enforce coding standards by analyzing Ada source programs with respect to a set of *rules* supplied at tool invocation.

It can also be used as a static analysis tool to detect potential errors (problematic or dangerous code patterns) or find areas of code improvement.

A number of rules are predefined in `gnatcheck` and are described in *Predefined Rules*. In addition, it is possible to write new rules as described in *Writing Your Own Rules* using a dedicated pattern matching language called *LKQL*, used to implement all the predefined rules.

Invoking `gnatcheck` on the command line has the form:

```
$ gnatcheck [switches] {filename}
      [-files=arg_list_filename]
      [-r rule_names]
      [--rule-file=lkql_rule_filename]
      [-cargs gcc_switches]
```

or the deprecated form:

```
$ gnatcheck [switches] {filename}
      [-files=arg_list_filename]
      -rules rule_options
      [-cargs gcc_switches]
```

where

- *switches* specify the *General GNATcheck Switches* such as `-P project`
- Each *filename* is the name (including the extension) of a source file to process, the file name may contain path information.
- *arg_list_filename* is the name (including the extension) of a text file containing the names of the source files to process, separated by spaces or line breaks.
- *rules_names* is a list of rule to enable for the GNATcheck run.
- *lkql_rule_filename* is the name (including the extension) of an LKQL rule configuration file used to control and configure the enabled rules for the GNATcheck run (see *LKQL Rule Files* for more information about it).
- *rule_options* is a list of options for controlling a set of rules to be checked by `gnatcheck` (*GNATcheck Rule Options*).

- *gcc_switches* is a list of switches for `gcc`. They will be passed on to a compiler invocation made by `gnatcheck` to collect compiler warnings and to add them to the report file and to check the legality of argument sources if `--check-semantic` option is specified. Here you can provide e.g. `-gnatxx` switches such as `-gnat2012`, etc.

Either a `filename` or an `arg_list_filename` must be supplied.

1.1 Example of GNATcheck Usage

Here is a complete example. Suppose that in the current directory we have a project file named `gnatcheck_example.gpr` with the following content:

```
project Gnatcheck_Example is

   for Source_Dirs use ("src");
   for Object_Dir use "obj";
   for Main use ("main.adb");

end Gnatcheck_Example;
```

And the file named `coding_standard.lkql` is also located in the current directory and has the following content

```
# This is a sample gnatcheck coding standard file

# The 'rules' value must be an object value containing rule configs
val rules = @{
   # Turning on rules directly implemented in GNATcheck
   Abstract_Type_Declarations,
   Anonymous_Arrays,
   Local_Packages,
   Float_Equality_Checks,
   EXIT_Statements_With_No_Loop_Name,

   # Turning on a compiler check
   Style_Checks: [{arg: "e"}]
}
```

And the subdirectory `src` contains the following Ada sources:

`pack.ads`:

```
package Pack is
   type T is abstract tagged private;
   procedure P (X : T) is abstract;

   package Inner is
      type My_Float is digits 8;
      function Is_Equal (L, R : My_Float) return Boolean;
   end Inner;
private
   type T is abstract tagged null record;
end;
```

`pack.adb`:

```
package body Pack is
   package body Inner is
      function Is_Equal (L, R : My_Float) return Boolean is
      begin
         return L = R;
      end;
   end Inner;
end Pack;
```

and `main.adb`:

```
with Pack; use Pack;
procedure Main is

   pragma Annotate
     (gnatcheck, Exempt_On, "Anonymous_Arrays", "this one is fine");
   Float_Array : array (1 .. 10) of Inner.My_Float;
   pragma Annotate (gnatcheck, Exempt_Off, "Anonymous_Arrays");

   Another_Float_Array : array (1 .. 10) of Inner.My_Float;

   use Inner;

   B : Boolean := False;

begin
   for J in Float_Array'Range loop
      if Is_Equal (Float_Array (J), Another_Float_Array (J)) then
         B := True;
         exit;
      end if;
   end loop;
end Main;
```

And suppose we call `gnatcheck` from the current directory using the project file as the only parameter of the call:

```
gnatcheck -Pgnatcheck_example.gpr --rule-file coding_standard.lkql
```

As a result, `gnatcheck` is called to check all the files from the project `gnatcheck_example.gpr` using the coding standard defined by the file `coding_standard.lkql`. The `gnatcheck` report file named `gnatcheck.out` will be created in the `obj` directory, and it will have the following content:

```
GNATCheck report

date              : YYYY-MM-DD HH:MM
gnatcheck version : gnatcheck XX.Y
command line      : gnatcheck -Pgnatcheck_example.gpr
runtime           : <default>
coding standard   : coding_standard
list of sources   : gnatcheck-source-list.out

1. Summary
```

(continues on next page)

(continued from previous page)

```
   fully compliant sources               : 0
   sources with exempted violations only : 0
   sources with non-exempted violations  : 3
   unverified sources                    : 0
   total sources                         : 3
   ignored sources                       : 0

   non-exempted violations               : 8
   rule exemption warnings               : 0
   compilation errors                    : 0
   exempted violations                   : 1
   internal errors                       : 0

2. Exempted Coding Standard Violations

main.adb:6:18: anonymous array type
   (this one is fine)

3. Non-exempted Coding Standard Violations

main.adb:9:26: anonymous array type
main.adb:19:10: exit statement with no loop name
pack.adb:5:17: use of equality operation for float values
pack.adb:6:07: "end Is_Equal" required
pack.ads:2:14: declaration of abstract type
pack.ads:5:12: declaration of local package
pack.ads:10:14: declaration of abstract type
pack.ads:11:01: "end Pack" required

4. Rule exemption problems

   no rule exemption problems detected

5. Language violations

   no language violations detected

6. Gnatcheck internal errors

   no internal error detected
```

CHAPTER

TWO

USING GNATCHECK

2.1 General GNATcheck Switches

The following switches control the general `gnatcheck` behavior

`-V, --version`
Display Copyright and version, then exit disregarding all other options.

`-h, --help`
Display usage, then exit disregarding all other options.

`-P file`
Indicates the name of the project file that describes the set of sources to be processed. The exact set of argument sources depends on other options specified, see below.

`-U`
If a project file is specified and no argument source is provided, process all units of the closure of the argument project. If explicit argument sources are specified directly through the command-line alongside the `-U` flag, they will be considered as closure roots for source fetching and not as explicit source files. If explicit argument sources are passed through *-files*, this option has no effect.

`--no-subprojects`
If a project file is specified and no argument source is explicitly specified (either directly or by means of `-files` option), process all the units of the root argument project. Otherwise this option has no effect.

`-Xname=value`
Indicates that external variable *name* in the argument project has the *value* value. Has no effect if no project is specified as tool argument.

`--subdirs=dir`
Use the specified subdirectory of the project objects file (or of the project file directory if the project does not specify an object directory) for tool output files. Has no effect if no project is specified as tool argument.

Note: This option can be used to safely run concurrent parallel jobs of GNATcheck or GNATkp.

`--no_objects_dir`
Put gnatcheck output files in the current directory instead of using the project file's object directory.

`-eL`
Follow all symbolic links when processing project files. By default, symbolic links are not resolved and kept as is. In some cases, resolving the target of symbolic links is needed for proper loading of project files.

`-vP level`
Set the verbosity level when parsing a project file. The level is expressed as a natural number between `0` and `2`; `0` being the less verbose and `2` being very verbose (default is `0`).

`--ignore-project-switches`
Ignore gnatcheck switches specified in the package `Check` of the main project file.

`--target=targetname`
Specify a target for cross platforms, this is needed to locate the proper runtime library.

`--RTS=rts-path`
Specifies the default location of the runtime library.

`--list-rules`
List all available GNATcheck rules and exit.

`-j=n`
Use *n* processes to analyze the source files. On a multi-core machine, this speeds up processing by analyzing subset of files separately under multiple processes running in parallel. If `n` is 0, then the maximum number processes is the number of core processors detected on the platform.

> **Attention:** Please read the *Performance and Memory Usage* section before using this flag.

`-l`
Use full source locations references in the report file.

`-log`
Duplicate all the output sent to `stderr` into a log file. The log file is named `gnatcheck.log`. If a project file is specified as `gnatcheck` parameter then it is located in the project objects directory (or in the project file directory if no object directory is specified). Otherwise it is located in the current directory.

`-m=n`
Maximum number of diagnostics to be sent to `stdout`, where *n* is in the range 0...1000; the default value is 0, which means that there is no limitation on the number of diagnostic messages to be output.

`-q`
Quiet mode. All the diagnostics about rule violations are placed in the `gnatcheck` report file only, without duplication on `stdout`.

`-s`
Short format of the report file (no version information, no list of applied rules, no list of checked sources is included)

`-xml`
Generate the report file in XML format.

`-nt`
Do not generate the report file in text format. Enforces `-xml`.

`-files=filename`
Take the argument source files from the specified file. This file should be an ordinary text file containing file names separated by spaces or line breaks. This switch can be specified only once, but can be combined with an explicit list of files. If you want to specify a source file with spaces, you need to surround it with double quotes ("). If a line in the file starts with `--` then the whole line is ignored (considered as a comment).

`--ignore=filename`
Do not process the sources listed in a specified file, using the same syntax as for the `-files` switch.

--rule-file=filename
Load the given file as an LKQL rule options file (see *LKQL Rule Files* for more information). If not absolute, the provided path is relative to the current working directory.

-r, --rule [rule_name]
Enable the given `rule_name` for the current GNATcheck run; you can pass this option multiple times to enable more that one rule. Note that you can enable all rules by passing "all" as `rule_name`. You cannot provide parameters to a rule through this command-line option, to do so, please use an *LKQL Rule Files*.

--show-rule
Add the corresponding rule name to the diagnosis generated for its violation. If the rule has a user-defined synonym, both gnatcheck and user-defined rule names are used as rule annotation: `[user_synonym|gnatcheck_rule_name]`.

--show-instantiation-chain
For reported generic instantiation constructs, display a chain of source location going from the generic unit to the instantiation.

--brief
Brief mode, report detections to Stderr. This switch also implies `-q` in terms of verbosity, and `-s`.

--check-redefinition
For a parametrized rule check if a rule parameter is defined more than once in the set of rule options specified and issue a warning if parameter redefinition is detected

--check-semantic
Check semantic validity of the source files by running gprbuild with the `-gnatc` switch, and report any legality error as part of the GNATcheck messages. By default, GNATcheck does not check that sources are semantically valid and will perform a best effort when encountering invalid source files. If you want to ensure and detect that your source files are valid as part of running GNATcheck, you should use this switch.

--charset=charset
Specify the charset of the source files. By default, `ISO-8859-1` is used if no charset is specified.

--lkql-path=dir
Specify directory to add to the `LKQL_PATH` environment variable when GNATcheck is spawning the LKQL engine. You can specify this option multiple times to add multiple directories.

--rules-dir=dir
Specify an alternate directory containing rule files. You can specify this switch multiple times. Each of the directories specified will be scanned and all files with the extension `.lkql` will be loaded by `GNATcheck` to provide additional rules.

--include-file=file
Append the content of the specified text file to the report file

--emit-lkql-rule-file
Emit a file named `rules.lkql` containing the rule configuration of the current GNATcheck run. This file is emitted besides the given project file if there is one, otherwise, it is generated in the current directory. Be careful, if a `rules.lkql` file already exists, there will be an error.

-t
Print out execution time.

-v
Verbose mode; `gnatcheck` generates version information and then a trace of sources being processed.

-W, --warnings-as-errors
Treat warnings raised by GNATcheck as errors, ensuring an erroneous return code.

-o report_file
Set name of the text report file to *report_file*.

-ox report_file
Set name of the XML report file to *report_file*. Enforces `-xml`.

-rules rules_options
Provide rule options for the current GNATcheck run through the command-line. All switches and options provided after this flag will be parse as *rule options*.

> **Attention:** This CLI section is **deprecated**, consider converting your rule configuration to the new *LKQL rule file* format using the `--emit-lkql-rule-file` switch.

If a project file is specified and no argument source is explicitly specified (either directly or by means of `-files` option), and no `-U` or `--no-subprojects` is specified, then the set of processed sources is determined in the following way. If root project file has attribute `Main` declared and all specified mains are Ada sources, then combined closure of those mains is processed. if root project does not have attribute `Main` declared, or at least one of the mains is not an Ada source, then all sources of non-externally built projects in the project hierarchy are processed.

If the argument project file is an aggregate project, and it aggregates more than one (non-aggregate) project, gnatcheck runs separately for each (non-aggregate) project being aggregated by the argument project, and a separate report file is created for each of these runs. Also such a run creates an umbrella report file that lists all the (non-aggregate) projects that are processed separately and for each of these projects contains the reference for the corresponding report file.

If the argument project file defines an aggregate project that aggregates only one (non-aggregate) project, the gnatcheck behavior is the same as for the case of non-aggregate argument project file.

2.2 The *Check* GPR Package

In addition to the command-line options, you can use attributes offered by the `Check` package to configure a GNATcheck run. In order to do this you may add the `Check` package in the GPR file you're providing to GNATcheck through the `-P` command line options, example given:

```
project My_Project is
   package Check is
      ...
   end Check;
end My_Project;
```

Inside this package you can define the following attributes to configure GNATcheck:

Log
Value is whether to duplicate all GNATcheck outputs in a log file. Accepted values are `True` and `False`.

Rules
Value is a list of rules to enable when invoking `gnatcheck` on this project. Values provided in this attribute behave as the ones provided with the `--rule` switch.

If the `--rule` switch is set when calling `gnatcheck` on a project file defining this attribute, then, values are concatenated.

Rule_File
Value is a path to a LKQL rule file. If not absolute, the path is relative to the project file that defines this attribute. See *LKQL Rule Files* for more information.

If the `--rule-file` switch is set when calling `gnatcheck` on a project file defining this attribute, then, an error is emitted and `gnatcheck` will exit with an error code.

`Lkql_Path`

Value is a list of directories to add to the `LKQL_PATH` environment variable when GNATcheck is spawning the LKQL engine. This variable is used to resolve module importations in LKQL sources. If not absolute, paths provided through this attribute are relatives to the project file defining it.

This attributes may work combined with the `--lkql-path` switch, in that case, all directories are added to the `LKQL_PATH` environment variable.

`Switches`

Index is a language name. Value is a list of additional switches to be used when invoking `gnatcheck`.

If a switch is provided in both command-line and `Switches` attribute, then, the value provided through the command-line is used.

> **Attention:** There are several command-line switches that you cannot pass through the `Switches` attribute:
>
> - `-V, --version`
> - `-h, --help`
> - `--list-rules`
> - `-P`
> - `-U`
> - `-Xname=value`
> - `--no-subprojects`
> - `-vP`
> - `-eL`
> - `-log` (use `Log` attribute instead)
> - `-r, --rule [rule_name]` (use `Rules` attribute instead)
> - `--rule-file=filename` (use `Rule_File` attribute instead)
> - `--lkql-path=dir` (use `Lkql_Path` attributes instead)
> - `--target` (use the `Target` GPR attribute instead)
> - `--RTS` (use the `Runtime` GPR attribute instead)
>
> If you're providing one of those switches through the `Switches` or the `Default_Switches` attribute, GNATcheck will emit an error message and exit with an error code.

`Default_Switches`

Same as `Switches`, but provided additional switches will apply only if there is no applicable `Switches` attribute.

2.3 Sources pre-processing

GNATcheck is handling Ada sources pre-processing, meaning that sources lines that are "excluded" by the Ada pre-processor are also ignored during the GNATcheck analysis. For example, given the following source:

```
procedure Main is
begin
   # if Foo = "Bar" then
   goto lbl;
   # else
   null;
   # end if;
end Main;
```

Running GNATcheck with the `Goto_Statements` rule enabled on this Ada code will flag the `goto lbl;` if, and only if, the preprocessor symbol `Foo` is set to `"Bar"`.

To configure pre-processing, you can use the following GPR attributes:

- `Builder.Global_Compilation_Switches`
- `Builder.Default_Switches`
- `Builder.Switches`
- `Compiler.Default_Switches`
- `Compiler.Switches`

Attention: There is a limitation to the GNATcheck's pre-processing handling regarding conditioned `with` clauses. Meaning that no matter how symbols are defined, all `with` clauses are going to be analyzed and used by GNATcheck to resolve the closure of files to analyze.

2.4 LKQL Rule Files

You can configure GNATcheck rules using an LKQL file, provided through the `--rule-file` command-line option or implicitly fetched by GNATcheck (as described in the following paragraph).

By default, GNATcheck will look for a `rules.lkql` file besides the specified project file if any. If one is found and no other rule configuration has been provided (either through the LKQL `--rule-file` option, `--rule` option, or by the now deprecated legacy `-rules` options), GNATcheck will load the rule configuration file as if it was provided by the `--rule-file` option.

Note: You can use the `--emit-lkql-rule-file` CLI switch to generate an LKQL rule file from a legacy rule configuration provided by the `-rules` section.

An LKQL rule file can be any valid LKQL file, the only requirement is that it must export a `rules` top-level symbol. This symbol defines an object value containing rules configuration; keys are GNATcheck rules to enable; and values are list of objects, each one representing an instance of the rule with its parameters. A rule parameter value can be of the boolean, the integer, the string, or the list of strings type, as shown in the simple example below:

```
val rules = @{
  Goto_Statements,
  Forbidden_Attributes: {Forbidden: ["GNAT"], Allowed: ["First", "Last"]}
}
```

Using the `@` object notation is strongly advised to make your configuration file way more understandable:

Please read the *Predefined Rules* documentation to view examples on how to provide parameters to rules through LKQL rule files.

Attention: You cannot provide the same key twice; thus, the following code will result in a GNATcheck error.

```
val rules = @{
  Forbidden_Attributes,
  Forbidden_Attributes: {Forbidden: ["GNAT"], Allowed: ["First", "Last"]}
}
```

If you want to create multiple instances of the same rule, you can associate a list value to the rule name in the rule configuration object. Elements of this list must be parameter objects containing an additional `instance_name` parameter defining the name of the instance described by the enclosing object. If none is provided, the instance is named after the rule it is instantiated from, as shown in the following example:

```
val rules = @{
  Goto_Statements,
  Forbidden_Attributes: [
    # "Forbidden_Attributes" instance of the "Forbidden_Attributes" rule, checking␣
↪for 'First and 'Last
    {Forbidden: ["First", "Last"]},

    # "Length_Attr" instance of the "Forbidden_Attributes" rule, checking for 'Length
    {Forbidden: ["Length"], instance_name: "Length_Attr"}
  ]
}
```

Moreover, each instance must be identifiable through a unique name, thus the following configuration is invalid and will lead to a GNATcheck error:

```
val rules = @{
  Forbidden_Attributes: [
    {Forbidden: ["First", "Last"], instance_name: "Instance"},
    {Forbidden: ["Length"], instance_name: "Instance"},
  ]
}
```

However, if two instances have the same name **and** the same parameters, GNATcheck will just ignore the duplicate instance and emit a warning instead of an error so that the analysis can nevertheless be executed.

Additionally to the `rules` top-level symbol, an LKQL rule file may export `ada_rules` and `spark_rules` symbols to enable associated rules, respectively, only on Ada code or only on SPARK code. Those symbols must also refer to an object value formatted like the `rules` value.

```
# Rules to run on both Ada and SPARK code
val rules = @{
  Goto_Statements
```

(continues on next page)

(continued from previous page)

```
}

# Rules to run only on Ada code
val ada_rules = @{
  Forbidden_Attributes: {Forbidden: ["GNAT"]}
}

# Rules to run only on SPARK code
val spark_rules = @{
  Ada_2022_In_Ghost_Code
}
```

Please note that compiler based rules (*Warnings*, *Restrictions* and *Style_Checks*) cannot be restricted to Ada or SPARK code. Consequently, the following configuration will raise an error:

```
val spark_rules = @{
  Warnings: {Arg: "a"}
}
```

Attention: Instance uniqueness must also be respected between all rule sets, meaning that such config is invalid:

```
val rules = @{
  # Clashing with "Goto_Statement" in ada_rules
  Goto_Statements,

  # Clashing with "Forbid_Attr" instance in spark_rules
  Forbidden_Attributes: {Forbidden: ["GNAT"], instance_name: "Forbid_Attr"}
}

val ada_rules = @{
  Goto_Statements
}

val spark_rules = @{
  Forbidden_Attributes: {Forbidden: ["Length"], instance_name: "Forbid_Attr"}
}
```

You cannot provide more than **one** LKQL rule file when running GNATcheck. In order to compose a rule file with another you have to use the *LKQL importation mechanism* and combine rule objects. Here is an example of LKQL rule file composition:

```
# common_rules.lkql

val rules = @{
  Goto_Statements
}
```

```
# specific_rules.lkql

import common_rules
```

(continues on next page)

(continued from previous page)

```
val rules = common_rules.rules.combine(
  @{ Redundant_Null_Statements }
)
```

Then you can run GNATcheck with the `specific_rules.lkql` file as coding standard to perform rules defined in `common_rules.lkql` combined to the ones defined in `specific_rules.lkql`.

Note: You can use the `--lkql-path` command-line switch and the `Check'Lkql_Path` GPR attribute to configure directories LKQL rule files are going to be searched in.

You can enable the same rule in multiple files, but the constraint about the instance name uniqueness remains valid: when two instances have the same name, GNATcheck will emit an error if the instances have different parameters but ignore duplicate instances that are configured identically. That means such a configuration is invalid:

```
# common_rules.lkql

val rules = @{
  Forbidden_Attributes: {Forbidden: ["First"], instance_name: "Forbid_Attr"}
}
```

```
# specific_rules.lkql

import common_rules

val rules = common_rules.rules.combine(@{
  Forbidden_Attributes: {
    Forbidden: ["Last"],
    instance_name: "Forbid_Attr"
  }
})
# error: This rule configuration defines two instances with the same name: "Forbid_Attr"
```

Note that GNATcheck will also detect instances that run the same check (i.e., instances that have different names but are configured with the same parameters). In such cases, GNATcheck will emit a warning so that duplicate checks can be easily detected when combining rules object.

Same restrictions apply when combining LKQL rules files with rules specified with the command line interface (using the `--rule` *switch*), or through the `Check` *GPR package*.

Attention: Combining *LKQL rule file* with the deprecated *GNATcheck Rule Options* is undefined behavior.

2.5 GNATcheck Rule Options

Attention: Rules options are **deprecated**, consider converting your rule configuration to the new *LKQL rule file* format using the `--emit-lkql-rule-file` CLI switch.

The following options control the processing performed by `gnatcheck`. You can provide as many rule options as you want after the `-rules` switch.

+R[:instance_name:]rule_id[:param{,param}]

Create and enable an instance of the specified rule with the specified parameter(s), if any. *rule_id* must be the identifier of one of the currently implemented rules (use `--list-rules` for the list of implemented rules). Rule identifiers are not case-sensitive.

Each *param* item must be a non-empty string representing a valid parameter for the specified rule. If the part of the rule option that follows the colon character contains any space characters then this part must be enclosed in quotation marks.

instance_name is a user-defined name for the created rule instance. If this is not specified, the instance name is set to the rule name (normalized to lower case). You can create as much instances as you want for a single rule, as long as they have distinct names (names aren't case sensitive either). If an instance of the same rule with the same name already exists GNATcheck will raise an error.

For example:

```
--  Create and enable an instance of "Goto_Statements" named
--  "goto_statements".
+RGoto_Statements

--  Create and enable an second instance of "Goto_Statements" named
--  "custom_name".
+R:custom_name:Goto_Statements

--  Create and enable an instance of "Recursive_Subprograms" named
--  "other_name".
+R:other_name:Recursive_Subprograms

--  This will cause a GNATcheck error because the "goto_statement" instance
--  already exists.
+RGoto_Statements
```

This feature can be used to map `gnatcheck` rules onto a user's coding standard.

-R[:instance_name:]rule_id

Remove the designated rule instance, disabling it at the same time.

Note: By removing a rule instance, all previously given instance parameter(s) are cleared from the GNATcheck memory.

Attention: No parameters are allowed for the `-R` rule option. Since rule instances are immutable, you cannot modify a parameter set once the instance has been created by a `+R` option.

-from=rule_option_filename
Read the rule options from the text file *rule_option_filename*, referred to as a 'coding standard file' below.

The default behavior is that all the rule checks are disabled.

If a rule option is given in a rule file, it can contain spaces and line breaks. Otherwise there should be no spaces between the components of a rule option.

If more than one rule option is specified for the same rule, with the same instance name, GNATcheck will raise an error and stop its execution.

Attention: Unlike in older versions of GNATcheck, rule instances aren't mutable, so you cannot change options for an instance after its instantiation.

A coding standard file is a text file that contains a set of rule options described above.

The file may contain empty lines and Ada-style comments (comment lines and end-of-line comments). There can be several rule options on a single line (separated by a space).

A coding standard file may reference other coding standard files by including more `-from=rule_option_filename` options, each such option being replaced with the content of the corresponding coding standard file during processing. In case a cycle is detected (that is, `rule_file_1` reads rule options from `rule_file_2`, and `rule_file_2` reads (directly or indirectly) rule options from `rule_file_1`), processing fails with an error message.

If the name of the coding standard file does not contain a path information in absolute form, then it is treated as being relative to the current directory if gnatcheck is called without a project file or as being relative to the project file directory if gnatcheck is called with a project file as an argument.

2.6 Mapping GNATcheck Rules Onto Coding Standards

If you want to use `GNATcheck` to check if your code follows a given coding standard, you can use the following approach to simplify mapping your coding standard requirements onto `GNATcheck` rules:

- when specifying rule configuration, use instance names that are relevant to your coding standard:

```
val rules = @{
  Gnatcheck_Rule_1: {instance_name: "My_Coding_Rule_1", param1: "value"},
  ...
  Gnatcheck_Rule_N: {instance_name: "My_Coding_Rule_N"}
}
```

  or with the deprecated rule options:

```
+R:My_Coding_Rule_1:Gnatcheck_Rule_1:param1
...
+R:My_Coding_Rule_N:Gnatcheck_Rule_N
```

- call `gnatcheck` with the `--show-rule` flag that adds the rule names the generated diagnoses. If a instance name is defined in the rule configuration, then this name will be used to annotate the diagnosis of the rule name:

```
foo.adb:2:28: something is wrong here [My_Coding_Rule_1|Gnatcheck_Rule_1]
...
bar.ads:17:3: this is not good [My_Coding_Rule_N|Gnatcheck_Rule_N]
```

Attention: A custom coding rule name can be any sequence of non-whitespace characters. Moreover, the ":" (colon) character is forbidden in those names for parsing purposes.

2.7 GNATcheck Exit Codes

`gnatcheck` returns the following exit codes at the end of its run:

- `0`: No tool failure, no missing argument source and no rule violation was detected.
- `1`: No tool failure, no missing argument source and at least one rule violation was detected.
- `2`: A tool failure was detected (in this case the results of the gnatcheck run cannot be trusted).
- `3`: No tool failure, no problem with rule specification, but there is at least one missing argument source.
- `4`: Provided rule configuration file doesn't exist.
- `5`: The name of an unknown rule in a rule option or some problem with rule parameters.
- `6`: Any other problem with specifying the rules to check.

If the exit code corresponds to some problem with defining the rules to check then the result of the gnatcheck run cannot be fully trusted because the set of rules that has been actually used may be different from user intent.

If gnatcheck is called with the `--brief` option, it will return the exit code `0` instead of `1` when some violation is detected (and no tool failure).

2.8 Format of the Report File

The `gnatcheck` tool outputs on `stderr` all messages concerning rule violations except if running in quiet mode. By default it also creates a text file that contains the complete report of the last gnatcheck run, this file is named `gnatcheck.out`. A user can specify generation of the XML version of the report file (its default name is `gnatcheck.xml`) If `gnatcheck` is called with a project file, the report file is located in the object directory defined by the project file (or in the directory where the argument project file is located if no object directory is defined), if `--subdirs` option is specified, the file is placed in the subdirectory of this directory specified by this option. Otherwise it is located in the current directory; the `-o` or `-ox` option can be used to change the name and/or location of the text or XML report file. This text report contains:

- general details of the `gnatcheck` run: date and time of the run, the version of the tool that has generated this report, full parameters of the `gnatcheck` invocation, reference to the list of checked sources and applied rules (coding standard);
- summary of the run (number of checked sources and detected violations);
- list of exempted coding standard violations;
- list of non-exempted coding standard violations;
- list of problems in the definition of exemption sections;
- list of language violations (compile-time errors) detected in processed sources;

The references to the list of checked sources and applied rules are references to the text files that contain the corresponding information. These files could be either files supplied as `gnatcheck` parameters or files created by `gnatcheck`; in the latter case these files are located in the same directory as the report file.

The content of the XML report is similar to the text report except that it explores the set of files processed by gnatcheck and the coding standard used for checking these files.

2.9 Rule Exemption

One of the most useful applications of `gnatcheck` is to automate the enforcement of project-specific coding standards, for example in safety-critical systems where particular features must be restricted in order to simplify the certification effort. However, it may sometimes be appropriate to violate a coding standard rule, and in such cases the rationale for the violation should be provided in the source program itself so that the individuals reviewing or maintaining the program can immediately understand the intent.

The `gnatcheck` tool supports this practice with the notion of a 'rule exemption' covering a specific source code section. Normally rule violation messages are issued both on `stderr` and in a report file. In contrast, exempted violations are not listed on `stderr`; thus users invoking `gnatcheck` interactively (e.g. in its GNAT Studio interface) do not need to pay attention to known and justified violations. However, exempted violations along with their justification are documented in a special section of the report file that `gnatcheck` generates.

2.9.1 Using pragma `Annotate` to control rule and instance exemption

Rule and instance exemption is controlled by pragma `Annotate` when its first argument is 'gnatcheck'. The syntax of `gnatcheck`'s exemption control annotations is as follows:

```
<pragma_exemption>  ::= pragma Annotate (gnatcheck, <exemption_control>, <exempted_name>␣
↪[, <justification>]);

<exemption_control> ::= Exempt_On | Exempt_Off

<exempted_name>     ::= <string_literal>

<justification>     ::= <expression>
```

An expression used as an exemption justification should be a static string expression. A string literal is enough in most cases, but you may want to use concatenation of string literals if you need a long message but you have to follow line length limitation.

When a `gnatcheck` annotation has more than four arguments, `gnatcheck` issues a warning and ignores the additional arguments. If the arguments do not follow the syntax above, `gnatcheck` emits a warning and ignores the annotation.

The `exempted_name` argument should be the name of some existing `gnatcheck` rule, or the name of a rule instance. Otherwise a warning message is generated and the pragma is ignored. If `exempted_name` doesn't denote an activated rule or a valid instance in the given `gnatcheck` call, the pragma is ignored and no warning is issued. The exception from this rule is that exemption sections for `Warnings` rule are fully processed when `Restrictions` rule is activated.

> **Attention:** Please not that for now it isn't possible to provide an exempted name which designates an instance of a compiler-based rule (*Warnings*, *Style_Checks* and *Restrictions*) with a custom name.

A source code section where an exemption is active for a given rule is delimited by an `exempt_on` and `exempt_off` annotation pair:

```
pragma Annotate (gnatcheck, Exempt_On, "Rule_Name", "justification");
-- source code section
pragma Annotate (gnatcheck, Exempt_Off, "Rule_Name");
```

Using such annotations will exempt all violations of the rule designated by `Rule_Name` inside the exempted source section. But you can also provide the name of a rule instance to only exempt violations raised by this instance.

For some rules it is possible specify rule parameter(s) when defining an exemption section for a rule or an instance of it. This means that only the checks corresponding to the given rule parameter(s) are exempted in this section:

```
pragma Annotate (gnatcheck, Exempt_On, "Rule_Name: Par1, Par2", "justification");
-- source code section
pragma Annotate (gnatcheck, Exempt_Off, "Rule_Name: Par1, Par2");
```

A parametric exemption section can be defined for a rule if a rule has parameters and these parameters change the scope of the checks performed by a rule. For example, if you define an exemption section for 'Restriction' rule with the parameter 'No_Allocators', then in this section only the checks for `No_Allocators` will be exempted, and the checks for all the other restrictions from your coding standard will be performed as usual.

See the description of individual rules to check if parametric exemptions are available for them and what is the format of the rule parameters to be used in the corresponding parameters of the `Annotate` pragmas.

If a rule has a parameter, but its documentation does not explicitly say that the parameter can be used when defining exemption sections for the rule, this means that the parametric exemption cannot be used for this rule.

You may also use pragma `GNAT_Annotate` instead of pragma `Annotate`, this pragma has exactly the same format. This may be needed if you are using an old version of the GNAT compiler that does not support the format of pragma `Annotate` given above. Old GNAT versions may issue warning about unknown pragma when compiling a source that contains pragma `GNAT_Annotate`.

2.9.2 GNATcheck Annotations Rules

- An `Exempt_Off` annotation can only appear after a corresponding 'Exempt_On' annotation.
- An `Exempt_On` annotation should have a justification. Conversely, an `Exempt_Off` annotation should *not* have a justification.
- Exempted source code sections are only based on the source location of the annotations. Any source construct between the two annotations is part of the exempted source code section.
- Exempted source code sections for different rules are independent. They can be nested or intersect with one another without limitation. Creating nested or intersecting source code sections for the same rule is not allowed.
- A matching 'Exempt_Off' annotation pragma for an 'Exempt_On' pragma that defines a parametric exemption section is the pragma that contains exactly the same set of rule parameters for the same exempted name.
- Parametric exemption sections for the same rule with different parameters can intersect or overlap in case if the parameter sets for such sections have an empty intersection.
- Malformed exempted source code sections are reported by a warning, and the corresponding rule exemptions are ignored.
- When an exempted source code section does not contain at least one violation of the exempted name, a warning is emitted on `stderr`.
- If an 'Exempt_On' annotation pragma does not have a matching 'Exempt_Off' annotation pragma in the same compilation unit, a warning is issued and the exemption section is considered to last until the end of the compilation unit source.

2.9.3 Using comments to control rule and instance exemption

As an alternative to the `pragma Annotate` syntax, it is also possible to use a syntax based on comments, with the following syntax:

```
<comment_exemption> ::= --## rule (on | off) <exempted_name> [## <exemption_
↪justification>]
```

Here is an example:

```
--## rule off implicit_in ## Exemption justification
procedure Bar (A : Integer);
--## rule on implicit_in
```

Attention: Please note that a comment starting with `--##` but not respecting the above syntax will not trigger a warning, in order to not emit false positives. Also note that in its current iteration, this syntax does not support passing parameters to rule names

The rules mentioned in *GNATcheck Annotations Rules* are relaxed, in particular:

- Justifications are not checked and are optional;
- Anything between the exempted name and `##` will be ignored;
- Rules regarding parametric exemption do not apply, as per the notice above.

The `rule on` marker corresponds to `Exempt_Off` and `rule off` corresponds to `Exempt_On`. Apart from that, you can expect those rule exemptions to work in a similar fashion as the ones described above.

In addition, a shorthand syntax is available to exempt a rule just for one line:

```
<line_comment_exemption> ::= --## rule line off <exempted_name> [## <rule_justification>]
```

For instance, from the previous example:

```
procedure Bar (A : Integer); --## rule line off implicit_in ## Exemption justification
```

This will exempt the given rule or instance only for the line on which this comment is placed, and automatically turn it back on on the next line.

2.10 Using GNATcheck as a Known Problem Detector

If you are a GNAT Pro Assurance customer, you have access to a special packaging of GNATcheck called `gnatkp` (GNAT Known Problem detector) where the `gnatcheck` executable is replaced by `gnatkp`, and the coding standard rules are replaced by rules designed to detect constructs affected by known problems in official compiler releases. Note that GNATkp comes in addition and not as a replacement of GNATcheck.

You can use the command `gnatkp --help` to list all the switches relevant to GNATkp. GNATkp mostly accepts the same command arguments as GNATcheck and behaves in a similar way, but there are some differences that are described below.

The easiest way to use GNATkp is by specifying the version of GNAT Pro that you have and letting `gnatkp` run all known problem detectors registered for this version, via the switch `--kp-version`. For example:

```
gnatkp -Pproject --kp-version=21.2 --target=<my_target> --RTS=<my_runtime>
```

will run all detectors relevant to GNAT Pro 21.2 on all files in the project. The list of detectors will be displayed as info messages, and will also be listed in the file `gnatkp-rule-list.out`. The list of detected source locations will be generated on standard error, as well as in a file called `gnatkp.out`.

You can display the list of detectors without running them by specifying additionally the `--list-rules` switch, e.g.:

```
gnatkp --kp-version=21.2 --list-rules --target=<my_target> --RTS=<my_runtime>
```

You can also combine the `--kp-version` switch with the `--target` switch to filter out detectors not relevant for your target, e.g:

```
gnatkp -Pproject --kp-version=21.2 --target=powerpc-elf --RTS=<my_runtime>
```

will only enable detectors relevant to GNAT Pro 21.2 and to the `powerpc-elf` target.

Note that you need to have the corresponding target GNAT compiler installed to use this option. By default, detectors for all targets are enabled.

It is also possible to specify the custom list of detectors for GNATkp to run using the switch `-r`:

```
gnatkp -Pproject --target=<my_target> --RTS=<my_runtime> -r kp_xxxx_xxx [-r kp_xxxx_xxx]
```

where `kp_xxxx_xxx` is the name of a relevant known-problem to detect. You can get the list of available detectors via the command `gnatkp --list-rules`. When combined with the `--kp-version` and possibly `--target` switches, `gnatkp --list-rules` will only list the detectors relevant to the version (and target) specified.

Attention: You must provide explicit target and runtime (either through the command-line or with a provided project file) when running GNATkp to ensure the result soundness.

Note: The exemption mechanism is available for GNATkp as well but you have to change pragmas and comments a bit to avoid conflict with GNATcheck exemptions. Thus, pragmas annotations' first argument must be `gnatkp` instead of `gnatcheck`:

```
pragma Annotate (gnatkp, Exempt_On, "kp_19198", "Justification");
```

And exemption comments' first word must be `kp` instead of `rule`, example:

```
--## kp off kp_19198 ## Justification
```

You can check via the GNAT Tracker interface which known problems are relevant to your version of GNAT and your target before deciding which known problems may impact you: most known problems are only relevant to a specific version of GNAT, a specific target, or a specific usage profile. Do not hesitate to contact the AdaCore support if you need help identifying the entries that may be relevant to you.

2.11 Performance and Memory Usage

GNATcheck performances are closely related to rules you're enabling and to the size of the codebase you're running it on, and sometimes it can take a lot of time to perform all checks. You can use the `-j` switch to run GNATcheck in multi-core mode and decrease the checking time. However, you have to be careful about memory usage when running GNATcheck with this mode enabled:

You should count around 3.5 GB of available memory per million source code lines, per process. Meaning that for a project with `l` source code lines, if you run GNATcheck while providing `n` as parameter of the `-j` switch, you will need `(l / 1,000,000) * 3.5 * n` GB of available memory (this formula isn't valid if `n = 0`).

Attention: Out-of-memory errors are hard to debug and can lead to system freezes, invalid results, or non-deterministic behavior. Thus, make sure you have enough memory before running GNATcheck.

2.12 Transition from ASIS-based GNATcheck

Originally `gnatcheck` was implemented on top of the ASIS technology and starting with version 23, it was re-implemented on top of the libadalang technology. This new implementation has kept most of the old gnatcheck interface and functionality, so transition from the old `gnatcheck` to the current version should be smooth and transparent, except possibly for a few aspects to be taken into account by users of the old technology.

2.12.1 Switches No Longer Supported

The following switches from the old `gnatcheck` are no longer supported:

`-a`
: In order to process GNAT Run-Time library units, you need to explicitly include them in a project file.

`--incremental`
: GNATcheck no longer makes the distinction between "local" and "global" rules, so this switch is no longer supported. You can use the `-j` switch instead which provides a significant speed up compared to the old version.

`--write-rules=template_file`
: This switch is no longer supported. You can use the GNAT Studio rule editor instead to create a coding standard file.

2.12.2 The new rule instance system

The new `gnatcheck` implementation is introducing a new rule instance system which allows you to instantiate a rule multiple times under different names, and with potentially different rule parameters. You can now define more that one "alias" for the same rule to map your coding standard on the `gnatcheck` rules. However, rules aren't mutable anymore, which means that you cannot modify parameters of a rule once it has been created (instantiated). For example:

```
--  The rule "Goto_Statements" is instantiated here
+RGoto_Statements

--  We try to create a new instance of the "Goto_Statements", this will fail
+RGoto_Statements:Only_Unconditional
```

While with the old system this rule file would just mutate the previously enabled "Goto_Statements" rule, with the new instance system, this will cause an error during the `gnatcheck` run, telling you that the "goto_statement" instance already exists. To correct this error, you have define a custom name for the second "Goto_Statements" instance:

```
--  The rule "Goto_Statements" is instantiated here
+RGoto_Statements

--  The rule "Goto_Statements" is also instantiated here,
--  under the "Uncond_Goto" name.
+R:Uncond_Goto:Goto_Statements:Only_Unconditional
```

The same way, you have to rewrite rule options such as:

```
+RForbidden_Pragmas:GNAT
+RForbidden_Pragmas:Annotate
+RForbidden_Pragmas:Assert
```

into a single rule option using the comma separated notation, like:

```
+RForbidden_Pragmas:GNAT,
                    Annotate,
                    Assert
```

This new instance system also suppress the possibility to disable a rule (or an instance) with a parameter. Thus, the `-R` rule option doesn't accept parameters anymore.

2.12.3 Rule Aliases No Longer Supported

Because of historical reasons the old `gnatcheck` allowed aliases for some rules. These aliases are not documented, but there is some possibility that they could be used in some legacy rule files. `GNATcheck` no longer supports these aliases. Here is the (alphabetically ordered) list of all the aliases formerly accepted and their replacement:

Old Rule Alias	Replacement
Abstr_Types	Abstract_Type_Declarations
Bool_Relation_Ops	Boolean_Relational_Operators
Contr_Types	Controlled_Type_Declarations
Control_Structure_Nesting	Overly_Nested_Control_Structures
Decl_Blocks	Declarations_In_Blocks
Default_Par	Default_Parameters
Derived_Types	Non_Tagged_Derived_Types
Discr_Rec	Discriminated_Records
Explicit_Discrete_Ranges	Explicit_Full_Discrete_Ranges
Functionlike_Procedures	Function_Style_Procedures
Global_Loop_Exit	Outer_Loop_Exits
Goto	GOTO_Statements
Implicit_IN_Parameter_Mode	Implicit_IN_Mode_Parameters
LL_Subpr	Library_Level_Subprograms
Local_Pckg	Local_Packages
Misnamed_Identifiers	Identifier_Suffixes
Missing_Small_For_Fixed_Point_Type	Implicit_SMALL_For_Fixed_Point_Types
Non_Marked_BEGIN_In_Package_Body	Uncommented_BEGIN_In_Package_Bodies
Non_Named_Blocks_And_Loops	Unnamed_Blocks_And_Loops

continues on next page

Table 1 – continued from previous page

Old Rule Alias	Replacement
One_Entry_In_PO	Multiple_Entries_In_Protected_Definitions
Parameter_Mode_Ordering	Parameters_Out_Of_Order
Positional_Component_Associations	Positional_Components
Positional_Generic_Associations	Positional_Generic_Parameters
Positional_Parameter_Associations	Positional_Parameters
Pragma_Usage	Forbidden_Pragmas
Predefined_Exceptions	Raising_Predefined_Exceptions
Proper_Returns	Improper_Returns
Qualified_Aggr	Non_Qualified_Aggregates
Restrict_Name_Space	Name_Clashes
Simple_Loop_Exit_Names	Expanded_Loop_Exit_Names
SPARK_Attributes	Non_SPARK_Attributes
Unconstr_Array_Return	Unconstrained_Array_Returns
Universl_Ranges	Universal_Ranges
Unreasonable_Places_For_Instantiations	Improperly_Located_Instantiations
Use_Pckg_Clauses	USE_PACKAGE_Clauses
Use_Of_Non_Short_Circuit	Non_Short_Circuit_Operators
Visible_Exceptions	Raising_External_Exceptions
Volatile_Requires_Addr_Clause	Volatile_Objects_Without_Address_Clauses

2.12.4 New Defaults For Recursive_Subprograms Rule

The `Recursive_Subprograms` rule now defaults to skipping dispatching calls and a new parameter `Follow_Dispatching_Calls` is available (the old `Skip_Dispatching_Calls` is still accepted for compatibility and is ignored since it's the default). In addition, implicit calls made via default object initialization are not taken into account.

2.12.5 Argument Sources Legality And Project Files

The old `gnatcheck` compiled its argument sources to create the so-called ASIS tree files. This had two important consequences: first, `gnatcheck` could analyze only legal Ada sources, and second, for each legal argument source `gnatcheck` had full static semantic information. The situation with the current `gnatcheck` is different.

First, `gnatcheck` can now analyze Ada sources that are not legal, and it is trying to do its best to check the rules specified. This may result in false negatives caused by the absence of necessary semantic information or by some other problems in the argument source that impede a full check of some rules. You can use the `--check-semantic` option to check if your Ada sources are legal sources.

Second, if `gnatcheck` is called for some Ada source and it does not have a project file as a parameter, it will see only the information contained in the sources specified and will not follow the semantic dependencies on other sources if any. This is why it is strongly recommended to call `gnatcheck` with a project file. When called with a project file, `gnatcheck` follows all the semantic dependencies for sources located in the project file source directories.

This page is intentionally left blank.

CHAPTER

THREE

PREDEFINED RULES

The description of the rules currently implemented in `gnatcheck` is given in this chapter. The rule identifier is used as a key for LKQL rule configuration objects (see *LKQL rule file*), and as first parameter of `gnatcheck`'s `+R` or `-R` switches.

Be aware that most of these rules apply to specialized coding requirements developed by individual users and may well not make sense in other environments. In particular, there are many rules that conflict with one another. Proper usage of gnatcheck involves selecting the rules you wish to apply by looking at your independently developed coding standards and finding the corresponding gnatcheck rules.

Unless documentation is specifying some, rules don't have any parameters.

If not otherwise specified, a rule does not do any check for the results of generic instantiations.

GNATcheck's predefined rules' parameters may have the following types:

bool

The parameter represents a boolean value, toggling a rule behavior. In a LKQL rule file you have to associate a boolean value to the parameter name:

```
val rules = @{
   My_Rule: {Bool_Param: true}
}
```

To specify a boolean parameter through a `+R` option, you just have to provide the parameter's name to set it to true:

```
+RMy_Rule:Bool_Param  -- 'Bool_Param' value is set to true
```

int

The parameter is an integer value. In a LKQL rule options file, you have to associate an integer value to the parameter name:

```
val rules = @{
   My_Rule: {N: 5} # If the rule param is named 'N'
}
```

To specify it with a `+R` option, you can write its value right after the rule name:

```
+RMy_Rule:5  -- 'My_Rule' integer param is set to 5
```

string

The parameter value is a string, sometimes with formatting constraints. In a LKQL rule options file, you just have to provide a string value:

```
val rules = @{
   My_Rule: {Str: "i_am_a_string"} # If the rule param is named 'Str'
}
```

You can specify it through the +R option also by passing a string right after the rule name:

```
+RMy_Rule:i_am_a_string  -- 'My_Rule' string param is set to "i_am_a_string"
```

list[string]

The parameter value is a list of string. In a LKQL rule options file, you can use the LKQL list type to specify the parameter value:

```
val rules = @{
   My_Rule: {Lst: ["One", "Two", "Three"]} # If the rule param is named 'Lst'
}
```

Through the +R option, you can specify it as a collection of string parameters separated by commas:

```
+RMy_Rule:One,Two,Three  -- 'My_Rule' string list param is set to ["One", "Two",
↪"Three"]
```

3.1 Style-Related Rules

The rules in this section may be used to enforce various feature usages consistent with good software engineering, for example as described in Ada 95 Quality and Style.

3.1.1 Tasking

The rules in this subsection may be used to enforce various feature usages related to concurrency.

3.1.1.1 Multiple_Entries_In_Protected_Definitions

Flag each protected definition (i.e., each protected object/type declaration) that declares more than one entry. Diagnostic messages are generated for all the entry declarations except the first one. An entry family is counted as one entry. Entries from the private part of the protected definition are also checked.

Example

```
protected PO is
   entry Get (I :     Integer);
   entry Put (I : out Integer);    --  FLAG
   procedure Reset;
   function Check return Boolean;
private
   Val : Integer := 0;
end PO;
```

3.1.1.2 Volatile_Objects_Without_Address_Clauses

Flag each volatile object that does not have an address specification. Only variable declarations are checked.

An object is considered as being volatile if a pragma or aspect Volatile is applied to the object or to its type, if the object is atomic or if the GNAT compiler considers this object as volatile because of some code generation reasons.

Example

```
with Interfaces, System, System.Storage_Elements;
package Foo is
   Variable: Interfaces.Unsigned_8
      with Address => System.Storage_Elements.To_Address (0), Volatile;

   Variable1: Interfaces.Unsigned_8                                    --  FLAG
      with Volatile;

   type My_Int is range 1 .. 32 with Volatile;

   Variable3 : My_Int;                                                 --  FLAG

   Variable4 : My_Int
     with Address => Variable3'Address;
end Foo;
```

3.1.2 Object Orientation

The rules in this subsection may be used to enforce various feature usages related to Object-Oriented Programming.

3.1.2.1 Constructors

Flag any declaration of a primitive function of a tagged type that has a controlling result and no controlling parameter. If a declaration is a completion of another declaration then it is not flagged.

Example

```
type T is tagged record
   I : Integer;
end record;

function Fun (I : Integer) return T;                 -- FLAG
function Bar (J : Integer) return T renames Fun;     -- FLAG
function Foo (K : Integer) return T is ((I => K));   -- FLAG
```

3.1.2.2 Deep_Inheritance_Hierarchies

Flags a tagged derived type declaration or an interface type declaration if its depth (in its inheritance hierarchy) exceeds the value specified by the *N* rule parameter. Types in generic instantiations which violate this rule are also flagged; generic formal types are not flagged. This rule also does not flag private extension declarations. In the case of a private extension, the corresponding full declaration is checked.

In most cases, the inheritance depth of a tagged type or interface type is defined as 0 for a type with no parent and no progenitor, and otherwise as 1 + max of the depths of the immediate parent and immediate progenitors. If the declaration of a formal derived type has no progenitor, or if the declaration of a formal interface type has exactly one progenitor, then the inheritance depth of such a formal derived/interface type is equal to the inheritance depth of its parent/progenitor type, otherwise the general rule is applied.

If the rule flags a type declaration inside the generic unit, this means that this type declaration will be flagged in any instantiation of the generic unit. But if a type is derived from a format type or has a formal progenitor and it is not flagged at the place where it is defined in a generic unit, it may or may not be flagged in instantiation, this depends of the inheritance depth of the actual parameters.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Integer not less than -1 specifying the maximal allowed depth of any inheritance hierarchy. If the rule parameter is set to -1, the rule flags all the declarations of tagged and interface types.

Example

```
type I0 is interface;
type I1 is interface and I0;
type I2 is interface and I1;

type T0 is tagged null record;
type T1 is new T0 and I0 with null record;
type T2 is new T0 and I1 with null record;
type T3 is new T0 and I2 with null record; -- FLAG (if rule parameter is 2)
```

3.1.2.3 Direct_Calls_To_Primitives

Flag any non-dispatching call to a dispatching primitive operation, except for:

- a call to the corresponding primitive of the parent type. (This occurs in the common idiom where a primitive subprogram for a tagged type directly calls the same primitive subprogram of the parent type.)
- a call to a primitive of an untagged private type, even though the full type may be tagged, when the call is made at a place where the view of the type is untagged.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Except_Constructors: bool
If `true`, do not flag non-dispatching calls to functions if the function has a controlling result and no controlling parameters (in a traditional OO sense such functions may be considered as constructors).

Example

```
package Root is
   type T_Root is tagged private;

   procedure Primitive_1 (X : in out T_Root);
   procedure Primitive_2 (X : in out T_Root);
private
   type T_Root is tagged record
      Comp : Integer;
   end record;
end Root;

package Root.Child is
   type T_Child is new T_Root with private;

   procedure Primitive_1 (X : in out T_Child);
   procedure Primitive_2 (X : in out T_Child);
private
   type T_Child is new T_Root with record
      B : Boolean;
   end record;
end Root.Child;

package body Root.Child is

   procedure Primitive_1 (X : in out T_Child) is
   begin
      Primitive_1 (T_Root (X));       --  NO FLAG
      Primitive_2 (T_Root (X));       --  FLAG
      Primitive_2 (X);                --  FLAG
   end Primitive_1;

   procedure Primitive_2 (X : in out T_Child) is
   begin
      X.Comp  := X.Comp + 1;
   end Primitive_2;

end Root.Child;
```

3.1.2.4 Downward_View_Conversions

Flag downward view conversions.

This rule will also flag downward view conversions done through access types.

Example

```
package Foo is
   type T1 is tagged private;
   procedure Proc1 (X : in out T1'Class);

   type T2 is new T1 with private;
   procedure Proc2 (X : in out T2'Class);

private
   type T1 is tagged record
      C : Integer := 0;
   end record;

   type T2 is new T1 with null record;
end Foo;

package body Foo is

   procedure Proc1 (X : in out T1'Class) is
      Var : T2 := T2 (X);                          --  FLAG
      X_Acc : T1_Access := X'Unrestricted_Access;
      Var_2 : T2_Access := T2_Access (X_Acc);      --  FLAG
   begin
      Proc2 (T2'Class (X));                        --  FLAG
   end Proc1;

   procedure Proc2 (X : in out T2'Class) is
   begin
      X.C := X.C + 1;
   end Proc2;

end Foo;
```

3.1.2.5 No_Inherited_Classwide_Pre

Flag a declaration of an overriding primitive operation of a tagged type if at least one of the operations it overrides or implements does not have (explicitly defined or inherited) Pre'Class aspect defined for it.

Example

```
package Foo is

   type Int is interface;
   function Test (X : Int) return Boolean is abstract;
   procedure Proc (I : in out Int) is abstract with Pre'Class => Test (I);

   type Int1 is interface;
   procedure Proc (I : in out Int1) is abstract;

   type T is tagged private;
```

(continues on next page)

(continued from previous page)

```
    type NT1 is new T and Int with private;
    function Test (X : NT1) return Boolean;         --  FLAG
    procedure Proc (X : in out NT1);

    type NT2 is new T and Int1 with private;
    procedure Proc (X : in out NT2);                --  FLAG

   private
   type T is tagged record
      I : Integer;
   end record;

   type NT1 is new T and Int with null record;
   type NT2 is new T and Int1 with null record;

end Foo;
```

3.1.2.6 Specific_Parent_Type_Invariant

Flag any record extension definition or private extension definition if a parent type has a Type_Invariant aspect defined for it. A record extension definition is not flagged if it is a part of a completion of a private extension declaration.

Example

```
package Pack1 is
   type PT1 is tagged private;
   type PT2 is tagged private
     with Type_Invariant => Invariant_2 (PT2);

   function Invariant_2   (X : PT2) return Boolean;

private
   type PT1 is tagged record
      I : Integer;
   end record;

   type PT2 is tagged record
      I : Integer;
   end record;

   type PT1_N is new PT1 with null record;
   type PT2_N is new PT2 with null record;     --  FLAG
end Pack1;

package Pack2 is
   type N_PT1 is new Pack1.PT1 with private;
   type N_PT2 is new Pack1.PT2 with private;  --  FLAG
private
   type N_PT1 is new Pack1.PT1 with null record;
```

(continues on next page)

(continued from previous page)

```
   type N_PT2 is new Pack1.PT2 with null record;
end Pack2;
```

3.1.2.7 Specific_Pre_Post

Flag a declaration of a primitive operation of a tagged type if this declaration contains specification of Pre or/and Post aspect.

Example

```
type T is tagged private;
function Check1 (X : T) return Boolean;
function Check2 (X : T) return Boolean;

procedure Proc1 (X : in out T)           --  FLAG
   with Pre => Check1 (X);

procedure Proc2 (X : in out T)           --  FLAG
   with Post => Check2 (X);

function Fun1 (X : T) return Integer     --  FLAG
   with Pre  => Check1 (X),
        Post => Check2 (X);

function Fun2 (X : T) return Integer
   with Pre'Class  => Check1 (X),
        Post'Class => Check2 (X);

function Fun3 (X : T) return Integer     --  FLAG
   with Pre'Class  => Check1 (X),
        Post'Class => Check2 (X),
        Pre        => Check1 (X),
        Post       => Check2 (X);
```

3.1.2.8 Specific_Type_Invariants

Flag any definition of (non-class-wide) Type_Invariant aspect that is a part of a declaration of a tagged type or a tagged extension. Definitions of Type_Invariant'Class aspects are not flagged. Definitions of (non-class-wide) Type_Invariant aspect that are parts of declarations of non-tagged types are not flagged.

Example

```
type PT is private
   with Type_Invariant => Test_PT (PT);
function Test_PT (X : PT) return Boolean;

type TPT1 is tagged private
   with Type_Invariant => Test_TPT1 (TPT1);         --  FLAG
function Test_TPT1 (X : TPT1) return Boolean;

type TPT2 is tagged private
   with Type_Invariant'Class => Test_TPT2 (TPT2);
function Test_TPT2 (X : TPT2) return Boolean;
```

3.1.2.9 Too_Many_Parents

Flag any tagged type declaration, interface type declaration, single task declaration or single protected declaration that has more than *N parents*, where *N* is a parameter of the rule. A *parent* here is either a (sub)type denoted by the subtype mark from the parent_subtype_indication (in case of a derived type declaration), or any of the progenitors from the interface list (if any).

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximal allowed number of parents/progenitors.

Example

```
type I1 is interface;
type I2 is interface;
type I3 is interface;
type I4 is interface;

type T_Root is tagged private;

type T_1 is new T_Root with private;
type T_2 is new T_Root and I1 with private;
type T_3 is new T_Root and I1 and I2 with private;
type T_4 is new T_Root and I1 and I2 and I3 with private; -- FLAG (if rule parameter is␣
↪3 or less)
```

3.1.2.10 Too_Many_Primitives

Flag any tagged type declaration that has more than N user-defined primitive operations (counting both inherited and not overridden and explicitly declared, not counting predefined operators). Only types declared in visible parts of packages, generic packages and package instantiations are flagged.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximal number of primitives when the type is not flagged.

Example

```
package Foo is
   type PT is tagged private;     --  FLAG (if rule parameter is 3 or less)

   procedure P1 (X : in out PT);
   procedure P2 (X : in out PT) is null;
   function F1 (X : PT) return Integer;
   function F2 (X : PT) return Integer is (F1 (X) + 1);

   type I1 is interface;

   procedure P1 (X : in out I1) is abstract;
   procedure P2 (X : in out I1) is null;

   type I2 is interface and I1;   --  FLAG (if rule parameter is 3 or less)
   function F1 (X : I2) return Integer is abstract;
   function F2 (X : I2) return Integer is abstract;

private
   type PT is tagged record
      I : Integer;
   end record;
end Foo;
```

3.1.2.11 Visible_Components

Flag all the type declarations located in the visible part of a library package or a library generic package that can declare a visible component. A visible component can be declared in a *record definition* which appears on its own or as part of a record extension. The *record definition* is flagged even if it contains no components.

Record definitions located in private parts of library (generic) packages or in local (generic) packages are not flagged. *Record definitions* in private packages, in package bodies, and in the main subprogram body are not flagged.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Tagged_Only: bool
If `true`, only declarations of tagged types are flagged.

Example

```
with Types;
package Foo is
   type Null_Record is null record;                                    --  FLAG

   type Not_Null_Record is record                                      --  FLAG
      I : Integer;
      B : Boolean;
   end record;

   type Tagged_Not_Null_Record is tagged record                        --  FLAG
      I : Integer;
      B : Boolean;
   end record;

   type Private_Extension is new Types.Tagged_Private with private;

   type NoN_Private_Extension is new Types.Tagged_Private with record  --  FLAG
      B : Boolean;
   end record;

private
   type Rec is tagged record
      I : Integer;
   end record;

   type Private_Extension is new Types.Tagged_Private with record
      C : Rec;
   end record;
end Foo;
```

3.1.3 Portability

The rules in this subsection may be used to enforce various feature usages that support program portability.

3.1.3.1 Bit_Records_Without_Layout_Definition

Flag record type declarations if a record has a component of a modular type and the record type is packed but does not have a record representation clause applied to it.

Example

```
package Pack is
   type My_Mod is mod 8;

   type My_Rec is record   --  FLAG
      I : My_Mod;
   end record;
```

(continues on next page)

(continued from previous page)

```
   pragma Pack (My_Rec);
end Pack;
```

3.1.3.2 Forbidden_Aspects

Flag each use of the specified aspects. The aspects to be detected are named in the rule's parameters.

This rule has the following parameters for the +R option and for LKQL rule options file:

Forbidden: list[string]
: Adds the specified aspects to the set of aspects to be detected and sets the detection checks for all the specified attributes ON. Note that if some aspect exists also as class-wide aspect, the rule treats its normal and class-wide versions separately. (If you specify `Pre` as the rule parameter, the rule will not flag the `Pre'Class` aspect, and the other way around - specifying `Pre'Class` as the rule parameter does not mean that the rule will flag the `Pre` aspect).

Allowed: string
: A semi-colon separated list of aspects to remove from the set of aspects to be detected. You have to use the named parameter formatting to specify it.

All: bool
: If `true`, all aspects are detected; this sets the rule ON.

Parameters are case insensitive. If an element of *Forbidden* or *Allowed* does not have the syntax of an Ada identifier, it is (silently) ignored, but if such a parameter is given for the +R option, this turns the rule ON.

The +R option with no parameters doesn't create any instance for the rule, thus, it has no effect.

Note: In LKQL rule options files, the `Allowed` parameter should be a list of strings:

```
val rules = @{
   Forbidden_Aspects: {Forbidden: ["one", "two"], Allowed: ["two"]}
}
```

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are *Forbidden*.

Example

```
--  if the rule is activated as +RForbidden_Aspects:Pack,Pre
package Foo is
   type Arr is array (1 .. 10) of Integer with Pack;    --  FLAG

   type T is tagged private;

   procedure Proc1 (X : in out T)
     with Pre => Predicate1;                            --  FLAG

   procedure Proc2 (X : in out T)
     with Pre'Class => Predicate2;                      --  NO FLAG
```

(continues on next page)

(continued from previous page)

```
--  if the rule is activated as +RForbidden_Aspects:ALL,Allowed=Pack;Pre
package Foo is
   type Arr is array (1 .. 10) of Integer with Pack;   --  NOFLAG (because of 'Allowed'␣
↪rule arg)

   type T is tagged private;

   procedure Proc1 (X : in out T)
     with Pre => Predicate1;                           --  NOFLAG (because of 'Allowed'␣
↪rule arg)

   procedure Proc2 (X : in out T)
     with Pre'Class => Predicate2;                     --  FLAG
```

3.1.3.3 Forbidden_Attributes

Flag each use of the specified attributes. The attributes to be detected are named in the rule's parameters.

This rule has the following parameters for the +R option and for LKQL rule options file:

Forbidden: list[string]
: Adds the specified attributes to the set of attributes to be detected and sets the detection checks for all the specified attributes ON. If an element does not denote any attribute defined in the Ada standard or in the GNAT Reference Manual, it is treated as the name of unknown attribute. If an element is equal to `GNAT` (case insensitive), then all GNAT-specific attributes are added to the set of attributes to be detected.

Allowed: string
: A semi-colon separated list of attributes to remove from the set of attributes to be detected. You have to use the named parameter formatting to specify it.

All: bool
: If `true`, all attributes are detected; this sets the rule ON.

Parameters are not case sensitive. If an element of *Forbidden* or *Allowed* does not have the syntax of an Ada identifier and therefore can not be considered as a (part of an) attribute designator, a diagnostic message is generated and the corresponding parameter is ignored. (If an attribute allows a static expression to be a part of the attribute designator, this expression is ignored by this rule.)

The +R option with no parameters doesn't create any instance for the rule, thus, it has no effect.

Note: In LKQL rule options files, the `Allowed` parameter should be a list of strings:

```
val rules = @{
   Forbidden_Attributes: {Forbidden: ["X", "Y", "GNAT"], Allowed: ["Z"]}
}
```

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are *Attribute_Designators*. Each *Attribute_Designator* used as a rule exemption parameter should denote a predefined or GNAT-specific attribute.

Example

```
--  if the rule is activated as +RForbidden_Attributes:Range,First,Last
procedure Foo is
   type Arr is array (1 .. 10) of Integer;
   Arr_Var : Arr;

   subtype Ind is Integer range Arr'First .. Arr'Last; --  FLAG (twice)
begin

   for J in Arr'Range loop                             --  FLAG
      Arr_Var (J) := Integer'Succ (J);

--  if the rule is activated as +RForbidden_Attributes:ALL,Allowed=First,Last
procedure Foo is
   type Arr is array (1 .. 10) of Integer;
   Arr_Var : Arr;

   subtype Ind is Integer range Arr'First .. Arr'Last; --  NOFLAG (because of 'Allowed'␣
↪rule arg)
begin

   for J in Arr'Range loop                             --  FLAG
      Arr_Var (J) := Integer'Succ (J);
```

3.1.3.4 Forbidden_Pragmas

Flag each use of the specified pragmas. The pragmas to be detected are named in the rule's parameters.

This rule has the following parameters for the +R option and for LKQL rule options file:

Forbidden: list[string]
: Adds the specified pragmas to the set of pragmas to be checked and sets the checks for all the specified pragmas ON. An element of this list is treated as a name of a pragma. If it does not correspond to any pragma name defined in the Ada standard or to the name of a GNAT-specific pragma defined in the GNAT Reference Manual, it is treated as the name of unknown pragma. If an element is equal to `GNAT` (case insensitive), then all GNAT-specific pragmas are added to the set of attributes to be detected.

Allowed: string
: A semi-colon separated list of pragmas to remove from the set of pragmas to be detected. You have to use the named parameter formatting to specify it.

All: bool
: If `true`, all pragmas are detected; this sets the rule ON.

Parameters are not case sensitive. If an element of *Forbidden* or *Allowed* does not have the syntax of an Ada identifier and therefore can not be considered as a pragma name, a diagnostic message is generated and the corresponding parameter is ignored.

The +R option with no parameters doesn't create any instance for the rule, thus, it has no effect.

Note that in case when the rule is enabled with *All* parameter, then the rule will flag also pragmas `Annotate` used to exempt rules, see *Rule Exemption*. Even if you exempt this *Forbidden_Pragmas* rule then the pragma `Annotate` that closes the exemption section will be flagged as non-exempted. To avoid this, remove the pragma `Annotate` from the "to be flagged" list by using `+RForbidden_Pragmas:ALL,Allowed=Annotate` rule option.

Note: In LKQL rule options files, you can specify a named `Allowed` parameter as a list of strings. This way you can exempt some pragmas from being flagged. Example:

```
val rules = @{
   Forbidden_Pragmas: {Forbidden: ["gnat"], Allowed: ["Annotate"]}
}
```

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are pragma names. Each name used as a rule exemption parameter should denote a predefined or GNAT-specific pragma.

Example

```
--  if the rule is activated as +RForbidden_Pragmas:Pack
package Foo is

   type Arr is array (1 .. 8) of Boolean;
   pragma Pack (Arr);                        --  FLAG

   I : Integer;
   pragma Atomic (I);

end Foo;
```

3.1.3.5 Implicit_SMALL_For_Fixed_Point_Types

Flag each fixed point type declaration that lacks an explicit representation clause to define its `'Small` value. Since `'Small` can be defined only for ordinary fixed point types, decimal fixed point type declarations are not checked.

Example

```
package Foo is
   type Fraction is delta 0.01 range -1.0 .. 1.0;
   type Fraction1 is delta 0.01 range -1.0 .. 1.0; --  FLAG

   type Money is delta 0.01 digits 15;

   for Fraction'Small use 0.01;
end Foo;
```

3.1.3.6 Incomplete_Representation_Specifications

Flag all record types that have a layout representation specification but without Size and Pack representation specifications.

Example

```
package Pack is
   type Rec is record  --  FLAG
      I : Integer;
      B : Boolean;
   end record;

   for Rec use record
      I at 0 range 0 ..31;
      B at 4 range 0 .. 7;
   end record;
end Pack;
```

3.1.3.7 Membership_For_Validity

Flag membership tests that can be replaced by a `'Valid` attribute. Two forms of membership tests are flagged:

- X in Subtype_Of_X
- X in Subtype_Of_Y'Range
- X in Subtype_Of_X'First .. Subtype_Of_X'Last

where X is a data object except for a loop parameter, and `Subtype_Of_X` is the subtype of the object as given by the corresponding declaration.

Example

```
   subtype My_Int is Integer range 1 .. 10;
   X : My_Int;
   Y : Integer;
begin
   if X in My_Int then                          --  FLAG
```

3.1.3.8 No_Explicit_Real_Range

Flag a declaration of a floating point type or a decimal fixed point type, including types derived from them if no explicit range specification is provided for the type.

Example

```
type F1 is digits 8;                          --  FLAG
type F2 is delta 0.01 digits 8;               --  FLAG
```

3.1.3.9 No_Scalar_Storage_Order_Specified

Flag each record type declaration, record extension declaration, and untagged derived record type declaration if a record_representation_clause that has at least one component clause applies to it (or an ancestor), but neither the type nor any of its ancestors has an explicitly specified Scalar_Storage_Order aspect.

Example

```
with System;
package Foo is

   type Rec1 is record      --  FLAG
      I : Integer;
   end record;

   for Rec1 use record
      I at 0 range 0 .. 31;
   end record;

   type Rec2 is record      --  NO FLAG
      I : Integer;
   end record
     with Scalar_Storage_Order => System.Low_Order_First;

   for Rec2 use record
      I at 0 range 0 .. 31;
   end record;

end Foo;
```

3.1.3.10 Predefined_Numeric_Types

Flag each explicit use of the name of any numeric type or subtype declared in package `Standard`.

The rationale for this rule is to detect when the program may depend on platform-specific characteristics of the implementation of the predefined numeric types. Note that this rule is overly pessimistic; for example, a program that uses `String` indexing likely needs a variable of type `Integer`. Another example is the flagging of predefined numeric types with explicit constraints:

```
subtype My_Integer is Integer range Left .. Right;
Vy_Var : My_Integer;
```

This rule detects only numeric types and subtypes declared in package `Standard`. The use of numeric types and subtypes declared in other predefined packages (such as `System.Any_Priority` or `Ada.Text_IO.Count`) is not flagged

Example

```
package Foo is
   I : Integer;                                  -- FLAG
   F : Float;                                    -- FLAG
   B : Boolean;

   type Arr is array (1 .. 5) of Short_Float; -- FLAG

   type Res is record
      C1 : Long_Integer;                         -- FLAG
      C2 : Character;
   end record;

end Foo;
```

3.1.3.11 Printable_ASCII

Flag source code text characters that are not part of the printable ASCII character set, a line feed, or a carriage return character (i.e. values 10, 13 and 32 .. 126 of the ASCII Character set).

3.1.3.12 Separate_Numeric_Error_Handlers

Flags each exception handler that contains a choice for the predefined `Constraint_Error` exception, but does not contain the choice for the predefined `Numeric_Error` exception, or that contains the choice for `Numeric_Error`, but does not contain the choice for `Constraint_Error`.

Example

```
exception
   when Constraint_Error =>  --  FLAG
      Clean_Up;
end;
```

3.1.4 Program Structure

The rules in this subsection may be used to enforce feature usages related to program structure.

3.1.4.1 Deep_Library_Hierarchy

Flag any library package declaration, library generic package declaration or library package instantiation that has more than N parents and grandparents (that is, the name of such a library unit contains more than N dots). Child subprograms, generic subprograms subprogram instantiations and package bodies are not flagged.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximal number of ancestors when the unit is not flagged.

Example

```
package Parent.Child1.Child2 is  -- FLAG  (if rule parameter is 1)
   I : Integer;
end;
```

3.1.4.2 Deeply_Nested_Generics

Flag a generic declaration nested in another generic declaration if the nesting level of the inner generic exceeds the value specified by the *N* rule parameter. The nesting level is the number of generic declarations that enclose the given (generic) declaration. Formal packages are not flagged by this rule.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Non-negative integer specifying the maximum nesting level for a generic declaration.

Example

```
package Foo is

   generic
   package P_G_0 is
      generic
      package P_G_1 is
         generic              --  FLAG (if rule parameter is 1)
         package P_G_2 is
            I  : Integer;
         end;
      end;
   end;

end Foo;
```

3.1.4.3 Deeply_Nested_Instantiations

Flag a generic package instantiation if it contains another instantiation in its specification and this nested instantiation also contains another instantiation in its specification and so on, and the length of these nested instantiations is more than N where N is a rule parameter.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Non-negative integer specifying the maximum nesting level for instantiations.

Example

```
procedure Proc is

   generic
   procedure D;

   procedure D is
   begin
      null;
   end D;

   generic
   package C is
      procedure Inst is new D;
   end C;

   generic
   package B is
      package Inst is new C;
   end B;

   generic
   package A is
       package Inst is new B;
   end A;

   package P is
      package Inst is new A;    --  FLAG
   end P;
```

3.1.4.4 Local_Packages

Flag all local packages declared in package and generic package specs. Local packages in bodies are not flagged.

Example

```
package Foo is
   package Inner is     --  FLAG
      I : Integer;
   end Inner;
end Foo;
```

3.1.4.5 Maximum_Expression_Complexity

Flag any expression that is not directly a part of another expression which contains more than *N* expressions of the following kinds (each count for 1) as its subcomponents, *N* is a rule parameter:

- Identifiers;
- Numeric, string or character literals;
- Conditional expressions;
- Quantified expressions;
- Aggregates;
- @ symbols (target names).

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximum allowed number of expression subcomponents.

Example

```
I := 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;  --  FLAG if N < 10
I := F (I);   --  FLAG if N < 2
I := F5 (1 + 2 + 3 + 4 + 5, 2, 3, 4, 5);   --  FLAG (twice) if N < 5
```

3.1.4.6 Maximum_Lines

Flags the file containing the source text of a compilation unit if this file contains more than N lines where N is a rule parameter

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximum allowed number of lines in the compilation unit source text.

3.1.4.7 Maximum_Subprogram_Lines

Flag handled sequences of statements of subprogram bodies exceeding *N* textual lines (*N* is the rule parameter). Lines are counted from the beginning of the first to the end of the last statement, including blank and comment lines

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximum allowed number of lines in the subprogram statement sequence.

Example

```
--  If the rule parameter is 3
procedure P (I : in out Integer) is
begin
   I := I + 1;    --  FLAG
   I := I + 2;
   I := I + 3;
   I := I + 4;
end P;
```

3.1.4.8 Non_Visible_Exceptions

Flag constructs leading to the possibility of propagating an exception out of the scope in which the exception is declared. Two cases are detected:

- An exception declaration located immediately within a subprogram body, task body or block statement is flagged if the body or statement does not contain a handler for that exception or a handler with an `others` choice.
- A `raise` statement in an exception handler of a subprogram body, task body or block statement is flagged if it (re)raises a locally declared exception. This may occur under the following circumstances: * it explicitly raises a locally declared exception, or
  - it does not specify an exception name (i.e., it is simply `raise;`) and the enclosing handler contains a locally declared exception in its exception choices.

Renamings of local exceptions are not flagged.

Example

```
procedure Bar is
   Var : Integer :=- 13;

   procedure Inner (I : in out Integer) is
      Inner_Exception_1 : exception;          --  FLAG
      Inner_Exception_2 : exception;
   begin
      if I = 0 then
         raise Inner_Exception_1;
      elsif I = 1 then
         raise Inner_Exception_2;
      else
         I := I - 1;
      end if;
   exception
      when Inner_Exception_2 =>
         I := 0;
         raise;                               --  FLAG
   end Inner;

begin
   Inner (Var);
end Bar;
```

3.1.4.9 One_Tagged_Type_Per_Package

Flag all package declarations with more than one tagged type declaration in the visible part.

Example

```
package P is  --  FLAG

   type T is tagged null record;
   type T2 is tagged null record;

end P;
```

3.1.4.10 Outside_References_From_Subprograms

Within a subprogram body or an expression function flag any identifier that denotes a non global data object declared outside this body.

This rule analyzes generic instantiations and ignores generic packages to avoid flagging all references to formal objects.

Example

```
procedure Enclosing is
   Var : Integer;

   procedure Proc (I : in out Integer) is
   begin
      I := I + Var;       --  FLAG
```

3.1.4.11 Raising_External_Exceptions

Flag any `raise` statement, in a program unit declared in a library package or in a generic library package, for an exception that is neither a predefined exception nor an exception that is also declared (or renamed) in the visible part of the package.

Example

```
package Exception_Declarations is
   Ex : exception;
end Exception_Declarations;
package Foo is
   procedure Proc (I : in out Integer);
end Foo;
with Exception_Declarations;
package body Foo is
   procedure Proc (I : in out Integer) is
   begin
      if I < 0 then
```

(continues on next page)

(continued from previous page)

```
         raise Exception_Declarations.Ex;    --  FLAG
      else
         I := I - 1;
      end if;
   end Proc;
end Foo;
```

3.1.4.12 Same_Instantiations

Flag each generic package instantiation when it can be determined that a set of the `gnatcheck` argument sources contains another instantiation of the same generic with the same actual parameters. This determination is conservative, it checks currently for the following matching parameters:

- integer, character, and string literals;
- Ada names that denote the same entity.

Generic packages that have no parameters are ignored.

If some instantiation is marked by the rule, additional investigation is required to decide if one of the duplicated instantiations can be removed to simplify the code. In particular, the rule does not check if these instantiations declare any global variable or perform some non-trivial actions as a part of their elaboration.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Library_Level_Only: bool
If `true`, only check library level instantiations.

Example

```
generic
   type T is private;
   X : Integer;
package Gen is
end Gen;

with Gen;

package Inst1 is
   package Inst_1 is new Gen (Integer, 2);  --  FLAG
   package Inst_2 is new Gen (Integer, 3);  --  NO FLAG
end Inst1;

with Gen;

package Inst2 is
   package Inst_3 is new Gen (Integer, 2);  --  FLAG
end Inst2;
```

3.1.4.13 Too_Many_Generic_Dependencies

Flags a `with` clause that mentions a generic unit that in turn directly depends (mentions in its `with` clause) on another generic unit, and so on, and the length of the chain of these dependencies on generics is more than N where N is a rule parameter.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Non-negative integer specifying the maximal allowed length of the chain of dependencies on generic units.

Example

```
generic
package D is
end D;

with D;
generic
package C is
end C;

with C;
generic
package B is
end B;

with B;
generic
package A is
end A;

with A;         --  FLAG (if N <= 3)
package P is
   procedure Proc;
end P;
```

3.1.5 Programming Practice

The rules in this subsection may be used to enforce feature usages that relate to program maintainability.

3.1.5.1 Access_To_Local_Objects

Flag any `'Access` attribute reference if its prefix denotes an identifier defined by a local object declaration or a subcomponent thereof. An object declaration is considered as local if it is located anywhere except library-level packages or bodies of library-level packages (including packages nested in those). Here both package declarations and package instantiations are considered as packages. If the attribute prefix is a dereference or a subcomponent thereof, the attribute reference is not flagged.

Example

```
package body Pack
   procedure Proc is
      type Int_A is access all Integer;
      Var1 : aliased Integer;
      Var2 :         Int_A := Var1'Access;  --  FLAG
```

3.1.5.2 Actual_Parameters

Flag situations when a specific actual parameter is passed for a specific formal parameter in the call to a specific subprogram. Subprograms, formal parameters and actual parameters to check are specified by the rule parameters.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Forbidden: list[string]
: A list of strings formatted as following: `subprogram:formal:actual` where `subprogram` should be a full expanded Ada name of a subprogram, `formal` should be an identifier, it is treated as the name of a formal parameter of the `subprogram` and `actual` should be a full expanded Ada name of a function or a data object declared by object declaration, number declaration, parameter specification, generic object declaration or object renaming declaration.

Note: In LKQL rule options files, the `Forbidden` parameter should be a list of three-elements tuples. Mapping `subprogram:formal:actual` to `(<subprogram>, <formal>, <actual>)`. For example:

```
val rules = @{
   Actual_Parameters: {Forbidden: [("P.SubP", "Param", "Value")]}
}
```

For all the calls to `subprogram` the rule checks if the called subprogram has a formal parameter named as `formal`, and if it does, it checks if the actual for this parameter is either a call to a function denoted by `actual` or a reference to the data object denoted by `actual` or one of the above in parenthesis, or a type conversion or a qualified expression applied to one of the above. References to object components or explicit dereferences are not checked.

Be aware that the rule does not follow renamings. The rule checks only calls that use the `subprogram` part of the rule parameter as a called name, and if this name is declared by a subprogram renaming, the rule does not pay attention to the calls that use subprogram name being renamed. When looking for the parameter to check, the rule assumes that a formal parameter denoted by the `formal` part of the rule parameter is declared as a part of the declaration of `subprogram`. The same for the `actual` part of the rule parameter - only those actual parameters that use `actual` as the name of a called function are considered. This is a user responsibility to provide as the rule parameters all needed combinations of subprogram name and formal parameter name for the subprogram of interest in case if renamings are used for the subprogram, and all possible aliases if renaming is used for a function of interest if its calls may be used as actuals.

Note also, that the rule does not make any overload resolution, so it will consider all possible subprograms denoted by the `subprogram` part of the rule parameter, and all possible function denoted by the `actual` part.

Example

```
-- Suppose the rule parameter is P.Proc:Par2:Q.Var
package P is
   procedure Proc (B : Boolean; I : Integer);
   procedure Proc (Par1 : Character; Par2 : Integer);
end P;

package Q is
   Var : Integer;
end Q;

with P; use P;
with Q; use Q;
procedure Main is
begin
   Proc (True, Var);    -- NO FLAG
   Proc (1, Var);       -- FLAG
```

3.1.5.3 Ada05_Formal_Packages

Flag formal package declarations that are not allowed in Ada 95. Ada 95 allows the box symbol (<>) to be used alone as a whole formal package actual part only.

Example

```
generic
   with package NP is new P (T => <>);  --  FLAG
package Pack_G is
```

3.1.5.4 Ada_2022_In_Ghost_Code

Flag usages of Ada 2022 specific constructions used outside of Ghost code and Assertion code.

This check is meant to allow users to use the new standard in code that is not shipped with the final executable version of their application.

You can check this page https://learn.adacore.com/courses/whats-new-in-ada-2022/index.html for a quick overview of the new features of Ada 2022.

Example

```
procedure Test_Ghost_Code is
   A : String := "hello";

   B : String := A'Image; --  FLAG

   procedure Foo
      with Pre => A'Image = "hello";  --  NOFLAG
```

(continues on next page)

(continued from previous page)

```
   B : String := A'Image with Ghost;  --  NOFLAG

   function Bar return String is (A'Image) with Ghost;  --  NOFLAG

   package P with Ghost is
      B : String := A'Image; --  NOFLAG
   end P;

   generic
   package Gen_Pkg is
      B : String := A'Image;  --  FLAG (via instantiation line 23)
   end Gen_Pkg;

   package Inst is new Gen_Pkg;
begin
   null;
end Test_Ghost_Code;
```

3.1.5.5 Address_Attribute_For_Non_Volatile_Objects

Flag any 'Address attribute reference if its prefix denotes a data object defined by a variable object declaration and this object is not marked as Volatile. An entity is considered as being marked volatile if it has an aspect Volatile, Atomic or Shared declared for it.

Example

```
Var1 : Integer with Volatile;
Var2 : Integer;

X : Integer with Address => Var1'Address;
Y : Integer with Address => Var2'Address;   --  FLAG
```

3.1.5.6 Address_Specifications_For_Initialized_Objects

Flag address clauses and address aspect definitions if they are applied to object declarations with explicit initializations.

Example

```
I : Integer := 0;
Var0 : Integer with Address => I'Address;

Var1 : Integer := 10;
for Var1'Address use Var0'Address;              --  FLAG
```

3.1.5.7 Address_Specifications_For_Local_Objects

Flag address clauses and address aspect definitions if they are applied to data objects declared in local subprogram bodies. Data objects declared in library subprogram bodies are not flagged.

Example

```
package Pack is
   Var : Integer;
   procedure Proc (I : in out Integer);
end Pack;
package body Pack is
   procedure Proc (I : in out Integer) is
      Tmp : Integer with Address => Pack.Var'Address;    --  FLAG
   begin
      I := Tmp;
   end Proc;
end Pack;
```

3.1.5.8 Anonymous_Arrays

Flag all anonymous array type definitions (by Ada semantics these can only occur in object declarations).

Example

```
type Arr is array (1 .. 10) of Integer;
Var1 : Arr;
Var2 : array (1 .. 10) of Integer;       --  FLAG
```

3.1.5.9 Binary_Case_Statements

Flag a case statement if this statement has only two alternatives, one containing exactly one choice, the other containing exactly one choice or the `others` choice.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Except_Enums: bool
If `true`, do not flag case statements whose selecting expression is of an enumeration type.

Example

```
case Var is                    --  FLAG
   when 1 =>
      Var := Var + 1;
   when others =>
      null;
end case;
```

3.1.5.10 Boolean_Negations

Flag any infix call to the predefined NOT operator for the predefined Boolean type if its argument is an infix call to a predefined relation operator or another call to the predefined NOT operator. Such expressions can be simplified by excluding the outer call to the predefined NOT operator. Calls to NOT operators for the types derived from Standard.Boolean are not flagged.

Example

```
Is_Data_Available := not (Buffer_Length = 0);    --  FLAG
```

3.1.5.11 Calls_In_Exception_Handlers

Flag an exception handler if its sequence of statements contains a call to one of the subprograms specified as a rule parameter.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Subprograms: list[string]
A list of full expanded Ada name of subprograms.

Note that if a rule parameter does not denote the name of an existing subprogram, the parameter itself is (silently) ignored and does not have any effect except for turning the rule ON.

Be aware that the rule does not follow renamings. So if a subprogram name specified as a rule parameter denotes the name declared by subprogram renaming, the rule will flag only exception handlers that calls this subprogram using this name and does not respect and will pay no attention to the calls that use original subprogram name, and the other way around. This is a user responsibility to provide as the rule parameters all needed subprogram names the subprogram of interest in case if renamings are used for this subprogram.

Note also, that the rule does not make any overload resolution, so if a rule parameter refers to more than one overloaded subprograms, the rule will treat calls to all these subprograms as the calls to the same subprogram.

Example

```
-- Suppose the rule parameter is P.Unsafe
package P is
   procedure Safe;
   procedure Unsafe;
end P;

with P; use P;
procedure Proc is
begin
   ...
exception
   when Constraint_Error =>    --  NO FLAG
      Safe;
   when others =>              --  FLAG
      Unsafe;
end Proc;
```

3.1.5.12 Calls_Outside_Elaboration

Flag subprogram calls outside library package elaboration code. Only calls to the subprograms specified as a rule parameter are considered, renamings are not followed.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Forbidden: list[string]
A list of full expanded Ada name of subprograms.

Note that if a rule parameter does not denote the name of an existing subprogram, the parameter itself is (silently) ignored and does not have any effect except for turning the rule ON.

Note also, that the rule does not make any overload resolution, so if a rule parameter refers to more than one overloaded subprograms, the rule will treat calls to all these subprograms as the calls to the same subprogram.

Example

```
--  Suppose the rule is activated as +RCalls_Outside_Elaboration:P.Fun;
package P is
   I : Integer := Fun (1);          --  NO FLAG
   J : Integer;

   procedure Proc (I : in out Integer);
end P;

package body P is
   procedure Proc (I : in out Integer) is
   begin
      I := Another_Fun (Fun (1));   --  FLAG
   end Proc;
begin
   J := Fun (I);                    --  NO FLAG
```

3.1.5.13 Concurrent_Interfaces

Flag synchronized, task, and protected interfaces.

Example

```
type Queue is limited interface;                                --  NO FLAG
type Synchronized_Queue is synchronized interface and Queue;    --  FLAG
type Synchronized_Task is task interface;                       --  FLAG
type Synchronized_Protected is protected interface;             --  FLAG
```

3.1.5.14 Constant_Overlays

Flag an overlay definition that has a form of an attribute definition clause `for Overlaying'Address use Overlaid'Address;` or a form of aspect definition `Address => Overlaid'Address`, and `Overlaid` is a data object defined by a constant declaration or a formal or generic formal parameter of mode `IN` if at least one of the following is true:

- the overlaying object is not a constant object;
- overlaying object or overlaid object is marked as Volatile;

Example

```
C : constant Integer := 1;
V : Integer;
for V'Address use C'Address;     --  FLAG
```

3.1.5.15 Default_Values_For_Record_Components

Flag a record component declaration if it contains a default expression. Do not flag record component declarations in protected definitions. Do not flag discriminant specifications.

Example

```
type Rec (D : Natural := 0) is record
   I : Integer := 0;                    -- FLAG
   B : Boolean;

   case D is
      when 0 =>
         C : Character := 'A';          -- FLAG
      when others =>
         F : Float;
   end case;
end record;
```

3.1.5.16 Deriving_From_Predefined_Type

Flag derived type declaration if the ultimate ancestor type is a predefined Ada type. Do not flag record extensions and private extensions. The rule is checked inside expanded generics.

Example

```
package Foo is
   type T is private;
   type My_String is new String;  --  FLAG
private
   type T is new Integer;         --  FLAG
end Foo;
```

3.1.5.17 Direct_Equalities

Flag infix calls to the predefined = and /= operators when one of the operands is a name of a data object provided as a rule parameter.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Actuals: list[string]
: A list of full expanded Ada name of a data objects declared by object declaration, number declaration, parameter specification, generic object declaration or object renaming declaration. Any other parameter does not have any effect except of turning the rule ON.

Be aware that the rule does not follow renamings. It checks if an operand of an (un)equality operator is exactly the name provided as rule parameter (the short name is checked in case of expanded name given as (un)equality operator), and that this name is given on its own, but not as a component of some other expression or as a call parameter.

Example

```
-- suppose the rule parameter is P.Var
package P is
   Var : Integer;
end P;

with P; use P;
procedure Proc (I : in out Integer) is
begin
   if Var = I then    --  FLAG
      I := 0;
   end if;
end Proc;
```

3.1.5.18 Duplicate_Branches

Flag a sequence of statements that is a component of an `if` statement or of a `case` statement alternative, if the same `if` or `case` statement contains another sequence of statements as its component (or a component of its `case` statement alternative) that is syntactically equivalent to the sequence of statements in question. The check for syntactical equivalence of operands ignores line breaks, white spaces and comments.

Small sequences of statements are not flagged by this rule. The rule has two optional parameters that allow to specify the maximal size of statement sequences that are not flagged:

- ***min_stmt: int***
  An integer literal. All statement sequences that contain more than *min_stmt* statements (*Stmt* as defined in Libadalang) as subcomponents are flagged;

- ***min_size: int***
  An integer literal. All statement sequences that contain more than *min_size* lexical elements (*SingleTokNode* in Libadalang terms) are flagged.

You have to use the `param_name=value` formatting to pass arguments through the `+R` options. Example: `+RDuplicate_Branches:min_stmt=20,min_size=42`.

If at least one of the two thresholds specified by the rule parameters is exceeded, a statement sequence is flagged. The following defaults are used: `min_stmt=4,min_size=14`.

Example

```
if X > 0 then
   declare       --  FLAG: code duplicated at line 11
      A : Integer := X;
      B : Integer := A + 1;
      C : Integer := B + 1;
      D : Integer := C + 1;
   begin
      return D;
   end;
else
   declare
      A : Integer := X;
      B : Integer := A + 1;
      C : Integer := B + 1;
      D : Integer := C + 1;
   begin
      return D;
   end;
end if;
```

3.1.5.19 Enumeration_Ranges_In_CASE_Statements

Flag each use of a range of enumeration literals as a choice in a `case` statement. All forms for specifying a range (explicit ranges such as `A .. B`, subtype marks and `'Range` attributes) are flagged. An enumeration range is flagged even if contains exactly one enumeration value or no values at all. A type derived from an enumeration type is considered as an enumeration type.

This rule helps prevent maintenance problems arising from adding an enumeration value to a type and having it implicitly handled by an existing `case` statement with an enumeration range that includes the new literal.

Example

```
procedure Bar (I : in out Integer) is
   type Enum is (A, B, C, D, E);
   type Arr is array (A .. C) of Integer;

   function F (J : Integer) return Enum is separate;
begin
   case F (I) is
```

(continues on next page)

(continued from previous page)

```
      when Arr'Range  =>  --  FLAG
         I := I + 1;
      when D .. E =>      --  FLAG
         null;
   end case;
end Bar;
```

3.1.5.20 Exception_Propagation_From_Callbacks

Flag an `'Address` or `'Access` attribute if:

- this attribute is a reference to a subprogram;
- this subprogram may propagate an exception;
- this attribute is an actual parameter of a subprogram call, and both the subprogram called and the corresponding formal parameter are specified by a rule parameter.

A subprogram is considered as not propagating an exception if:

- its body has an exception handler with `others` exception choice;
- no exception handler in the body contains a raise statement nor a call to `Ada.Exception.Raise_Exception` or `Ada.Exception.Reraise_Occurrence`.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Callbacks: list[string]
: A list of strings which should have the following structure `subprogram_name.parameter`. `subprogram_name` should be a full expanded Ada name of a subprogram. `parameter` should be a simple name of a parameter of a subprogram defined by the `subprogram_name` part of the rule parameter. For such a rule parameter for calls to all the subprograms named as `subprogram_name` the rule checks if a reference to a subprogram that may propagate an exception is passed as an actual for parameter named `parameter`.

Note: In LKQL rule options files, the `Callbacks` parameter should be a list of two-elements tuples. Mapping `subprogram_name.parameter` to `(<subprogram_name>, <parameter>)`. For example:

```
val rules = @{
   Exception_Propagation_From_Callbacks: {Forbidden: [("P.SubP", "Param")]}
}
```

Note that if a rule parameter does not denote the name of an existing subprogram or if its `parameter` part does not correspond to any formal parameter of any subprogram defined by `subprogram_name` part, the parameter itself is (silently) ignored and does not have any effect except for turning the rule ON.

Be aware that `subprogram_name` is the name used in subprogram calls to look for callback parameters that may raise an exception, and `parameter` is the name of a formal parameter that is defined in the declaration that defines `subprogram_name`. This is a user responsibility to provide as the rule parameters all needed combinations of subprogram name and parameter name for the subprogram of interest in case if renamings are used for this subprogram.

Note also, that the rule does not make any overload resolution, so calls to all the subprograms corresponding to `subprogram_name` are checked.

Note: Note that you can use both fully qualified names to instantiated or non-instantiated generic subprograms, depending on the granularity you wish for. However **you cannot use a mix of the two**, so the names need to be either

fully instantiated or fully uninstantiated.

Example

```
-- Suppose the rule parameter is P.Take_CB.Param1
package P is
   procedure Good_CB; --  does not propagate an exception
   procedure Bad_CB;  --  may propagate an exception
   procedure Take_CB
     (I : Integer;
      Param1 : access procedure;
      Param2 : access procedure);
end P;

with P; use P;
procedure Proc is
begin
   Take_CB (1, Bad_CB'Access, Good_CB'Access);    --  FLAG
   Take_CB (1, Good_CB'Access, Bad_CB'Access);    --  NO FLAG
end Proc;
```

3.1.5.21 Exception_Propagation_From_Export

Flag a subprogram body if aspect or pragma `Export` or `Convention` is applied to this subprogram and this subprogram may propagate an exception.

A subprogram is considered as not propagating an exception if:

- its body has an exception handler with `others` exception choice;
- no exception handler in the body contains a raise statement nor a call to `Ada.Exception.Raise_Exception` or `Ada.Exception.Reraise_Occurrence`.

Example

```
package P is
   procedure Proc (I : in out Integer) with Export;
end P;

package body P is
   procedure Proc (I : in out Integer) is    --  FLAG
   begin
      I := I + 10;
   end Proc;
end P;
```

3.1.5.22 Exception_Propagation_From_Tasks

Flag a task body if it does not contain and exception handler with `others` exception choice or if it contains an exception handler with a raise statement or a call to `Ada.Exception.Raise_Exception` or `Ada.Exception.Reraise_Occurrence`.

Example

```
task T;

task body T is   --  FLAG
begin
   ...
exception
   when Constraint_Error => null;
end T;
```

3.1.5.23 Exceptions_As_Control_Flow

Flag each place where an exception is explicitly raised and handled in the same subprogram body. A `raise` statement in an exception handler, package body, task body or entry body is not flagged.

Example

```
procedure Bar (I : in out Integer) is

begin
   if I = Integer'Last then
      raise Constraint_Error;    --  FLAG
   else
     I := I - 1;
   end if;
exception
   when Constraint_Error =>
      I := Integer'First;
end Bar;
```

3.1.5.24 EXIT_Statements_With_No_Loop_Name

Flag each `exit` statement that does not specify the name of the loop being exited.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Nested_Only: bool
If `true`, flag only those exit statements with no loop name that exit from nested loops.

Example

```
procedure Bar (I, J : in out Integer) is
begin
   loop
      exit when I < J;  --  FLAG
      I := I - 1;
      J := J + 1;
   end loop;
end Bar;
```

3.1.5.25 Exits_From_Conditional_Loops

Flag any exit statement if it transfers the control out of a `for` loop or a `while` loop. This includes cases when the `exit` statement applies to a `for` or `while` loop, and cases when it is enclosed in some `for` or `while` loop, but transfers the control from some outer (unconditional) `loop` statement.

Example

```
function Bar (S : String) return Natural is
   Result : Natural := 0;
begin
   for J in S'Range loop
      exit when S (J) = '@';  --  FLAG
      Result := Result + J;
   end loop;

   return 0;
end Bar;
```

3.1.5.26 Final_Package

Check that package declarations annotated as final don't have child packages

---

Note: We don't do a transitive check, so grandchild packages won't be flagged. We consider this is not necessary, because the child package will be flagged anyway.

Here is an example:

```
package Pkg with Annotate => (GNATcheck, Final) is
end Pkg;

package Pkg.Child is -- FLAG
end Pkg.Child;

package Pkg.Child.Grandchild is -- NOFLAG
end Pkg.Child.Grandchild;
```

3.1.5.27 Function_OUT_Parameters

Flag any function declaration, function body declaration, expression function declaration, function body stub, or generic function declaration which has at least one formal parameter of mode `out` or `in out`.

A function body declaration or function body stub is only flagged if there is no separate declaration for this function.

Example

```
function F_1 (I : Integer) return Integer;
function F_2 (I : out Integer) return Integer;         --  FLAG
function F_3 (I : in out Integer) return Integer;      --  FLAG
function F_4 (I : in out Integer) return Integer is    --  FLAG
  (I + 42);

function F_2 (I : out Integer) return Integer is       --  NOFLAG (declaration has␣
↪already been flagged)
begin
   return 0;
end F_2;
```

3.1.5.28 Global_Variables

Flag any variable declaration that appears immediately within the specification of a library package or library generic package. Variable declarations in nested packages and inside package instantiations are not flagged.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Only_Public: bool
: If `true`, do not flag variable declarations in private library (generic) packages and in package private parts.

Example

```
package Foo is
    Var1 : Integer;     --  FLAG
    procedure Proc;
private
    Var2 : Boolean;     --  FLAG
end Foo;
```

3.1.5.29 GOTO_Statements

Flag each occurrence of a `goto` statement.

This rule has the following optional parameter for the +R option and for LKQL rule options files:

Only_Unconditional: bool
: If `true`, Only flag unconditional goto statements, that is, goto statements that are not directly enclosed in an if or a case statement.

Example

```
for K in 1 .. 10 loop
   if K = 6 then
      goto Quit; -- FLAG only if Only_Unconditional is false
   end if;
   null;
end loop;
goto Next; -- FLAG in any case
<<Quit>>

<<Next>>
null;
return;
```

3.1.5.30 Improper_Returns

Flag each explicit `return` statement in procedures, and multiple `return` statements in functions. Diagnostic messages are generated for all `return` statements in a procedure (thus each procedure must be written so that it returns implicitly at the end of its statement part), and for all `return` statements in a function after the first one. This rule supports the stylistic convention that each subprogram should have no more than one point of normal return.

Example

```
procedure Proc (I : in out Integer) is
begin
   if I = 0 then
      return;                          --  FLAG
   end if;

   I := I * (I + 1);
end Proc;

function Factorial (I : Natural) return Positive is
begin
   if I = 0 then
      return 1;
   else
      return I * Factorial (I - 1);    --  FLAG
   end if;
exception
   when Constraint_Error =>
      return Natural'Last;             --  FLAG
end Factorial;
```

3.1.5.31 Integer_Types_As_Enum

Flag each integer type declaration (including types derived from integer types) if this integer type may benefit from being replaced by an enumeration type. An integer type is considered as being potentially replaceable by an enumeration type if all the following conditions are true:

- there is no infix calls to any arithmetic or bitwise operator for objects of this type;
- this type is not referenced in an actual parameter of a generics instantiation;
- there is no type conversion from or to this type;
- no type is derived from this type;
- no subtype is declared for this type.

Example

```
procedure Proc is
   type Enum is range 1 .. 3;     --  FLAG
   type Int is range 1 .. 3;      --  NO FLAG

   X : Enum := 1;
   Y : Int := 1;
begin
   X := 2;
   Y := Y + 1;
end Proc;
```

3.1.5.32 Local_Instantiations

Non library-level generic instantiations are flagged.

The rule has an optional parameter(s) for the +R option and for LKQL rule options files:

Packages: list[string]
A list of fully expanded Ada names of generic units to flag local instantiations of.

If the rule is activated without parameters, all local instantiations are flagged, otherwise only instantiations of the generic units which names are listed as rule parameters are flagged. Note that a rule parameter should be a generic unit name but not the name defined by generic renaming declaration. Note also, that if a rule parameter does not denote an existing generic unit or if it denotes a name defined by generic renaming declaration, the parameter itself is (silently) ignored and does not have any effect, but the presence of at least one of such a parameter already means that the rule will not flag any instantiation if the full expanded Ada name of the instantiated generic unit is listed as a rule parameter.

Example

```
generic
package Pack_G is
   I : Integer;
end Pack_G;

with Pack_G;
package Pack_I is new Pack_G;    --  NO FLAG
```

(continues on next page)

(continued from previous page)

```
with Pack_G;
procedure Proc is
   package Inst is new Pack_G;  --  FLAG
begin
   ...
```

3.1.5.33 Local_USE_Clauses

Use clauses that are not parts of compilation unit context clause are flagged.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Except_USE_TYPE_Clauses: bool
If `true`, do not flag local use type clauses.

Example

```
with Pack1;
with Pack2;
procedure Proc is
   use Pack1;                --  FLAG

   procedure Inner is
      use type Pack2.T;      --  FLAG (if Except_USE_TYPE_Clauses is not set)
   ...
```

3.1.5.34 Maximum_OUT_Parameters

Flag any subprogram declaration, subprogram body declaration, expression function declaration, null procedure declaration, subprogram body stub or generic subprogram declaration if the corresponding subprogram has more than *N* formal parameters of mode `out` or `in out`, where *N* is a parameter of the rule.

A subprogram body, an expression function, a null procedure or a subprogram body stub is flagged only if there is no separate declaration for this subprogram. Subprogram renaming declarations and subprogram instantiations, as well as declarations inside expanded generic instantiations are never flagged.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximum allowed total number of subprogram formal parameters of modes `out` and `in out`.

Example

```
procedure Proc_1 (I : in out Integer);           --  NO FLAG
procedure Proc_2 (I, J : in out Integer);        --  NO FLAG
procedure Proc_3 (I, J, K : in out Integer);     --  NO FLAG
procedure Proc_4 (I, J, K, L : in out Integer); --  FLAG (if rule parameter is 3)
```

3.1.5.35 Maximum_Parameters

Flag any subprogram declaration, subprogram body declaration, expression function declaration, null procedure declaration, subprogram body stub or generic subprogram declaration if the corresponding subprogram has more than *N* formal parameters, where *N* is a parameter of the rule.

A subprogram body, an expression function, a null procedure or a subprogram body stub is flagged only if there is no separate declaration for this subprogram. Subprogram renaming declarations and subprogram instantiations, as well as declarations inside expanded generic instantiations are never flagged.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximum allowed total number of subprogram formal parameters.

Example

```
package Foo is

   procedure Proc_1 (I : in out Integer);
   procedure Proc_2 (I, J : in out Integer);
   procedure Proc_3 (I, J, K : in out Integer);
   procedure Proc_4 (I, J, K, L : in out Integer); --  FLAG (if rule parameter is 3)

   function Fun_4                                  --  FLAG (if rule parameter is 3)
     (I : Integer;
      J : Integer;
      K : Integer;
      L : Integer) return Integer is (I + J * K - L);

end Foo;
```

3.1.5.36 Misplaced_Representation_Items

Flag a representation item if there is any Ada construct except another representation item for the same entity between this clause and the declaration of the entity it applies to. A representation item in the context of this rule is either a representation clause or one of the following representation pragmas:

- Atomic J.15.8(9/3)
- Atomic_Components J.15.8(9/3)
- Independent J.15.8(9/3)
- Independent_Components J.15.8(9/3)
- Pack J.15.3(1/3)

- Unchecked_Union J.15.6(1/3)
- Volatile J.15.8(9/3)
- Volatile_Components J.15.8(9/3)

Example

```
type Int1 is range 0 .. 1024;
type Int2 is range 0 .. 1024;

for Int2'Size use 16;          --  NO FLAG
for Int1'Size use 16;          --  FLAG
```

3.1.5.37 Nested_Paths

Flag the beginning of a sequence of statements that is immediately enclosed by an IF statement if this sequence of statement can be moved outside the enclosing IF statement. The beginning of a sequence of statements is flagged if:

- The enclosing IF statement contains IF and ELSE paths and no ELSIF path;
- This sequence of statements does not end with a breaking statement but the sequence of statement in another path does end with a breaking statement.

  OR

  This sequence of statements is in the ELSE path, ends with a breaking statement, and the IF path also ends with a breaking statement of a different kind from the one in the ELSE path (this is done to preserve potential "if-else" symmetry).

A breaking statement is either a raise statement, or a return statement, or an unconditional exit statement, or a goto statement or a block statement without an exception handler with the enclosed sequence of statements that ends with some breaking statement.

Example

```
loop
   if I > K then
      K := K + I;    --  FLAG
      I := I + 1;
   else
      L := 10;
      exit;
   end if;
end loop;

if Condition then
   return 1;         --  NOFLAG
else
   return 2;         --  NOFLAG
end if;

if Condition then
```

(continues on next page)

(continued from previous page)

```
   return 1;
else
   goto <<label>>;  --  FLAG
end if;
```

3.1.5.38 Nested_Subprograms

Flag any subprogram declaration, subprogram body declaration, subprogram instantiation, expression function declaration or subprogram body stub that is not a completion of another subprogram declaration and that is declared within subprogram body (including bodies of generic subprograms), task body or entry body directly or indirectly (that is - inside a local nested package). Protected subprograms are not flagged. Null procedure declarations are not flagged. Procedure declarations completed by null procedure declarations are not flagged.

Example

```
procedure Bar (I, J : in out Integer) is

   procedure Foo (K : Integer) is null;
   procedure Proc1;                      --  FLAG

   procedure Proc2 is separate;          --  FLAG

   procedure Proc1 is
   begin
      I := I + J;
   end Proc1;

begin
```

3.1.5.39 No_Closing_Names

Flag any program unit that is longer than N lines where N is a rule parameter and does not repeat its name after the trailing `END` keyword.

This rule has the following (mandatory) parameter for the `+R` option and for LKQL rule options files:

N: int
: Positive integer specifying the maximal allowed number of lines in the program unit that allows not to repeat the unit name at the end.

Example

```
procedure Proc (I : in out Integer) is -- FLAG is rule parameter is 3 or less
begin
   I := I + 1;
end;
```

3.1.5.40 No_Others_In_Exception_Handlers

Flag handled sequences of statements that do not contain exception handler with `others`, depending on the rule parameter(s) specified.

This rule has three parameters for the +R option and for LKQL rule options files:

Subprogram: bool
: If `true`, flag a subprogram body if the handled sequence of statements of this body does not contain an exception handler with `others` choice. This includes the case when the body does not contain any exception handler at all. The diagnostic message points to the beginning of the subprogram body.

Task: bool
: If `true`, flag a task body if the handled sequence of statements of this body does not contain an exception handler with `others` choice. This includes the case when the body does not contain any exception handler at all. The diagnostic message points to the beginning of the task body.

All_Handlers: bool
: If `true`, flag a handled sequence of statements if it does contain at least one exception handler, but it does not contain an exception handler with `others` choice. If a handled sequence of statements does not have any exception handler, nothing is flagged for it. The diagnostic message points to the `EXCEPTION` keyword in the handled sequence of statements.

At least one parameter should be specified for the rule. If more than one parameter is specified, each of the specified parameters has its effect.

Example

```
procedure Other (I, J : in out Integer) is
begin
   begin
      I := I + 1;
   exception                 --  FLAG (if All_Handlers parameter is set)
      when Constraint_Error => null;
   end;

exception                     --  NO FLAG
   when Constraint_Error =>
      I := Integer'Last;
   when others =>
      I := J;
      raise;
end Other;
```

3.1.5.41 Non_Component_In_Barriers

Flag a barrier condition expression in an entry body declaration if this expression contains a reference to a data object that is not a (sub)component of the enclosing record the entry belongs to.

Example

```
protected Obj is
   entry E1;
   entry E2;
private
   Value  : Integer;
   Is_Set : Boolean := False;
end Obj;

Global_Bool : Boolean := False;

protected body Obj is

   entry E1
      when Is_Set and then Value > 0 is  --  NO FLAG
   begin
      Value  := Value - 1;
      Is_Set := False;
   end E1;

   entry E2
      when Global_Bool is                --  FLAG
   begin
      Is_Set := True;
   end E2;

end Obj;
```

3.1.5.42 Non_Constant_Overlays

Flag an overlay definition that has a form of an attribute definition clause `for Overlaying'Address use Overlaid'Address;` or a form of aspect definition `Address => Overlaid'Address`, and `Overlaid` is a data object defined by a variable declaration , a formal parameter of mode `IN OUT` or `OUT` or a generic formal parameter of mode `IN OUT` if at least one of the following is true:

- the overlaying object is a constant object;
- overlaying object is not marked as Volatile;
- if overlaid object is not a parameter, it is not marked as Volatile;

Example

```
V : Integer with Volatile;
C : constant Integer := 1;
for C'Address use V'Address;     --  FLAG
```

3.1.5.43 Non_Short_Circuit_Operators

Flag all calls to predefined `and` and `or` operators for any boolean type. Calls to user-defined `and` and `or` and to operators defined by renaming declarations are not flagged. Calls to predefined `and` and `or` operators for modular types or boolean array types are not flagged.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Except_Assertions: bool
: If `true`, do not flag the use of non-short-circuit_operators inside assertion-related pragmas or aspect specifications.

A pragma or an aspect is considered as assertion-related if its name is from the following list:

- `Assert`
- `Assert_And_Cut`
- `Assume`
- `Contract_Cases`
- `Debug`
- `Default_Initial_Condition`
- `Dynamic_Predicate`
- `Invariant`
- `Loop_Invariant`
- `Loop_Variant`
- `Post`
- `Postcondition`
- `Pre`
- `Precondition`
- `Predicate`
- `Predicate_Failure`
- `Refined_Post`
- `Static_Predicate`
- `Type_Invariant`

Example

```
B1 := I > 0 and J > 0;        --  FLAG
B2 := I < 0 and then J < 0;
B3 := I > J or J > 0;         --  FLAG
B4 := I < J or else I < 0;
```

3.1.5.44 Nonoverlay_Address_Specifications

Flag an attribute definition clause that defines `'Address` attribute if it does not have the form `for Overlaying'Address use Overlaid'Address;` where `Overlaying` is an identifier defined by an object declaration and `Overlaid` is an identifier defined either by an object declaration or a parameter specification. Flag an Address aspect specification if this aspect specification is not a part of an object declaration and if the aspect value does not have the form `Overlaid'Address` where `Overlaid` is an identifier defined either by an object declaration or a parameter specification.

Address specifications given for program units are not flagged.

Example

```
type Rec is record
   C : Integer;
end record;

Var_Rec : Rec;
Var_Int : Integer;

Var1 : Integer with Address => Var_Int'Address;
Var2 : Integer with Address => Var_Rec.C'Address;  --  FLAG
```

3.1.5.45 Not_Imported_Overlays

Flag an attribute definition clause that defines 'Address attribute and has the form `for Overlaying'Address use Overlaid'Address;` where `Overlaying` and `Overlaid` are identifiers both defined by object declarations if `Overlaying` is not marked as imported. Flag an Address aspect specification if this aspect specification is a part of an object declaration of the object `Overlaying` and if the aspect value has the form `Overlaid'Address` where `Overlaid` is an identifier defined by an object declaration if the object `Overlaying` is not marked as imported.

Example

```
package Pack is
   I : Integer;

   J : Integer with Address => I'Address;             --  FLAG

   L : Integer;
   for L'Address use I'Address;                       --  NO FLAG
   pragma Import (C, L);
end Pack;
```

3.1.5.46 Null_Paths

Flag a statement sequence that is a component of an `if`, `case` or `loop` statement if this sequences consists of NULL statements only.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Except_Enums: bool
If `true`, do not flag null paths inside case statements whose selecting expression is of an enumeration type.

Example

```
if I > 10 then
   J := 5;
elsif I > 0 then
   null;                  --  FLAG
else
  J := J + 1;
end if;

case J is
   when 1 =>
      I := I + 1;
   when 2 =>
      null;               --  FLAG
   when 3 =>
      J := J + 1;
   when others =>
      null;               --  FLAG
end case;
```

3.1.5.47 Objects_Of_Anonymous_Types

Flag any object declaration located immediately within a package declaration or a package body (including generic packages) if it uses anonymous access or array type definition. Record component definitions and parameter specifications are not flagged. Formal object declarations defined with anonymous access definitions are flagged.

Example

```
package Foo is
   type Arr is array (1 .. 10) of Integer;
   type Acc is access Integer;

   A : array (1 .. 10) of Integer;  --  FLAG
   B : Arr;

   C : access Integer;              --  FLAG
   D : Acc;

   generic
      F1 : access Integer;          --  FLAG
```

(continues on next page)

(continued from previous page)

```
      F2 : Acc;
   procedure Proc_G
     (P1 : access Integer;
      P2 : Acc);
end Foo;
```

3.1.5.48 Operator_Renamings

Flag subprogram renaming declarations that have an operator symbol as the name of renamed subprogram.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Name_Mismatch: bool
If `true`, only flag when the renamed subprogram is also an operator with a different name.

Example

```
function Foo (I, J : Integer)           --  FLAG
  return Integer renames Standard."+";
function "-" (I, J : Integer)           --  NO FLAG
  return Integer renames Bar;
```

3.1.5.49 OTHERS_In_Aggregates

Flag each use of an `others` choice in extension aggregates. In record and array aggregates, an `others` choice is flagged unless it is used to refer to all components, or to all but one component.

If, in case of a named array aggregate, there are two associations, one with an `others` choice and another with a discrete range, the `others` choice is flagged even if the discrete range specifies exactly one component; for example, `(1..1 => 0, others => 1)`.

Example

```
package Foo is
   type Arr is array (1 .. 10) of Integer;

   type Rec is record
      C1 : Integer;
      C2 : Integer;
      C3 : Integer;
      C4 : Integer;
   end record;

   type Tagged_Rec is tagged record
      C1 : Integer;
   end record;

   type New_Tagged_Rec is new Tagged_Rec with record
      C2 : Integer;
```

(continues on next page)

(continued from previous page)

```
      C3 : Integer;
      C4 : Integer;
   end record;

   Arr_Var1 : Arr := (others => 1);
   Arr_Var2 : Arr := (1 => 1, 2 => 2, others => 0);  --  FLAG

   Rec_Var1 : Rec := (C1 => 1, others => 0);
   Rec_Var2 : Rec := (1, 2, others => 3);            --  FLAG

   Tagged_Rec_Var : Tagged_Rec := (C1 => 1);

   New_Tagged_Rec_Var : New_Tagged_Rec := (Tagged_Rec_Var with others => 0); -- FLAG
end Foo;
```

3.1.5.50 OTHERS_In_CASE_Statements

Flag any use of an `others` choice in a `case` statement.

The rule has an optional parameter for the +R option and for LKQL rule options files:

N: int
: If specified, only flag if the others choice can be determined to span less than `N` values (0 means no minimum value).

Example

```
case J is
   when 1 =>
      I := I + 1;
   when 3 =>
      J := J + 1;
   when others =>         --  FLAG
      null;
end case;
```

3.1.5.51 OTHERS_In_Exception_Handlers

Flag any use of an `others` choice in an exception handler.

Example

```
exception
   when Constraint_Error =>
      I:= Integer'Last;
   when others =>                    --  FLAG
      I := I_Old;
      raise;
```

3.1.5.52 Outbound_Protected_Assignments

Flag an assignment statement located in a protected body if the variable name in the left part of the statement denotes an object declared outside this protected type or object.

Example

```
package Pack is
   Var : Integer;

   protected P is
      entry E (I : in out Integer);
      procedure P (I : Integer);
   private
      Flag : Boolean;
   end P;

end Pack;
package body Pack is
   protected body P is
      entry E (I : in out Integer) when Flag is
      begin
         I   := Var + I;
         Var := I;           --  FLAG
      end E;

      procedure P (I : Integer) is
      begin
         Flag := I > 0;
      end P;
   end P;
end Pack;
```

3.1.5.53 Overly_Nested_Control_Structures

Flag each control structure whose nesting level exceeds the value provided in the rule parameter.

The control structures checked are the following:

- `if` statement
- `case` statement
- `loop` statement
- selective accept statement
- timed entry call statement
- conditional entry call statement
- asynchronous select statement

The rule has the following (optional) parameters for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximal control structure nesting level that is not flagged. Defaults to 3 if not specified.

Loops_Only: bool
If `true`, only loop statements are counted.

Example

```
if I > 0 then
    for Idx in I .. J loop
       if J < 0 then
          case I is
             when 1 =>
                if Idx /= 0 then  --  FLAG (if rule parameter is 3)
                   J := J / Idx;
                end if;
             when others =>
                J := J + Idx;
          end case;
       end if;
    end loop;
end if;
```

3.1.5.54 Overly_Nested_Scopes

Flag a nested scope if the nesting level of this scope is more than the rule parameter. The following declarations are considered as scopes by this rule:

- package and generic package declarations and bodies;
- subprogram and generic subprogram declarations and bodies;
- task type and single task declarations and bodies;
- protected type and single protected declarations and bodies;
- entry bodies;

- block statements;

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Non-negative integer specifying the maximal allowed depth of scope constructs.

Example

```
with P; use P;
package Pack is
   package Pack1 is
      package Pack2 is
         generic
         package Pack_G is
            procedure P;            --  FLAG if rule parameter is 3 or less

            package Inner_Pack is   --  FLAG if rule parameter is 3 or less
               I : Integer;
            end Inner_Pack;
         end Pack_G;
      end Pack2;
   end Pack1
end Pack;
```

3.1.5.55 Parameters_Aliasing

Flags subprogram calls for which it can be statically detected that the same variable (or a variable and a subcomponent of this variable) is given as an actual to more than one `OUT` or `IN OUT` parameter. The rule resolves object renamings.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

In_Parameters: bool
Whether to consider aliasing between `OUT`, `IN OUT` and `IN` parameters, except for those `IN` parameters that are of a by-copy type, see the definition of by-copy parameters in the Ada Standard.

Example

```
package Pack is
   type Arr is array (1 .. 5) of Integer;

   type Rec is record
      Comp : Arr;
   end record;

   procedure Proc (P1 : in out : Rec; P2 : out Integer);
end Pack;

with Pack; use Pack;
procedure Test (I : Integer) is
   Var : Rec;
```

(continues on next page)

(continued from previous page)

```
begin
   Proc (Var, Var.Comp (I));   --  FLAG
```

3.1.5.56 POS_On_Enumeration_Types

Flag 'Pos attribute in case if the attribute prefix has an enumeration type (including types derived from enumeration types).

Example

```
procedure Bar (Ch1, Ch2 : Character; I : in out Integer) is
begin
   if Ch1'Pos in 32 .. 126           --  FLAG
     and then
      Ch2'Pos not in 0 .. 31         --  FLAG
   then
      I := (Ch1'Pos + Ch2'Pos) / 2;  --  FLAG (twice)
   end if;
end Bar;
```

3.1.5.57 Positional_Actuals_For_Defaulted_Generic_Parameters

Flag each generic actual parameter corresponding to a generic formal parameter with a default initialization, if positional notation is used.

Example

```
package Foo is
   function Fun_1 (I : Integer) return Integer;
   function Fun_2 (I : Integer) return Integer;

   generic
      I_Par1 : Integer;
      I_Par2 : Integer := 1;
      with function Fun_1 (I : Integer) return Integer is <>;
      with function Fun_3 (I : Integer) return Integer is Fun_2;
   package Pack_G is
      Var_1 : Integer := I_Par1;
      Var_2 : Integer := I_Par2;
      Var_3 : Integer := Fun_1 (Var_1);
      Var_4 : Integer := Fun_3 (Var_2);
   end Pack_G;

   package Pack_I_1 is new Pack_G (1);

   package Pact_I_2 is new Pack_G
     (2, I_Par2 => 3, Fun_1 => Fun_2, Fun_3 => Fun_1);
```

(continues on next page)

(continued from previous page)

```
   package Pack_I_3 is new Pack_G (1,
                                   2,             --  FLAG
                                   Fun_2,         --  FLAG
                                   Fun_1);        --  FLAG

end Foo;
```

3.1.5.58 Positional_Actuals_For_Defaulted_Parameters

Flag each actual parameter to a subprogram or entry call where the corresponding formal parameter has a default expression, if positional notation is used.

Example

```
   procedure Proc (I : in out Integer; J : Integer := 0) is
   begin
      I := I + J;
   end Proc;

begin
   Proc (Var1, Var2);   --  FLAG
```

3.1.5.59 Positional_Components

Flag each array, record and extension aggregate that includes positional notation.

Example

```
package Foo is
   type Arr is array (1 .. 10) of Integer;

   type Rec is record
      C_Int  : Integer;
      C_Bool : Boolean;
      C_Char : Character;
   end record;

   Var_Rec_1 : Rec := (C_Int => 1, C_Bool => True, C_Char => 'a');
   Var_Rec_2 : Rec := (2, C_Bool => False, C_Char => 'b');   --  FLAG
   Var_Rec_3 : Rec := (1, True, 'c');                        --  FLAG

   Var_Arr_1 : Arr := (1 => 1, others => 10);
   Var_Arr_2 : Arr := (1, others => 10);                     --  FLAG
end Foo;
```

3.1.5.60 Positional_Generic_Parameters

Flag each positional actual generic parameter except for the case when the generic unit being instantiated has exactly one generic formal parameter.

Example

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Conversion;
procedure Bar (I : in out Integer) is
   type My_Int is range -12345 .. 12345;

   function To_My_Int is new Ada.Unchecked_Conversion
     (Source => Integer, Target => My_Int);

   function To_Integer is new Ada.Unchecked_Conversion
     (My_Int, Integer);                                  --  FLAG (twice)

   package My_Int_IO is new  Ada.Text_IO.Integer_IO (My_Int);
```

3.1.5.61 Positional_Parameters

Flag each positional parameter notation in a subprogram or entry call, except for the following:

- Parameters of calls to attribute subprograms are not flagged;
- Parameters of prefix or infix calls to operator functions are not flagged;
- If the called subprogram or entry has only one formal parameter, the parameter of the call is not flagged;
- If call expression only carries one actual parameter, corresponding formal parameter doesn't have a default value, and all other formal parameters do, then the sole actual is not flagged.
- If a subprogram call uses the *Object.Operation* notation, then * the first parameter (that is, *Object*) is not flagged;
  - if the called subprogram has only two parameters, the second parameter of the call is not flagged;

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

All: bool
: If `true`, all the positional parameter associations that can be replaced with named associations according to language rules are flagged, except parameters of the calls to operator functions.

Example

```
procedure Bar (I : in out Integer) is
   function My_Max (Left, Right : Integer) return Integer renames Integer'Max;

   procedure Proc1 (I : in out Integer) is
   begin
      I := I + 1;
   end Proc1;

   procedure Proc2 (I, J : in out Integer) is
```

(continues on next page)

(continued from previous page)

```
   begin
      I := I + J;
   end Proc2;

   L, M : Integer := 1;
begin
   Proc1 (L);
   Proc2 (L, M);                                --  FLAG (twice)
   Proc2 (I => M, J => L);

   L := Integer'Max (10, M);
   M := My_Max (100, Right => L);               --  FLAG

end Bar;
```

3.1.5.62 Potential_Parameters_Aliasing

This rule is a complementary rule for the *Parameters_Aliasing* rule - it flags subprogram calls where the same variable (or a variable and its subcomponent) is given as an actual to more than one `OUT` or `IN OUT` parameter, but the fact of aliasing cannot be determined statically because this variable is an array component, and the index value(s) is(are) not known statically. The rule resolves object renamings.

Note that this rule does not flag calls that are flagged by the *Parameters_Aliasing* rule and vice versa.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

In_Parameters: bool
: Whether to consider aliasing between `OUT`, `IN OUT` and `IN` parameters, except for those `IN` parameters that are of a by-copy type, see the definition of by-copy parameters in the Ada Standard.

Example

```
package Pack is
   procedure Proc (P1 : out Integer; P2 : in out Integer);
   type Arr is array (1 .. 10 ) of Integer;
end Pack;

with Pack; use Pack;
procedure Proc (X : in out Arr; I, J : Integer) is
begin
   Proc (X (I), X (J));    --  FLAG
```

3.1.5.63 Recursive_Subprograms

Flags specs (and bodies that act as specs) of recursive subprograms. A subprogram is considered as recursive in a given context if there exists a chain of direct calls starting from the body of, and ending at this subprogram within this context. A context is provided by the set of Ada sources specified as arguments of a given `gnatcheck` call. Neither dispatching calls nor calls through access-to-subprograms are considered as direct calls by this rule. If *Follow_Dispatching_Calls* rule parameter is set, `gnatcheck` considers a dispatching call as a set of calls to all the subprograms the dispatching call may dispatch to, otherwise dispatching calls are ignored. The current rule limitation is that when processing dispatching calls the rule does not take into account type primitive operations declared in generic instantiations.

This rule does not take into account calls to subprograms whose bodies are not available because of any reason (a subprogram is imported, the Ada source containing the body is not provided as `gnatcheck` argument source etc.). The *Unavailable_Body_Calls* rule can be used to detect these cases.

Generic subprograms and subprograms detected in generic units are not flagged. Recursive subprograms in generic instantiations are flagged.

Ghost code and assertion code such as pre & post conditions or code inside of *pragma Assert* is not flagged either by default.

The rule does not take into account implicit calls that are the result of computing default initial values for an object or a subcomponent thereof as a part of the elaboration of an object declaration.

The rule also does not take into account subprogram calls inside aspect definitions.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Follow_Dispatching_Calls: bool
Whether to treat a dispatching call as a set of calls to all the subprograms the dispatching call may dispatch to.

Follow_Ghost_Code: bool
Whether to analyze ghost code and assertion code, which isn't analyzed by this check by default.

Example

```
function Factorial (N : Natural) return Positive is  --  FLAG
begin
   if N = 0 then
      return 1;
   else
      return N * Factorial (N - 1);
   end if;
end Factorial;
```

3.1.5.64 Redundant_Boolean_Expressions

Flag constructs including boolean operations that can be simplified. The following constructs are flagged:

- `if` statements that have `if` and `else` paths (and no `elsif` path) if both paths contain a single statement that is either:
  - an assignment to the same variable of `True` in one path and `False` in the other path
  - a return statement that in one path returns `True` and in the other path `False`

  where `True` and `False` are literals of the type `Standard.Boolean` or any type derived from it. Note that in case of assignment statements the variable names in the left part should be literally the same (case insensitive);

- `if` expressions that have `if` and `else` paths (without any `elseif`) if one path expression is `True` and the other is `False`, where `True` and `False` are literals of the `Standard.Boolean` type (or any type derived from it).
- infix call to a predefined = or /= operator when the right operand is `True` or `False` where `True` and `False` are literals of the type `Standard.Boolean` or any type derived from it.
- infix call to a predefined `not` operator whose argument is an infix call to a predefined ordering operator.

Example

```
if I + J > K then   --  FLAG
   return True;
else
   return False;
end if;
```

3.1.5.65 Redundant_Null_Statements

Flag null statements that serve no purpose and can be removed. If a null statement has a label it is not flagged.

Example

```
 if I > 0 then
    null;       --  FLAG
    pragma Assert (J > 0);
 end if;
```

3.1.5.66 Restrictions

Flags violations of Ada predefined and GNAT-specific restrictions according to the rule parameter(s) specified.

`gnatcheck` does not check Ada or GNAT restrictions itself, instead it compiles an argument source with a configuration file that defines restrictions of interest, analyses the style warnings generated by the GNAT compiler and includes the information about restriction violations detected into the `gnatcheck` messages.

This rule allows parametric rule exemptions, the parameters that are allowed in the definition of exemption sections are the names of the restrictions except for the case when a restriction requires a non-numeric parameter, in this case the parameter should be the name of the restriction with the parameter, as it is given for the rule.

The rule should have a parameter, the format of the rule parameter is the same as the parameter of the pragma `Restrictions` or `Restriction_Warnings`.

Note: In LKQL rule options files, this rule should have an `Arg` named parameter associated to a list of strings. Each element of this list should be a restriction parameter, for example:

```
val rules = @{
   Restrictions: {Arg: ["Max_Task_Entries=>2", "No_Access_Subprograms"]}
}
```

Attention: It is forbidden to provide the same restriction name in multiple instances of the `Restrictions` rule. Meaning that such configuration is invalid and will cause GNATcheck to issue an error message:

```
val rules = @{
   Restrictions: [
      {Arg: ["Max_Task_Entries=>2", "No_Access_Subprograms"]},
      {Arg: ["Max_Task_Entries=>6"], instance_name: "Another_Instance"}
            # ^^^^^^^^^^^^^^^^ The "Max_Task_Entries" name is provided in multiple␣
↪instances of "Restrictions"
   ]
}
```

If your code contains pragmas `Warnings` with parameter `Off`, this may result in false negatives for this rule, because the corresponding warnings generated during compilation will be suppressed. The workaround is to use for `gnatcheck` call a configuration file that contains `pragma Ignore_Pragma (Warnings);`.

Warning: Note, that some restriction checks cannot be performed by gnatcheck because they are either dynamic or require information from the code generation phase. For such restrictions gnatcheck generates the corresponding warnings and disables the `Restrictions` rules.

Example

```
with Ada.Finalization;      --  FLAG (+RRestrictions:No_Dependence=>Ada.Finalization)
procedure Proc is
   type Access_Integer is access Integer;
   Var : Access_Integer;
begin
   Var := new Integer'(1);  --  FLAG (+RRestrictions:No_Allocators)
end Proc;
```

3.1.5.67 Same_Logic

Flags expressions that contain a chain of infix calls to the same boolean operator (`and`, `or`, `and then`, `or else`, `xor`) if an expression contains syntactically equivalent operands.

Example

```
B := Var1 and Var2;            --  NO FLAG
return A or else B or else A;  --  FLAG
```

3.1.5.68 Same_Operands

Flags infix calls to binary operators /, =, /=, >, >=, <, <=, -, mod, rem (except when operating on floating point types) if operands of a call are syntactically equivalent.

Example

```
Y := (X + 1) / (X - 1);          --  NO FLAG
Z := (X + 1) / (X + 1);          --  FLAG
```

3.1.5.69 Same_Tests

Flags condition expressions in `if` statements or `if` expressions if a statement or expression contains another condition expression that is syntactically equivalent to the first one.

Example

```
if Str = A then                  --  FLAG: same test at line 5
   Put_Line("Hello, tata!");
elsif Str = B then
   Put_Line("Hello, titi!");
elsif Str = A then
   Put_Line("Hello, toto!");
else
   Put_Line("Hello, world!");
end if;
```

3.1.5.70 Side_Effect_Parameters

Flag subprogram calls and generic instantiations that have at least two actual parameters that are expressions containing a call to the same function as a subcomponent. Only calls to the functions specified as a rule parameter are considered.

The rule has an optional parameter(s) for the +R option and for LKQL rule options files:

Functions: list[string]
A list of fully expanded Ada names of functions to flag parameters from.

Note that a rule parameter should be a function name but not the name defined by a function renaming declaration. Note also, that if a rule parameter does not denote the name of an existing function or if it denotes a name defined by a function renaming declaration, the parameter itself is (silently) ignored and does not have any effect.

Note also, that the rule does not make any overload resolution, so if a rule parameter refers to more than one overloaded functions with the same name, the rule will treat calls to all these function as the calls to the same function.

Example

```
--  Suppose the rule is activated as +RSide_Effect_Parameters:P.Fun
package P is
  function Fun return Integer;
  function Fun  (I : Integer) return Integer;
  function Fun1 (I : Integer) return Integer;
end P;

with P; use P;
with Bar;
procedure Foo is
begin
  Bar (Fun, 1, Fun1 (Fun));    --  FLAG
```

3.1.5.71 Silent_Exception_Handlers

Flag any exception handler in which there exists at least one an execution path that does not raise an exception by a `raise` statement or a call to `Ada.Exceptions.Raise_Exception` or to `Ada.Exceptions.Reraise_Occurrence` nor contains a call to some subprogram specified by the rule parameter *Subprograms*.

The rule has the following parameter for the +R option and for LKQL rule options files:

Subprograms: list[string]

List of names of subprograms. An exception handler is not flagged if it contains a call to a subprogram that has a fully expanded Ada names that matches an element of this list. This list may contains fully expanded Ada names *AND* case-insensitive regular expression. From a +R option, you can specify a regular expression by providing an Ada string literal, and from an LKQL rule options file, you have to append the | character at the beginning of your regular expression. For example:

```
+RSilent_Exception_Handlers:My.Expanded.Name,"My\.Regex\..*"
```

maps to:

```
val rules = @{
   Silent_Exception_Handlers: {Subprograms: ["My.Expanded.Name", "|My\.Regex\..*"]}
}
```

Note that if you specify the rule with parameters in a command shell, you may need to escape its parameters. The best and the safest way of using this rule is to place it into an LKQL rule file and to use this rule file with the `--rule-file` switch, no escaping is needed in this case.

Example

```
with Ada.Exceptions; use Ada.Exceptions;

procedure Exc is
   procedure Log (Msg : String) with Import;
   --  Suppose the rule parameters are:
   --     ada.exceptions.exception_message,"\.Log$"
   I : Integer := 0;
begin
```

(continues on next page)

(continued from previous page)

```
   begin
      I := I + 1;
   exception
      when others =>    --  FLAG
         null;
   end;

exception
   when Constraint_Error =>  --  NO FLAG
      raise;
   when Program_Error =>     --  NO FLAG
      Log ("");
   when E : others =>        --  NO FLAG
      I := 0;
      Log (Exception_Message (E));
end Exc;
```

3.1.5.72 Single_Value_Enumeration_Types

Flag an enumeration type definition if it contains a single enumeration literal specification

Example

```
type Enum3 is (A, B, C);
type Enum1 is (D);       --  FLAG
```

3.1.5.73 Size_Attribute_For_Types

Flag any ‘Size attribute reference if its prefix denotes a type or a subtype. Attribute references that are subcomponents of attribute definition clauses of aspect specifications are not flagged.

Example

```
type T is record
   I : Integer;
   B : Boolean;
end record;

Size_Of_T : constant Integer := T'Size  --  FLAG
```

3.1.5.74 SPARK_Procedures_Without_Globals

Flags SPARK procedures that don't have a global aspect.

Example

```
package Test is
    procedure P with SPARK_Mode => On; -- FLAG

    procedure Q is null; -- NOFLAG

    function Foo return Integer  -- NOFLAG
    is (12)
    with SPARK_Mode => On;

    V : Integer;

    procedure T with Global => V;  -- NOFLAG

    function Bar return Integer with SPARK_Mode => On;  -- NOFLAG
end Test;
```

3.1.5.75 Suspicious_Equalities

Flag 'or' expressions whose left and right operands are unequalities referencing the same entity and a literal and 'and' expressions whose left and right operands are equalities referencing the same entity and a literal.

Example

```
procedure tmp is
   X : Integer := 0;
begin
   if X /= 1 or x /= 2 then -- FLAG
      null;
   end;
   if x = 1 and then X = 2 then -- Flag
      null;
   end;
end;
```

3.1.5.76 Trivial_Exception_Handlers

Flag exception handlers that contain a raise statement with no exception name as their first statement unless the enclosing handled sequence of statements also contains a handler with OTHERS exception choice that starts with any statement but not a raise statement with no exception name.

Example

```
   exception
      when My_Exception =>    --  FLAG
         raise;
   end;
exception
   when Constraint_Error =>  --  NO FLAG
      raise;
   when others =>
      null;
end;
```

3.1.5.77 Unavailable_Body_Calls

Flag any subprogram call if the set of argument sources does not contain a body of the called subprogram because of any reason. Calls to formal subprograms in generic bodies are not flagged. This rule can be useful as a complementary rule for the *Recursive_Subprograms* rule - it flags potentially missing recursion detection and identify potential missing checks.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Indirect_Calls: bool
Whether to flag all the indirect calls (that is, calls through access-to-subprogram values).

Example

```
procedure Calls is
   procedure Unknown with Import;

   type Proc_A is access procedure (X : Integer);
   X : Proc_A := Some_Proc'Access;
begin
   Unknown;      --  FLAG
   X (1);        --  FLAG (if Indirect_Calls is enabled)
```

3.1.5.78 Unchecked_Address_Conversions

Flag instantiations of `Ada.Unchecked_Conversion` if the actual for the formal type Source is the `System.Address` type (or a type derived from it), and the actual for the formal type `Target` is an access type (including types derived from access types). This include cases when the actual for `Source` is a private type and its full declaration is a type derived from `System.Address`, and cases when the actual for `Target` is a private type and its full declaration is an access type. The rule is checked inside expanded generics unless the `No_Instantiations` parameter is set.

The rule has the following optional parameters for the +R option and for LKQL rule options files:

All: bool
If `true`, all instantiations of Unchecked_Conversion to or from System.Address are flagged.

No_Instantiations: bool
If `true`, Do not check inside expanded generics.

Example

```
with Ada.Unchecked_Conversion;
with System;
package Foo is
   type My_Address is new System.Address;

   type My_Integer is new Integer;
   type My_Access is access all My_Integer;

   function Address_To_Access is new Ada.Unchecked_Conversion  --  FLAG
     (Source => My_Address,
      Target => My_Access);
end Foo;
```

3.1.5.79 Unchecked_Conversions_As_Actuals

Flag call to instantiation of `Ada.Unchecked_Conversion` if it is an actual in procedure or entry call or if it is a default value in a subprogram or entry parameter specification.

Example

```
with Ada.Unchecked_Conversion;
procedure Bar (I : in out Integer) is
   type T1 is array (1 .. 10) of Integer;
   type T2 is array (1 .. 10) of Integer;

   function UC is new Ada.Unchecked_Conversion (T1, T2);

   Var1 : T1 := (others => 1);
   Var2 : T2 := (others => 2);

   procedure Init (X : out T2; Y : T2 := UC (Var1)) is   --  FLAG
   begin
      X := Y;
   end Init;

   procedure Ident (X : T2; Y : out T2) is
   begin
      Y := X;
   end Ident;

begin
   Ident (UC (Var1), Var2);                              --  FLAG
end Bar;
```

3.1.5.80 Uninitialized_Global_Variables

Flag an object declaration that does not have an explicit initialization if it is located in a library-level package or generic package or bodies of library-level package or generic package (including packages and generic packages nested in those). Do not flag deferred constant declarations.

Example

```
package Foo is
   Var1 : Integer;       --  FLAG
   Var2 : Integer := 0;
end Foo;
```

3.1.5.81 Unnamed_Blocks_And_Loops

Flag each unnamed block statement. Flag a unnamed loop statement if this statement is enclosed by another loop statement or if it encloses another loop statement.

Example

```
procedure Bar (S : in out String) is
   I : Integer := 1;
begin
   if S'Length > 10 then
      declare                                    --  FLAG
         S1   : String (S'Range);
         Last : Positive := S1'Last;
         Idx  : Positive := 0;
      begin
         for J in S'Range loop                   --  FLAG
            S1 (Last - Idx) := S (J);
            Idx             := Idx + 1;

            for K in S'Range loop                --  FLAG
               S (K) := Character'Succ (S (K));
            end loop;

         end loop;

         S := S1;
      end;
   end if;
end Bar;
```

3.1.5.82 Unnamed_Exits

Flags exit statements with no loop names that exit from named loops.

Example

```
Named: for I in 1 .. 10 loop
    while J < 0 loop
       J := J + K;
       exit when J = L;  --  NO FLAG
    end loop;

    exit when J > 10;     --  FLAG
end loop Named;
```

3.1.5.83 Use_Array_Slices

Flag `for` loops if a loop contains a single assignment statement, and this statement is an assignment between array components or between an array component and a constant value, and such a loop can be replaced by a single assignment statement with array slices or array objects as the source and the target of the assignment.

Example

```
   type Table_Array_Type is array (1 .. 10) of Integer;
   Primary_Table   : Table_Array_Type;
   Secondary_Table : Table_Array_Type;

begin
   for I in Table_Array_Type'Range loop   --  FLAG
      Secondary_Table (I) := Primary_Table (I);
   end loop;

   for I in 2 .. 5 loop                   --  FLAG
      Secondary_Table (I) := 1;
   end loop;
```

3.1.5.84 Use_Case_Statements

Flag an `if` statement if this statement could be replaced by a `case` statement. An `if` statement is considered as being replaceable by a `case` statement if:

- it contains at least one `elsif` alternative;
- all the conditions are infix calls to some predefined relation operator, for all of them one operand is the reference to the same variable of some discrete type;
- for calls to relation operator another operand is some static expression;

Example

```
if I = 1 then      --  FLAG
   I := I + 1;
elsif I > 2 then
   I := I + 2;
else
   I := 0;
end if;
```

3.1.5.85 Use_For_Loops

Flag `while` loops which could be replaced by a `for` loop. The rule detects the following code patterns:

```
   ...
   Id : Some_Integer_Type ...;
   ... -- no write reference to Id
begin
   ...
   while Id <relation_operator> <expression> loop
      ...  -- no write reference to Id
      Id := Id <increment_operator> 1;
   end loop;
   ...  -- no reference to Id
end;
```

where relation operator in the loop condition should be some predefined relation operator, and increment_operator should be a predefined "+" or "-" operator.

Note, that the rule only informs about a possibility to replace a `while` loop by a `for`, but does not guarantee that this is really possible, additional human analysis is required for all the loops marked by the rule.

This rule has the following (optional) parameters for the +R option and for LKQL rule options files:

No_Exit: bool
If `true`, flag only loops that do not include an exit statement that applies to them.

No_Function: bool
If `true`, <expression> must not contain any non-operator function call.

Example

```
   Idx : Integer := 1;
begin
   while Idx <= 10 loop    --  FLAG
      Idx := Idx + 1;
   end loop;
end;
```

3.1.5.86 Use_For_Of_Loops

Flag `for ... in` loops which could be replaced by a `for ... of` loop, that is, where the loop index is used only for indexing a single object on a one dimension array.

The rule detects the following code patterns:

```
for Index in <array>'Range loop
   --  where <array> is an array object
   [all references to Index are of the form <array> (Index)]
end loop;
```

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

N: int
Non-negative integer, indicates the minimal number of references of the form `<array> (Index)` in the loop to make the loop to be flagged.

If no parameter is used for the rule, this corresponds to the parameter value 1.

Example

```
for J in Arr'Range loop    --  FLAG
   Sum := Sum + Arr (J);
end loop;

for K in Left'Range loop
   Res := Left (J) + Right (J);
end loop;
```

3.1.5.87 Use_If_Expressions

Flag `if` statements which could be replaced by an `if` expression. This rule detects the following code patterns:

```
if ... then
   return ...;
elsif ... then    --  optional chain of elsif
   return ...;
else
   return ...;
end if;
```

and:

```
if ... then
   <LHS> := ...;
elsif ... then    --  optional chain of elsif
   <LHS> := ...;
else
   <LHS> := ...;  --  same LHS on all branches
end if;
```

Example

```
if X = 1 then   --  FLAG
   return 1;
elsif X = 2 then
   return 2;
else
   return 3;
end if;

if X >= 2 then   --  FLAG
   X := X + 1;
elsif X <= 0 then
   X := X - 1;
else
   X := 0;
end if;
```

3.1.5.88 Use_Memberships

Flag expressions that could be rewritten as membership tests. Only expressions that are not subexpressions of other expressions are flagged. An expression is considered to be replaceable with an equivalent membership test if it is a logical expression consisting of a call to one or more predefined `or` operation(s), each relation that is an operand of the `or` expression is a comparison of the same variable of one of following forms:

- a call to a predefined = operator, the variable is the left operand of this call;
- a membership test applied to this variable;
- a range test of the form `Var >= E1 and Var <= E2` where `Var` is the variable in question and >=, and and <= are predefined operators;

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Short_Circuit: bool
Whether to consider the short circuit `and then` and `or else` operations along with the predefined logical `and` and `or` operators.

Example

```
begin
   Bool1 := A = 100         -- FLAG (if Short_Circuit is true)
            or (A >=  1 and then A <= B);

   Bool2 := A = 100         --  NO FLAG
            or B in S;

   Bool3 := A = 1           --  FLAG
            or
             A = B
            or
             A = B + A;
```

3.1.5.89 USE_PACKAGE_Clauses

Flag all `use` clauses for packages; `use type` clauses are not flagged.

Example

```
with Ada.Text_IO;
use Ada.Text_IO;                                -- FLAG
procedure Bar (S : in out String) is
```

3.1.5.90 Use_Ranges

Flag expressions of the form `Name'First .. Name'Last` that can be replaced by `Name'Range` or simply `Name`. Also flag expressions of the form `Name'Range` that can be replaced with `Name`.

Example

```
procedure Proc (S : String; I : in out Integer) is
begin
   for J in Integer'First .. Integer'Last loop   -- FLAG

      if I in Natural'Range then                 -- FLAG
         for K in S'Range loop                   -- NO FLAG
            I := I + K;
         end loop;
      end if;
   end loop;
end Proc;
```

3.1.5.91 Use_Record_Aggregates

Flag the first statement in the sequence of assignment statements if the targets of all these assignment statements are components of the same record objects, all the components of this objects get assigned as the result of such a sequence, and the type of the record object does not have discriminants. This rule helps to detect cases when a sequence of assignment statements can be replaced with a single assignment statement with a record aggregate as an expression being assigned, there is no guarantee that it detects all such sequences.

Example

```
   type Rec is record
      Comp1, Comp2 : Integer;
   end record;

   Var1, Var2 : Rec;
begin
   Var1.Comp1 := 1;  -- FLAG
   Var1.Comp2 := 2;
```

(continues on next page)

(continued from previous page)

```
   Var2.Comp1 := 1;  --  NO FLAG
   I := 1;
   Var2.Comp2 := 2;
```

3.1.5.92 Use_Simple_Loops

Flag `while` loop statements that have a condition statically known to be `TRUE`. Such loop statements can be replaced by simple loops.

Example

```
while True loop       --  FLAG
   I := I + 10;
   exit when I > 0;
end loop;
```

3.1.5.93 Use_While_Loops

Flag simple loop statements that have the exit statement completing execution of such a loop as the first statement in their sequence of statements. Such loop statements can be replaced by `WHILE` loops.

Example

```
loop      --  FLAG
   exit when I > 0;
   I := I + 10;
end loop;
```

3.1.5.94 Variable_Scoping

Flag local object declarations that can be moved into declare blocks nested into the declaration scope. A declaration is considered as movable into a nested scope if:

- The declaration does not contain an initialization expression;
- The declared object is used only in a nested block statement, and this block statement has a declare part;
- the block statement is not enclosed into a loop statement.

Example

```
procedure Scope is
   X : Integer;  --  FLAG
begin
   declare
      Y : Integer := 42;
   begin
      X := Y;
   end;
end;
```

3.1.5.95 Warnings

Flags construct that would result in issuing a GNAT warning if an argument source would be compiled with warning options corresponding to the rule parameter(s) specified. For GNAT warnings and corresponding warning control options see the Warning Message Control section of the GNAT User's Guide.

`gnatcheck` does not check itself if this or that construct would result in issuing a warning, instead it compiles the project sources with the needed warning control compilation options combined with the `-gnatc` switch, analyses the warnings generated by GNAT and adds the relevant information to the `gnatcheck` messages.

The rule should have a parameter, the format of the parameter should be a valid `static_string_expression` listing GNAT warnings switches (the letter following `-gnatw` in the *Warning Message Control* section mentioned above).

Note: In LKQL rule options files, this rule should have an `Arg` named parameter associated to a string corresponding to the wanted GNAT warning switches. Example:

```
val rules = @{ Warnings: {Arg: "u"} }
```

You can also use the shortcut argument format by associating a simple string to the rule name:

```
val rules = @{
   Warnings: "u"
}
```

Attention: It is forbidden to provide the same parameter in multiple instance of the `Warnings` rule. Meaning that such configuration is invalid and will cause GNATcheck to issue an error message:

```
val rules = @{
   Warnings: [
      {Arg: "u"},
      {Arg: "u", instance_name: "Another_Instance"}
           # ^-- The "u" parameter is provided in multiple instances of "Warnings"
   ]
}
```

Note that `s` and `e` parameters, corresponding respectively to GNAT `-gnatws` and `-gnatwe` options, are not allowed for the `Warnings` GNATcheck rule since they may have side effects on other rules.

Note also that some GNAT warnings are only emitted when generating code, these warnings will not be generated by this rule. In other words, this rule will only generate warnings that are enabled when using `-gnatc`.

If your code contains pragmas `Warnings` with parameter `Off`, this may result in false negatives for this rule, because the corresponding warnings generated during compilation will be suppressed. The workaround is to use a configuration file that contains `pragma Ignore_Pragma (Warnings);` when running `gnatcheck`.

This rule allows parametric rule exemptions, the parameters that are allowed in the definition of exemption sections are the same as the parameters of the rule itself.

Example

```
with Ada.Text_IO;                          --  FLAG (+RWarnings:u)
procedure Proc (I : in out Integer) is
begin
   pragma Unrecognized;                    --  FLAG (+RWarnings:g)

   I := I + 1;
end Proc;
```

3.1.6 Readability

The rules described in this subsection may be used to enforce feature usages that contribute towards readability.

3.1.6.1 End_Of_Line_Comments

Flags comments that are located in the source lines that contains Ada code.

Example

```
package A is
   -- NO FLAG
   I : Integer;  -- FLAG
end A;  --  FLAG
```

3.1.6.2 Headers

Check that the source text of a compilation unit starts from the text fragment specified as a rule parameter.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

Header: string
The name of a header file.

A header file is a plain text file. The rule checks that the beginning of the compilation unit source text is literally the same as the content of the header file. Blank lines and trailing spaces are not ignored and are taken into account, casing is important. The format of the line breaks (DOS or UNIX) is not important.

3.1.6.3 Identifier_Casing

Flag each defining identifier that does not have a casing corresponding to the kind of entity being declared. All defining names are checked. For the defining names from the following kinds of declarations a special casing scheme can be defined:

- type and subtype declarations;
- enumeration literal specifications (not including character literals) and function renaming declarations if the renaming entity is an enumeration literal;
- constant and number declarations (including object renaming declarations if the renamed object is a constant);
- exception declarations and exception renaming declarations.

The rule may have the following parameters for +R option and for LKQL rule options files:

Type: casing_scheme
Specifies casing for names from type and subtype declarations.

Enum: casing_scheme
Specifies the casing of defining enumeration literals and for the defining names in a function renaming declarations if the renamed entity is an enumeration literal.

Constant: casing_scheme
Specifies the casing for defining names from constants and named number declarations, including the object renaming declaration if the renamed object is a constant

Exception: casing_scheme
Specifies the casing for names from exception declarations and exception renaming declarations.

Others: casing_scheme
Specifies the casing for all defining names for which no special casing scheme is specified. If this parameter is not set, the casing for the entities that do not correspond to the specified parameters is not checked.

Exclude: string
The name of a dictionary file to specify casing exceptions. The name of the file may contain references to environment variables (e.g. $REPOSITORY_ROOT/my_dict.txt), they are replaced by the values of these variables.

Where *casing_scheme* is a string and:

```
casing_scheme ::= upper|lower|mixed
```

upper means that the defining identifier should be upper-case. *lower* means that the defining identifier should be lower-case *mixed* means that the first defining identifier letter and the first letter after each underscore should be upper-case, and all the other letters should be lower-case

You have to use the `param_name=value` formatting to pass arguments through the +R options. Example: `+RIdentifier_Casing:Type=mixed,Others=lower`.

If a defining identifier is from a declaration for which a specific casing scheme can be set, but the corresponding parameter is not specified for the rule, then the casing scheme defined by `Others` parameter is used to check this identifier. If `Others` parameter also is not set, the identifier is not checked.

Exclude is the name of the text file that contains casing exceptions. The way how this rule is using the casing exception dictionary file is consistent with using the casing exception dictionary in the GNAT pretty-printer *gnatpp*, see GNAT User's Guide.

There are two kinds of exceptions:

identifier
If a dictionary file contains an identifier, then each occurrence of that (defining) identifier in the checked source should use the casing specified included in *dictionary_file*

wildcard
A wildcard has the following syntax

```
wildcard ::= *simple_identifier* |
                    *simple_identifier |
                    simple_identifier*
simple_identifier ::= letter{letter_or_digit}
```

`simple_identifier` specifies the casing of subwords (the term 'subword' is used below to denote the part of a name which is delimited by '_' or by the beginning or end of the word and which does not contain any '_' inside). A wildcard of the form `simple_identifier*` defines the casing of the first subword of a defining name to check, the wildcard of the form `*simple_identifier` specifies the casing of the last subword, and the wildcard of the form `*simple_identifier*` specifies the casing of any subword.

If for a defining identifier some of its subwords can be mapped onto wildcards, but some other cannot, the casing of the identifier subwords that are not mapped onto wildcards from casing exception dictionary is checked against the casing scheme defined for the corresponding entity.

If some identifier is included in the exception dictionary both as a whole identifier and can be mapped onto some wildcard from the dictionary, then it is the identifier and not the wildcard that is used to check the identifier casing.

If more than one dictionary file is specified, or a dictionary file contains more than one exception variant for the same identifier, the new casing exception overrides the previous one.

Casing check against dictionary file(s) has a higher priority than checks against the casing scheme specified for a given entity/declaration kind.

The rule activation option should contain at least one parameter.

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are:

Type
Exempts check for type and subtype name casing

Enum
Exempts check for enumeration literal name casing

Constant
Exempts check for constant name casing

Exception
Exempts check for exception name casing

Others
Exempts check for defining names for which no special casing scheme is specified.

Exclude
Exempts check for defining names for which casing schemes are specified in exception dictionaries

Example

```
--  if the rule is activated as '+RIdentifier_Casing:Type=upper,Others=mixed'
package Foo is
   type ENUM_1 is (A1, B1, C1);
   type Enum_2 is (A2, B2, C2);      --  FLAG

   Var1 : Enum_1 := A1;
   VAR2 : ENUM_2 := A2;              --  FLAG
end Foo;
```

3.1.6.4 Identifier_Prefixes

Flag each defining identifier that does not have a prefix corresponding to the kind of declaration it is defined by. The defining names in the following kinds of declarations are checked:

- type and subtype declarations (task, protected and access types are treated separately);
- enumeration literal specifications (not including character literals) and function renaming declarations if the renaming entity is an enumeration literal;
- exception declarations and exception renaming declarations;
- constant and number declarations (including object renaming declarations if the renamed object is a constant).

Defining names declared by single task declarations or single protected declarations are not checked by this rule.

The defining name from the full type declaration corresponding to a private type declaration or a private extension declaration is never flagged. A defining name from an incomplete type declaration is never flagged.

The defining name from a subprogram renaming-as-body declaration is never flagged.

For a deferred constant, the defining name in the corresponding full constant declaration is never flagged.

The defining name from a body that is a completion of a program unit declaration or a proper body of a subunit is never flagged.

The defining name from a body stub that is a completion of a program unit declaration is never flagged.

Note that the rule checks only defining names. Usage name occurrence are not checked and are never flagged.

The rule may have the following parameters for the +R option and for LKQL rule options files:

Type: string
Specifies the prefix for a type or subtype name.

Concurrent: string
Specifies the prefix for a task and protected type/subtype name. If this parameter is set, it overrides for task and protected types the prefix set by the Type parameter.

Access: string
Specifies the prefix for an access type/subtype name. If this parameter is set, it overrides for access types the prefix set by the `Type` parameter.

Class_Access: string
Specifies the prefix for the name of an access type/subtype that points to some class-wide type. If this parameter is set, it overrides for such access types and subtypes the prefix set by the `Type` or `Access` parameter.

Subprogram_Access: string
Specifies the prefix for the name of an access type/subtype that points to a subprogram. If this parameter is set, it overrides for such access types/subtypes the prefix set by the `Type` or `Access` parameter.

Derived: string
Specifies the prefix for a type that is directly derived from a given type or from a subtype thereof. The parameter must have the `string1:string2` format where *string1* should be a full expanded Ada name of the ancestor type (starting from the full expanded compilation unit name) and *string2* defines the prefix to check. If this parameter is set, it overrides for types that are directly derived from the given type the prefix set by the `Type` parameter.

Constant: string
Specifies the prefix for defining names from constants and named number declarations, including the object renaming declaration if the renamed object is a constant.

> **Attention:** For legacy reasons, formal object declarations are not considered constant, even if they are declared with the `in` mode. Consequently, this rule will flag each generic formal object declarations that have the prefix specified by this parameter value.

Enum: string
Specifies the prefix for defining enumeration literals and for the defining names in a function renaming declarations if the renamed entity is an enumeration literal.

Exception: string
Specifies the prefix for defining names from exception declarations and exception renaming declarations.

Exclusive: bool
If `true`, check that only those kinds of names for which specific prefix is defined have that prefix (e.g., only type/subtype names have prefix *T_*, but not variable or package names), and flag all defining names that have any of the specified prefixes but do not belong to the kind of entities this prefix is defined for. By default the exclusive check mode is ON.

You have to use the `param_name=value` formatting to pass arguments through the `+R` options. Example: `+RIdentifier_Prefixes:Type=Type_,Enum=Enum_`.

The `+RIdentifier_Prefixes` option (with no parameter) does not create a new instance for the rule; thus, it has no effect on the current GNATcheck run.

There is no default prefix setting for this rule. All checks for name prefixes are case-sensitive

If any error is detected in a rule parameter, that parameter is ignored. In such a case the options that are set for the rule are not specified.

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are:

Type
Exempts check for type and subtype name prefixes

Concurrent
Exempts check for task and protected type/subtype name prefixes

Access
Exempts check for access type/subtype name prefixes

Class_Access
Exempts check for names of access types/subtypes that point to some class-wide types

Subprogram_Access
Exempts check for names of access types/subtypes that point to subprograms

Derived
Exempts check for derived type name prefixes

Constant
Exempts check for constant and number name prefixes

Exception
Exempts check for exception name prefixes

Enum
Exempts check for enumeration literal name prefixes

Exclusive
Exempts check that only names of specific kinds of entities have prefixes specified for these kinds

Example

```
--  if the rule is activated as '+RIdentifier_Prefixes:Type=Type_,Constant=Const_,
↪ExceptioN=X_'
package Foo is
   type Type_Enum_1 is (A1, B1, C1);
   type Enum_2      is (A2, B2, C2);          --  FLAG

   Const_C1 : constant Type_Enum_1 := A1;
   Const2   : constant Enum_2      := A2;     --  FLAG

   X_Exc_1 : exception;
   Exc_2   : exception;                       --  FLAG
end Foo;
```

3.1.6.5 Identifier_Suffixes

Flag the declaration of each identifier that does not have a suffix corresponding to the kind of entity being declared. The following declarations are checked:

- type declarations
- subtype declarations
- object declarations (variable and constant declarations, but not number, declarations, record component declarations, parameter specifications, extended return object declarations, formal object declarations)
- package renaming declarations (but not generic package renaming declarations)

The default checks (enforced by the *Default* rule parameter) are:

- type-defining names end with `_T`, unless the type is an access type, in which case the suffix must be `_A`
- constant names end with `_C`
- names defining package renamings end with `_R`
- the check for access type objects is not enabled

Defining identifiers from incomplete type declarations are never flagged.

For a private type declaration (including private extensions), the defining identifier from the private type declaration is checked against the type suffix (even if the corresponding full declaration is an access type declaration), and the defining identifier from the corresponding full type declaration is not checked.

For a deferred constant, the defining name in the corresponding full constant declaration is not checked.

Defining names of formal types are not checked.

Check for the suffix of access type data objects is applied to the following kinds of declarations:

- variable and constant declaration
- record component declaration
- return object declaration
- parameter specification
- extended return object declaration
- formal object declaration

If both checks for constant suffixes and for access object suffixes are enabled, and if different suffixes are defined for them, then for constants of access type the check for access object suffixes is applied.

The rule may have the following parameters for `+R` option and for LKQL rule options files:

Default: bool
If `true`, sets the default listed above for all the names to be checked.

Type_Suffix: string
Specifies the suffix for a type name.

Access_Suffix: string
Specifies the suffix for an access type name. If this parameter is set, it overrides for access types the suffix set by the `Type_Suffix` parameter. For access types, this parameter may have the following format: *suffix1(suffix2)*. That means that an access type name should have the *suffix1* suffix except for the case when the designated type is also an access type, in this case the type name should have the *suffix1 & suffix2* suffix.

Class_Access_Suffix: string
Specifies the suffix for the name of an access type that points to some class-wide type. If this parameter is set, it overrides for such access types the suffix set by the `Type_Suffix` or `Access_Suffix` parameter.

Class_Subtype_Suffix: string
Specifies the suffix for the name of a subtype that denotes a class-wide type.

Constant_Suffix: string
Specifies the suffix for a constant name.

Renaming_Suffix: string
Specifies the suffix for a package renaming name.

Access_Obj_Suffix: string
Specifies the suffix for objects that have an access type (including types derived from access types).

Interrupt_Suffix: string
Specifies the suffix for protected subprograms used as interrupt handlers.

You have to use the `param_name=value` formatting to pass arguments through the `+R` options. Example: `+RIdentifier_Prefixes:Type=_T,Constant=_C`.

The `+RIdentifier_Prefixes` option (with no parameter) does not create a new instance for the rule; thus, it has no effect on the current GNATcheck run.

The *string* value must be a valid suffix for an Ada identifier (after trimming all the leading and trailing space characters, if any). Parameters are not case sensitive, except the *string* part.

If any error is detected in a rule parameter, the parameter is ignored. In such a case the options that are set for the rule are not specified.

The rule allows parametric exemption, the parameters that are allowed in the definition of exemption sections are:

Type
Exempts check for type name suffixes

Access
Exempts check for access type name suffixes

Access_Obj
Exempts check for access object name suffixes

Class_Access
Exempts check for names of access types that point to some class-wide types

Class_Subtype
Exempts check for names of subtypes that denote class-wide types

Constant
Exempts check for constant name suffixes

Renaming
Exempts check for package renaming name suffixes

Example

```
--  if the rule is activated as '+RIdentifier_Suffixes:Access_Suffix=_PTR,Type_Suffix=_T,
↪Constant_Suffix=_C'
package Foo is
   type Int   is range 0 .. 100;       --  FLAG
   type Int_T is range 0 .. 100;

   type Int_A   is access Int;         --  FLAG
   type Int_PTR is access Int;

   Const   : constant Int := 1;        --  FLAG
   Const_C : constant Int := 1;

end Foo;
```

3.1.6.6 Lowercase_Keywords

Flag Ada keywords that are not purely lowercase, such as `BEGIN` or `beGin`. The rule is language version sensitive.

The rule has an optional parameter for the `+R` option and for LKQL rule options files:

Language_Version: string
The version of the language on which to run the checker (supported versions are Ada_83, Ada_95, Ada_2005, Ada_2012, and, Ada_2022 which is also the default).

Example

```
packagE Foo is -- FLAG
END Foo; -- FLAG
```

3.1.6.7 Max_Identifier_Length

Flag any defining identifier that has length longer than specified by the rule parameter. Defining identifiers of enumeration literals are not flagged.

The rule has a mandatory parameter for the +R option and for LKQL rule options files:

N: int
The maximal allowed identifier length specification.

Example

```
type My_Type is range -100 .. 100;
My_Variable_With_A_Long_Name : My_Type;  -- FLAG (if rule parameter is 27 or smaller)
```

3.1.6.8 Min_Identifier_Length

Flag any defining identifier that has length shorter than specified by the rule parameter. Defining identifiers of objects and components of numeric types are not flagged.

The rule has a mandatory parameter for the +R option and for LKQL rule options files:

N: int
The minimal allowed identifier length specification.

Example

```
I : Integer;              --  NO FLAG
J : String (1 .. 10);     --  FLAG
```

3.1.6.9 Misnamed_Controlling_Parameters

Flag a declaration of a dispatching operation, if the first parameter is not a controlling one and its name is not `This` (the check for parameter name is not case-sensitive). Declarations of dispatching functions with a controlling result and no controlling parameter are never flagged.

A subprogram body declaration, subprogram renaming declaration, or subprogram body stub is flagged only if it is not a completion of a prior subprogram declaration.

Example

```
package Foo is
   type T is tagged private;

   procedure P1 (This : in out T);
   procedure P2 (That : in out T);              --  FLAG
   procedure P1 (I : Integer; This : in out T); --  FLAG
```

3.1.6.10 Name_Clashes

Check that certain names are not used as defining identifiers. The names that should not be used as identifiers must be listed in a dictionary file that is a rule parameter. A defining identifier is flagged if it is included in a dictionary file specified as a rule parameter, the check is not case-sensitive. Only the whole identifiers are checked, not substrings thereof. More than one dictionary file can be specified as the rule parameter, in this case the rule checks defining identifiers against the union of all the identifiers from all the dictionary files provided as the rule parameters.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

Dictionary_File: string
: The name of a dictionary file. The name may contain references to environment variables (e.g. $REPOSITORY_ROOT/my_dict.txt), they are replaced by the values of these variables.

A dictionary file is a plain text file. The maximum line length for this file is 1024 characters. If the line is longer than this limit, extra characters are ignored.

If the name of the dictionary file does not contain any path information and the rule option is specifies in a rule file, first the tool tries to locate the dictionary file in the same directory where the rule file is located, and if the attempt fails - in the current directory.

Each line can be either an empty line, a comment line, or a line containing a list of identifiers separated by space or HT characters. A comment is an Ada-style comment (from -- to end-of-line). Identifiers must follow the Ada syntax for identifiers. A line containing one or more identifiers may end with a comment.

Example

```
--  If the dictionary file contains names 'One' and 'Two":
One          : constant Integer := 1;     --  FLAG
Two          : constant Float   := 2.0;   --  FLAG
Constant_One : constant Float   := 1.0;
```

3.1.6.11 No_Dependence

Flag every explicit dependency (with clause) to any of the library units designated by names passed as parameters.

This rule has the following optional parameter for the +R option and for LKQL rule options files:

Unit_Names: list[string]
: List of fully qualified names designating the library units that should not be explicitly depended upon.

The list of unit names is case insensitive. Any case can be used both in the parameter or in the code's with clauses.

Example

```
--  if the rule is activated as +RNo_Dependence:Unchecked_Conversion
with Unchecked_Conversion; -- FLAG

package Foo is
end Foo;
```

3.1.6.12 Numeric_Format

Flag each numeric literal which does not satisfy at least one of the following requirements:

- the literal is given in the conventional decimal notation given, or, if its base is specified explicitly, this base should be 2, 8, 10 or 16 only;
- if the literal base is 8 or 10, an underscore should separate groups of 3 digits starting from the right end of the literal;
- if the literal base is 2 or 16, an underscore should separate groups of 4 digits starting from the right end of the literal;
- all letters (exponent symbol and digits above 9) should be in upper case.

Example

```
D : constant := 16#12AB_C000#;          --  NO FLAG
E : constant := 3.5E3;                  --  NO FLAG

F : constant := 1000000;                --  FLAG
G : constant := 2#0001000110101011#;    --  FLAG
```

3.1.6.13 Object_Declarations_Out_Of_Order

Flag any object declaration that is located in a library unit body if this is preceding by a declaration of a program unit spec, stub or body.

Example

```
procedure Proc is
   procedure Proc1 is separate;

   I : Integer;     -- FLAG
```

3.1.6.14 One_Construct_Per_Line

Flag any statement, declaration or clause if the code line where this construct starts contains some other Ada code symbols preceding or following this construct. The following constructs are not flagged:

- enumeration literal specification;
- parameter specifications;
- discriminant specifications;
- mod clauses;
- loop parameter specification;
- entry index specification;
- choice parameter specification;

In case if we have two or more declarations/statements/clauses on a line and if there is no Ada code preceding the first construct, the first construct is flagged

Example

```
procedure Swap (I, J : in out Integer) is
   Tmp : Integer;
begin
   Tmp := I;
   I := J; J := Tmp;       --  FLAG
end Swap;
```

3.1.6.15 Overriding_Indicators

Check that overriding subprograms are explicitly marked as such.

This applies to all subprograms of a derived type that override a primitive operation of the type, for both tagged and untagged types. In particular, the declaration of a primitive operation of a type extension that overrides an inherited operation must carry an overriding indicator. Another case is the declaration of a function that overrides a predefined operator (such as an equality operator).

Attention: This doesn't apply to primitives of multiple untagged types, and as such, won't ever flag such overriding primitives.

Example

```
package Foo is
  type A is null record;
  procedure Prim (Self : A) is null;

  type B is new A;

  procedure Prim (Self : B) is null; -- FLAG
end Foo;
```

3.1.6.16 Profile_Discrepancies

Flag subprogram or entry body (or body stub) if its parameter (or parameter and result) profile does not follow the lexical structure of the profile in the corresponding subprogram or entry declaration.

Example

```
package Pack is
   procedure Proc
     (I : Integer;
      J : Integer);
end Pack;

package body Pack is
   procedure Proc (I, J : Integer) is    --  FLAG
```

3.1.6.17 Style_Checks

Flags violations of the source code presentation and formatting rules specified in the Style Checking section of the GNAT User's Guide according to the rule parameter(s) specified.

gnatcheck does not check GNAT style rules itself, instead it compiles an argument source with the needed style check compilation options, analyses the style messages generated by the GNAT compiler and includes the information about style violations detected into the gnatcheck messages.

This rule takes a parameter in one of the following forms:

- *All_Checks*, which enables the standard style checks corresponding to the -gnatyy GNAT style check option,
- A string with the same structure and semantics as the string_LITERAL parameter of the GNAT pragma Style_Checks (see Pragma Style_Checks in the GNAT Reference Manual).

For instance, the +RStyle_Checks:0 rule option activates the compiler style check that corresponds to -gnaty0 style check option.

Note: In LKQL rule options files, this rule should have an Arg named parameter associated to a string corresponding to the wanted GNAT style checks switches. Example:

```
val rules = @{ Style_Checks: {Arg: "xz"} }
```

You can also use the shortcut argument format by associating a simple string to the rule name:

```
val rules = @{
   Style_Checks: "xz"
}
```

Attention: It is forbidden to provide the same parameter in multiple instances of the Style_Checks rule. Meaning that such configuration is invalid and will cause GNATcheck to issue an error message:

```
val rules = @{
   Style_Checks: [
      {Arg: "xz"},
      {Arg: "x", instance_name: "Another_Instance"}
           # ^-- The "x" parameter is provided in multiple instances of "Style_Checks"
   ]
}
```

This rule allows parametric rule exemptions, the parameters that are allowed in the definition of exemption sections are the same as the parameters of the rule itself.

Example

```
package Pack is
   I : Integer;
end;    -- FLAG (for +RStyle_Checks:e)
```

3.1.6.18 Uncommented_BEGIN

Flags BEGIN keywords in program unit bodies if the body contains both declarations and a statement part and if there is no trailing comment just after the keyword (on the same line) with the unit name as the only content of the comment, the casing of the unit name in the comment should be the same as the casing of the defining unit name in the unit body declaration.

Example

```
procedure Proc (I : out Integer) is
   B : Boolean;
begin
   I := Var;
end Proc;
```

3.1.6.19 Uncommented_BEGIN_In_Package_Bodies

Flags BEGIN keywords in package bodies if the body contains both declarations and a statement part and if there is no trailing comment just after the keyword (on the same line) with the package name as the only content of the comment, the casing of the package name in the comment should be the same as the casing of the defining unit name in the package body.

Example

```
package body Foo is
   procedure Proc (I : out Integer) is
   begin
      I := Var;
   end Proc;

   package body Inner is
      procedure Inner_Proc (I : out Integer) is
      begin
         I := Inner_Var;
      end  ;
   begin  -- Inner
      Inner_Var := 1;
   end Inner;
begin                   --  FLAG
   Var := Inner.Inner_Var + 1;
end Foo;
```

3.1.6.20 Uncommented_End_Record

Flags END keywords that are trailing keywords in record definitions if a record definition is longer than N lines where N is a rule parameter, and the line that contains the END keyword does not contain a trailing comment immediately after this END. This trailing comment should start with the name of the type that contains this record definition as (a part of) its type definition, and it may contain any other information separated from the type name by a space or a comma.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Non-negative integer specifying the maximum size (in source code lines) of a record definition that does not require the type name as a trailing comment.

Example

```
--  If the rule parameter is 3:
type R1 is record
   I : Integer;
end record;       --  NO FLAG

type R2 is record
   I : Integer;
   B : Boolean;
end record; -- R2      NO FLAG

type R3 is record
   C : Character;
   F : Float;
end record;       -- FLAG
```

3.2 Feature-Related Rules

The rules in this section can be used to enforce specific usage patterns for a variety of language features.

3.2.1 Abort_Statements

Flag abort statements.

Example

```
if Flag then
   abort T;    --  FLAG
end if;
```

3.2.2 Abstract_Type_Declarations

Flag all declarations of abstract types, including generic formal types. For an abstract private type, the full type declarations is flagged only if it is itself declared as abstract. Interface types are not flagged.

Example

```
package Foo is
   type Figure is abstract tagged private;              --  FLAG
   procedure Move (X : in out Figure) is abstract;
private
   type Figure is abstract tagged null record;          --  FLAG
end Foo;
```

3.2.3 Anonymous_Access

Flag object declarations, formal object declarations and component declarations with anonymous access type definitions. Discriminant specifications and parameter specifications are not flagged.

Example

```
procedure Anon (X : access Some_Type) is    --  NO FLAG
   type Square
     (Location : access Coordinate)         --  NO FLAG
   is record
      null;
   end record;

   type Cell is record
      Some_Data : Integer;
      Next      : access Cell;              --  FLAG
   end record;

   Link : access Cell;                      --  FLAG
```

3.2.4 Anonymous_Subtypes

Flag all uses of anonymous subtypes except for the following:

- when the subtype indication depends on a discriminant, this includes the cases of a record component definitions when a component depends on a discriminant, and using the discriminant of the derived type to constraint the parent type;
- when a self-referenced data structure is defined, and a discriminant is constrained by the reference to the current instance of a type;

A use of an anonymous subtype is any instance of a subtype indication with a constraint, other than one that occurs immediately within a subtype declaration. Any use of a range other than as a constraint used immediately within a subtype declaration is considered as an anonymous subtype.

The rule does not flag ranges in the component clauses from a record representation clause, because the language rules do not allow to use subtype names there.

An effect of this rule is that `for` loops such as the following are flagged (since `1..N` is formally a 'range')

```
for I in 1 .. N loop   --  FLAG
   ...
end loop;
```

Declaring an explicit subtype solves the problem:

```
subtype S is Integer range 1..N;
...
for I in S loop        --  NO FLAG
   ...
end loop;
```

3.2.5 At_Representation_Clauses

Flag at clauses and mod clauses (treated as obsolescent features in the Ada Standard).

Example

```
Id : Integer;
for Id use at Var'Address;   --  FLAG

type Rec is record
   Field : Integer;
end record;

for Rec use
   record at mod 2;          --  FLAG
end record;
```

3.2.6 Blocks

Flag each block statement.

Example

```
if I /= J then
   declare              --  FLAG
      Tmp : Integer;
   begin
      TMP := I;
      I   := J;
      J   := Tmp;
   end;
end if;
```

3.2.7 Complex_Inlined_Subprograms

Flag a subprogram (or generic subprogram, or instantiation of a subprogram) if pragma Inline is applied to it and at least one of the following conditions is met:

- it contains at least one complex declaration such as a subprogram body, package, task, protected declaration, or a generic instantiation (except instantiation of `Ada.Unchecked_Conversion`);
- it contains at least one complex statement such as a loop, a case or an if statement;
- the number of statements exceeds a value specified by the *N* rule parameter;

Subprogram renamings are also considered.

This rule has the following (mandatory) parameter for the `+R` option and for LKQL rule options files:

N: int
Positive integer specifying the maximum allowed total number of statements in the subprogram body.

Example

```
procedure Swap (I, J : in out Integer) with Inline => True;

procedure Swap (I, J : in out Integer) is   --  FLAG
begin

   if I /= J then
      declare
         Tmp : Integer;
      begin
         TMP := I;
         I   := J;
         J   := Tmp;
      end;
   end if;

end Swap;
```

3.2.8 Conditional_Expressions

Flag use of conditional expression.

This rule has the following (optional) parameter for the `+R` option and for LKQL rule options files:

Except_Assertions: bool
If `true`, do not flag a conditional expression if it is a subcomponent of the following constructs:

argument of the following pragmas

Language-defined

- `Assert`

GNAT-specific

- `Assert_And_Cut`
- `Assume`

- Contract_Cases
- Debug
- Invariant
- Loop_Invariant
- Loop_Variant
- Postcondition
- Precondition
- Predicate
- Refined_Post

definition of the following aspects

Language-defined

- Static_Predicate
- Dynamic_Predicate
- Pre
- Pre'Class
- Post
- Post'Class
- Type_Invariant
- Type_Invariant'Class

GNAT-specific

- Contract_Cases
- Invariant
- Invariant'Class
- Predicate
- Refined_Post

Example

```
Var1 : Integer := (if I > J then 1 else 0);  --  FLAG
Var2 : Integer := I + J;
```

3.2.9 Controlled_Type_Declarations

Flag all declarations of controlled types. A declaration of a private type is flagged if its full declaration declares a controlled type. A declaration of a derived type is flagged if its ancestor type is controlled. Subtype declarations are not checked. A declaration of a type that itself is not a descendant of a type declared in `Ada.Finalization` but has a controlled component is not checked.

Example

```
with Ada.Finalization;
package Foo is
   type Resource is new Ada.Finalization.Controlled with private;  --  FLAG
```

3.2.10 Declarations_In_Blocks

Flag all block statements containing local declarations. A `declare` block with an empty *declarative_part* or with a *declarative part* containing only pragmas and/or `use` clauses is not flagged.

Example

```
if I /= J then
   declare                          --  FLAG
      Tmp : Integer;
   begin
      TMP := I;
      I   := J;
      J   := Tmp;
   end;
end if;
```

3.2.11 Deeply_Nested_Inlining

Flag a subprogram (or generic subprogram) if pragma Inline has been applied to it, and it calls another subprogram to which pragma Inline applies, resulting in potential nested inlining, with a nesting depth exceeding the value specified by the *N* rule parameter.

This rule requires the global analysis of all the compilation units that are `gnatcheck` arguments; such analysis may affect the tool's performance. If gnatcheck generates warnings saying that "*body is not analyzed for ...*", this means that such an analysis is incomplete, this may result in rule false negatives.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
: Positive integer specifying the maximum level of nested calls to subprograms to which pragma Inline has been applied.

Example

```
procedure P1 (I : in out integer) with Inline => True;   --  FLAG
procedure P2 (I : in out integer) with Inline => True;
procedure P3 (I : in out integer) with Inline => True;
procedure P4 (I : in out integer) with Inline => True;

procedure P1 (I : in out integer) is
begin
   I := I + 1;
   P2 (I);
end;

procedure P2 (I : in out integer) is
begin
   I := I + 1;
   P3 (I);
end;

procedure P3 (I : in out integer) is
begin
   I := I + 1;
   P4 (I);
end;

procedure P4 (I : in out integer) is
begin
   I := I + 1;
end;
```

3.2.12 Default_Parameters

Flag formal part (in subprogram specifications and entry declarations) if it defines more than N parameters with default values, when N is a rule parameter. If no parameter is provided for the rule then all the formal parts with defaulted parameters are flagged.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

N: int
Integer not less than 0 specifying the minimal allowed number of defaulted parameters.

Example

```
procedure P (I : in out Integer; J : Integer := 0);  -- No FLAG (if parameter is 1)
procedure Q (I : in out Integer; J : Integer);
procedure R (I, J : Integer := 0; K : Integer := 0); --  FLAG (if parameter is 2 or less)
procedure S (I : Integer; J, K : Integer := 0);      --  FLAG (if parameter is 2 or less)
```

3.2.13 Discriminated_Records

Flag all declarations of record types with discriminants. Only the declarations of record and record extension types are checked. Incomplete, formal, private, derived and private extension type declarations are not checked. Task and protected type declarations also are not checked.

Example

```
type Idx is range 1 .. 100;
type Arr is array (Idx range <>) of Integer;
subtype Arr_10 is Arr (1 .. 10);

type Rec_1 (D : Idx) is record        --  FLAG
   A : Arr (1 .. D);
end record;

type Rec_2 (D : Idx) is record        --  FLAG
   B : Boolean;
end record;

type Rec_3 is record
   B : Boolean;
end record;
```

3.2.14 Enumeration_Representation_Clauses

Flag enumeration representation clauses.

Example

```
type Enum1 is (A1, B1, C1);
for Enum1 use (A1 => 1, B1 => 11, C1 => 111);     --  FLAG
```

3.2.15 Explicit_Full_Discrete_Ranges

Flag each discrete range that has the form `A'First .. A'Last`.

Example

```
   subtype Idx is Integer range 1 .. 100;
begin
   for J in Idx'First .. Idx'Last loop   --  FLAG
      K := K + J;
   end loop;

   for J in Idx loop
      L := L + J;
   end loop;
```

3.2.16 Explicit_Inlining

Flag a subprogram declaration, a generic subprogram declaration or a subprogram instantiation if this declaration has an Inline aspect specified or an Inline pragma applied to it. If a generic subprogram declaration has an Inline aspect specified or pragma Inline applied, then only generic subprogram declaration is flagged but not its instantiations.

Example

```
procedure Swap (I, J : in out Integer);                    --  FLAG
pragma Inline (Swap);

function Increment (I : Integer) return Integer is (I + 1) --  FLAG
  with Inline;
```

3.2.17 Expression_Functions

Flag each expression function declared in a package specification (including specification of local packages and generic package specifications).

Example

```
package Foo is

   function F (I : Integer) return Integer is   --  FLAG
     (if I > 0 then I - 1 else I + 1);
```

3.2.18 Fixed_Equality_Checks

Flag all explicit calls to the predefined equality operations for fixed-point types. Both '=' and '/=' operations are checked. User-defined equality operations are not flagged, nor are uses of operators that are renamings of the predefined equality operations. Also, the '=' and '/=' operations for floating-point types are not flagged.

Example

```
package Pack is
     type Speed is delta 0.01 range 0.0 .. 10_000.0;
     function Get_Speed return Speed;
end Pack;

with Pack; use Pack;
procedure Process is
     Speed1 : Speed := Get_Speed;
     Speed2 : Speed := Get_Speed;

     Flag : Boolean := Speed1 = Speed2;     --  FLAG
```

3.2.19 Float_Equality_Checks

Flag all explicit calls to the predefined equality operations for floating-point types and private types whose completions are floating-point types. Both '=' and '/=' operations are checked. User-defined equality operations are not flagged. Also, the '=' and '/=' operations for fixed-point types are not flagged. Uses of operators that are renamings of the predefined equality operations will be flagged if *Follow_Renamings* is true.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Follow_Renamings: bool
Whether to take renamings of predefined equality operations into account.

Example

```
package Pack is
     type Speed is digits 0.01 range 0.0 .. 10_000.0;
     function Get_Speed return Speed;
end Pack;

with Pack; use Pack;
procedure Process is
     Speed1 : Speed := Get_Speed;
     Speed2 : Speed := Get_Speed;

     Flag : Boolean := Speed1 = Speed2;      --  FLAG
```

3.2.20 Function_Style_Procedures

Flag each procedure that can be rewritten as a function. A procedure can be converted into a function if it has exactly one parameter of mode `out` and no parameters of mode `in out`, with no `Global` aspect specified or with explicit specification that its `Global` aspect is set to `null` (either by aspect specification or by pragma Global). Procedure declarations, formal procedure declarations, and generic procedure declarations are always checked. Procedure bodies and body stubs are flagged only if they do not have corresponding separate declarations. Procedure renamings and procedure instantiations are not flagged.

If a procedure can be rewritten as a function, but its `out` parameter is of a limited type, it is not flagged.

Protected procedures are not flagged. Null procedures also are not flagged.

Example

```
procedure Cannot_be_a_function (A, B : out Boolean);
procedure Can_be_a_function (A : out Boolean);          --  FLAG
```

3.2.21 Generic_IN_OUT_Objects

Flag declarations of generic formal objects of mode IN OUT.

Example

```
generic
   I :        Integer;
   J : in     Integer;
   K : in out Integer;             --  FLAG
package Pack_G is
```

3.2.22 Generics_In_Subprograms

Flag each declaration of a generic unit in a subprogram. Generic declarations in the bodies of generic subprograms are also flagged. A generic unit nested in another generic unit is not flagged. If a generic unit is declared in a local package that is declared in a subprogram body, the generic unit is flagged.

Example

```
procedure Proc is

   generic                                  --  FLAG
      type FT is range <>;
   function F_G (I : FT) return FT;
```

3.2.23 Implicit_IN_Mode_Parameters

Flag each occurrence of a formal parameter with an implicit `in` mode. Note that `access` parameters, although they technically behave like `in` parameters, are not flagged.

Example

```
procedure Proc1 (I :    Integer);          --  FLAG
procedure Proc2 (I : in Integer);
procedure Proc3 (I :    access Integer);
```

3.2.24 Improperly_Located_Instantiations

Flag all generic instantiations in library-level package specs (including library generic packages) and in all subprogram bodies.

Instantiations in task and entry bodies are not flagged. Instantiations in the bodies of protected subprograms are flagged.

Example

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Proc is
   package My_Int_IO is new Integer_IO (Integer);    --  FLAG
```

3.2.25 Library_Level_Subprograms

Flag all library-level subprograms (including generic subprogram instantiations).

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Proc is                          --  FLAG
```

3.2.26 Membership_Tests

Flag use of membership test expression.

This rule has the following (optional) parameters for the +R option and for LKQL rule options files:

Multi_Alternative_Only: bool
If `true`, flag only those membership test expressions that have more than one membership choice in the membership choice list.

Float_Types_Only: bool
If `true`, flag only those membership test expressions that checks objects of floating point type and private types whose completions are floating-point types.

Except_Assertions: bool
If `true`, do not flag a membership test expression if it is a subcomponent of the following constructs:

argument of the following pragmas

Language-defined

- `Assert`

GNAT-specific

- `Assert_And_Cut`
- `Assume`
- `Contract_Cases`
- `Debug`
- `Invariant`
- `Loop_Invariant`
- `Loop_Variant`
- `Postcondition`
- `Precondition`
- `Predicate`
- `Refined_Post`

definition of the following aspects

Language-defined

- `Static_Predicate`
- `Dynamic_Predicate`
- `Pre`
- `Pre'Class`
- `Post`
- `Post'Class`
- `Type_Invariant`
- `Type_Invariant'Class`

GNAT-specific

- `Contract_Cases`
- `Invariant`
- `Invariant'Class`
- `Predicate`
- `Refined_Post`

These three parameters are independent on each other.

Example

```
procedure Proc (S : in out Speed) is
begin
   if S in Low .. High then      --  FLAG
```

3.2.27 Non_Qualified_Aggregates

Flag each non-qualified aggregate. A non-qualified aggregate is an aggregate that is not the expression of a qualified expression. A string literal is not considered an aggregate, but an array aggregate of a string type is considered as a normal aggregate. Aggregates of anonymous array types are not flagged.

Example

```
type Arr is array (1 .. 10) of Integer;

Var1 : Arr := (1 => 10, 2 => 20, others => 30);              --  FLAG
Var2 : array (1 .. 10) of Integer := (1 => 10, 2 => 20, others => 30);
```

3.2.28 Number_Declarations

Number declarations are flagged.

Example

```
Num1 : constant := 13;                  --  FLAG
Num2 : constant := 1.3;                 --  FLAG

Const1 : constant Integer := 13;
Const2 : constant Float := 1.3;
```

3.2.29 Numeric_Indexing

Flag numeric literals, including those preceded by a predefined unary minus, if they are used as index expressions in array components. Literals that are subcomponents of index expressions are not flagged (other than the aforementioned case of unary minus).

Example

```
procedure Proc is
   type Arr is array (1 .. 10) of Integer;
   Var : Arr;
begin
   Var (1) := 10;       --  FLAG
```

3.2.30 Numeric_Literals

Flag each use of a numeric literal except for the following:

- a literal occurring in the initialization expression for a constant declaration or a named number declaration, or
- a literal occurring in an aspect definition or in an aspect clause, or
- an integer literal that is less than or equal to a value specified by the *N* rule parameter, or
- an integer literal that is the right operand of an infix call to an exponentiation operator, or
- an integer literal that denotes a dimension in array types attributes `First`, `Last` and `Length`, or
- a literal occurring in a declaration in case the *Statements_Only* rule parameter is given.

This rule may have the following parameters for the +R option and for LKQL rule options files:

N: int
: An integer literal used as the maximal value that is not flagged (i.e., integer literals not exceeding this value are allowed).

All: bool
: If `true`, all integer literals are flagged.

Statements_Only: bool
: If `true`, numeric literals are flagged only when used in statements.

If no parameters are set, the maximum unflagged value is 1, and the check for literals is not limited by statements only.

The last specified check limit (or the fact that there is no limit at all) is used when multiple +R options appear.

Example

```
C1 : constant Integer := 10;
V1 :          Integer := C1;
V2 :          Integer := 20;      --  FLAG
```

3.2.31 Parameters_Out_Of_Order

Flag each parameter specification if it does not follow the required ordering of parameter specifications in a formal part. The required order may be specified by the following rule parameters:

in
: `in` non-access parameters without initialization expressions;

access
: `access` parameters without initialization expressions;

in_out
: `in out` parameters;

out
: `out` parameters;

defaulted_in
: parameters with initialization expressions (the order of `access` and non-access parameters is not checked.

When the rule is used with parameters, all the five parameters should be given, and each parameter should be specified only once.

The rule can be called without parameters, in this case it checks the default ordering that corresponds to the order in which the rule parameters are listed above.

Example

```
procedure Proc1 (I : in out Integer; B : Boolean) is    --  FLAG
```

3.2.32 Predicate_Testing

Flag a membership test if at least one of its membership choice contains a subtype mark denoting a subtype defined with (static or dynamic) subtype predicate.

Flags 'Valid attribute reference if the nominal subtype of the attribute prefix has (static or dynamic) subtype predicate.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Except_Assertions: bool
: If `true`, do not flag the use of non-short-circuit_operators inside assertion-related pragmas or aspect specifications.

A pragma or an aspect is considered as assertion-related if its name is from the following list:

- `Assert`

- Assert_And_Cut
- Assume
- Contract_Cases
- Debug
- Default_Initial_Condition
- Dynamic_Predicate
- Invariant
- Loop_Invariant
- Loop_Variant
- Post
- Postcondition
- Pre
- Precondition
- Predicate
- Predicate_Failure
- Refined_Post
- Static_Predicate
- Type_Invariant

Example

```
with Support; use Support;
package Pack is
   subtype Even is Integer with Dynamic_Predicate => Even mod 2 = 0;

   subtype Small_Even is Even range -100 .. 100;

   B1 : Boolean := Ident (101) in Small_Even;       --  FLAG
```

3.2.33 Quantified_Expressions

Flag use of quantified expression.

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Except_Assertions: bool
If true, do not flag a conditional expression if it is a subcomponent of the following constructs:

argument of the following pragmas

Language-defined

- Assert

GNAT-specific

- Assert_And_Cut

- Assume
- Contract_Cases
- Debug
- Invariant
- Loop_Invariant
- Loop_Variant
- Postcondition
- Precondition
- Predicate
- Refined_Post

definition of the following aspects

Language-defined

- Static_Predicate
- Dynamic_Predicate
- Pre
- Pre'Class
- Post
- Post'Class
- Type_Invariant
- Type_Invariant'Class

GNAT-specific

- Contract_Cases
- Invariant
- Invariant'Class
- Predicate
- Refined_Post

Example

```
subtype Ind is Integer range 1 .. 10;
type Matrix is array (Ind, Ind) of Integer;

function Check_Matrix (M : Matrix) return Boolean is
  (for some I in Ind =>                              --  FLAG
     (for all J in Ind => M (I, J) = 0));            --  FLAG
```

3.2.34 Raising_Predefined_Exceptions

Flag each `raise` statement that raises a predefined exception (i.e., one of the exceptions `Constraint_Error`, `Numeric_Error`, `Program_Error`, `Storage_Error`, or `Tasking_Error`).

Example

```
begin
   raise Constraint_Error;    --  FLAG
```

3.2.35 Relative_Delay_Statements

Relative delay statements are flagged. Delay until statements are not flagged.

Example

```
if I > 0 then
   delay until Current_Time + Big_Delay;
else
   delay Small_Delay;                    --  FLAG
end if;
```

3.2.36 Renamings

Flag renaming declarations.

Example

```
I : Integer;
J : Integer renames I;     --  FLAG
```

3.2.37 Representation_Specifications

Flag each record representation clause, enumeration representation clause and representation attribute clause. Flag each aspect definition that defines a representation aspect. Also flag any pragma that is classified by the Ada Standard as a representation pragma, and the definition of the corresponding aspects.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Record_Rep_Clauses_Only: bool
If `true`, only record representation clauses are flagged.

Example

```
type State         is (A,M,W,P);
type Mode          is (Fix, Dec, Exp, Signif);

type Byte_Mask     is array (0..7)  of Boolean
  with Component_Size => 1;                                  --  FLAG

type State_Mask    is array (State) of Boolean
  with Component_Size => 1;                                  --  FLAG

type Mode_Mask     is array (Mode)  of Boolean;
for Mode_Mask'Component_Size use 1;                          --  FLAG
```

3.2.38 Separates

Flags subunits.

Example

```
package body P is

   procedure Sep is separate;

end P;

separate(P)        --  FLAG
procedure Sep is
   procedure Q is separate;
begin
   null;
end Sep;
```

3.2.39 Simple_Loop_Statements

Flags simple loop statements (loop statements that do not have iteration schemes).

Example

```
loop                     --  FLAG
   I := I + 1;
   exit when I > 10;
end loop;

while I > 0 loop         --  NO FLAG
   I := I - 1;
end loop;
```

3.2.40 Subprogram_Access

Flag all constructs that belong to access_to_subprogram_definition syntax category, and all access definitions that define access to subprogram.

Example

```
type Proc_A is access procedure ( I : Integer);       --  FLAG

procedure Proc
  (I       : Integer;
   Process : access procedure (J : in out Integer));  --  FLAG
```

3.2.41 Too_Many_Dependencies

Flag a library item or a subunit that immediately depends on more than N library units (N is a rule parameter). In case of a dependency on child units, implicit or explicit dependencies on all their parents are not counted.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
Positive integer specifying the maximal number of dependencies when the library item or subunit is not flagged.

Example

```
--  if rule parameter is 5 or smaller:
with Pack1;
with Pack2;
with Pack3;
with Pack4;
with Pack5;
with Pack6;
procedure Main is                --  FLAG
```

3.2.42 Unassigned_OUT_Parameters

Flag subprograms' `out` parameters that are not assigned.

An `out` parameter is flagged if neither the *declarative part* nor the *sequence of statements* of the subprogram body (before the subprogram body's exception part, if any) contain an assignment to the parameter.

An `out` parameter is flagged if an *exception handler* contains neither an assignment to the parameter nor a raise statement nor a call to a procedure marked No_Return.

Bodies of generic subprograms are also considered.

The following are treated as assignments to an `out` parameter:

- an assignment statement, with the parameter or some component as the target
- passing the parameter (or one of its components) as an `out` or `in out` parameter.

The rule has an optional parameter for the +R option and for LKQL rule options files:

Ignore_Component_Assignments: bool
: Whether to ignore assignments to subcomponents of an `out` parameter when detecting if the parameter is assigned.

Note: An assignment to a subprogram's parameter can occur in the subprogram body's *declarative part* in the presence of a nested subprogram declaration which itself contains an assignment to the enclosing subprogram's parameter.

> **Warning:** This rule only detects the described cases of unassigned variables and doesn't provide a full guarantee that there is no uninitialized access. It is only a partial replacement for the validity checks provided by CodePeer.

Example

```
procedure Proc                  --  FLAG
  (I    : Integer;
   Out1 : out Integer;
   Out2 : out Integer)
is
begin
   Out1 := I + 1;
end Proc;
```

3.2.43 Unconditional_Exits

Flag unconditional `exit` statements.

Example

```
procedure Find_A (S : String; Idx : out Natural) is
begin
   Idx := 0;

   for J in S'Range loop
      if S (J) = 'A' then
         Idx := J;
         exit;              --  FLAG
      end if;
   end loop;
end Find_A;
```

3.2.44 Unconstrained_Array_Returns

Flag each function returning an unconstrained array. Function declarations, function bodies (and body stubs) having no separate specifications, and generic function instantiations are flagged. Function calls and function renamings are not flagged.

Generic function declarations, and function declarations in generic packages, are not flagged. Instead, this rule flags the results of generic instantiations (that is, expanded specification and expanded body corresponding to an instantiation).

This rule has the following (optional) parameter for the +R option and for LKQL rule options files:

Except_String: bool
: If `true`, do not flag functions that return the predefined `String` type or a type derived from it, directly or indirectly.

Example

```
type Arr is array (Integer range <>) of Integer;
subtype Arr_S is Arr (1 .. 10);

function F1 (I : Integer) return Arr;      --  FLAG
function F2 (I : Integer) return Arr_S;
```

3.2.45 Unconstrained_Arrays

Unconstrained array definitions are flagged.

Example

```
type Idx is range -100 .. 100;

type U_Arr is array (Idx range <>) of Integer;      --  FLAG
type C_Arr is array (Idx) of Integer;
```

3.2.46 USE_Clauses

Flag names mentioned in use clauses. Use type clauses and names mentioned in them are not flagged.

This rule has the following optional parameter for the +R option and for LKQL rule options files:

Exempt_Operator_Packages: bool
: If `true`, do not flag a package name in a package use clause if it refers to a package that only declares operators in its visible part.

Note: This rule has another parameter, only available when using an LKQL rule options file: `allowed`. It is a list of Ada names describing packages to exempt from begin flagged when used in "use" clauses. Strings in this list are case insensitive. Example:

```
val rules = @{
   Use_Clauses: {Allowed: ["Ada.Strings.Unbounded", "Other.Package"]}
}
```

Example

```
package Pack is
   I : Integer;
end Pack;

package Operator_Pack is
   function "+" (L, R : Character) return Character;
end Operator_Pack;

with Pack, Operator_Pack;
use Pack;                    --  FLAG if "Pack" is not in Allowed
use Operator_Pack;           --  FLAG only if Exempt_Operator_Packages is false
```

3.3 Metrics-Related Rules

The rules in this section can be used to enforce compliance with specific code metrics, by checking that the metrics computed for a program lie within user-specifiable bounds.

The name of any metrics rule consists of the prefix `Metrics_` followed by the name of the corresponding metric: `Essential_Complexity`, `Cyclomatic_Complexity`, or `LSLOC`. (The 'LSLOC' acronym stands for 'Logical Source Lines Of Code'.) The meaning and the computed values of the metrics are the same as in *gnatmetric*.

3.3.1 Metrics_Cyclomatic_Complexity

The `Metrics_Cyclomatic_Complexity` rule takes a positive integer as upper bound. A program unit that is an executable body exceeding this limit will be flagged.

This rule has the following parameters for the `+R` option and for LKQL rule options files:

N: int
: Maximum cyclomatic complexity a body can have without being flagged, all bodies with a higher index will be flagged.

Exempt_Case_Statements: bool
: Whether to count the complexity introduced by `case` statement or `case` expression as 1.

The McCabe cyclomatic complexity metric is defined in http://www.mccabe.com/pdf/mccabe-nist235r.pdf The goal of cyclomatic complexity metric is to estimate the number of independent paths in the control flow graph that in turn gives the number of tests needed to satisfy paths coverage testing completeness criterion.

Example

```
--  if the rule parameter is 6 or less
procedure Proc (I : in out Integer; S : String) is   --  FLAG
begin
   if I in 1 .. 10 then
      for J in S'Range loop

         if S (J) = ' ' then
            if I < 10 then
               I := 10;
            end if;
         end if;

         I := I + Character'Pos (S (J));
      end loop;
   elsif S = "abs" then
      if I > 0 then
         I := I + 1;
      end if;
   end if;
end Proc;
```

3.3.2 Metrics_Essential_Complexity

The Metrics_Essential_Complexity rule takes a positive integer as upper bound. A program unit that is an executable body exceeding this limit will be flagged.

The Ada essential complexity metric is a McCabe cyclomatic complexity metric counted for the code that is reduced by excluding all the pure structural Ada control statements.

This rule has the following (mandatory) parameter for the +R option and for LKQL rule options files:

N: int
: Maximum essential complexity a body can have without being flagged, all bodies with a higher index will be flagged.

Example

```
--  if the rule parameter is 3 or less
procedure Proc (I : in out Integer; S : String) is   --  FLAG
begin
   if I in 1 .. 10 then
      for J in S'Range loop

         if S (J) = ' ' then
            if I > 10 then
               exit;
            else
               I := 10;
            end if;
         end if;
```

(continues on next page)

(continued from previous page)

```
         I := I + Character'Pos (S (J));
      end loop;
   end if;
end Proc;
```

3.3.3 Metrics_LSLOC

The Metrics_LSLOC rule takes a positive integer as upper bound. A program unit declaration or a program unit body exceeding this limit will be flagged.

The metric counts the total number of declarations and the total number of statements.

This rule has the following parameters for the +R option and for LKQL rule options files:

N: int
Maximum number of logical source lines a program declaration / body can have without being flagged, all bodies with a higher number of LSLOC will be flagged.

Subprograms: bool
Optional parameter specifying whether to check the rule for subprogram bodies only.

Example

```
--  if the rule parameter is 20 or less
package Pack is                              --  FLAG
   procedure Proc1 (I : in out Integer);
   procedure Proc2 (I : in out Integer);
   procedure Proc3 (I : in out Integer);
   procedure Proc4 (I : in out Integer);
   procedure Proc5 (I : in out Integer);
   procedure Proc6 (I : in out Integer);
   procedure Proc7 (I : in out Integer);
   procedure Proc8 (I : in out Integer);
   procedure Proc9 (I : in out Integer);
   procedure Proc10 (I : in out Integer);
end Pack;
```

3.4 SPARK-Related Rules

The rules in this section can be used to enforce compliance with the Ada subset allowed by the SPARK 2005 language.

More recent versions of SPARK support these language constructs, so if you want to further restrict the SPARK constructs allowed in your coding standard, you can use some of the following rules.

3.4.1 Annotated_Comments

Flags comments that are used as annotations or as special sentinels/markers. Such comments have the following structure:

```
--<special_character> <comment_marker>
```

where

<*special_character*> is a character (such as '#', '$', '|' etc.)
indicating that the comment is used for a specific purpose

<*comment_marker*> is a word identifying the annotation or special usage
(word here is any sequence of characters except white space)

There may be any amount of white space (including none at all) between <special_character> and <comment_marker>, but no white space is permitted between '--' and <special_character>. (A white space here is either a space character or horizontal tabulation)

<comment_marker> must not contain any white space.

<comment_marker> may be empty, in which case the rule flags each comment that starts with --<special_character> and that does not contain any other character except white space

The rule has the following mandatory parameter for the +R option and for LKQL rule options files:

S: list[string]
List of string with the following interpretation: the first character is the special comment character, and the rest is the comment marker. Items must not contain any white space.

The rule is case-sensitive.

Example:

The rule

```
+RAnnotated_Comments:#hide
```

will flag the following comment lines

```
--#hide
--# hide
--#           hide

   I := I + 1; --# hide
```

But the line

```
-- # hide
```

will not be flagged, because of the space between '–' and '#'.

The line

```
--#Hide
```

will not be flagged, because the string parameter is case sensitive.

3.4.2 Boolean_Relational_Operators

Flag each call to a predefined relational operator ('<', '>', '<=', '>=', '=' and '/=') for the predefined Boolean type. (This rule is useful in enforcing the SPARK language restrictions.)

Calls to predefined relational operators of any type derived from `Standard.Boolean` are not detected. Calls to user-defined functions with these designators, and uses of operators that are renamings of the predefined relational operators for `Standard.Boolean`, are likewise not detected.

Example

```
   procedure Proc (Flag_1 : Boolean; Flag_2 : Boolean; I : in out Integer) is
   begin
      if Flag_1 >= Flag_2 then      --  FLAG
```

3.4.3 Expanded_Loop_Exit_Names

Flag all expanded loop names in `exit` statements.

Example

```
procedure Proc (S : in out String) is
begin
   Search : for J in S'Range loop
      if S (J) = ' ' then
         S (J) := '_';
         exit Proc.Search;             --  FLAG
      end if;
   end loop Search;
end Proc;
```

3.4.4 Non_SPARK_Attributes

The SPARK language defines the following subset of Ada 95 attribute designators as those that can be used in SPARK programs. The use of any other attribute is flagged.

- 'Adjacent
- 'Aft
- 'Base
- 'Ceiling
- 'Component_Size
- 'Compose
- 'Copy_Sign
- 'Delta
- 'Denorm
- 'Digits

- 'Exponent
- 'First
- 'Floor
- 'Fore
- 'Fraction
- 'Last
- 'Leading_Part
- 'Length
- 'Machine
- 'Machine_Emax
- 'Machine_Emin
- 'Machine_Mantissa
- 'Machine_Overflows
- 'Machine_Radix
- 'Machine_Rounds
- 'Max
- 'Min
- 'Model
- 'Model_Emin
- 'Model_Epsilon
- 'Model_Mantissa
- 'Model_Small
- 'Modulus
- 'Pos
- 'Pred
- 'Range
- 'Remainder
- 'Rounding
- 'Safe_First
- 'Safe_Last
- 'Scaling
- 'Signed_Zeros
- 'Size
- 'Small
- 'Succ
- 'Truncation

- 'Unbiased_Rounding
- 'Val
- 'Valid

Example

```
type Integer_A is access all Integer;

Var : aliased Integer := 1;
Var_A : Integer_A := Var'Access;  --  FLAG
```

3.4.5 Non_Tagged_Derived_Types

Flag all derived type declarations that do not have a record extension part.

Example

```
type Coordinates is record
   X, Y, Z : Float;
end record;

type Hidden_Coordinates is new Coordinates;    --  FLAG
```

3.4.6 Outer_Loop_Exits

Flag each `exit` statement containing a loop name that is not the name of the immediately enclosing `loop` statement.

Example

```
Outer : for J in S1'Range loop
   for K in S2'Range loop
      if S1 (J) = S2 (K) then
         Detected := True;
         exit Outer;                       --  FLAG
      end if;
   end loop;
end loop Outer;
```

3.4.7 Overloaded_Operators

Flag each function declaration that overloads an operator symbol. A function body or an expression function is checked only if it does not have a separate spec. Formal functions are also checked. For a renaming declaration, only renaming-as-declaration is checked.

Example

```
type Rec is record
   C1 : Integer;
   C2 : Float;
end record;

function "<" (Left, Right : Rec) return Boolean;     --  FLAG
```

3.4.8 Slices

Flag all uses of array slicing

Example

```
procedure Proc (S : in out String; L, R : Positive) is
   Tmp : String := S (L .. R);         --  FLAG
begin
```

3.4.9 Universal_Ranges

Flag discrete ranges that are a part of an index constraint, constrained array definition, or `for`-loop parameter specification, and whose bounds are both of type *universal_integer*. Ranges that have at least one bound of a specific type (such as `1 .. N`, where `N` is a variable or an expression of non-universal type) are not flagged.

Example

```
L : Positive := 1;

S1 : String (L .. 10);
S2 : String (1 .. 10);      --  FLAG
```

CHAPTER

FOUR

WRITING YOUR OWN RULES

All the predefined rules in GNATcheck are implemented via a pattern matching language called LKQL (LangKit Query Language) which is a functional, Turing-complete language, and provides as an advanced usage the ability to write an infinite number of custom checks.

The general description of this language can be found in the *LKQL Language Reference*. The APIs available in LKQL are described in the *LKQL API*.

In addition to the examples available in this document, you will find all the predefined checks implemented in GNATcheck under the `share/lkql` directory which can be used as examples and starting points for your own rules.

This chapter gives some additional information on how to use this language to create your own rules and checkers.

4.1 How To Write Rules

Rules are written in the LKQL language, and put either in the predefined `share/lkql` directory or under any other directory specified via the `--rules-dir` switch. Each `.lkql` file found in these directories will be loaded by gnatcheck and represents a distinct rule (or a set of helper functions). The naming convention of the rules is `lowercase_with_underscores`.

Here is a simple rule example, that will just flag every body node:

```
@check
fun bodies(node) = node is BodyNode
```

Adding this source in the `bodies.lkql` file in a directory listed via `--rules-dir` will add a rule to GNATcheck dynamically, without the need to modify GNATcheck itself. For example:

```
$ gnatcheck -Pprj --rules-dir=. -r bodies
```

4.1.1 Boolean Rules

Boolean rules are functions that take a node and return a boolean and are marked with the `@check` decorator. They usually contain an `is` pattern match as the main expression:

```
@check
fun goto(node) = node is GotoStmt
```

But are not limited to this, and can contain arbitrary expressions as long as they return a boolean, e.g:

```
@check
fun goto_or_loop(node) =
    match node
    | GotoStmt     => true
    | BaseLoopStmt => true
    | *            => false
```

4.1.2 Unit Rules

Unit rules are functions that take an analysis unit and return a list of objects containing a message and a location. They're meant to be ultimately flexible, and fulfill the needs that boolean rules can't fulfill, for example:

- Customizing messages.
- Having a non 1 to 1 relationship between messages and nodes.
- Having token based rules.

The returned objects must have two keys:

- `message`: Contains the message to be displayed.
- `loc`: Either a node or a token, used as the source location for the error message.

These functions are marked with the `@unit_check` decorator:

```
@unit_check
fun goto_line(unit) = [
   {message:
      "go to line " &
      img(node.f_label_name.p_referenced_decl().token_start().start_line),
    loc: node.f_label_name}
   for node in (from unit.root select GotoStmt)
]
```

The above rule will find each goto statement and generate a message for each, listing the line where the target label of the goto is defined.

For example given this code:

```
1 procedure Go_To is
2 begin
3    goto Foo;
4    ...
5 <<Foo>>
6    ...
7 end Go_To;
```

The following gnatcheck call (assuming the file `goto_line.lkql` is found in the current directory) will output:

```
$ gnatcheck -d go_to.adb --rules-dir=. -r goto_line
go_to.adb:3:09: go to line 5
```

4.1.3 Rule Arguments

Rules can take different optional arguments:

- `message`: The custom message that is to be shown for a given rule on the command line. Defaults to the name of the rule if not specified.
- `help`: The help message that is to be shown via `gnatcheck --list-rules`. Defaults to message if not specified.
- `follow_generic_instantiations`: Whether to follow generic instantiations during the traversal of given Ada units. If `true`, generic instantiations will be traversed in instantiated form. Defaults to `false`.
- `category`, `subcategory`: The category (and subcategory) associated with this rule, used by gnatcheck as part of its `-hx` output. Defaults to `Misc`.
- remediation: A string with the following possible values:
  - EASY
  - MEDIUM
  - MAJOR

  Used by `gnatcheck --list-rules` and by the SonarQube integration to compute technical debt. Defaults to *MEDIUM*.

Here is an example rule:

```
@check(message="integer object declaration", follow_generic_instantiations=true)
fun int_obj_decl(node) =
    |" Will flag object declarations for which the type is the standard
    |" ``Integer`` type
    node is o@ObjectDecl(
        p_type_expression(): SubtypeIndication(
            p_designated_type_decl(): t@* when t == o.p_std_entity("Integer")
        )
    )
```

4.2 Debugging Your Rules

When writing new rules, you should first enable the gnatcheck switch `-d` so that any LKQL runtime error (such as type mismatches, wrong nodes or syntax errors) are reported as part of the gnatcheck output.

You can then use one (or a mix) of the approaches described in the following sections.

4.2.1 The LKQL interpreter

LKQL comes with a command line interpreter. The interpreter is part of the `lkql` executable, shipped with your GNATcheck distribution. `lkql` has an interpreter mode, that you can access via the `lkql run` command:

```
$ lkql run -S my_script.lkql [-P my_project] [sources]
```

This is different from how GNATcheck interfaces with the LKQL language, because GNATcheck only executes LKQL code through the *check* functions, whereas with the interpreter you can execute arbitrary LKQL code. Here is an example LKQL script:

```
# Prints the first basic declaration
print(select first BasicDecl)
```

This interpreter has an interactive mode, which is also commonly known as an REPL (Read-Eval-Print-Loop). This REPL allows you to elaborate and verify all your LKQL expressions line by line, as well as explore the available properties and functions via the code completion provided by this interactive environment.

```
$ lkql run -i -P prj
```

where prj is your project file `prj.gpr`. From there you have access to an interactive shell which provides a history of commands available via e.g. the up and down keys, as well as automatic completion. To exit this shell, you can use the `Control-D` key combination.

Here is an example session:

```
$ lkql run -i -Pprj

.-.   .-. .-..----. .-.
| |   | |/ //  {}  \| |       Welcome to LKQL repl
| `--.| |\ \\      /| `--.    type 'help' for more information
`----'`-' `-'`-----``----'

 > val root=select first AdaNode
()
 > print(root)
<CompilationUnit file1.adb:1:1-41:11>
()
 > root.dump
CompilationUnit[1:1-41:11]
|f_prelude:
|  AdaNodeList[1:1-1:1]: <empty list>
|f_body:
|  LibraryItem[1:1-41:11]
[...]
 > val ops=select BinOp
()
 > print ops
[<BinOp file1.adb:3:54-3:59>, <RelationOp file1.adb:6:56-6:62>, ...]
 > ops[1].dump
BinOp[3:54-3:59]
|f_left:
|  Id[3:54-3:55]: L
|f_op:
|  OpMinus[3:56-3:57]
|f_right:
|  Id[3:58-3:59]: R
()
 > print ops[1].f_left
<Id "L" file1.adb:3:54-3:55>
 > print ops[1].f_left.p_referenced_decl()
<ParamSpec ["L", "R"] file1.adb:3:19-3:33>
 > select ParamSpec
[<ParamSpec ["L", "R"] file1.adb:2:19-2:33>, <ParamSpec ["L", "R"] file1.adb:3:19-3:33>,␣
↪...]
```

(continues on next page)

(continued from previous page)

```
 > select p@ParamSpec when [n for n in p.f_ids.children if n.f_name.p_name_is("Str")]
[<ParamSpec ["Str"] file1.adb:1:18-1:37>, <ParamSpec ["Str"] file2.adb:1:18-1:37>]
 > ^D
Do you really want to exit ([y]/n)? y
```

4.2.2 Print Technique

Another option to verify at various steps that your rule is doing the right thing is to insert calls to `print`, `dump` or `img` functions in a block:

```
fun do_this(node) = {
    print(node);
    print("parent node is: " & img(node.parent));
    node.parent.dump;
    do_that()
}
```

Inside a boolean expression, you can also insert a call to `print` or `dump` which will always evaluate to `false`:

```
node is GotoStmt and (print(node) or real_expression())
```

Note that print statements will be output immediately on standard output, while gnatcheck messages are stored internally and dumped at the end. In addition, the default gnatcheck output may interfere with your print statements, so it is recommended to use the `-v` or `--brief` switches to avoid or reduce the interference.

4.3 A Complete Step By Step Example

In this section, we will implement step by step a rule to detect integer types that could be replaced by an enumeration type.

To find such types, we first need to define a `@check` looking for all type declarations, with an associated message:

```
@check(message="integer type may be replaced by an enumeration")
fun integer_types_as_enum(node) = node is TypeDecl
```

Then let's refine the rule to only consider integer type declarations, by using the libadalang `p_is_int_type` property:

```
@check(message="integer type may be replaced by an enumeration")
fun integer_types_as_enum(node) = node is TypeDecl(p_is_int_type(): true)
```

Now, we'll add a first criteria to consider: there should be no use of any arithmetic or bitwise operator on this type anywhere in the sources. To achieve that, we need to perform a global query on the whole project, which is done via a `select` query, to find all the references to arithmetic operators:

```
select (
    BinOp(f_op: OpDiv | OpMinus | OpMod | OpMult | OpPlus | OpPow | OpRem | OpXor |␣
↪OpAnd | OpOr)
    | UnOp(f_op: OpAbs | OpMinus | OpPlus | OpNot)
)
```

we then create a function that will compute all the types associated with these expressions in a list:

```
fun arithmetic_ops() =
    |" Return a list of all types referenced in any arithmetic operator
    [op.p_expression_type()
     for op in select (
         BinOp(f_op: OpDiv | OpMinus | OpMod | OpMult |
                     OpPlus | OpPow | OpRem | OpXor |
                     OpAnd | OpOr)
         | UnOp(f_op: OpAbs | OpMinus | OpPlus | OpNot)
     )
    ].to_list
```

and we update our rule accordingly to find all integer types for which no arithmetic operator is found. To achieve that, we use a list comprehension to iterate over the list returned by `arithmetic_ops` and take advantage of the semantic of list comprehensions when used in a boolean expression: a list with no element evaluates to `false`, and a list with at least one element evaluates to `true`:

```
fun integer_types_as_enum(node) =
     node is TypeDecl(p_is_int_type(): true)
     when not [t for t in arithmetic_ops() if t == node]
```

Running this rule we realize that it finds some interesting matches, but also too many false positives. In particular types referenced in type conversions also need to be filtered out. So let's define another helper function that will list all types referenced as a target of a type conversion. In the libadalang tree, a type conversion appears as a `CallExpr` whose referenced declaration (`p_referenced_decl` property) is a type declaration (`TypeDecl`). We perform another global `select` query:

```
fun types() =
    [c.p_referenced_decl()
     for c in select CallExpr(p_referenced_decl(): TypeDecl)].to_list
```

And we update our rule accordingly:

```
fun integer_types_as_enum(node) =
     node is TypeDecl(p_is_int_type(): true)
     when not [t for t in arithmetic_ops() if t == node]
      and not [t for t in types() if t == node]
```

So we're now filtering target types in type conversions, but that's not enough, we also need to filter source types in type conversions, so let's refine our `types` function by also using the `f_suffix` which is a `ParamAssocList` in this context with a single element, where we compute the type of the expression via the `p_expression_type` property:

```
c.f_suffix[1].f_r_expr.p_expression_type()
```

We then use the `concat` builtin function to concatenate the previous result with this new one and create a single dimension list of type declarations with both source and target types of conversions:

```
fun types() =
    concat([[c.p_referenced_decl(), c.f_suffix[1].f_r_expr.p_expression_type()]
            for c in select CallExpr(p_referenced_decl(): TypeDecl)].to_list)
```

This gives much better results and much fewer false positives! We then realize that we need to perform a similar filtering on subtype declarations: types references in subtype declarations should not be flagged. We use another global `select` on subtype declarations, and list all the referenced types:

```
[s.f_subtype.f_name.p_referenced_decl() for s in select SubtypeDecl]
```

We combine this with the previous results:

```
fun types() =
    |" Return a list of TypeDecl matching all type conversions (both as source
    |" and target) and subtype declarations in the project.
    concat([[c.p_referenced_decl(), c.f_suffix[1].f_r_expr.p_expression_type()]
            for c in select CallExpr(p_referenced_decl(): TypeDecl)].to_list)
    & [s.f_subtype.f_name.p_referenced_decl() for s in select SubtypeDecl].to_list
```

We're getting even less false positives now, and quickly realize that we need to do the same for type derivations:

```
[c.f_type_def.f_subtype_indication.f_name.p_referenced_decl()
 for c in select TypeDecl(f_type_def: DerivedTypeDef)].to_list
```

We combine again the results, which gives us our final `types` function:

```
fun types() =
    |" Return a list of TypeDecl matching all type conversions (both as source
    |" and target), subtype declarations and type derivations in the project.
    concat([[c.p_referenced_decl(), c.f_suffix[1].f_r_expr.p_expression_type()]
            for c in select CallExpr(p_referenced_decl(): TypeDecl)].to_list)
    & [s.f_subtype.f_name.p_referenced_decl() for s in select SubtypeDecl].to_list
    & [c.f_type_def.f_subtype_indication.f_name.p_referenced_decl()
       for c in select TypeDecl(f_type_def: DerivedTypeDef)].to_list
```

Running our rule again, we find a final source of false positives: types referenced as parameter of generic instantiations also need to be filtered out, so we define a new function to compute all declarations referenced as parameters of a generic instantiation, via two `select`: a global query returning all generic instantiations:

```
select GenericInstantiation
```

and we then inject the result of this query into another select to list all identifiers referenced by all these instantiations:

```
from (select GenericInstantiation) select Identifier
```

which gives us the following function:

```
fun instantiations() =
    |" Return a list of all declarations referenced in any generic instantiation
    [id.p_referenced_decl()
     for id in from (select GenericInstantiation) select Identifier].to_list
```

Updating our rule this gives us:

```
fun integer_types_as_enum(node) =
     node is TypeDecl(p_is_int_type(): true)
     when not [t for t in arithmetic_ops() if t == node]
      and not [t for t in types() if t == node]
      and not [t for t in instantiations() if t == node]
```

That's good enough in terms of results, but we also realize that running this rule is very slow, so let's look at how to optimize it.

The first thing to do is to avoid repeated calls to the very costly global select contained in functions `arithmetic_ops`, `types` and `instantiations`. We achieve that easily by marking our functions with the `@memoized` decorator, so that these function calls will be cached after the first evaluation. In addition, to avoid checking multiple times the same type declarations, we can take advantage of the `unique` builtin in each of our helper function, e.g:

```
@memoized
fun instantiations() =
    unique([id.p_referenced_decl()
            for id in from (select GenericInstantiation) select Identifier].to_list)
```

Finally, we notice that there are many more arithmetic operators to check in a project than type conversion or generic instantiations, so we swap the order of the tests:

```
fun integer_types_as_enum(node) =
     node is TypeDecl(p_is_int_type(): true)
     when not [t for t in types() if t == node]
      and not [t for t in instantiations() if t == node]
      and not [t for t in arithmetic_ops() if t == node]
```

which gives us this complete rule:

```
@memoized
fun arithmetic_ops() =
    |" Return a list of all types referenced in any arithmetic operator
    unique([op.p_expression_type()
            for op in select (
                BinOp(f_op: OpDiv | OpMinus | OpMod | OpMult |
                            OpPlus | OpPow | OpRem | OpXor |
                            OpAnd | OpOr) |
                UnOp(f_op: OpAbs | OpMinus | OpPlus | OpNot))].to_list)

@memoized
fun instantiations() =
    |" Return a list of all declarations referenced in any generic instantiation
    unique([id.p_referenced_decl()
            for id in from (select GenericInstantiation) select Identifier].to_list)

@memoized
fun types() =
    |" Return a list of TypeDecl matching all type conversions (both as source
    |" and target), subtype declarations and type derivations in the project.
    unique(concat([[c.p_referenced_decl(),
                    c.f_suffix[1].f_r_expr.p_expression_type()]
                   for c in select CallExpr(p_referenced_decl(): TypeDecl)].to_list) &
           [s.f_subtype.f_name.p_referenced_decl() for s in select SubtypeDecl].to_list &
           [c.f_type_def.f_subtype_indication.f_name.p_referenced_decl()
            for c in select TypeDecl(f_type_def: DerivedTypeDef)].to_list)

@check(message="integer type may be replaced by an enumeration")
fun integer_types_as_enum(node) =
     node is TypeDecl(p_is_int_type(): true)
     when not [t for t in types() if t == node]
      and not [t for t in instantiations() if t == node]
      and not [t for t in arithmetic_ops() if t == node]
```

CHAPTER

FIVE

LKQL LANGUAGE REFERENCE

LKQL (short for LangKit Query Language) is a query language enabling users to run queries on top of source code.

LKQL today is the mixture of two language subsets:

- The first is a dynamically typed, functional, small but general purpose programming language, including function definitions, common expressions, very basic support for numeric types and computations, list comprehensions, etc.
- The second is a tree query language, allowing the user to express very concisely predicates over a node and its syntactic and semantic relatives, and tree traversal logics.

Those two subsets will be documented separately. The general language will be documented first, because its knowledge is needed for understanding the tree query language.

LKQL is based upon the langkit technology. As such, it is theoretically capable of running queries on any language with a Langkit frontend. In practice for the moment, LKQL is hardwired for Ada (and Libadalang).

Attention: While mostly stable, LKQL is not completely done yet. The language will keep being extended with new constructs, and from time to time syntax might change to accommodate new language constructs/enhance the language ergonomics/fix design mistakes. Read the *Language Changes* section for more information.

5.1 General Purpose Language Subset

This language subset is composed of a reduced set of declarations and expressions that forms a minimal but turing complete language.

For the time being, *it has no side effects*, which is intended since the purpose of LKQL is strictly to express queries.

5.1.1 Data Types

LKQL has a very limited number of data types for the moment. Here are the current data types:

5.1.1.1 Basic Data Types

`Unit`
A type used to represents empty values.

`Int`
Basic integer type, supporting arbitrary sized values. Supports simple arithmetic.

`Str`
Built-in string type, that supports concatenation.

`Bool`
Built-in boolean type, that supports the usual expected boolean relational operators.

`Node`
Coming from the queried language (in the common case, Ada). Nodes correspond to the syntax nodes of the source files being queried. They can be explored as part of the general language subset, through *Field access*, or via the *Query language subset*.

`Token`
Also coming from the queried language. Tokens correspond to lexical units of the queried source files.

`Pattern`
Values of this type are compiled regular expressions that can be used in a few contexts to match a string against, notably in the string built-in functions `contains` and `find`.

`Function`
LKQL functions are first class citizens, thus, any expression can has this type which represents values that can be called with a *call expression*.

5.1.1.2 Composite Data Types

`Tuple`
Tuples are heterogeneous groups of values with a fixed size. They can be indexed to access inner values, a bit like Python tuples.

`List`
Lists are contiguous immutable sequences of items that can be indexed. Lists also support concatenation.

`Stream`
Streams are lazy sequences of items, which means an element will not be computed until it is observed. They can also be indexed (which will force the computation until the indexed item).

> **Attention:** Tuples, Lists and Streams are indexed starting from 1, like in Lua/R/.., unlike in Python/Java/..

`Object`
Objects are heterogeneous records that can contain any number of key to value mappings, where keys are labels and values are any valid LKQL value.

5.1.2 Declarations

Declarations in LKQL only belong at the top level. There is no support currently for nested declarations, except in the *block expression*.

5.1.2.1 Function Declaration

Allows the user to declare an LKQL function that can be used to factor some computation:

```
fun add(x, y) = x + y
```

The syntax is simple, you only declare argument names and an expression after the =.

If you need to declare temporary named values in the body of your function, you can use a *block expression*:

```
fun add(x, y) = {
    |" Add two integers
    val ret = x + y;
    ret
}
```

Note: A function can have annotations. For the moment, this is used only in the context of LKQL checkers:

```
@check(message="Bla detected")
fun is_bla() = node is Bla
```

Functions can also be nested in other functions, and closures are allowed, ie. you can return a function that references the environment in which it was declared:

```
fun make_closure(closure_var) = {
    fun use_closure() = closure_var + 1;
    use_closure
}

# This will display the functional value "use_closure"
print(make_closure(12))
```

Note: Functions can be memoized via the @memoized annotation. In a language such as lkql that is purely functional, this will give a way for users to express/optimize computationally expensive things. Here is a simple example:

```
@memoized
fun fib(a) =
    if a == 0 then 0
    else (if a == 1 then 1
          else fib(a - 1) + fib (a - 2))

print(fib(30))
```

5.1.2.2 Value Declaration

Declare a named value (often called a variable or constant in other languages):

```
val a = 12 + 15
```

Note that the value is immutable.

5.1.2.3 Docstrings

Declarations can have assorted docstrings. They're part of the AST and are directly attached to the declaration:

```
# Docstrings

fun make_closure(closure_var) =
|" Make a function that will capture ``closure_var`` and return the sum of
|" it plus its first argument
{
    fun use_closure(x) = closure_var + x;
    use_closure
}

|" Function that will add 12 to its first argument
val adder = make_closure(12)

print(make_closure(12))
```

5.1.3 Expressions

5.1.3.1 Block Expression

The block expression is useful to declare temporary named values and execute intermediate expressions. This can be useful to share the result of a temporary calculation, to name an intermediate value to make the code more readable, or to print debug values:

```
{
    val x = 40;
    val y = 2;
    print("DEBUG : " & (x + y).img);
    x + y
}
```

As you can see in the example above, value declarations and intermediate expressions are ended by semicolons. After the last one, you write the block's result expression, without an ending semicolon.

5.1.3.2 Field Access

A field access returns the contents of a field. In the following example, we get the content of the `f_type_expr` syntax field on a node of type `ObjectDecl`:

```
object_decl.f_type_expr
```

A regular field access on a nullable variable is illegal and leads to a runtime error, which is why field access has a variant, which is called a "safe access":

```
object_decl?.f_type_expr
```

The safe access will return null if the left hand side is null. This allows users to chain accesses without having to checks for nulls at every step.

In the context of rewriting features usage, you may want to get a reference to a field of a node. You can access such references with a dot-access notation on node kinds:

```
val ref_to_f_child = MyNodeKind.f_child
```

Such values can be used when calling `RewritingContext`'s methods.

5.1.3.3 Unwrap Expression

When you have a nullable object and you want to make it non nullable, you can use the unwrap expression. This is useful after a chain of safe accesses/calls, for example:

```
object_decl?.p_type_expr()?.p_designated_type_decl()!!
```

Unwrap will raise an error if the value is null.

5.1.3.4 Call Expression

LKQL values of the `Function` type can be invoked with the call expression:

```
fun add(a, b) = a + b

val c = add(12, 15)
val d = add(a=12, b=15)
```

Parameters can be passed via positional or named associations.

Calls have a "safe" variant, that will return `null` if the callee is null:

```
fun add(a, b) = a + b
val fn = if true then null else add
fn?(1, 2) # Returns null
```

Additionally, you can also call selectors via the call syntax. Selector calls take only one argument, which is the starting point of the selector call chain:

```
children(select first AdaNode)
```

5.1.3.5 Constructor call

You can call node constructors to create new nodes possibly used for the tree rewriting layer of LKQL. The result of a constructor call is a value of the `RewritingNode` type.

```
val token_node = new BooleanLiteral("Hello!")
val list_node = new SomeListNode(child_1, child_2)
val composite_node = new CompositeNode(
    f_child_1=token_node,
    f_child_2=list_node
)
```

As function calls, you can pass arguments via positional or named associations for composite nodes. About token and list nodes, you may only pass arguments through the positional format.

To know whether a node is a token, list or composite one, you may refer to the Langkit specification of the language you're querying.

5.1.3.6 Indexing Expression

Indexing expression allows the user to access elements of a `Tuple`, `List`, `Stream`, or `Node`.

When using the indexing expression on a node value:

- for list nodes, it will access the different elements of the list
- for regular nodes, it will access children in lexical order

Here are some examples of indexing expressions:

```
# Indexing a tuple
(1, 2, 3)[1]

# Indexing a list
list[1]

# Indexing a node with an arbitrary index
{
    val x = 2;
    node[x]
}
```

Indexing also has a safe variant, that will return `unit` instead of raising when an out of bound access is done:

```
val lst = [1, 2, 3]

# This will display "()"
print(lst?[5])
```

5.1.3.7 Comparison Expression

Comparison expressions are used to compare an object to another object, or pattern. All those constructions are evaluated as booleans.

Membership Expression

The membership expression verifies that a collection (`List`/`Stream`) contains the given value:

```
12 in list
```

Is Expression

The `is` expression verifies if a value matches a given *pattern*:

```
val a = select AdaNode
val b = a[1] is ObjectDecl
```

Comparison Operators

The usual comparison operators are available:

```
12 < 15
a == b
b != c
```

Order dependent operators (</>/...) are only usable on integers.

5.1.3.8 Object Literal

An object literal is a literal representation of an object value:

```
# Object literal
{a: 1, b: "foo", c: null, d: [1, 2, 3, 4]}
```

"@" object literals are quite the same as standard objects literals, but each associated value is wrapped in a list (if not already one). You are also allowed to omit the associated expression when adding a key in the object. The default associated value is a list with only one element: an empty object.

```
# @-object literal
@{a: "Hello", b, c: 42, d}

# Is similar to
{a: ["Hello"], b: [{}], c: [42], d: [{}]}
```

This "@" object notation are mainly used to express coding standards in LKQL rule configuration files, however, you can use it in any context.

Object keys may contain upper-case characters at declaration, but the LKQL engine will lower them. This means that object keys are case-insensitive:

```
val o = {lower: "Hello", UPPER: "World"}

# This will display "Hello World"
print(o.lower & " " & o.upper)
```

Please note that objects are immutable.

5.1.3.9 List Literal

A list literal is simply a literal representation of a list:

```
# Simple list literal
[1, 2, 3, 4]
```

Lists being immutable, lists literals are the primary way to create new lists from nothing, with *list comprehension* being the way to create new lists from existing lists.

5.1.3.10 List Comprehension

A list comprehension allows the user to create a new list by iterating on an existing collection, applying a mapping operation, and eventually a filtering logic:

```
# Simple list comprehension that'll double every number in int_list if it
# is prime
[a * 2 for a in int_list if is_prime(a)]

# Complex example interleaving two collections
val subtypes = select SubtypeIndication
val objects = select ObjectDecl
print(
    [
        o.image & " " & st.image
        for o in objects, st in subtypes
        if (o.image & " " & st.image).length != 64
    ].to_list
)
```

A list comprehension is a basic language construct, that, since LKQL is purely functional, replaces traditional for loops. A list comprehension expression returns a value of the `Stream` type, meaning that elements in the result aren't computed until queried:

```
val lazy = [a * 2 for a in int_list if is_prime(a)]

# This will display "Stream"
print(lazy)

# To display all elements of a stream, you have to convert it to a list
print(lazy.to_list)
```

5.1.3.11 If Expression

If expressions are traditional conditional expressions composed of a condition, an expression executed when the condition is true, and and expression executed when the condition is false:

```
# No parentheses required
val x = if b < 12 then c() else d()
```

The `else` branch is optional and its default value is `true`, this can be useful to express an implication logic:

```
# Without "else" expression
val y = if b < 12 then a == 0
```

5.1.3.12 Match Expression

This expression is a pattern matching expression, and reuses the same patterns as the query part of the language. Matchers will be evaluated in order against the match's target expression. The first matcher to match the object will trigger the evaluation of the associated expression in the match arm:

```
match nodes[1]
| ObjectDecl(p_has_aliased(): aliased @ *) => aliased
| ParamSpec(p_has_aliased(): aliased @ *) => aliased
| * => false
```

Note: For the moment, there is no static check that the matcher is complete. A match expression where no arm has matched will raise an exception at runtime.

5.1.3.13 Tuple Literal

The tuple literal is used to create a value of the `Tuple` composite type:

```
val t = (1, 2)
val tt = ("hello", "world")
val ttt = (t[1], tt[1])
print(t)
print(tt)
print(ttt)
```

Tuples are useful as function return values, or to aggregate data, since LKQL doesn't have structs yet.

5.1.3.14 Anonymous Function

LKQL supports first class functions, and anonymous functions expressions (or lambdas). Thus, you can create anonymous functional values:

```
fun mul_y(y) = (x) => x * y
val mul_2 = mul_y (2)
val four = mul_2 (2)
```

5.1.3.15 Literals and Operators

LKQL has literals for booleans, integers, strings, unit, and null values:

```
val a = true     # Boolean
val b = 12       # Integer
val c = "hello"  # String
val d = ()       # Unit
val e = null     # Null
```

Note: The LKQL `null` literal is used to represent a null node value, thus, it is different from the `()` (`unit`) value.

LKQL has multi-line string literals, called block-strings but they're a bit different than in Python or other languages:

```
val a = |" Hello
        |" This is a multi line string
        |" Bue
```

Note: The first character after the " should be a whitespace. This is not enforced at parse-time but at run-time, so `|"hello` is still a syntactically valid block-string, but will raise an error when evaluated.

LKQL has a few built-in operators available:

- Basic arithmetic operators on integers

```
val calc = a + 2 * 3 / 4 == b
val smaller_or_eq = a <= b
val greater_or_eq = b >= c
```

- Basic relational operators on booleans

```
true and false or (a == b) and (not c)
```

- Value concatenation

```
# Strings concatenation
"Hello " & name
```

```
# Lists concatenation
[1, 2, 3] & [4, 5, 6]
```

5.1.3.16 Module Importation

LKQL has a very simple module system. Basically every file in LKQL is a module, and you can import modules from other files with the `import` clause. When importing a module, you are associating its name to the namespace produced by the evaluation of its source (all declarations in its top-level):

```
# foo.lkql
fun bar() = 12
```

(continues on next page)

(continued from previous page)

```
# bar.lkql
import foo

print(foo.bar())
```

LKQL will search for files:

1. That are in the same directory as the current file
2. That are in the `LKQL_PATH` environment variable

In case of multiple LKQL modules with the same name (two LKQL files named the same), an error is raised by the interpreter.

Note: There is no way to create hierarchies of modules for now, only flat modules are supported.

Attention: Circular dependencies are forbidden, thus the following files will raise an error at runtime:

```
# foo.lkql
import bar

# bar.lkql
import foo
```

Attention: In case of an ambiguous importation, the LKQL engine will raise a runtime error. For example, the following example will raise an error if the `subdir` directory is in the `LKQL_PATH` environment variable:

```
# foo.lkql
val x = 42

# subdir/foo.lkql
val y = 50

# bar.lkql
import foo
print(foo.x)
```

5.2 Query Language Subset

The query language subset is mainly composed of three language constructs: patterns, queries and selectors.

Patterns allow the user to express filtering logic on trees and graphs, akin to what regular expressions allow for strings.

A lot of the ideas behind patterns are similar to ideas in XPath, or even in CSS selectors

However, unlike in CSS or xpath, a pattern is just the filtering logic, not the traversal, even though filtering might contain sub traversals via selectors.

Here is a very simple example of a *query expression*, that will select object declarations that have the aliased qualifier:

```
# Queries are expressions, so their result can be stored in a named value
val a = select ObjectDecl(p_has_aliased(): true)
```

This will query every source file in the LKQL context, filter their nodes according to the provided *pattern*, and return the `List` containing all nodes matching the pattern.

Finally, selectors are a way to express "traversal" logic on the node graph. Syntactic nodes, when explored through their syntactic children, form a tree. However:

- There are different ways to traverse this tree (for example, you can explore the parents starting from a node)
- There are non syntactic ways to explore nodes, for example using semantic properties such as going from references to their declarations, or going up the tree of base types for a given tagged type.

All those traversals, including the most simple built-in one, use what is called selectors in LKQL. Those are a way to specify a traversal, which will return a `Stream` of nodes as a result. Here is an example of a selector that will go up the parent chain:

```
selector parent
| AdaNode => rec(*this.parent, this)
| *       => ()
```

Read the *Selector Declaration* section for more information about selectors.

5.2.1 Query Expression

The query expression is extremely simple, and most of the complexity lies in the upcoming sections about patterns.

A query traverses one or several trees, from one or several root nodes, applying the pattern on every node, and then returns a `List` containing all nodes that matched the pattern:

```
# Will select all non null nodes
select AdaNode
```

By default the query's roots are implicit and set by the context. However, you can specify them with the `from` keyword, followed either by a `Node` value, or a `List` of nodes:

```
# Select all non null nodes starting from node a
from a select AdaNode

# Select all non null nodes starting from all nodes in list
from [a, b, c] select AdaNode
```

You can also run a query that will only select the first element, this can be useful to avoid visiting all the parsing tree:

```
# Select first basic declaration
select first BasicDecl
```

5.2.1.1 Specifying the selector

By default, queries traverse the syntactic tree from the root node to leaves. This behavior is equivalent to going through the nodes returned via the `children` built-in selector (read the *Built-in Selectors* section for more information).

But you can also specify which selector you're using to do the traversal, and even use your custom defined selectors. This is done using the `through` keyword:

```
# Selects the parents of the first basic declaration
from (select first BasicDecl) through parent select *
```

Attention: There is a special case for Ada, where you can specify `follow_generics` as a selector name, even though `follow_generics` is not a selector. This allows traversal of the tree going through instantiated generic trees, but is directly hard-coded into the engine for performance reasons.

```
# Selects all nodes following generic instantiations
through follow_generics select *
```

5.2.2 Pattern

Patterns are by far the most complex part of the query language subset, but at its core, the concept of a pattern is very simple: it is a construction that you will match against a value. LKQL will check that the value matches the pattern, and produce `true` if it does. In the context of a query, that will add the value to the result of the query.

---

todo

Patterns are not yet expressions, but they certainly could be and should be, so we're planning on improving that at a later stage.

5.2.2.1 Node patterns

Simple Node Patterns

Matching one or many node kinds is the simplest atom for node patterns. It can be either:

- a node kind name, matching all nodes of this kind
- an *or pattern*, matching on multiple node kinds
- a *wildcard pattern*, matching on all node kinds

```
select *                          # Will select every node
select BasicDecl                  # Will select every basic declaration
select (ObjectDecl | BaseTypeDecl) # Will select every object and type declaration
```

In a more complex form, those can have sub-patterns in an optional part between parentheses, which brings us to the next section.

Nested Sub Patterns

Inside the optional parentheses of node patterns, the user can add sub-patterns that will help refine the query. Those patterns can be of three different kinds:

Selector Predicate

A selector predicate is a sub-pattern that allows you to run a sub-query and to match its results:

```
select Body(any children: ForLoopStmt)
```

The quantifier part (`any` in the previous example) can be either `any` or `all`, which will alter how the sub-pattern matches:

- `all` will match only if all nodes returned by the selector match the condition
- `any` will match as soon as at least one child matches the condition

Any of the *built-in selectors* can be used, or even custom selectors.

Note: All selectors have three optional parameters that allows controlling the depth of the traversal, `depth`, `max_depth` and `min_depth`. Read *Selector Declaration* section for more information.

Field Predicate

A field predicate is a sub-pattern that allows you to match a sub-pattern against a specific field in the parent object. We have already seen such a construct in the introduction, and it's one of the simplest kind of patterns:

```
select ObjectDecl(f_default_expr: IntLiteral)
```

Property Call Predicate

A property predicate is very similar to a field predicate, except that a property of the node is called, instead of a field accessed. Syntactically, this is denoted by the parentheses after the property name:

```
select BaseId(p_referenced_decl(): ObjectDecl)
```

5.2.2.2 Regular Values Patterns

Not only nodes can be matched in LKQL: Any value can be matched via a pattern, including basic and composite data types.

Integer Pattern

You can match simple integer values with this pattern:

```
v is 12
```

Bool Pattern

You can match simple boolean values with this pattern:

```
v is true
```

Regex Pattern

You can match simple string values with this pattern, but you can also do more complicated matching based on regular expressions. The regex syntax follows Python's `re` module flavor.

```
v is "hello"
v is "hello.*?world"
```

Tuple Pattern

You can match tuple values with this pattern, elements being matched with component patterns:

```
match i
| (1, 2, 3) => print("un, dos, tres")
| *          => print("un pasito adelante maria")

match i
| (1, a@*, b@*, 4) => { print(a); print(b) }
```

List Pattern

You can match list values with this pattern, destructuring them and matching their elements against arbitrary value patterns:

```
match lst
| [1, 2, 3]   => "[1, 2, 3]"
| [1, a@*, 3] => "[1, a@*, 3], with a = " & img(a)
```

You can use the *splat pattern* at the end of a list pattern to match remaining elements:

```
match lst
| [11, 12, ...] => "[11, 12, ...]"
| [1, c@...]    => "[1, c@...] with b = " & img(b) & " & c = " & img(c)
| [...]         => "Any list"
```

Object Pattern

You can match object values with this pattern, associating each object key with an arbitrary value pattern:

```
match obj
| {a: 12}  => "{a: 12}"
| {a: a@*} => "Any object with an a key. Bind the result to a"
```

You can use the "splat" pattern anywhere in an object pattern to match remaining elements:

```
match obj
| {a@..., b: "hello"} => "Bind keys that are not b to var a"
| {a@...}             => "Bind all the object to a"
```

5.2.2.3 Special and Composite Patterns

Null Pattern

You can match all null nodes with this pattern:

```
match node
| BasicDecl => "A BasicDecl node"
| null      => "Node is null!"
```

Wildcard Pattern

You can match all values with this pattern, it will always return `true`:

```
match any_val
| BasicDecl => "A BasicDecl node"
| *         => "Any other value"
```

Splat Pattern

This pattern is used inside *List Pattern* and *Object Pattern* as a pattern to match all remaining values, collecting them in a collection of the same type as it is used in:

```
match v
| [1, rem@...]    => "A list with 1 as first element followed by " & img(rem)
| {a: 1, rem@...} => "An object with a=1 and " & img(rem)
```

Not Pattern

You can use this pattern to negate another one:

```
match v
| not BasicDecl => "Everything except a BasicDecl node"
| *             => "A BasicDecl node"
```

Or Pattern

You can use this pattern to combine any number of other patterns, and match any value matching one of those:

```
match v
| (BasicDecl | 1) => "A BasicDecl node or 1"
| *               => "Any other value"
```

5.2.2.4 Filtered Patterns and Binding Patterns

While you can express a lot of things via the regular pattern syntax mentioned above, sometimes it is necessary to be able to express an arbitrary boolean condition in patterns; this is done via the `when` clause:

```
select BasicDecl when bool_condition
```

However, in order to be able to express conditions on the currently matched objects, or arbitrary objects in the query, naming those objects is necessary. This is done via binding patterns:

```
select b @ BaseId # Same as "select BaseId", but now every BaseId object
                  # that is matched has a name that can be used in the whole
                  # pattern clause.

# Example usage:
val a = select first BasicDecl
select b @ BaseId when b.p_referenced_decl() == a
```

5.2.3 Selector Declaration

Selectors are a special form of functions that return a `LasyList` of values. They're widely used in the query subset of LKQL, allowing the easy expression of traversal blueprints.

For example, by default, a *query expression* explores the tree via the `children` built-in selector.

While you can't add parameters to the definition of a selector, selector calls (a *call expression* or a *selector predicate*) can take three optional arguments that allows the control of depth:

- `min_depth` allows you to filter nodes for which the traversal depth is lower than a certain value
- `max_depth` allows you to filter nodes for which the traversal depth is higher than a certain value
- `depth` allows you to only receive nodes that are exactly at the given traversal depth

Here are some examples of calling selectors with those parameters:

```
# Calling a selectors directly
val c = children(node, depth=3)

# Calling a selector in a nested sub-pattern
select AdaNode(any children(min_depth=3): BasicDecl)
```

You've already seen selectors used in previous sections, and, most of the time, you might not need to define your own, but in case you need to, here is how they work.

5.2.3.1 Defining a Selector

A selector is a recursive function. In the body of the selector, there is a binding from `this` to the current node. A selector has an implicit top level *match expression* matching on `this`.

In the branch of a selector, you can express whatever computation you want for the current node. **There is a high-level requirement though, which is that the expression returned by a selector branch must be a `RecExpr` , which can be created via the call to the `rec` built-in operation.**

The `rec` built-in operation looks like a function call.

It takes one or two expressions, which can be prefixed by the splat operator `*`.

- The first expression represents what has to be added to the recurse list (either an item, or a list of items, if prefixed by *). The recurse list is the list of items on which the selector will be called next. Items are added at the end of the list
- The second expression represents what has to be added to the result list (either an item, or a list of items, if prefixed by *). The result list is the list of items that will be yielded, piece-by-piece, to the user.
- You can pass only one expression, in which case it is used both for the result list and for the recurse list.

> **Attention:** Please note that selector call results are `LasyList`, thus, their elements are computed on demand (when accessed).

Here is for example how the `super_types` selector is expressed in LKQL:

```
selector super_types
| BaseTypeDecl => rec(*this.p_base_types())
| *            => ()
```

While selectors are in the vast majority of cases used to express tree traversals of graph of nodes, you can use selectors to generate or process more general sequences:

```
selector infinite_sequence
|" Infinite sequence generator
| nb => rec(
    nb + 1, # Recurse with value nb + 1
    nb # Add nb to the result list
)

fun my_map(lst, fn) =
|" User defined map function. Uses an inner selector to return a lazy
|" iterator result
{
```

(continues on next page)

(continued from previous page)

```
    selector internal
    | idx => rec(
        idx + 1,     # Recurse with value idx + 1
        fn(lst[idx]) # Add the result of calling fn on list[idx] to the result list
    );

    internal(1)
}

val mpd = my_map(infinite_sequence(0), (x) => x * 4)
print(mpd)
print(mpd[51])
```

Attention: The user interface for selectors is not optimal at the moment, so we might change it again soon.

5.2.3.2 Built-in Selectors

The built-in selectors are:

- `parent`: parent nodes
- `children`: child nodes
- `prev_siblings`: sibling nodes that are before the current node
- `next_siblings`: sibling nodes that are after the current node
- `super_types`: if the current node is a type, then all its parent types

5.3 Language changes

Under this section, we'll document language changes chronologically, and categorize them by AdaCore GNATcheck release.

Note: Changes marked as "**breaking**" indicates that your LKQL code bases need to be migrated when moving to the referred GNATcheck version. The LKQL executable provides a sub-command named `refactor` to help you doing this (run `lkql refactor --help` for more information).

5.3.1 25.0

5.3.1.1 Conditional expression alternatives are now optional

Now you can write a conditional expression without providing any alternative expression. This way, if the condition is evaluated as `true`, then the consequence expression is evaluated, else the `true` value is returned. You can use this feature to express logical implication when performing boolean operation, example:

```
if node.p_has_something() then node.p_check_something_else()
```

5.3.1.2 Syntax of pattern details (breaking)

Pattern details were specified with the syntax `<left_part> is <pattern>`, and are now specified with the syntax `<left_part>: <pattern>`.

5.3.1.3 Syntax of selectors recursion definition (breaking)

The syntax for defining a recursion in selectors has completely changed. The old *rec* and *skip* keywords have been replaced by a single *rec* construct that allows to specify what elements will be recursed upon, and what elements will be yielded by the selector:

```
selector parent
| AdaNode => rec(*this.parent, this)
#                ^ Add parent to the recurse list
#                ^             ^ Add this to the return list
| *       => ()
```

Warning: This syntax is more general than the previous one, but is still not optimal, and might change again in a further release. Please take that into account when using selectors in your own code.

More details in the *Selector Declaration* section.

5.3.1.4 Or patterns syntax (breaking)

Or patterns were defined with the `<pattern> or <pattern>` syntax, and are now defined with the `<pattern> | <pattern>` syntax.

5.3.1.5 Binding patterns without value pattern

Patterns binding any value to a name can simply be expressed with a binding name now:

```
match d
| BasicDecl(p_doc(): doc) => print(doc)
```

5.3.1.6 More patterns

So far, only node values had corresponding patterns to match them. Now, patterns can be used to match other values:

```
v is 12
v is true
v is "hello"
v is "hello.*?world"

match i
| (1, 2, 3) => print("un, dos, tres")
| *         => print("un pasito adelante maria")

match i
| (1, a@*, b@*, 4) => { print(a); print(b) }
```

(continues on next page)

(continued from previous page)

```
match lst
| [1, 2, 3]   => "[1, 2, 3]"
| [1, a@*, 3] => "[1, a@*, 3], with a = " & img(a)

match lst
| [11, 12, ...] => "[11, 12, ...]"
| [1, c@...]    => "[1, c@...] with b = " & img(b) & " & c = " & img(c)
| [...]         => "Any list"

match obj
| {a: 12}  => "{a: 12}"
| {a: a@*} => "Any object with an a key. Bind the result to a"

match obj
| {a@..., b: "hello"} => "Bind keys that are not b to var a"
| {a@...}             => "Bind all the object to a"
```

5.4 LKQL API

5.4.1 Libadalang API

The libadalang API can be called from LKQL and is the basis for most of the GNATcheck rules.

In addition, LKQL comes with a built-in standard library described in *Standard library*, as well as a LKQL stdlib module described in *stdlib's API doc*.

5.4.2 Standard library

5.4.2.1 Builtin functions

base_name(*file_name*)

Given a string that represents a file name, returns the basename

concat(*list*)

Given a list, return the result of the concatenation of all its elements

context()

Return the analysis context used to parse units

doc(*function*)

Given any object, return the documentation associated with it

document_builtins()

Return a string in the RsT format containing documentation for all built-ins

get_unit(*name*, *kind*)

Use the context unit provider to get the analysis unit matching the provided name and kind

help(*callable*)

Print formatted help for the given object

img(*string*)

Return a string representation of an object

nil()

Return an empty stream

node_checker(*root*)

Given a root, execute all node checkers while traversing the tree

pattern(*regex*, *case_sensitive*)

Given a regex pattern string, create a pattern object

print(*to_print*, *new_line*)

Built-in print function. Prints the argument

profile(*callable*)

Given any object, if it is a callable, return its profile as text

reduce(*iterable*, *function*, *init_value*)

Given a collection, a reduction function, and an initial value reduce the result

repeat(*times*, *function*)

Call the given function N times

specified_units()

Return a list of units specified by the user

unique(*iterable*)

Given a collection, create a list with all duplicates removed

unit_checker(*unit*)

Given a unit, apply all the unit checkers on it

units()

Return a list of all units

5.4.2.2 Builtin methods

Methods for *AnalysisUnit*

AnalysisUnit.**doc**(*this*)

Given any object, return the documentation associated with it

AnalysisUnit.**help**(*this*)

Print formatted help for the given object

AnalysisUnit.**img**(*this*)

Return a string representation of an object

AnalysisUnit.**name**(*this*)

Return the name for this unit

AnalysisUnit.**print**(*this*)

Built-in print function. Prints the argument

`AnalysisUnit.`**root**(*this*)
Return the root for this unit

`AnalysisUnit.`**text**(*this*)
Return the text for this unit

`AnalysisUnit.`**tokens**(*this*)
Return the tokens for this unit

Methods for *Bool*

`Bool.`**doc**(*this*)
Given any object, return the documentation associated with it

`Bool.`**help**(*this*)
Print formatted help for the given object

`Bool.`**img**(*this*)
Return a string representation of an object

`Bool.`**print**(*this*)
Built-in print function. Prints the argument

Methods for *Function*

`Function.`**doc**(*this*)
Given any object, return the documentation associated with it

`Function.`**help**(*this*)
Print formatted help for the given object

`Function.`**img**(*this*)
Return a string representation of an object

`Function.`**print**(*this*)
Built-in print function. Prints the argument

Methods for *Int*

`Int.`**doc**(*this*)
Given any object, return the documentation associated with it

`Int.`**help**(*this*)
Print formatted help for the given object

`Int.`**img**(*this*)
Return a string representation of an object

`Int.`**print**(*this*)
Built-in print function. Prints the argument

Methods for *List*

List.**all**(*this*, *predicate*)

Given a collection and a predicate, returns true if all elements satisfies the predicate.

List.**any**(*this*, *predicate*)

Given a collection and a predicate, returns true if any element satisfies the predicate.

List.**combine**(*this*, *right*, *recursive*)

Combine two LKQL values if possible and return the result, recursively if required

List.**doc**(*this*)

Given any object, return the documentation associated with it

List.**enumerate**(*this*)

Return the content of the iterable object with each element associated to its index in a tuple: [(<index>, <elem>), ...]

List.**flat_map**(*this*, *function*)

Given an iterable and a function that takes one argument and return another iterable value, return a new iterable, result of the function application on all elements, flatten in a sole iterable value. The returned iterable value is lazy.

List.**flatten**(*this*)

Given an iterable of iterables, flatten all of them in a resulting iterable value. The returned value is lazy.

List.**help**(*this*)

Print formatted help for the given object

List.**img**(*this*)

Return a string representation of an object

List.**length**(*this*)

Return the length of the list

List.**map**(*this*, *function*)

Given an iterable and a function that takes one argument and return a value, return a new iterable, result of the application of the function on all iterable elements. The returned iterable value is lazy.

List.**print**(*this*)

Built-in print function. Prints the argument

List.**reduce**(*this*, *function*, *init_value*)

Given a collection, a reduction function, and an initial value reduce the result

List.**sublist**(*this*, *low*, *high*)

Return a sublist of *list* from *low_bound* to *high_bound*

List.**to_list**(*this*)

Transform into a list. WARNING: This may never return in case of an infinite stream.

List.**to_stream**(*this*)

Transform into a stream

List.**unique**(*this*)

Given a collection, create a list with all duplicates removed

Methods for *MemberReference*

`MemberReference.`**`doc`**(*this*)
Given any object, return the documentation associated with it

`MemberReference.`**`help`**(*this*)
Print formatted help for the given object

`MemberReference.`**`img`**(*this*)
Return a string representation of an object

`MemberReference.`**`print`**(*this*)
Built-in print function. Prints the argument

Methods for *Namespace*

`Namespace.`**`doc`**(*this*)
Given any object, return the documentation associated with it

`Namespace.`**`help`**(*this*)
Print formatted help for the given object

`Namespace.`**`img`**(*this*)
Return a string representation of an object

`Namespace.`**`print`**(*this*)
Built-in print function. Prints the argument

Methods for *Node*

`Node.`**`children`**(*this*)
Return the node's children

`Node.`**`children_count`**(*this*)
Return the node's children count

`Node.`**`doc`**(*this*)
Given any object, return the documentation associated with it

`Node.`**`dump`**(*this*)
Dump the node's content in a structured tree

`Node.`**`help`**(*this*)
Print formatted help for the given object

`Node.`**`image`**(*this*)
Return the node's image

`Node.`**`img`**(*this*)
Return a string representation of an object

`Node.`**`kind`**(*this*)
Return the node's kind

`Node.`**`parent`**(*this*)

Return the node's parent

`Node.`**`print`**(*this*)

Built-in print function. Prints the argument

`Node.`**`same_tokens`**(*this*, *right_node*)

Return whether two nodes have the same tokens, ignoring trivias

`Node.`**`text`**(*this*)

Return the node's text

`Node.`**`tokens`**(*this*)

Return the node's tokens

`Node.`**`unit`**(*this*)

Return the node's analysis unit

Methods for *Object*

`Object.`**`combine`**(*this*, *right*, *recursive*)

Combine two LKQL values if possible and return the result, recursively if required

`Object.`**`doc`**(*this*)

Given any object, return the documentation associated with it

`Object.`**`help`**(*this*)

Print formatted help for the given object

`Object.`**`img`**(*this*)

Return a string representation of an object

`Object.`**`print`**(*this*)

Built-in print function. Prints the argument

Methods for *Pattern*

`Pattern.`**`doc`**(*this*)

Given any object, return the documentation associated with it

`Pattern.`**`help`**(*this*)

Print formatted help for the given object

`Pattern.`**`img`**(*this*)

Return a string representation of an object

`Pattern.`**`print`**(*this*)

Built-in print function. Prints the argument

Methods for *PropertyReference*

`PropertyReference.`**`doc`**(*this*)

Given any object, return the documentation associated with it

`PropertyReference.`**`help`**(*this*)

Print formatted help for the given object

`PropertyReference.`**`img`**(*this*)

Return a string representation of an object

`PropertyReference.`**`print`**(*this*)

Built-in print function. Prints the argument

Methods for *RecValue*

`RecValue.`**`doc`**(*this*)

Given any object, return the documentation associated with it

`RecValue.`**`help`**(*this*)

Print formatted help for the given object

`RecValue.`**`img`**(*this*)

Return a string representation of an object

`RecValue.`**`print`**(*this*)

Built-in print function. Prints the argument

Methods for *RewritingContext*

`RewritingContext.`**`add_first`**(*this*, *node*, *new_node*)

Insert *new_node* at the beginning of *list_node*

`RewritingContext.`**`add_last`**(*this*, *node*, *new_node*)

Insert *new_node* at the end of *list_node*

`RewritingContext.`**`create_from_template`**(*this*, *template*, *grammar_rule*, *arguments*)

Create a new node from the provided template, filling '{}' with provided argument, and parsing the template with the specified grammar rule. Example:

```
# Create a new BinOp node with OpAdd as operator, representing the addition of the␣
↪value
# expressed by `my_other_node`, and "42".
ctx.create_from_template(
    "{} + 42",
    "expr_rule",
    [my_other_node]
)
```

`RewritingContext.`**`doc`**(*this*)

Given any object, return the documentation associated with it

`RewritingContext.`**`help`**(*this*)
Print formatted help for the given object

`RewritingContext.`**`img`**(*this*)
Return a string representation of an object

`RewritingContext.`**`insert_after`**(*this*, *node*, *new_node*)
Insert *new_node* after *node* (*node*'s parent needs to be a list node)

`RewritingContext.`**`insert_before`**(*this*, *node*, *new_node*)
Insert *new_node* before *node* (*node*'s parent needs to be a list node)

`RewritingContext.`**`print`**(*this*)
Built-in print function. Prints the argument

`RewritingContext.`**`remove`**(*this*, *obj_to_remove*)
Delete the given node from its parent (parent needs to be a list node)

`RewritingContext.`**`replace`**(*this*, *old_node*, *new_node*)
Replace old node by the new one

`RewritingContext.`**`set_child`**(*this*, *node*, *member_ref*, *new_value*)
Set the node child, following the given member reference, to the new value

Methods for *RewritingNode*

`RewritingNode.`**`clone`**(*this*)
Given a rewriting node, clone it and return its copy

`RewritingNode.`**`doc`**(*this*)
Given any object, return the documentation associated with it

`RewritingNode.`**`help`**(*this*)
Print formatted help for the given object

`RewritingNode.`**`img`**(*this*)
Return a string representation of an object

`RewritingNode.`**`print`**(*this*)
Built-in print function. Prints the argument

Methods for *Selector*

`Selector.`**`doc`**(*this*)
Given any object, return the documentation associated with it

`Selector.`**`help`**(*this*)
Print formatted help for the given object

`Selector.`**`img`**(*this*)
Return a string representation of an object

`Selector.`**`print`**(*this*)
Built-in print function. Prints the argument

Methods for *Str*

Str.**base_name**(*this*)

Given a string that represents a file name, returns the basename

Str.**combine**(*this*, *right*, *recursive*)

Combine two LKQL values if possible and return the result, recursively if required

Str.**contains**(*this*, *to_find*)

Search for to_find in the given string. Return whether a match is found. to_find can be either a pattern or a string

Str.**doc**(*this*)

Given any object, return the documentation associated with it

Str.**ends_with**(*this*, *suffix*)

Returns whether string ends with given prefix

Str.**find**(*this*, *to_find*)

Search for to_find in the given string. Return position of the match, or -1 if no match. to_find can be either a pattern or a string

Str.**help**(*this*)

Print formatted help for the given object

Str.**img**(*this*)

Return a string representation of an object

Str.**is_lower_case**(*this*)

Return whether the string is in lowercase

Str.**is_mixed_case**(*this*)

Return whether the given string is written in mixed case, that is, with only lower case characters except the first one and every character following an underscore

Str.**is_upper_case**(*this*)

Return whether the string is in uppercase

Str.**length**(*this*)

Return the string's length

Str.**print**(*this*)

Built-in print function. Prints the argument

Str.**split**(*this*, *sep*)

Given a string, split it on the given separator, and return an iterator on the parts

Str.**starts_with**(*this*, *prefix*)

Returns whether string starts with given prefix

Str.**substring**(*this*, *start*, *end*)

Given a string and two indices (from and to), return the substring contained between indices from and to (both included)

Str.**to_lower_case**(*this*)

Return the string in lowercase

Str.**to_upper_case**(*this*)

Return the string in uppercase

Methods for *Stream*

Stream.**all**(*this*, *predicate*)

Given a collection and a predicate, returns true if all elements satisfies the predicate.

Stream.**any**(*this*, *predicate*)

Given a collection and a predicate, returns true if any element satisfies the predicate.

Stream.**doc**(*this*)

Given any object, return the documentation associated with it

Stream.**enumerate**(*this*)

Return the content of the iterable object with each element associated to its index in a tuple: [(<index>, <elem>), ...]

Stream.**flat_map**(*this*, *function*)

Given an iterable and a function that takes one argument and return another iterable value, return a new iterable, result of the function application on all elements, flatten in a sole iterable value. The returned iterable value is lazy.

Stream.**flatten**(*this*)

Given an iterable of iterables, flatten all of them in a resulting iterable value. The returned value is lazy.

Stream.**head**(*this*)

Return the head of the stream.

Stream.**help**(*this*)

Print formatted help for the given object

Stream.**img**(*this*)

Return a string representation of an object

Stream.**length**(*this*)

Return the length of the stream. WARNING: This may never return in case of an infinite stream.

Stream.**map**(*this*, *function*)

Given an iterable and a function that takes one argument and return a value, return a new iterable, result of the application of the function on all iterable elements. The returned iterable value is lazy.

Stream.**print**(*this*)

Built-in print function. Prints the argument

Stream.**reduce**(*this*, *function*, *init_value*)

Given a collection, a reduction function, and an initial value reduce the result

Stream.**tail**(*this*)

Return the tail of the stream.

Stream.**to_list**(*this*)

Transform into a list. WARNING: This may never return in case of an infinite stream.

Stream.**to_stream**(*this*)

Transform into a stream

Stream.**unique**(*this*)

Given a collection, create a list with all duplicates removed

Methods for *Token*

Token.**doc**(*this*)
Given any object, return the documentation associated with it

Token.**end_column**(*this*)
Return the end column

Token.**end_line**(*this*)
Return the end line

Token.**help**(*this*)
Print formatted help for the given object

Token.**img**(*this*)
Return a string representation of an object

Token.**is_equivalent**(*this*, *other*)
Return whether two tokens are structurally equivalent

Token.**is_trivia**(*this*)
Return whether this token is a trivia

Token.**kind**(*this*)
Return the kind for this token

Token.**next**(*this*, *ignore_trivia*)
Return the next token

Token.**previous**(*this*, *exclude_trivia*)
Return the previous token

Token.**print**(*this*)
Built-in print function. Prints the argument

Token.**start_column**(*this*)
Return the start column

Token.**start_line**(*this*)
Return the start line

Token.**text**(*this*)
Return the text for this token

Token.**unit**(*this*)
Return the unit for this token

Methods for *Tuple*

Tuple.**doc**(*this*)
Given any object, return the documentation associated with it

Tuple.**help**(*this*)
Print formatted help for the given object

`Tuple.`**`img`**`(`*this*`)`

Return a string representation of an object

`Tuple.`**`print`**`(`*this*`)`

Built-in print function. Prints the argument

Methods for *Unit*

`Unit.`**`doc`**`(`*this*`)`

Given any object, return the documentation associated with it

`Unit.`**`help`**`(`*this*`)`

Print formatted help for the given object

`Unit.`**`img`**`(`*this*`)`

Return a string representation of an object

`Unit.`**`print`**`(`*this*`)`

Built-in print function. Prints the argument

5.4.3 stdlib's API doc

5.4.3.1 Functions

`all``(`*iterable*`)`

Return whether all elements in the given iterable are truthy

`any``(`*iterable*`)`

Return whether at least one element in the given iterable is truthy

`children_no_nested``(`*node*`)`

Return all children nodes starting from a base subprogram body, but not entering in nested bodies.

`closest_enclosing_generic``(`*n*`)`

If `n` is part of a generic package or subprogram, whether it is instantiated or not, then return it.

`default_bit_order``()`

Return the value of `System.Default_Bit_Order`.

`depends_on_mutable_discriminant``(`*component_decl*`)`

Given a *ComponentDecl*, return whether it depends on a mutable discriminant value coming from its parent record declaration. The component depends on a discriminant if it uses it in its subtype constraint or if it is a variant.

`enclosing_block``(`*n*`)`

Return the first `DeclBlock` enclosing `n` if any, `null` otherwise.

`enclosing_body``(`*n*`)`

Return the first BodyNode enclosing n if any, null otherwise

`enclosing_package``(`*n*`)`

Return the first BasePackageDecl or PackageBody enclosing n if any, null otherwise

find_comment(*token*, *name*)

Return true if a comment token immediately following the previous "begin" keyword is found and contains only the provided name.

first_non_blank(*s*, *ind=1*)

Return the index of the first non blank character of s, starting at ind

full_root_type(*t*)

Return the full view of the root type of `t`, traversing subtypes, derivations and privacy.

get_formals(*subp_spec*)

Given a SubpSpec node, return a list of all its formal parameter defining names, each one associated to its ParamSpec node.

get_parameter(*params*, *actual*)

Given a `List[ParamActual]`, return the parameter corresponding to actual, null if actual is not found.

get_subp_body(*node*)

Return the SubpBody, TaskBody or ExprFunction corresponding to node, if any, null otherwise.

has_interfaces(*n*)

Return true if `n` is an interface or implements some interfaces

has_local_scope(*n*)

Return `true` if `n` is enclosed in a local scope

has_non_default_sso(*decl*)

Return true if `decl` has a `Scalar_Storage_Order` aspect whose value cannot be determined to be equal to `System.Default_Storage_Order`.

in_generic_instance(*n*)

Return true if `n` is part of a generic instantiation.

in_generic_template(*n*)

Return true if `n` is declared as part of a generic template (spec or body). Return false otherwise, including inside a generic instantiation.

is_assert_aspect(*s*)

Return `true` if the string `s` is the name of an assert aspect

is_assert_pragma(*s*)

Return `true` if the string `s` is the name of an assert pragma

is_by_copy(*param*)

Return true if `param` (a `ParamActual`) has a non aliased by-copy type

is_by_ref(*param*)

Get whether the provided parameter (a `ParamActual`) type is a by-reference" type as defined in the reference manual at 6.2(4-9).

is_classwide_type(*t*)

Return true if t is a classwide TypeDecl.

is_composite_type(*decl*)

Given a BaseTypeDecl, returns whether the declared type is a composite Ada type (record, array, task or protected).

is_constant_object(*node*)

Return true is node represents a constant object, false otherwise

is_constrained_subtype(*type_decl*)

Return whether the provided base type declaration declare a constrained type. This function also looks for constrains in parents of the type.

is_constructor(*spec*)

Return true if spec is a subprogram spec of a constructor, that is, has a controlling result and no controlling parameter.

is_controlling_param_type(*t*, *spec*)

Return true if *t* is a TypeExpr corresponding to a controlling parameter of the subprogram spec *spec*.

is_in_library_unit_body(*o*)

Return `true` if `o` is located in a library unit body

is_in_package_scope(*o*)

Return `true` if `o` is immediately in the scope of a package spec, body or generic package.

is_limited_type(*type*)

Return *true* if type is a limited type. This function expects *type* to be a BaseTypeDecl. .. WARNING:

```
This function is deprecated. Call `type.p_is_limited_type()` instead.
```

is_local_object(*o*)

Return `true` if `o` represents a local `ObjectDecl` or `ParamSpec`

is_negated_op(*node*)

Return whether `node` is a "not" unary operation, returning a standard boolean, and having as operand a predefined RelationOp or UnOp with OpNeq as operator.

is_predefined_op(*op*, *follow_renamings=false*)

Return true if `op` is a predefined operator; `op` can be an Op or a CallExpr.

is_predefined_type(*n*)

Return true if `n` is the name of a type declared in a predefined package spec.

is_program_unit(*n*)

Return `true` if `n` is a program unit spec, body or stub

is_standard_boolean(*n*)

Return true if the root type of `n` is `Standard.Boolean`.

is_standard_false(*node*)

Get whether the given node is a Name representing the False literal of the standard Boolean type, or of any type deriving from it.

is_standard_numeric(*n*)

Return `true` if `n` is the name of a numeric type or subtype in Standard

is_standard_true(*node*)

Get whether the given node is a Name representing the True literal of the standard Boolean type, or of any type deriving from it.

is_subject_to_predicate(*decl*)

Return whether the provided declaration is subject to a dynamic or static predicate.

is_subtype_indication_constrained(*subtype_indication*)

Return whether the provided subtype indication declare constraints.

is_tasking_construct(*node*)

Returns whether the given node is a construct related to Ada tasking, in other words: All constructs described in the section 9 of Ada RM.

is_unchecked_conversion(*node*)

Return true if node represents an instantiation of the *Ada.Unchecked_Conversion* subprogram

is_unchecked_deallocation(*node*)

Return true if node represents an instantiation of the *Ada.Unchecked_Deallocation* subprogram

list_of_units()

Return a (cached) list of all known units

max(*x*, *y*)

Return the max value between x and y

negate_op(*node*)

Assumes that `node` is either a RelationOp or UnOp with the OpNot as operator. Returns the negated form of the operation as a rewriting node. Examples: `negate_op("A = B") -> "A /= B"` `negate_op("A > B") -> "A <= B"` `negate_op("not A") -> "A"`

next_non_blank_token_line(*token*)

Return the start line of the next non blank token, or the next line for a comment, or 0 if none.

number_of_values(*type*)

Return the number of values covered by a given BaseTypeDecl, -1 if this value cannot be determined.

param_pos(*n*, *pos=0*)

Return the position of node `n` in its current list of siblings

previous_non_blank_token_line(*token*)

Return the end line of the previous non blank token, or the previous line for a comment, or 0 if none.

propagate_exceptions(*body*)

Return true if the given body may propagate an exception, namely if: - it has no exception handler with a `when others` choice; - or it has an exception handler containing a raise statement, or a call to `Ada.Exception.Raise_Exception` or `Ada.Exception.Reraise_Occurrence`.

range_values(*left*, *right*)

Return the number of values covered between left and right expressions, -1 if it cannot be determined.

sloc_image(*node*)

Return a string with basename:line corresponding to node's sloc

strip_conversions(*node*)

Strip `ParenExpr`, `QualExpr` and type conversions

strip_parenthesis(*node*)

Strip `ParenExpr` from the provided node.

ultimate_alias(*name*, *all_nodes=true*, *strip_component=false*)

Return the ultimately designated `ObjectDecl`, going through renamings This will not go through generic instantiations. If all_nodes is true, consider all kinds of nodes, otherwise consider only `BaseId` and `DottedName`. If `strip_component` is true, go to the prefix when encountering a component, otherwise stop at the `ComponentDecl`.

ultimate_designated_generic_subp(*subp_inst*)

Given a node representing an instantiation of a generic subprogram, return that non-instantiated subprogram after resolving all renamings.

ultimate_exception_alias(*name*)

Return the ultimately designated `ExceptionDecl`, going through renamings

ultimate_generic_alias(*name*)

Return the ultimately designated `GenericDecl`, going through renamings

ultimate_prefix(*n*)

Return `n.f_prefix` as long as `n` is a `DottedName` and designates a `ComponentDecl`, `n` otherwise.

ultimate_subprogram_alias(*name*)

Return the ultimately designated `BasicSubpDecl`, going through renamings

within_assert(*node*)

Return `true` if `node` is part of an assertion-related pragma or aspect.

5.4.3.2 Selectors

complete_super_types(*root*)

Yields the chain of super types of the given type in their most complete view. Hence, for a type T which public view derives from a type A but private view derives from a type B (which itself derives from A), invoking this selector on the public view of T will yield B and then A.

component_types(*root*)

Return all the `BaseTypeDecl` corresponding to all fields of a given type, including their full views, base types and subtypes.

full_parent_types(*root*)

Return all base (sub)types full views

parent_decl_chain(*root*)

Return all parent basic decl nodes starting from a given node, using semantic parent. When on a subprogram or package body, go to the declaration This allows us to, if in a generic template, always find back the generic formal.

semantic_parent(*root*)

Return all semantic parent nodes starting from a given node.

super_types(*root*)

Yields the chain of super types of the given type, as viewed from that type. Hence, for a type T which public view derives from a type A but private view derives from a type B (which itself derives from A), invoking this selector on the public view of T will yield A.

CHAPTER

SIX

LKQL DRIVER

Additionally to the `gnatcheck` executable, you can access the LKQL language through the LKQL driver. This is an executable file named `lkql` which defines several sub-commands, each being an entry point to the LKQL engine.

Attention: While being shipped alongside GNATcheck, not all sub-commands are considered as stable, some are even experimental or internal and should be used with extreme caution.

6.1 Sub-commands List

Additionally to their specific switches, each sub-command accepts the `--help` flag which triggers the display of its specific help message and exit.

6.1.1 `lkql refactor`

Hint: This sub-command is considered as stable and is officially supported.

This sub-command is used to perform automatic refactoring operations. It is mainly used to automatically migrate existing LKQL code-bases when a change is made in the language syntax or sematic.

`refactor` defines the following CLI switches:

`-i, --in-place`
: Apply refactoring directly into LKQL source files, modifying them.

`-r, --refactoring=<refactoring>`
: Name of the refactoring to apply to your LKQL files. You can view a list of available refactorings in the sub-command help message.

Additionally to those switches, the `refactor` sub-command expect a list of LKQL files to apply the specified refactoring on. Here is an example usage:

```
lkql refactor -i -r=IS_TO_COLON file_1.lkql file_2.lkql
```

6.1.2 lkql run

Caution: This sub-command is considered as a beta feature: while being pretty stable, its interface may change in the future, and relying on it should be considered as unsafe.

This is the LKQL interpreter entry point, through it you can access the current LKQL implementation to run any LKQL script or start a REPL. This is a good entry point to test the LKQL language and write custom GNATcheck rules in an iterative way.

`run` defines the following CLI switches:

`-C, --charset=<charset>`
Defines the charset to use for source decoding. The default is "utf-8".

`-i, --interactive`
Start an LKQL REPL (read-eval-print loop). This switch is incompatible with the `-S, --script-path` one.

`--missing-file-is-error`
If an Ada source file is missing, emit an error message instead of a warning one.

`-P, --project=<project>`
GPR file to fetch Ada sources from for the interpreter.

`--RTS=<runtime>`
Ada runtime to use when resolving sources.

`-S, --script-path=<script>`
Name of the LKQL script to run. This switch is incompatible with the `-i, --interactive` one.

`--target=<target>`
Hardware target used to resolved Ada runtime sources.

`-U, --recursive`
Process all units in the project tree, excluding externally built project.

`-v, --verbose`
Enable the verbose mode.

Additionally to those switches, you can provide to the `run` sub-command a list of Ada sources to use. Here is an example usage:

```
lkql run -S script.lkql main.adb
```

6.1.3 lkql check

Danger: This sub-command is considered as unstable and is not supported. Use it at your own risks.

This sub-command is used to run a set of LKQL rules on provided Ada sources. This is an internal entry point mainly used to test GNATcheck rules.

`check` defines the following CLI switches:

`-C, --charset=<charset>`
Defines the charset to use for source decoding. The default is "utf-8".

-j, --jobs=<n>
Number of jobs to use during analysis. If n is 0, spawn 1 job per CPU.

-P, --project=<project>
GPR file to fetch Ada sources from for the interpreter.

--RTS=<runtime>
Ada runtime to use when resolving sources.

--target=<target>
Hardware target used to resolved Ada runtime sources.

-U, --recursive
Process all units in the project tree, excluding externally built project.

-v, --verbose
Enable the verbose mode.

--missing-file-is-error
If an Ada source file is missing, emit an error message instead of a warning one.

-r, --rule=<rule>
Enable the given rule for the current run. This option is cumulative.

--rules-dir=<directory>
Additional directory to fetch LKQL rules from. This options is cumulative.

-a, --rule-arg=<rule>.<arg>=<value>
Provide a value for a specific argument of a rule. This option is cumulative.

Additionally to those switches, you can provide to the `check` sub-command a list of Ada sources to use during analysis. Here is an example usage:

```
lkql check main.adb main.ads -r my_rule -a "my_rule.arg=42"
```

6.1.4 lkql fix

> **Danger:** This sub-command is considered as an experimental feature. Use it at your own risks.

Sub-command to run a set of auto-fixing functions on a set of sources. This is an experimental entry point mainly used for testing purposes, but you can give it a try (be careful, this process may alter your Ada sources).

`fix` defines the same switches as `lkql check` sub-command, with some additional ones:

--auto-fix-mode=<mode>
The mode to use when applying auto-fixes. Available modes are:

- `DISPLAY`: Only display fixed sources in standard output, doesn't modify any source file
- `NEW_FILE`: For each source file, if it has some fixes, create a new file named `<filename>.patched` alongside the original one containing the patched source
- `PATCH_FILE`: Replace each source file that has fixes in them by their patched version

For now, there is no list of rules with an auto-fix function, but you can check if a rule can be used with this sub-command by reading its source code and checking for the `auto_fix` argument in its related `@check` annotation.

6.1.5 lkql doc-api

Danger: This sub-command is considered as unstable and is not supported. Use it at your own risks.

Entry point used to generate API documentation for LKQL modules in the RST format. Each LKQL file defines a module and all top level symbols are documented.

`doc-api` defines the following CLI switches:

-O, --output-dir=<directory>
Directory path to place generated RST files in.

--std
Additionally to other generated files, generate the documentation of the LKQL prelude and built-in functions.

Additionally to those switches, the `doc-api` sub-command expect a list of LKQL files to generate documentation for. Here is an example usage:

```
lkql doc-api -O=doc/ --std file_1.lkql file_2.lkql
```

6.1.6 lkql doc-rules

Danger: This sub-command is considered as unstable and is not supported. Moreover, some information are hard-coded in it, so it should be considered as an internal. Use it at your own risks.

Entry point used to generate documentation for a set of LKQL rules in the RST format.

`doc-rules` defines the following CLI switches:

-O, --output-dir=<directory>
Directory path to place generated RST files in.

-v, --verbose
Enable the verbose mode.

This sub-command also expect a list of directories containing LKQL rules to generate the documentation for. Here is an example usage:

```
lkql doc-rules -O=rules_doc/ rules/ other_rules/
```

6.1.7 lkql gnatcheck_worker

Danger: This sub-command is considered as internal and is not meant to be used from the command-line.

This is the entry point of the GNATcheck driver, and it is not meant to be used outside this context. That's why this entry point won't be documented any further.

APPENDIX

A

ALPHABETICAL LIST OF RULES

This section contains an alphabetized list of all the predefined GNATcheck rules.

- *Abort_Statements*
- *Abstract_Type_Declarations*
- *Access_To_Local_Objects*
- *Actual_Parameters*
- *Ada05_Formal_Packages*
- *Ada_2022_In_Ghost_Code*
- *Address_Attribute_For_Non_Volatile_Objects*
- *Address_Specifications_For_Initialized_Objects*
- *Address_Specifications_For_Local_Objects*
- *Annotated_Comments*
- *Anonymous_Access*
- *Anonymous_Arrays*
- *Anonymous_Subtypes*
- *At_Representation_Clauses*
- *Binary_Case_Statements*
- *Bit_Records_Without_Layout_Definition*
- *Blocks*
- *Boolean_Negations*
- *Boolean_Relational_Operators*
- *Calls_In_Exception_Handlers*
- *Calls_Outside_Elaboration*
- *Complex_Inlined_Subprograms*
- *Concurrent_Interfaces*
- *Conditional_Expressions*
- *Constant_Overlays*
- *Constructors*

- *Controlled_Type_Declarations*
- *Declarations_In_Blocks*
- *Deep_Inheritance_Hierarchies*
- *Deep_Library_Hierarchy*
- *Deeply_Nested_Generics*
- *Deeply_Nested_Inlining*
- *Deeply_Nested_Instantiations*
- *Default_Parameters*
- *Default_Values_For_Record_Components*
- *Deriving_From_Predefined_Type*
- *Direct_Calls_To_Primitives*
- *Direct_Equalities*
- *Discriminated_Records*
- *Downward_View_Conversions*
- *Duplicate_Branches*
- *End_Of_Line_Comments*
- *Enumeration_Ranges_In_CASE_Statements*
- *Enumeration_Representation_Clauses*
- *Exception_Propagation_From_Callbacks*
- *Exception_Propagation_From_Export*
- *Exception_Propagation_From_Tasks*
- *Exceptions_As_Control_Flow*
- *EXIT_Statements_With_No_Loop_Name*
- *Exits_From_Conditional_Loops*
- *Expanded_Loop_Exit_Names*
- *Explicit_Full_Discrete_Ranges*
- *Explicit_Inlining*
- *Expression_Functions*
- *Final_Package*
- *Fixed_Equality_Checks*
- *Float_Equality_Checks*
- *Forbidden_Aspects*
- *Forbidden_Attributes*
- *Forbidden_Pragmas*
- *Function_OUT_Parameters*
- *Function_Style_Procedures*

- *Generic_IN_OUT_Objects*
- *Generics_In_Subprograms*
- *Global_Variables*
- *GOTO_Statements*
- *Headers*
- *Identifier_Casing*
- *Identifier_Prefixes*
- *Identifier_Suffixes*
- *Implicit_IN_Mode_Parameters*
- *Implicit_SMALL_For_Fixed_Point_Types*
- *Improper_Returns*
- *Improperly_Located_Instantiations*
- *Incomplete_Representation_Specifications*
- *Integer_Types_As_Enum*
- *Library_Level_Subprograms*
- *Local_Instantiations*
- *Local_Packages*
- *Local_USE_Clauses*
- *Lowercase_Keywords*
- *Max_Identifier_Length*
- *Maximum_Expression_Complexity*
- *Maximum_Lines*
- *Maximum_OUT_Parameters*
- *Maximum_Parameters*
- *Maximum_Subprogram_Lines*
- *Membership_For_Validity*
- *Membership_Tests*
- *Metrics_Cyclomatic_Complexity*
- *Metrics_Essential_Complexity*
- *Metrics_LSLOC*
- *Min_Identifier_Length*
- *Misnamed_Controlling_Parameters*
- *Misplaced_Representation_Items*
- *Multiple_Entries_In_Protected_Definitions*
- *Name_Clashes*
- *Nested_Paths*

- *Nested_Subprograms*
- *No_Closing_Names*
- *No_Dependence*
- *No_Explicit_Real_Range*
- *No_Inherited_Classwide_Pre*
- *No_Others_In_Exception_Handlers*
- *No_Scalar_Storage_Order_Specified*
- *Non_Component_In_Barriers*
- *Non_Constant_Overlays*
- *Non_Qualified_Aggregates*
- *Non_Short_Circuit_Operators*
- *Non_SPARK_Attributes*
- *Non_Tagged_Derived_Types*
- *Non_Visible_Exceptions*
- *Nonoverlay_Address_Specifications*
- *Not_Imported_Overlays*
- *Null_Paths*
- *Number_Declarations*
- *Numeric_Format*
- *Numeric_Indexing*
- *Numeric_Literals*
- *Object_Declarations_Out_Of_Order*
- *Objects_Of_Anonymous_Types*
- *One_Construct_Per_Line*
- *One_Tagged_Type_Per_Package*
- *Operator_Renamings*
- *OTHERS_In_Aggregates*
- *OTHERS_In_CASE_Statements*
- *OTHERS_In_Exception_Handlers*
- *Outbound_Protected_Assignments*
- *Outer_Loop_Exits*
- *Outside_References_From_Subprograms*
- *Overloaded_Operators*
- *Overly_Nested_Control_Structures*
- *Overly_Nested_Scopes*
- *Overriding_Indicators*

- *Parameters_Aliasing*
- *Parameters_Out_Of_Order*
- *POS_On_Enumeration_Types*
- *Positional_Actuals_For_Defaulted_Generic_Parameters*
- *Positional_Actuals_For_Defaulted_Parameters*
- *Positional_Components*
- *Positional_Generic_Parameters*
- *Positional_Parameters*
- *Potential_Parameters_Aliasing*
- *Predefined_Numeric_Types*
- *Predicate_Testing*
- *Printable_ASCII*
- *Profile_Discrepancies*
- *Quantified_Expressions*
- *Raising_External_Exceptions*
- *Raising_Predefined_Exceptions*
- *Recursive_Subprograms*
- *Redundant_Boolean_Expressions*
- *Redundant_Null_Statements*
- *Relative_Delay_Statements*
- *Renamings*
- *Representation_Specifications*
- *Restrictions*
- *Same_Instantiations*
- *Same_Logic*
- *Same_Operands*
- *Same_Tests*
- *Separate_Numeric_Error_Handlers*
- *Separates*
- *Side_Effect_Parameters*
- *Silent_Exception_Handlers*
- *Simple_Loop_Statements*
- *Single_Value_Enumeration_Types*
- *Size_Attribute_For_Types*
- *Slices*
- *SPARK_Procedures_Without_Globals*

- *Specific_Parent_Type_Invariant*
- *Specific_Pre_Post*
- *Specific_Type_Invariants*
- *Style_Checks*
- *Subprogram_Access*
- *Suspicious_Equalities*
- *Too_Many_Dependencies*
- *Too_Many_Generic_Dependencies*
- *Too_Many_Parents*
- *Too_Many_Primitives*
- *Trivial_Exception_Handlers*
- *Unassigned_OUT_Parameters*
- *Unavailable_Body_Calls*
- *Unchecked_Address_Conversions*
- *Unchecked_Conversions_As_Actuals*
- *Uncommented_BEGIN*
- *Uncommented_BEGIN_In_Package_Bodies*
- *Uncommented_End_Record*
- *Unconditional_Exits*
- *Unconstrained_Array_Returns*
- *Unconstrained_Arrays*
- *Uninitialized_Global_Variables*
- *Universal_Ranges*
- *Unnamed_Blocks_And_Loops*
- *Unnamed_Exits*
- *Use_Array_Slices*
- *Use_Case_Statements*
- *USE_Clauses*
- *Use_For_Loops*
- *Use_For_Of_Loops*
- *Use_If_Expressions*
- *Use_Memberships*
- *USE_PACKAGE_Clauses*
- *Use_Ranges*
- *Use_Record_Aggregates*
- *Use_Simple_Loops*

- *Use_While_Loops*
- *Variable_Scoping*
- *Visible_Components*
- *Volatile_Objects_Without_Address_Clauses*
- *Warnings*

This page is intentionally left blank.

APPENDIX

B

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **Document**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **Opaque**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

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 fewer than five), unless they release you from this requirement.

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", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

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. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

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 to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W