GNAT SAS

GNAT SAS

Copyright (C) 2018-2026, AdaCore under CC BY-SA

Published under a CC BY-SA license, which means that you can copy, redistribute, remix, transform, and build upon the content for any purpose, even commercially, as long as you give appropriate credit, provide a link to the license, and indicate if changes were made. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You can find license details on this page http://creativecommons.org/licenses/by-sa/4.0

GNATcheck

Introduction

GNATcheck Is...

- An **automated** coding standards checker
- Capable of expressing a variety of rules
  - GNAT compiler **warnings and style** checks
  - Language-defined and GNAT-defined **restrictions**
  - Complexity **metrics**
  - **Specific** GNATCHECK rules
- Qualified to DO-178 in several programs
- Integrated in GNAT STUDIO

Required by DO-178

Table A-5

Verification Of Outputs of Software Coding & Integration Process

	Objective		Applicability by SW Level			
	Description	Ref.	A	B	C	D
	...					
4	Source Code conforms to standards	6.3.4.d	○	○	○	
	...					

● Should be satisfied with independence

○ Should be satisfied

Conformance to Standards Requirement - DO-178

6.3.4 Reviews and Analyses of the Source Code

d. Conformance to standards

The objective is to **ensure that the Software Code Standards were followed** during the development of the code, especially **complexity restrictions and code constraints** that would be consistent with the system safety objectives.

Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions.

This analysis also ensures that **deviations to the standards are justified.**

GNATcheck Input Requirements

- Can analyze sources that are not legal
  - But may result in false negatives due to missing/incorrect semantic information
  - Switch `check-semantic` can check if sources are legal
- Can analyze standalone files
  - But will not parse dependencies
  - Use a GNAT Project File as input for better analysis

Getting Started

Basic Usage

Command Line Invocation

```
gnatcheck [options] filename -files=filename [-cargs gcc_switches] -rules rule_switches
```

Argument	Description
{filename}	File to analyze (wildcards allowed)
{files=filename}	`filename` specifies text file containing list of files to analyze
-rules rule_switches	Rules to apply for analysis

Where `rule_switches` can be any combination of the following:

Switch	Explanation
-from=filename	read rule options from `filename`
+R<rule_id>[:param]	turn ON a given rule [with given parameter]
-R<rule_id>	turn OFF a given rule
-R<rule_id>:param	turn OFF some of the checks for a given rule, depending on the specified parameter

Command Line Example Run

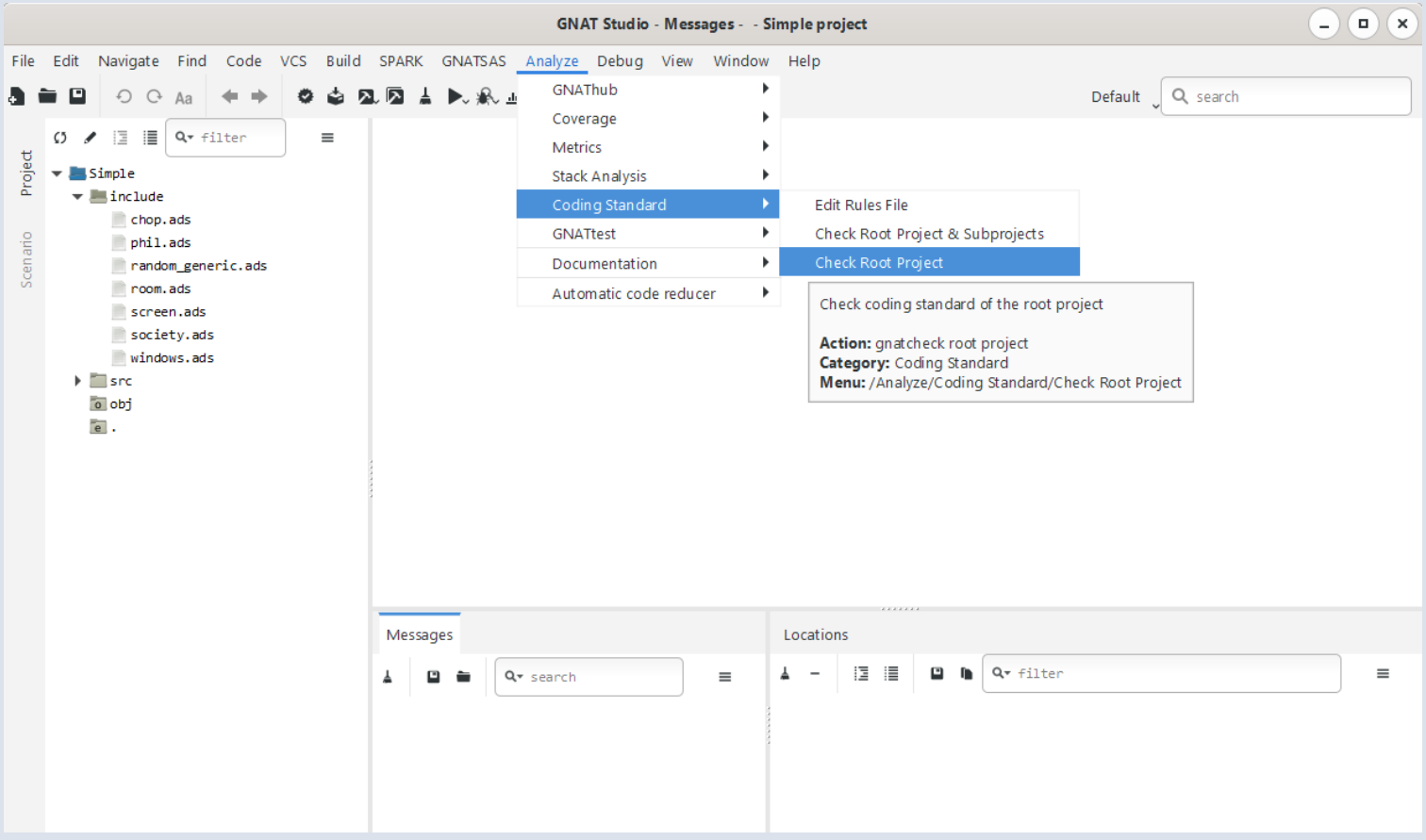
```
gnatcheck -P simple.gpr -rules -from=coding_standard.rules
```

```
chop.adb:14:11: PIck_Up does not have casing specified (mixed)
chop.ads:11:18: Stick does not start with subtype prefix T_
phil.adb:21:11: Think_Times does not start with subtype prefix T_
phil.adb:33:05: "Who_Am_I" is not modified, could be declared constant
phil.ads:12:03: violation of restriction "No_Tasking"
phil.ads:12:13: Philosopher does not start with subtype prefix T_
phil.ads:12:26: My_ID does not have casing specified (mixed)
phil.ads:19:08: States does not end with type suffix _Type
phil.ads:19:08: States does not start with subtype prefix T_
random_generic.ads:5:08: Result_Subtype does not end with type suffix _Type
random_generic.ads:5:08: Result_Subtype does not start with subtype prefix T_
room.adb:19:03: violation of restriction "No_Tasking"
room.adb:19:23: anonymous subtype
...
```

These messages are coming from rules specified in `coding_standard.rules`

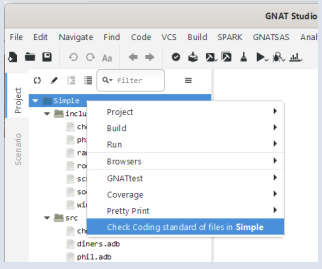
GNATcheck From GNAT Studio Main Menu

Analyze → Coding Standard → Check Root Project

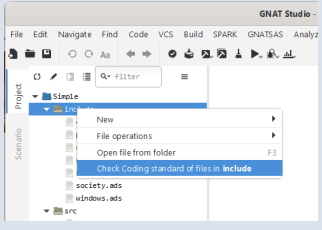


GNATcheck From **GNAT Studio** Right-Click

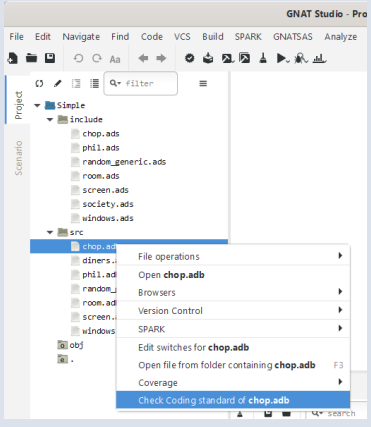
- Right-click on project in Project pane



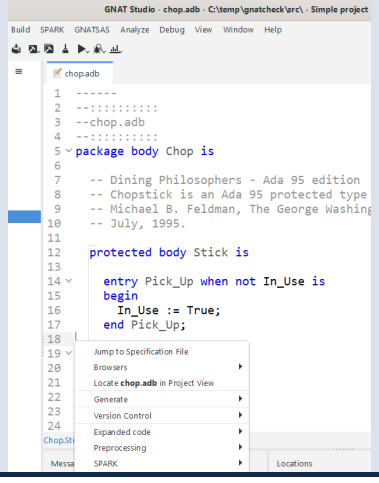
- Right-click on folder in Project pane



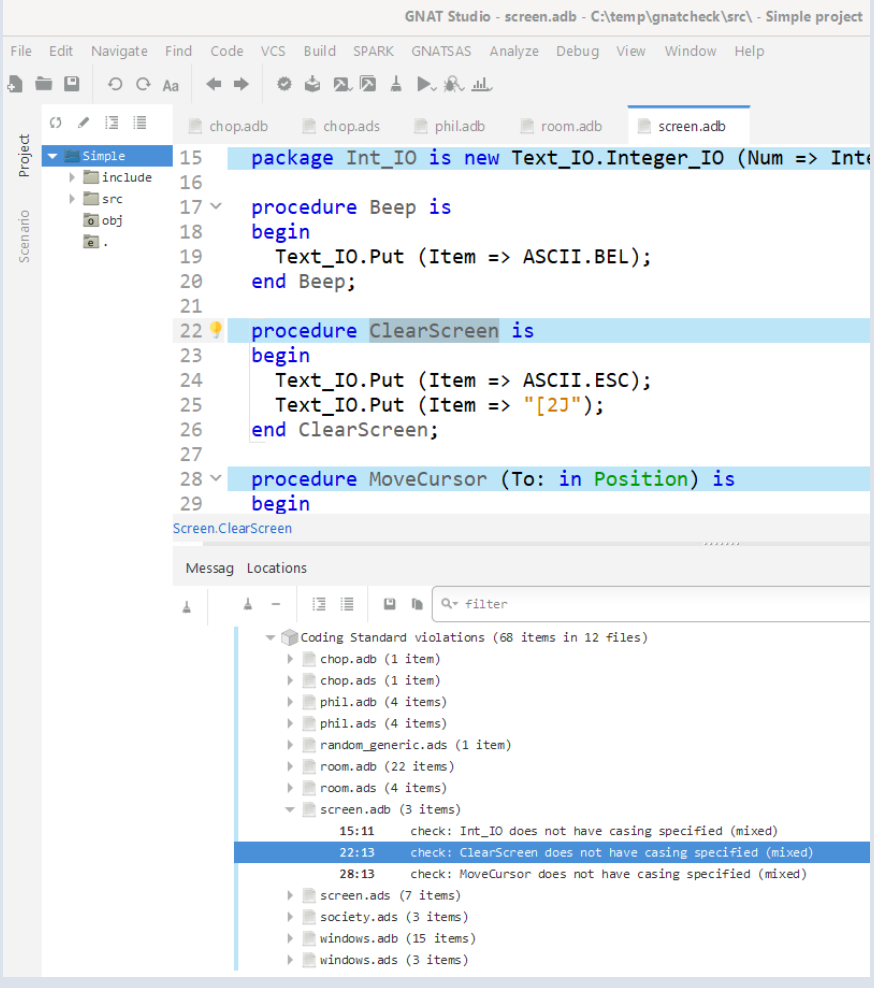
Right-click on file in Project pane



Right-click in source file editor window



GUI Example Run



Specifying Rules File

- Rules file can be specified on command line
  - `gnatcheck -rules -from=coding_standard.rules ...`
  - But more commonly defined in project file

```
project Simple is
   for Source_Dirs use ("./include", "./src");
   for Main use ("diners");
   for Object_Dir use "./obj";
   package Check is
      for Default_Switches ("ada") use
         ("-rules", "-from=coding_standard.rules");
   end Check;
end Simple;
```

Lab

GNATcheck Getting Started Lab

- Copy the `check_020_getting_started` folder from the course materials location
- Contents of the folder:
  - `simple.gpr` - project file
  - `include` - source directory
  - `src` - source directory
  - `coding_standard.rules` - GNATcheck rules to apply during analysis

Preparing the Command Line

1. Open a command prompt window and navigate to the `check_020_getting_started` folder
2. Type `gnatcheck` and press Enter to verify tool is on your path
   - If you see `gnatcheck:  No existing file to process`, you can go to the next step
   - If you see something like
     `'gnatcheck' is not recognized as an internal or external command`

     Add the appropriate folder to your path (On Windows, typically `C:\GNATSAS\<version>\bin` where version is the GNAT SAS version number)

     ```
     set PATH=C:\GNATSAS\24.0\bin;%PATH%
     ```
3. Type `gnatcheck -h` and press Enter to show list of predefined rules
   - Examine the output to see what kinds of rules are available
   - The keyword at the end (*Easy*, *Medium*, *Major*) indicates the difficulty in remediating the issue

Running From the Command Line

1. Perform gnatcheck on a single file in the `src` folder

   ```
   gnatcheck src\room.adb -rules -from=coding_standard.rules
   ```

   Examine the output to see what parts of the code failed analysis

2. Add the switch to indicate which rule caused the message

   ```
   gnatcheck src\room.adb --show-rule -rules -from=coding_standard.rules
   ```

   Note the actual rule now appears at the end of the message

Preparing the GUI

1. Use GNAT Studio to open the project `simple.gpr`
2. Set the coding standards for the project to `coding_standard.rules`
   - In the **Project** pane, right-click on the project name (GPR filename)
     - Click Project → Edit source file
   - Add the following package to the project source file

```
package Check is
   for Default_Switches ("ada") use
      ("-rules", "-from=coding_standard.rules");
end Check;
```

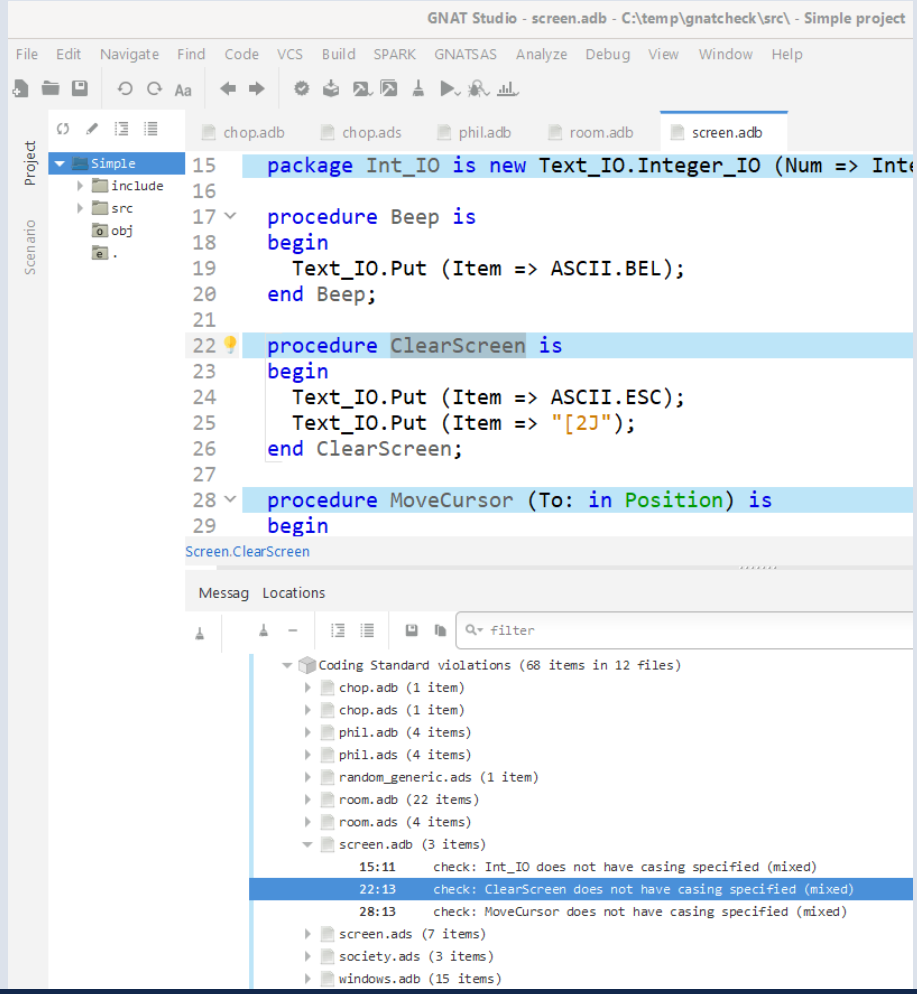
Running From the GUI

1. Perform Coding Analysis on the project

   Analyze -> **Coding Standard** -> **Check Root Project**

2. Double-click on any source line in the **Locations** window to go to the problematic code

   Try fixing the problem and re-running the analysis



Summary

Closing Remarks

- GNATcheck is a coding standards checker
  - *Rules* are how the tool decides what is the "standard"
- Rules are broken down into two categories:
  - Predefined rules - over 200 rules built into the tool
  - User-defined rules - ability for user to write their own rules

Predefined Rules

Introduction

Accessing Predefined Rules

- Over 200 predefined rules within GNATCHECK
  - Can be found via command `gnatcheck -h`
- Rules have been developed over many years for many uses
  - May have very specialized use cases
  - Some rules may contradict other rules
- Rules can be specified on the command line or via a file
  - Rule on the command line:

    ```
    gnatcheck screen.adb -rules +Runnamed_exits
    ```

    Apply **unnamed_exits** rule (*unnamed exit statement*) in analysis of file `screen.adb`
  - Rules file:

    ```
    gnatcheck screen.adb -rules -from=coding_standard.rules -ROTHERS_In_Aggregates
    ```

    Apply rules from `coding_standard.rules` except for rule **OTHERS_In_Aggregates**

Rules with Parameters

- Some rules have parameters

  `Too_Many_Primitives`

  Flag any tagged type declaration that has more than N user-defined primitive operations

- To specify a parameter, the value comes immediately after the rule separated only by a colon (:)
  - Incorrect

    ```
    gnatcheck screen.adb -rules +RToo_Many_Primitives
    ```

    ```
    gnatcheck: (too_many_primitives) parameter is required
    for +R
    ```

  - Correct

    ```
    gnatcheck screen.adb -rules +RToo_Many_Primitives:3
    ```

Note: Some parameters are optional

Predefined Rules Categories

Style-Related Rules

Tasking Example

`Volatile_Objects_Without_Address_Clauses`
Flag each volatile object without an address specification

Object Orientation Example

`Visible_Components`
Flag type declarations located in visible part of a library package or a library generic package that can declare visible component

Portability Example

`Forbidden_Pragmas`
Flag each use of the specified pragmas

Program Structure Example

`Local_Packages`
Flag local packages declared in package and generic package spec

Programming Practice Example

`Anonymous_Array`
Flag all anonymous array type definitions

Readability Example

`Style_Checks`
Flags violations of the source code presentation and formatting rules according to the rule parameter(s) specified

Feature Usage Rules

Examples

`Abort_Statements`
: *Flag abort statements*

`Numeric_Literals`
: *Flag each use of a numeric literal except for those matching certain requirements*

Metrics-Related Rules

Examples

`Metrics_Cyclomatic_Complexity`
: *Flag program units whose executable body exceeds the specified limit*

`Metrics_LSLOC`
: *Flag program units that exceed the specified limit*

SPARK Rules

Examples

`Overloaded_Operators`
: *Flag each function declaration that overloads an operator symbol*

`Slices`
: *Flag all uses of array slicing*

Introduction to LKQL

Introduction

Topics Covered

- **What is LKQL?**
  - How to write custom rules for GNATCHECK
- **General Purpose Programming Language**
  - Language used for LKQL function creation
- **Query Language**
  - Language used to interrogate source being examined
- **Building a Rule**

Overview

Libadalang (LAL)

- For parsing and semantic analysis of Ada code
- Building block for integration into other tools
  - IDEs, static analyzers, etc.
- Complete syntactic analysis with error recovery
  - When source is syntactically valid: precise syntax tree
  - When syntax errors are present: best-effort tree
- Semantic queries on the syntactic tree such as
  - Resolution of references (what a reference corresponds to)
  - Resolution of types (what is the type of an expression)
  - General cross reference queries (find all references to this entity)

LangKit Query Language (LKQL)

- LKQL is a functional language for querying source code
  - Based on langkit technology (link)
  - Currently hardwired for Ada (and LAL)
- Purely functional, high level, dynamically typed language
  - General purpose and tree query subsets
- Designed to be simple and concise
- Has a reference manual (link)

Why LKQL?

- GNATcheck uses LKQL to specify the rules it verifies
  - "Predefined" rules are just a text library of LKQL rules

```
gnatcheck -P default.gpr --rule renamings
```

```
<SAS installation>/share/lkql/renamings.lkql
```

```
@check(message="renaming declaration", category="Feature")
fun renamings(node) =
    |" Flag renaming declarations.
    |"
    |" .. rubric:: Example
    |"
    |" .. code-block:: ada
    |"    :emphasize-lines: 2
    |"
    |"    I : Integer;
    |"    J : Integer renames I;     --  FLAG
    node is RenamingClause
```

LKQL Construction

- Consists of two language subsets
- General Purpose Subset
  - Dynamically typed
  - Functional
  - Includes
    - Function definitions
    - Common expressions
    - Basic support for numeric types, list comprehension, etc.
- Tree Query Language
  - Allows expression of concise predicates over tree nodes and their relatives
  - Includes tree traversal logistics

General Purpose Subset

General Purpose Programming Language

- Composed of reduced set of declarations and expressions
- Minimal (but Turing-complete)
- Currently no side effects
  - LKQL is designed primarily to express queries

Basic Data Types

- `Unit`
  - Used to represent empty values
- `Int`
  - Basic integer type
  - Supports arbitrary-sized values
- `Str`
  - Built-in string type
  - Supports concatenation
- `Bool`
  - Built-in boolean type
  - Supports expected boolean relational operators
- `Node`
  - Corresponds to syntax nodes of queried source files
- `Token`
  - Corresponds to lexical units of queried source files
- `Pattern`
  - Compiled regular expressions
  - Can be used for string matching
- `Function`
  - Designated by keyword `fun`
  - LKQL functions are first class citizens
  - Represents values that can be called with a call expression

Composite Data Types

- `Tuple`
  - Heterogeneous groups of values with a fixed size
  - Can be indexed to access inner values
- `List`
  - Contiguous immutable sequences of items
  - Can be indexed
  - Supports concatenation
- `Object`
  - Heterogeneous records
  - Contains any number of key-to-value mappings
- `Stream`
  - Supports access by index
  - "Lazy" sequences of items
    - Forces computation until indexed item

Note

`Tuple`, `List`, and `Stream` index starts at 1 (like Ada)

Block Expressions

- Group local declarations and expressions
  - Local values can be declared in the block
    - Designated by keyword `val`
- Create a *local scope*

```
fun maximum_lines(unit, n: int = 10000) =
    {
        val tokens = unit.tokens.to_list;
        val tok    = tokens[tokens.length];

        if tok.end_line > n
        then [{message: "too many lines: " & img(tok.end_line),
               loc: tok}]
        else []
    }
```

- Value of the *block expression* comes from the `if` branches
  - But `tokens` and `tok` can hold temporary values

Declarations

- Functions are callable values
  - Function with a single expression

```
fun add(x, y) = x + y
```

  - Function with block expression

```
fun add(x, y) = {
    |" Add two integers
    val ret = x + y;
    ret
}
```

- Value declarations are used for named values

```
val a = 12 + 15
```

  - Values are immutable

- Docstrings are used for comments

```
|" Function that will add 12 to its first argument
val adder = make_closure(12)
```

  - Part of the syntax tree and are attached to the declaration

Literals and Operators

- Simple literals for booleans, integers, strings, unit, and null

```
val a = true      # Boolean
val b = 12        # Integer
val c = "hello"   # String
val d = ()        # Unit
val e = null      # Null
```

- Integer arithmetic

```
val area = height * width
```

- Relational operators

```
val flag = level > limit
```

- Boolean operators (`and`, `or`, `not`)
  - `and` and `or` use short-circuit evaluation

```
val test = power_on and is_ready(something)
# if "power_on" is false, "is_ready" will not be called
```

- Concatenation

```
"Hello " & name  # String concatenation
[1, 2] & [5, 6]  # List concatenation
```

A Word About Strings

- Simple strings look just like strings in most languages

```
val simple = "Hello"
```

- Multi-line strings are formatted differently
  - Require |" and whitespace before text on each line

```
val multi = |" Hello
            |" This is a multi-line string
            |" Goodbye!
```

- Multi-line string delimiters look just like *docstrings*
  - Differentiated by context

⚠ Warning

Omitting the whitespace after |" is a runtime error, **not** a parse error

Expressions

- *Block expressions* group computations into a single expression
  - May include local declarations
  - Final expression determines block's value

```
{
    val x = 40;
    val y = 2;
    print("DEBUG : " & (x + y).img);
    x + y
}
```

- *Field access* returns contents of a field

```
object_decl.f_type_expr
```

  - Get `f_type_expr` field from `object_decl` node

- *Call expressions* allow functions to call other functions

```
fun add(a, b) = a + b

val c = add(12, 15)
val d = add(a=12, b=15)
```

  - Parameters use either positional or named notation

- *Indexing expressions* allow access to elements of composite data types

```
(1, 2, 3)[1] # Indexing a tuple
list[1]      # Indexing a list
node[x]      # Indexing a node
```

  - List and tuple nodes use list-based indexing
  - Regular nodes use lexical ordering

Comparison Expressions

- Typical comparison operators are available

```
12 < 15
a == b
b != c
```

  - Order-dependent operators (<, >=, etc.) only work on integers

- *Membership expressions* verify value is in the collection
  - Work with `List` and `Stream`

```
12 in list
```

- *Is expressions* verify value matches a **pattern**

```
val a = select AdaNode
val b = a[1] is ObjectDecl
```

- *Pattern* is a construct to match against a value
  - Can match basic or composite values
  - Can match literals (numeric, string, `null`)

Query Language Subset

Query Language

- Composed of three language constructs
- **Patterns**
  - Allow user to express filtering logic on trees and graphs
  - Similar to regular expressions for strings
- **Queries**
  - Select syntax tree nodes based on certain criteria
- **Selectors**
  - Mechanism to perform traversal on the node graph
  - Tree traversal
    - E.g. find the parent of a node
  - Non-syntactic exploration
    - E.g. go from a reference to its declaration

Query Expression

- Simplest part of the query subset
- Simple query

```
# Select all non-null nodes
select AdaNode
```

  - Root of query is implicit
    - Set by context
- Specify query root via `from`

```
# Select all non-null nodes starting from node "theNode"
from theNode select AdaNode
```

- Specifying a selector using `through`

```
# Selects parents of first basic declaration
from (select first BasicDecl) through parent select *
```

Pattern

- A construct to match against a value
  - But can get complicated very quickly!
- LKQL checks that value matches the pattern
  - Matches get added to result of query

Pattern Categories

- Node patterns
  - Match one (or many) node kinds
    - Node kind name - match all nodes of this kind
    - Pattern - match multiple node kinds
    - Wildcard pattern - match all node kinds

```
select *                          # Selects every node
select BasicDecl                  # Selects every basic declaration
select (ObjectDecl | BaseTypeDecl) # Selects every object and type declaration
```

- Regular values patterns
  - Match a value based on a pattern

```
v is 12              # Integer pattern
v is "hello.*?world" # Regex pattern
```

- Filtered patterns
  - Use `when` for an arbitrary boolean condition

```
select BasicDecl when some_condition
```

Selectors

- A *selector* is a function that returns a `Stream` of values
  - Query expression explores tree via built-in selector `children`
- Selector does not have parameters
  - But can take optional arguments
    - `min_depth` - return values at this depth or deeper
    - `max_depth` - return values at this depth or shallower
    - `depth` - return values at exactly this depth

```
# Calling selectors directly
val kids = children(node, depth=3)

# Calling a selector in a nested sub-pattern
select AdaNode(any children(min_depth=3): BasicDecl)
```

Built-in Selectors

- `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

Building a Boolean Rule

Example of a Very Simple Rule

- We want a rule to find all objects of `Integer` types
  - Because we know `Integer` may not be completely portable
- We start by creating our rule file `integer_objects.lkql`
  - First determine if the node is an object

```
@check
fun integer_objects(node) =
    node is ObjectDecl
```

  - Returns `true` for any object declaration

Checking Object Type

- Object declaration has field `f_type_expr`
  - Returns **`TypeExpr`**

  We want to know the root type of the object
- Type expression has property `p_designated_type_decl()`
  - Returns **`BaseTypeDecl`**

  This is just the type declaration - we need the root type
- Type declaration has property `p_root_type()`
  - Returns **`BaseTypeDecl`**

  This is the root type of the derivation hierarchy
- Base type declaration has field `f_name`
  - Returns **`DefiningName`**

  We want the name as a string
- Defining Name has property `p_fully_qualified_name()`
  - Returns `str`

  We want to see if this is `Integer`

Putting It All Together

`integer_objects.lkql`

```
@check
fun integer_objects(node) =
    node is ObjectDecl and
    node.f_type_expr
        .p_designated_type_decl()
        .p_root_type()
        .f_name
        .p_fully_qualified_name() == "Standard.Integer"
```

`main.adb`

```
1  procedure Main is
2     type Integer_T is new Integer range -1_000 .. 1_000;
3     type Number_T is range -1_000 .. 1_000;
4     type Float_T is digits 6 range -1_000.0 .. 1_000.0;
5     One   : Integer_T := 11;
6     Two   : Number_T  := 222;
7     Three : Float_T   := 3.3e-3;
8  begin
9     One   := One + 1;
10    Two   := Two + 1;
11    Three := Three * 1.1;
12 end Main;
```

```
gnatcheck -P default.gpr --rules-dir=.  --rule integer_objects
```

```
main.adb:5:04: rule violation: integer_objects
main.adb:6:04: error: internal issue at integer_objects.lkql:18:07:
    Null receiver in dot access [integer_objects]
main.adb:7:04: error: internal issue at integer_objects.lkql:18:07:
    Null receiver in dot access [integer_objects]
```

Cleaning Up the Rule

```
Null receiver in dot access [integer_objects]
```

- Many fields and properties downstream from `ObjectDecl` can be null
  - But our rule assumes we can always dereference the results
- Check for null before accessing fields or calling properties

integer_objects.lkql

```
@check
fun integer_objects(node) =
{
    val is_object = node is ObjectDecl;
    val has_type = is_object and node.f_type_expr != null;
    val has_designated = has_type and
        node.f_type_expr.p_designated_type_decl() != null;
    val has_root = has_designated and
        node.f_type_expr.p_designated_type_decl().p_root_type() != null;
    val has_name = has_root and
        node.f_type_expr.p_designated_type_decl().p_root_type().f_name != null;
    has_name and node.f_type_expr
                .p_designated_type_decl()
                .p_root_type()
                .f_name
                .p_fully_qualified_name() == "Standard.Integer"
}
```

```
main.adb:5:04: rule violation: integer_objects
```

Note

An *if expression* would be cleaner - we'll see that later

Messages

- By default, rule violation message contains the name of the rule

```
main.adb:5:04: rule violation: integer_objects
```

- Customize the diagnostic using the `message` argument in `@check`

```
@check(message="Integer object found")
fun integer_objects(node) =
```

```
main.adb:5:04: rule violation: Integer object found
```

Lab

A Very Simple Rule

This lab is a hands-on walkthrough of creating your own LKQL rule for use with GNATcheck. You can use any text editor to create this rule file.

Our "imaginary" coding standard requires no abstract types in the code, so we just want to flag any abstract type when we find one.

Tip

The lab is a step-by-step guide. Many of the steps have hints in addition to solutions. If the page has empty space, pressing Page Down will usually get you that extra help!

Example Project

- Copy the `introduction_to_lkql` folder from the course materials location
- Contents of the folder:
  - `default.gpr` - project file
  - `src` - source directory
    - Contains some source code that happens to use abstract types

Step 1 - Create Rule File

- Create rule **flag_abstract_types**
  - In file `flag_abstract_types.lkql` in current directory
  - Use `@check` annotation
- For now, just return `true`
  - This should flag **every** node as a violation

Step 1 - Create Rule File

- Create rule **flag_abstract_types**
  - In file `flag_abstract_types.lkql` in current directory
  - Use `@check` annotation
- For now, just return `true`
  - This should flag **every** node as a violation

```
@check
fun flag_abstract_types(node) =
    true
```

- Run the rule using GNATcheck and specify
  - Directory containing the rule file
  - Name of rule

Step 1 - Create Rule File

- Create rule **flag_abstract_types**
  - In file `flag_abstract_types.lkql` in current directory
  - Use `@check` annotation
- For now, just return `true`
  - This should flag **every** node as a violation

```
@check
fun flag_abstract_types(node) =
    true
```

- Run the rule using GNATCHECK and specify
  - Directory containing the rule file
  - Name of rule

```
gnatcheck -P default.gpr --rules-dir=.  -r flag_abstract_types
```

If you do not get a lot of rule violations, you did it wrong!

```
loggers.adb:1:01: rule violation: flag_abstract_types
loggers.adb:1:08: rule violation: flag_abstract_types
loggers.adb:1:16: rule violation: flag_abstract_types
loggers.adb:1:17: rule violation: flag_abstract_types
...
```

Step 2 - Filter For Types

- Now, only return `true` if the node is a type declaration

Step 2 - Filter For Types

- Now, only return `true` if the node is a type declaration

```
@check
fun flag_abstract_types(node) =
    node is TypeDecl
```

Node types like `TypeDecl` *are in the LKQL API section of the documentation*

https://docs.adacore.com/live/wave/lkql/html/gnatcheck_rm/gnatcheck_rm/lkql_language_reference.html#node-types

- Running the command now generates a lot fewer messages

Step 2 - Filter For Types

- Now, only return `true` if the node is a type declaration

```
@check
fun flag_abstract_types(node) =
    node is TypeDecl
```

Node types like `TypeDecl` *are in the LKQL API section of the documentation*

https://docs.adacore.com/live/wave/lkql/html/gnatcheck_rm/gnatcheck_rm/lkql_language_reference.html#node-types

- Running the command now generates a lot fewer messages

```
loggers.ads:3:09: rule violation: flag_abstract_types
loggers.ads:4:09: rule violation: flag_abstract_types
loggers.ads:11:09: rule violation: flag_abstract_types
loggers.ads:18:09: rule violation: flag_abstract_types
...
```

Step 3 - Find Abstract Types

- Now, only return `true` if the type declaration has `abstract`
- Hint: `TypeDecl` is derived from `BaseTypeDecl`

Step 3 - Find Abstract Types

- Now, only return `true` if the type declaration has `abstract`
- Hint: `TypeDecl` is derived from `BaseTypeDecl`
- `BaseTypeDecl` has many fields and properties
  - So `TypeDecl` inherits all of them

```
@check
fun flag_abstract_types(node) =
    node is TypeDecl and
    node.p_is_abstract_type()
```

- Running the command now generates messages as appropriate

Step 3 - Find Abstract Types

- Now, only return `true` if the type declaration has `abstract`
- Hint: `TypeDecl` is derived from `BaseTypeDecl`
- `BaseTypeDecl` has many fields and properties
  - So `TypeDecl` inherits all of them

```
@check
fun flag_abstract_types(node) =
    node is TypeDecl and
    node.p_is_abstract_type()
```

- Running the command now generates messages as appropriate

```
loggers.ads:3:09: rule violation: flag_abstract_types
loggers.ads:11:09: rule violation: flag_abstract_types
loggers.ads:18:09: rule violation: flag_abstract_types
loggers.ads:22:09: rule violation: flag_abstract_types
loggers.ads:57:09: rule violation: flag_abstract_types
loggers.ads:58:09: rule violation: flag_abstract_types
loggers.ads:60:09: rule violation: flag_abstract_types
loggers.ads:62:09: rule violation: flag_abstract_types
```

Step 4 - Better Message

- The current error message is not very friendly
- Update the rule message to be more helpful

Step 4 - Better Message

- The current error message is not very friendly
- Update the rule message to be more helpful

```
@check(message="Abstract type found")
fun flag_abstract_types(node) =
    node is TypeDecl and
    node.p_is_abstract_type()
```

- Running the command now generates the new messages

Step 4 - Better Message

- The current error message is not very friendly
- Update the rule message to be more helpful

```
@check(message="Abstract type found")
fun flag_abstract_types(node) =
    node is TypeDecl and
    node.p_is_abstract_type()
```

- Running the command now generates the new messages

```
loggers.ads:3:09: rule violation: Abstract type found
loggers.ads:11:09: rule violation: Abstract type found
loggers.ads:18:09: rule violation: Abstract type found
loggers.ads:22:09: rule violation: Abstract type found
loggers.ads:57:09: rule violation: Abstract type found
loggers.ads:58:09: rule violation: Abstract type found
loggers.ads:60:09: rule violation: Abstract type found
loggers.ads:62:09: rule violation: Abstract type found
```

Extra Credit - Removing Duplication

- Note that `loggers.ads` flags the same type twice
  - Public view

```
package Loggers is

   type Abstract_Logger is abstract tagged limited private;
```

  - Private declaration

```
private

   type Abstract_Logger is abstract tagged limited null record;
```

  These are just some of the doubly-flagged types
- Update the rule to skip the public view
  - Hint: find a property that returns a type's private completion, if any

Extra Credit - Removing Duplication

- Note that `loggers.ads` flags the same type twice
  - Public view

```
package Loggers is

   type Abstract_Logger is abstract tagged limited private;
```

  - Private declaration

```
private

   type Abstract_Logger is abstract tagged limited null record;
```

  These are just some of the doubly-flagged types
- Update the rule to skip the public view
  - Hint: find a property that returns a type's private completion, if any

```
@check(message="Abstract type found")
fun flag_abstract_types(node) =
    node is TypeDecl and
    node.p_is_abstract_type() and
    node.p_private_completion() == null
```

- Running the command now generates one less message per type skipped

Extra Credit - Removing Duplication

- Note that `loggers.ads` flags the same type twice
  - Public view

```
package Loggers is

   type Abstract_Logger is abstract tagged limited private;
```

  - Private declaration

```
private

   type Abstract_Logger is abstract tagged limited null record;
```

  These are just some of the doubly-flagged types
- Update the rule to skip the public view
  - Hint: find a property that returns a type's private completion, if any

```
@check(message="Abstract type found")
fun flag_abstract_types(node) =
    node is TypeDecl and
    node.p_is_abstract_type() and
    node.p_private_completion() == null
```

- Running the command now generates one less message per type skipped

```
loggers.ads:57:09: rule violation: Abstract type found
loggers.ads:58:09: rule violation: Abstract type found
loggers.ads:60:09: rule violation: Abstract type found
loggers.ads:62:09: rule violation: Abstract type found
```

Summary

What We Covered

- **What is LKQL?**
  - Analyzes libadalang-parsed code and makes queries
- **General Purpose Programming Language**
  - Simple language for performing queries
  - Declarations and expressions
- **Query Language**
  - Queries parsed source code
  - Pattern-matching mechanisms
    - `is`, `select`, `when`
  - Selectors
    - Select nodes within parsed code
    - Automatic traversal
- **Building a Rule**
  - LKQL is not a very complicated language
  - Limitations can be overcome!

Intermediate LKQL

Introduction

Topics Covered

- **Expressions**
  - Expanding expressions with values, selectors, filters, and more
- **Functions**
  - More function categories, parameters, and annotations
- **Patterns**
  - Filtering query results through pattern matching
- **Building a Unit Rule**

Expressions

Review: Simple Expressions

- Previous module covered some simple LKQL expressions
  - Including blocks, comparisons, etc.
- This chapter covers the "next level" of expressions
  - Non-numeric literals
  - Control flow (`if` and `match`)
  - List comprehension
  - Dealing with null/empty values

Simple Literals

- LKQL supports the following basic *types*
  - Boolean
  - Integer
  - String
  - Unit
- Literals are available for these types

```
val flag = true    # Boolean
val count = 3      # Integer
val name = "Fred"  # String
val empty = ()     # Unit
```

Note

Unit is an **empty** result - use `null` when value doesn't exist

Composite Literals

- Object literal
  - Object with named fields
    - Field names are case-insensitive

```
val object = {lower: "Hello", UPPER: "World"}
print (object.lower & " " & object.upper)
```

- List literal
  - Contiguous list of values

```
val list = [1, 2, 3, 4]
```

- Tuple literal
  - Heterogeneous group of values

```
val tuple = (1, "Hello", true)
```

"if" and "match"

- `if` expression follows one of two paths
  - Based on a boolean condition

```
val expr_index = if is_return then 1 else 2;
```

  - Boolean `if` expressions do not require an `else`
    - If condition is false and no `else` expression, `true` is the value
- `match` expression allows pattern-matching to determine the path

```
match n.f_prefix.p_referenced_decl()
| BasicSubpDecl => true
| SubpBody      => true
| *             => false
```

```
match node
| ParenExpr => node.f_expr
| QualExpr  => node.f_suffix
```

  - Arms are evaluated in order
    - First matching arm is taken
    - If no matching arm, run-time exception is raised
    - Optional catch-all `*` should come last

List Comprehension

- LKQL is functional, so no `for` loops
- List comprehension allows construction by embedding the `for`

```
[<element_expression> for <identifier> in <query_expression>]
```

  - Build a sequence of *element_expression* elements such that
    - *Query_expression* returns results named *identifier*
    - *Identifier* can be used to build *element_expression*

```
[item for item in subprograms if name == item.to_lower_case]
```

  - For every `item` in the list of subprograms
    - If `name` matches the item (the subprogram name)
    - Then add `item` to the list being returned

Note

- List comprehension actually builds a `Stream`, not a list
- Elements of a `Stream` are not computed until referenced

"Safe" Variant

- Queries in LKQL can return mutliple literals indicating no result
  - **()** - no meaningful value ("Unit")
  - **[]** - list that has no values (empty list)
  - **null** - no node present
- `?.` is used to simplify accessing past a `null` value
  - Allows chaining of empty values

```
val params = node?.f_subp_params?.f_params;
```

  - `params` will contain a `ParamSpecList` if
    - `node` is not null
    - `node` is a subprogram specification
    - `node` has at least one parameter

Note

`?.` is also called *safe field access*

Unwrap Expression

- Safe field access returns a *nullable value*
  - Either `null` or an object
- Use the *unwrap expression* `!!` to require a non-null value
  - Typically used at the end of a chain of safe field accesses

```
object_decl?.p_type_expression()?.p_designated_type_decl()!!
```

⚠ Warning

If the chain returns `null`, the unwrap will raise an error

Functions

Review: Functions

- The previous module introduced basic functions
  - Building block of LKQL
- Simple function to add two numbers

```
fun add(left, right) = left + right
```

  - **Function name:** `add`
  - **Parameters:** `left`, `right`
    - As there is no global data, most functions will have parameters

Decorators (1/2)

- Preface functions with *decorators* to describe their behavior
- `@check` indicates rule returning a boolean value
  - Operates on every syntax tree node within the source code
  - Returns true if node violates the rule
- `@unit_check` indicates a unit rule
  - Operates on every library unit within the source code
  - Returns a *list* of failures within the unit
    - Each element of the list is a message and a location
    - Message and location indicate code that violates the rule

Note

Files can have **either** `@check` **or** `@unit_check` but not both

Files containing support functions might have neither

Decorators (2/2)

- `@memoized` is a function that caches its result
  - Calls with same arguments return cached value
    - Functions cannot have side effects
    - So result must be the same
- Decorators can have annotations
  - **message** - violation message to be displayed
  - **help** - text displayed in a help message
  - Others beyond the scope of this course

```
@check(message="renaming declaration")
fun renamings(node) = ...

@unit_check(help="usage of specified aspects")
fun forbidden_aspects(unit) = ...
```

Nesting

- LKQL allows nested functions
  - Closures allowed (return reference to local environment)

```
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))
```

Function "Pointers"

- Functions are first-class values
  - Named value can store a function value
    - Assigned different values based on a condition
    - Setting to `null` would indicate no call
  - `fn? (a, b)` is the syntax for a safe function call

```
fun div(a, b) = a / b

fun safe_div(a, b) = {
    val fn = if b == 0 then null else div;
    fn? (a, b) # Return () if b = 0, else a/b
}
```

- If `b == 0`, `fn` will be null, otherwise it points to `div`
- When treating `fn` as a function call
  - If `fn` is null, return `()`
  - Else calls `div(a, b)` and returns its result

Patterns

Review: Patterns

- Previous module covered basic patterns
- Node patterns
  - Select node(s) that match some criteria
- Value patterns
  - Compare two values
- Filtered patterns
  - Filter selection based on some criteria

Binding Patterns

- Sometimes we want to refer to the values in the pattern we matched

```
select Identifier # return all identifiers
```

  - How do we refer to each identifier returned?

- *Binding pattern* allows us to map the returned items to a name

```
select id @ Identifier # return all identifiers
```

  - We can now pass `id` as a reference to the identifier

Node Patterns

- Node patterns have optional parenthesized expressions
  - Used to refine the query
- **Selector Predicate**
  - Match results of a sub-query

```
select Body (any children: ForLoopStmt)
```

Select all bodies where any child has a `for` statement

  - `any` - at least one child matches
  - `all` - all children match

- **Field Predicate**
  - Matches if the specified field has the specified pattern

```
select ObjectDecl(f_default_expr: IntLiteral)
```

Select all object declarations where default expression is integer literal

- **Property Call Predicate**
  - Similar to field predicate
  - Matches a property call's result against a pattern

```
select Identifier(p_name_matches("some_string"): true)
```

Select identifiers whose name match the identifier `"some_string"`

Regular Value Patterns

- Pattern matching can extend to any values
  - Even for composite objects
- **Tuple Pattern**

```
match i
# match exactly 1, 2, 3
| (1, 2, 3) => print("One, Two, Three")
# match where first element is 1, third element is 3
#   then print the second element
| (1, a@*, 3) => { print(a) }
```

- **List Pattern**

```
match lst
# match exactly 1, 2, 3
| [1, 2, 3]   => { print ("[1, 2, 3]") }
# match where first element is 1, third element is 3
#   then print the second element
| [1, a@*, 3] => { print ("[1, a@*, 3], with a = " & img(a)) }
```

- **Object Pattern**

```
match obj
| {a: 12}  => "{a: 12}"
| {a: a@*} => "Any object with an a key. Bind the result to a"
```

Special Case Patterns

- **Null Pattern**
  - Match all null nodes with this pattern

```
match node
| BasicDecl => "A BasicDecl node"
| null      => "Node is null!"
```

- **Not Pattern**
  - Use this pattern to negate another one

```
match v
| not BasicDecl => "Everything except a BasicDecl node"
| *             => "A BasicDecl node"
```

- **Or Pattern**
  - Use this pattern to match one of a set of patterns
    - Match any value matching one of those

```
match v
| (BasicDecl | 1) => "A BasicDecl node or 1"
| *               => "Any other value"
```

Flexible Pattern Matching

- **Wildcard Pattern**
  - Match all values with this pattern
    - Will always return true

```
match any_val
| BasicDecl => "A BasicDecl node"
| *         => "Any other value"
```

- **Splat Pattern**
  - Used inside *List Pattern* and *Object Pattern*
    - Match all remaining values

```
match v
| [1, rem@...]     => "List with first element '1' followed by " &
                      img(rem)
| {a: 1, rem@...} => "An object with a=1 and " & img(rem)
```

Filtered Patterns

- Pattern using an arbitrary boolean condition
  - Uses `when` clause

```
select BasicDecl when some_condition
```

- Results of query need to be named
  - If passing to another query or function

```
select b @ Identifier when b.p_is_defining()
```

Unit Rules

What Is a Unit Rule?

- Previously our rules worked on nodes

```
@check
fun abort_statements(node) =
    node is AbortStmt
```

  - Return `true` if the node is an `abort` statement

- But some rules need more context
  - Check if number of lines in file exceeds some maximum
  - Flag any file containing end-of-line comments
- Both of these rules could be written as simple rules
  - But would generate messages for **every** node that matches
- We need rules to check if the **unit** fails, not the **node**

Basic Unit Rule

- Starts with `@unit_check` rather than `@check`
- Major differences from boolean rules
  - Operate on analysis units instead of syntax tree nodes
  - Return a message list rather than a boolean value
    - List can flag multiple violations in a single unit
    - List can be empty if no violations found
- Simple rules trigger once per unit
  - Unit length cannot exceed some maximum number of lines
  - Unit must start with some predefined text
- Other rules trigger multiple times per unit
  - Subprogram length cannot exceed some maximum number of lines
  - Use of disallowed aspects

Expanded Messages for Unit Rules

- In a boolean rule, the default message could be replaced with a text explanation

```
@check(message="Integer object found")
fun integer_object(node) = ...
```

- In a unit rule, messages must be in a list
  - So they are defined in the body of the rule

```
{message: "mark BEGIN with -- " & n.parent.p_defining_name().text,
 loc: n.token_start()}
```

  - **message** is what is displayed to the user
    - Can use concatenation to add specific information to the message
  - **loc** is the node (or token) that triggered the rule
    - Used to display where in the code that rule was triggered

```
main.adb:6:07: rule violation: mark BEGIN with -- Proc_One
main.adb:17:07: rule violation: mark BEGIN with -- Proc_Three
```

Simple Unit Rule

- Some rules need to flag a unit only one time
  - E.g. unit length rule
    - Violation should appear for the unit
    - Not for every node past the end of the line
- The rule still needs to return a **list** of messages

```
@unit_check
fun maximum_lines(unit, n: int = 10000) =
    {
        val tokens = unit.tokens.to_list;
        val tok    = tokens[tokens.length];

        if tok.end_line > n
        # Return a message as the only element of a list
        then [{message: "too many lines: " & img(tok.end_line), loc: tok}]
        # Return an empty list
        else []
    }
```

  - Flag unit if last token ends on a line greater than passed-in value

Rules That Trigger Multiple Times

- Rules can trigger multiple times (e.g. subprogram length exceeded)
  - **List comprehension** allows rule to build stream of multiple messages

```
import stdlib
@unit_check
fun end_of_line_comments(unit) =
    [
        {message: "end of line comment", loc: tok}
        for tok in unit.tokens
        if tok.kind == "comment" and
           stdlib.previous_non_blank_token_line(tok) == tok.start_line
    ]
```

- Build a list of messages such by looping through the unit tokens
  - If a token is a comment and there is code on the same line
    - Add message "end of line comment" for the current token location

```
6  -- Regular comment
7  One   : Integer_T := 11;      -- end of line comment
8  Two   : Number_T  := 222;
9  Three : Float_T   := 3.3e-3; -- end of line comment
10 -- Regular comment
```

```
main.adb:7:33: rule violation: end of line comment
main.adb:9:33: rule violation: end of line comment
```

Support Functions

- Support functions can return anything (boolean, integer, list, etc.)
  - No decorators required

```
fun count_lines(node) =
    node.token_end().end_line - node.token_start().start_line + 1

@unit_check(help="maximum number of lines in a subprogram",
            category="Style", subcategory="Program Structure")
fun maximum_subprogram_lines(unit, max_lines: int = 1000) =
    [
        {message: "too many lines in subprogram body: " &
                  img(count_lines(loop_node)),
         loc: loop_node.token_start().previous(exclude_trivia=true)}
        for loop_node in from unit.root
        select node@HandledStmts(parent: SubpBody)
            when count_lines(node) > max_lines
    ]
```

- Could rewrite the query as

```
select node@HandledStmts(parent: SubpBody)
  when max_lines <
       node.token_end().end_line - node.token_start().start_line + 1
```

  - But it would need to be repeated in the message
  - And the query reads better with a function call

"Memoized" Functions

```
@memoized
fun num_primitives(t) = t.p_get_primitives().length

@unit_check
fun too_many_primitives(unit, max_count : int = 5) =
    [
        {message: "tagged type has too many primitives (" &
                  img(num_primitives(loop_node)) & ")",
         loc: loop_node.p_defining_name()}
        for loop_node in from unit.root through follow_generics
        select node@TypeDecl
        when node.p_is_tagged_type() and node.parent.parent is PublicPart
         and num_primitives(node) > max_count
    ]
```

- `num_primitives` is called multiple times
  - Display information for the message
  - Part of the query for determining too many primitives
- Performing its own query (`p_get_primitives`) can get expensive
- But functions cannot have side-effects (no global data)
  - So if called with the same parameter, results are the same
- `@memoized` tells analyzer to cache the result
  - And return the value from the cache when called with the same value

Building a Unit Rule

Example of a Simple Unit Rule

- We want a rule to find complex expressions
  - As defined by "expressions with more than N subexpressions"
    - In our case, **N** will be 2

```
5 Value := 1 + 2 + 3;
6 Value := Value * 4;
7 Do_Something (5 + 6, 7, Value * 8 / 9);
```

```
main.adb:5:14: rule violation: expression has too many sub-expressions (3)
main.adb:7:05: rule violation: expression has too many sub-expressions (7)
main.adb:7:25: rule violation: expression has too many sub-expressions (3)
```

- Line 5 has 3 subexpressions (**1**, **2**, **3**)
- Line 6 has 2 subexpressions (**Value**, **4**)
- Line 7 has 7 subexpressions (**Do_Something**, **5**, **6**, **7**, **Value**, **8**, **9**)
  - Line 7 also flags the third parameter with 3 sub-expressions (**Value**, **8**, **9**)

Note

This rule already exists in GNATcheck as `maximum_expression_complexity`

Actual Rule Implementation

```
fun num_expr(node) =
    (from node select ((e@SingleTokNode when e is not Op) | CondExpr |
                       QuantifiedExpr | BaseAggregate | TargetName)).length

@unit_check(help="maximum complexity of an expression",
            category="Style", subcategory="Program Structure")
fun maximum_expression_complexity(unit, n: int = 10) =
    [
        {message: "expression has too many sub-expressions (" &
                  img(num_expr(node)) & ")", loc: node}
        for node in from unit.root select
        expr@Expr(parent: not Expr)
        when expr is not (Identifier | DefiningName | EndName)
         and num_expr(expr) > n
    ]
```

Note

The actual rule file contains **docstrings** for describing the functions

Parameterized Rules

- Many rules use some kind of limit to flag violations
  - Nesting depth
  - Number of lines in a subprogram
  - Etc.
- Making the limit configurable gives us a *parameterized* rule

```
fun maximum_expression_complexity(unit, n: int = 10) =
```

  - Default behavior is to flag expressions with complexity > 10

```
gnatcheck -P default.gpr -r maximum_expression_complexity
```

- `-r` enables the rule but does not allow parameter values
  - Use a *rules file* to specify a value

`rules_file.lkql`

```
val rules = @{
  maximum_expression_complexity: {N: 2}
}
```

Flag expressions whose complexity > 2

```
gnatcheck -P default.gpr --rule-file=rules_file.lkql
```

Examining the Rule - Query

```
for node in from unit.root select
# Loop over all nodes in the unit

expr@Expr(parent: not Expr)
# Match expressions whose immediate parent
# is not an expression and bind each to "expr"

when expr is not (Identifier | DefiningName | EndName)
          and num_expr(expr) > n
# Trigger the rule when "expr" is not a name
# and number of expressions is > "n"
```

Examining the Rule - Support Function

```
fun num_expr(node) =
    (from node select (
        (e@SingleTokNode when e is not Op) |
        CondExpr |
        QuantifiedExpr |
        BaseAggregate |
        TargetName)).length
```

- Build a list of nodes such that each element is one of
  - Single token, excluding operators
  - Conditional expression
  - Quantified expression
  - Aggregate
  - `@` (Ada 2022 "target" symbol)
- Return the number of matching nodes

Examining the Rule - Decorator

```
@unit_check(help="maximum complexity of an expression",
            category="Style", subcategory="Program Structure")
```

- **help** - sets help text for rule
- **category** - text used to describe broad classification of rule
- **subcategory** - text used to further describe rule usage

Lab

A Common Unit Rule

In this lab we are going to build a "unit rule" to validate that the number of lines in a subprogram do not exceed some value.

We are using a "unit rule" because we only want to search for subprograms within a unit - no need to run the rule for every node.

Note

This rule already exists in GNATCHECK as `maximum_subprogram_lines`. Because rule names cannot conflict, we're using a different name here as we learn how the delivered rule was created.

Tip

The lab is a step-by-step guide. Many of the steps have hints in addition to solutions. If the page has empty space, pressing Page Down will usually get you that extra help!

Example Project

- Copy the `intermediate_lkql` folder from the course materials location
- Contents of the folder:
  - `default.gpr` - project file
  - `src` - source directory
    - Contains similar source code to the previous chapter lab
    - Package body now has some (mostly harmless) subprogram implementations

Step 1 - Create Rule File

- Create rule `flag_large_subprograms`
  - In file `flag_large_subprograms.lkql`
  - Use `@unit_check` annotation
  - This rule will require a parameter `Lines`
    - So we only flag subprograms longer than `Lines`
- For now, just return a message indicating a violation

Step 1 - Create Rule File

- Create rule `flag_large_subprograms`
  - In file `flag_large_subprograms.lkql`
  - Use `@unit_check` annotation
  - This rule will require a parameter `Lines`
    - So we only flag subprograms longer than `Lines`
- For now, just return a message indicating a violation

```
@unit_check
fun flag_large_subprograms(unit, lines: int = 0) = [
   {message: "unit flagged", loc: unit.root}
]
```

Note

*This rule returns a **list literal** to build the list of values to return. Right now, the list contains just one element (for the selected unit)*

Step 2 - Dealing With Parameters

- We need a mechanism to pass a value to **Lines**
  - We cannot do it from the command line
- Instead, we will need to build a rules file and update the GNATcheck call
- Rules file looks like LKQL with a list of rules to apply

`rules_file.lkql`

```
val rules = @{
  Flag_Large_Subprograms: {Lines: 30}
}
```

where `Lines` *will be the parameter name and* 30 *the value*

- To run this, we now pass in the rules file to GNATcheck

```
gnatcheck -P default.gpr --rules-dir .  --rule-file=rules_file.lkql
```

```
loggers.adb:1:01: rule violation: unit flagged
loggers.ads:1:01: rule violation: unit flagged
```

Step 3 - Checking Each Subprogram

- As written, it is the unit itself that is being flagged
  - We want to flag subprograms
- Update the rule to select subprograms from the unit
  - Update the message to include the subprogram name in the message

Step 3 - Checking Each Subprogram

- As written, it is the unit itself that is being flagged
  - We want to flag subprograms
- Update the rule to select subprograms from the unit
  - Update the message to include the subprogram name in the message

```
@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "subprogram body flagged: " & subp.p_defining_name().text,
         loc: subp}
        for subp in from unit.root select SubpBody
    ]
```

- Run GNATCHECK to see the new results

Step 3 - Checking Each Subprogram

- As written, it is the unit itself that is being flagged
  - We want to flag subprograms
- Update the rule to select subprograms from the unit
  - Update the message to include the subprogram name in the message

```
@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "subprogram body flagged: " & subp.p_defining_name().text,
         loc: subp}
        for subp in from unit.root select SubpBody
    ]
```

- Run GNATCHECK to see the new results

```
loggers.adb:9:04: rule violation: subprogram body flagged: Severity_To_String
loggers.adb:31:04: rule violation: subprogram body flagged: Log_Message
loggers.adb:93:04: rule violation: subprogram body flagged: Enable_Color_Output
loggers.adb:135:04: rule violation: subprogram body flagged: Should_Log
loggers.adb:158:04: rule violation: subprogram body flagged: Create_Filter
```

Note

List comprehension means we now get an element for every subprogram body we find in the unit.

Step 4 - Finding the Subprogram Statements

- We now need to get the statements in the subprogram
- Update the query to select `HandledStmts` nodes whose parent is a `SubpBody`

Note

There is no easy way for the message to get the subprogram name from the statement list, so just remove that part of the `message` argument

Step 4 - Finding the Subprogram Statements

- We now need to get the statements in the subprogram
- Update the query to select `HandledStmts` nodes whose parent is a `SubpBody`

Note

There is no easy way for the message to get the subprogram name from the statement list, so just remove that part of the `message` argument

```
@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "subprogram body flagged: ",
         loc: subp}
        for subp in from unit.root
            select HandledStmts(parent: SubpBody)
]
```

- Run GNATcheck to see the new results (they're different!)

Step 4 - Finding the Subprogram Statements

- We now need to get the statements in the subprogram
- Update the query to select `HandledStmts` nodes whose parent is a `SubpBody`

Note

There is no easy way for the message to get the subprogram name from the statement list, so just remove that part of the `message` argument

```
@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "subprogram body flagged: ",
         loc: subp}
        for subp in from unit.root
            select HandledStmts(parent: SubpBody)
]
```

- Run GNATcheck to see the new results (they're different!)

```
loggers.adb:12:09: rule violation: subprogram body flagged:
loggers.adb:35:09: rule violation: subprogram body flagged:
loggers.adb:96:09: rule violation: subprogram body flagged:
loggers.adb:139:09: rule violation: subprogram body flagged:
loggers.adb:161:09: rule violation: subprogram body flagged:
```

- The line numbers are now tracking to the `begin` rather than subprogram declaration

Step 5 - Counting the Number of Lines

- We need a way to count how many lines are in the subprogram
  - We can measure the source span of the statement sequence
- Simplest calculation
  - Difference between line of first token on first line and last token on last line
- Hints:
  - Need to give a name to the object returned by the `select`
  - `AdaNode` has properties `token_start` and `token_end`
  - `Token` has fields `end_line` and `start_line`

Step 5 - Counting the Number of Lines

- We need a way to count how many lines are in the subprogram
  - We can measure the source span of the statement sequence
- Simplest calculation
  - Difference between line of first token on first line and last token on last line
- Hints:
  - Need to give a name to the object returned by the `select`
  - `AdaNode` has properties `token_start` and `token_end`
  - `Token` has fields `end_line` and `start_line`

```
@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "subprogram body flagged: ",
         loc: subp}
        for subp in from unit.root
            select stmts@HandledStmts(parent: SubpBody)
                when lines < stmts.token_end().end_line -
                             stmts.token_start().start_line + 1
    ]
```

- Run GNATcheck to see the "correct" results

Step 5 - Counting the Number of Lines

- We need a way to count how many lines are in the subprogram
  - We can measure the source span of the statement sequence
- Simplest calculation
  - Difference between line of first token on first line and last token on last line
- Hints:
  - Need to give a name to the object returned by the `select`
  - `AdaNode` has properties `token_start` and `token_end`
  - `Token` has fields `end_line` and `start_line`

```
@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "subprogram body flagged: ",
         loc: subp}
        for subp in from unit.root
            select stmts@HandledStmts(parent: SubpBody)
                when lines < stmts.token_end().end_line -
                             stmts.token_start().start_line + 1
    ]
```

- Run GNATCHECK to see the "correct" results

```
loggers.adb:35:09: rule violation: subprogram body flagged:
loggers.adb:161:09: rule violation: subprogram body flagged:
```

Step 6 - Using Functions for Modularity

- Rather than doing the calculation in-line, create a function
  - Hint: function parameter will be `HandledStmts`

Step 6 - Using Functions for Modularity

- Rather than doing the calculation in-line, create a function
  - Hint: function parameter will be `HandledStmts`

```
fun count_lines(stmts) =
    stmts.token_end().end_line - stmts.token_start().start_line + 1

@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "subprogram body flagged: ",
         loc: subp}
        for subp in from unit.root
            select stmts@HandledStmts(parent: SubpBody)
                when lines < count_lines(stmts)
    ]
```

- Run GNATcheck and verify messages haven't changed

Step 7 - Updating the Message for Clarity

- Now update the message to be more descriptive
  - Show number of lines in failing subprogram
  - Use start of statement list for error message
- Hints
  - The name of the object in the message is the iterator (`subp`)

Step 7 - Updating the Message for Clarity

- Now update the message to be more descriptive
  - Show number of lines in failing subprogram
  - Use start of statement list for error message
- Hints
  - The name of the object in the message is the iterator (`subp`)

```
fun count_lines(stmts) =
    stmts.token_end().end_line - stmts.token_start().start_line + 1

@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "too many lines in subprogram body: " & img(count_lines(subp)),
         loc: subp.token_start()}
        for subp in from unit.root
            select stmts@HandledStmts(parent: SubpBody)
                when lines < count_lines(stmts)
    ]
```

With the `count_lines` function, we have access to the calculation in multiple places.

- Run GNATcheck one last time!

Step 7 - Updating the Message for Clarity

- Now update the message to be more descriptive
  - Show number of lines in failing subprogram
  - Use start of statement list for error message
- Hints
  - The name of the object in the message is the iterator (`subp`)

```
fun count_lines(stmts) =
    stmts.token_end().end_line - stmts.token_start().start_line + 1

@unit_check
fun flag_large_subprograms(unit, lines: int = 1000) =
    [
        {message: "too many lines in subprogram body: " & img(count_lines(subp)),
         loc: subp.token_start()}
        for subp in from unit.root
            select stmts@HandledStmts(parent: SubpBody)
                when lines < count_lines(stmts)
    ]
```

With the `count_lines` function, we have access to the calculation in multiple places.

- Run GNATCHECK one last time!

```
loggers.adb:36:07: rule violation: too many lines in subprogram body: 51
loggers.adb:162:07: rule violation: too many lines in subprogram body: 32
```

Summary

What We Covered

- **Expressions**
  - Conditional expressions `if` and `match`
  - List Comprehension
    - Collection of generated by node traversal
- **Functions**
  - Boolean rules
  - Unit rules
  - Support functions
    - `@memoized` can provide better performance
- **Patterns**
  - `select` expression to find certain node types
    - Selector predicate to refine the selection
  - Matching tuples, lists, and objects
  - Using bindings to give a name to a selection result
- **Unit Rules**
  - Rules that apply to units rather than syntax tree nodes
  - Can report multiple violations
    - With messages and source locations

Advanced LKQL

Introduction

Topics Covered

- **Expressions In-Depth**
  - Less common but useful expression types
- **Selectors In-Depth**
  - Defining a selector
- **LKQL API**
  - Core capabilities

Expressions In-Depth

Fields and Properties

- User-created objects have fields
  - Access fields using "dot-notation"

```
val status = {msg: "Hello", flag: true}
print (status.msg)
```

- LKQL API exposes fields and properties of syntax tree nodes
  - Still referenced via "dot-notation"
  - Fields are prefaced with `f_`
  - Properties are prefaced with `p_`
    - Properties are **function calls** which may or may not take parameters

```
node is ExprFunction
when node.p_semantic_parent() is BasePackageDecl or
     (node.p_semantic_parent() is PrivatePart)

node is AttributeRef
when node.f_attribute?.p_name_is("Size")
```

Call Expressions

- Regular call expression

```
fun add(a, b) = a + b

val c = add(12, 15)
val d = add(a=12, b=15)
```

- "Safe" variant

```
fun add(a, b) = a + b
val fn = if true then null else add
fn?(1, 2) # Returns ()
```

  - Returns `()` (Unit) if `fn` is `null`
  - Notice that `fn` is just a nullable value

Traversing the Syntax Tree

- Default behavior for traversal is down through children

```
from node select SubpBody
# Return all subprogram bodies enclosed in "node"
```

  - Equivalent to

```
from node through children select SubpBody
```

- Can use `through` to indicate other traversals
  - Most common is `parent` (traverse upward)

```
from node through parent select SubpBody
# Return all subprogram bodies that enclose "node"
```

  - Also can use
    - `prev_siblings` - sibling nodes before current node
    - `next_siblings` - sibling nodes after current node
    - `super_types` - if current node is a type, then all parent types

Note

`follow_generics` is a special case indicating traversal should follow instantiated generic trees

Importing Modules

- Every rules file is considered a *module*
- Import modules to use functions and values defined in other rules
- Modules that are part of GNATCHECK
  - **control_flow** - perform control flow analysis
  - **metrics** - code complexity metrics
  - **parameters_aliasing** - determine parameter aliasing
  - **stdlib** - commonly used processing
- To import a module, include `import <module>` at the top of the file
- Function calls use dot-notation

```
import stdlib

@check
fun exception_propagation_from_tasks(node) =
    node is TaskBody when stdlib.propagate_exceptions(node)
```

Selectors In-Depth

Review: What Is a Selector?

- Special form of functions
  - Recursive `match` statement
    - Match expression treats current node as `this`
  - Return a `Stream` of values
  - Used in the query subset of LKQL
    - Allow easy expression of traversal blueprints
- Example: built-in selector `children`
  - Explores the tree node by node
- Selector declarations do not have parameters
  - But selector calls can specify, `min_depth`, `max_depth`, and `depth`

```
# Calling a selectors directly
val descendants = children(node, depth=3)
# descendants has all children at depth of 3

# Calling a selector in a nested sub-pattern
select AdaNode(any children(min_depth=3): BasicDecl)
# find all nodes with a "BasicDecl" descendant at depth 3 or deeper
```

Defining a Selector

- Each selector arm returns a `RecExpr` or `()` to stop recursion
  - Builds two lists
    - *Recursion list* is a list of items to be traversed next
    - *Return list* is a list of items to be returned to the user
    - Can add an item or a list of items (prefixed by `*`) to a list
    - Either list can be empty (`()`)
  - Selector returns a `Stream`
- `rec` operation creates the recursive expression
  - First parameter represents what is added to recursion list
  - Second parameter represents what is added to return list
    - If not specified, first parameter is added to both lists
- Return all defining names of a `BasicDecl`

```
selector defining_names
    | BasicDecl => rec(*[d for d in this.p_defining_names() if d != null])
    | *         => ()
```

  - If the node is a basic declaration
    - Build a list of all defining names for `list` that are not null
    - Add each item in that list to the recursion and return lists
  - Otherwise, stop recursing (return Unit)

LKQL API

Libadalang and LKQL

- Libadalang API can be called from LKQL
  - Basis for most GNATcheck rules
- Broken down by category
  - **Node types** - syntactic and semantic constructs
  - **Symbol types** - named values representing categories or kinds
  - **List types** - strongly-typed sequences of nodes
  - **Object types** - structured data (such as query results)
- *Origin* parameter
  - Some Libadalang properties accept optional `origin` parameter
  - Specifies the node from which visibility is evaluated
    - Default is the current node
  - For example, `p_most_visible_part` returns different data for a private type depending on the location of the node

Standard Library

- Built-in functions used throughout LKQL functions
- Built-in methods for tree nodes
- `stdlib` module defines supplied support functions
  - Not built-in so they need to be imported

Accumulation Across a Collection

- Sometimes we need a way to gather information about a collection
  - Count number of items
  - Sum of integer items
- `reduce` function provides that capability

```
reduce (collection, accumulating_function, initial_value)
```

- **Collection**
  - List or stream of items to traverse
- **Accumulating_Function**
  - Takes two parameters
    - Accumulator
    - Item from the collection
  - Can be either a function OR a parameter list and expression
- **Initial_Value** - initial value of accumulator

"reduce" Examples

- Counting number of children

```
fun add(x, y) = x + y
reduce(
    [1 for child in node.children],
    add,
    0)
```

  - Initialize accumulator to 0
  - Use list comprehension to add **1** to the list for every child
  - Call `add` to add the item to the accumulator

- Calculating a total

```
val total = reduce([1, 2, 3], (acc, x) => acc + x, 0)
```

  - Initialize accumulator `acc` to 0
  - For each item in the list, add its value to `acc`
    - `total` = 6

Lab

Advanced Rule Building

In this lab, we want a rule to check that nested subprograms are "relatively" small as part of our coding standard.

Again, we are using a "unit rule" because we only need to examine subprograms - no need to run the rule for every node.

Tip

The lab is a step-by-step guide. Many of the steps have hints in addition to solutions. If the page has empty space, pressing Page Down will usually get you that extra help!

Example Project

- Copy the `advanced_lkql` folder from the course materials location
- Contents of the folder:
  - `default.gpr` - project file
  - `src` - source directory
    - Contains a trivial collection of subprograms and nested subprograms

Step 1 - Create Rule File

- Create rule `long_nested_subprograms`
  - In file `long_nested_subprograms.lkql`
  - Use `@unit_check` annotation
- Our first step is to flag every subprogram in the unit
- Rule should use
  - List comprehension
  - Iteration over results of a query
  - Name of subprogram in violation message

Step 1 - Create Rule File

- Create rule `long_nested_subprograms`
  - In file `long_nested_subprograms.lkql`
  - Use `@unit_check` annotation
- Our first step is to flag every subprogram in the unit
- Rule should use
  - List comprehension
  - Iteration over results of a query
  - Name of subprogram in violation message

```
@unit_check
fun long_nested_subprograms(unit) =
    [
        {message: "subprogram name: " &
                  img(subp.p_defining_name().p_fully_qualified_name()),
         loc: subp}
        for subp in
            (from unit.root
                select BasicDecl(p_is_subprogram(): true))
    ]
```

Run rule to see behavior

Step 1 - Create Rule File

- Create rule `long_nested_subprograms`
  - In file `long_nested_subprograms.lkql`
  - Use `@unit_check` annotation
- Our first step is to flag every subprogram in the unit
- Rule should use
  - List comprehension
  - Iteration over results of a query
  - Name of subprogram in violation message

```
@unit_check
fun long_nested_subprograms(unit) =
    [
        {message: "subprogram name: " &
                  img(subp.p_defining_name().p_fully_qualified_name()),
         loc: subp}
        for subp in
            (from unit.root
                select BasicDecl(p_is_subprogram(): true))
    ]
```

Run rule to see behavior

```
gnatcheck -P default.gpr --rules-dir=.  --rule long_nested_subprograms
```

```
advanced.adb:3:04: rule violation: subprogram name: "Advanced.Visible1"
advanced.adb:4:07: rule violation: subprogram name: "Advanced.Visible1.Nested1"
advanced.adb:18:07: rule violation: subprogram name: "Advanced.Visible1.Nested2"
advanced.adb:26:04: rule violation: subprogram name: "Advanced.Visible2"
advanced.adb:27:07: rule violation: subprogram name: "Advanced.Visible2.Nested1"
...
```

Should get 34 total 'violations'

Step 2 - Focus on Nested Subprograms

- There is already a rule to count lines in a subprogram (`maximum_subprogram_lines`)
- We want to focus on nested subprograms
  - We need to see if any of the subprogram parents is a subprogram
  - Create a helper function to search the subprogram parents for `SubpBody`

Step 2 - Focus on Nested Subprograms

- There is already a rule to count lines in a subprogram (`maximum_subprogram_lines`)
- We want to focus on nested subprograms
  - We need to see if any of the subprogram parents is a subprogram
  - Create a helper function to search the subprogram parents for `SubpBody`

```
fun enclosing_subp(n) =
   from n through parent select first SubpBody
```

- Now add this function to our list comprehension as a filter

Step 2 - Focus on Nested Subprograms

- There is already a rule to count lines in a subprogram (`maximum_subprogram_lines`)
- We want to focus on nested subprograms
  - We need to see if any of the subprogram parents is a subprogram
  - Create a helper function to search the subprogram parents for `SubpBody`

```
fun enclosing_subp(n) =
   from n through parent select first SubpBody
```

- Now add this function to our list comprehension as a filter

```
@unit_check
fun long_nested_subprograms(unit) =
    [
        {message: "subprogram name: " &
                  img(subp.p_defining_name().p_fully_qualified_name()),
         loc: subp}
        for subp in
            (from unit.root select BasicDecl(p_is_subprogram(): true))
            if enclosing_subp(subp) != null
    ]
```

Run rule to see behavior

Step 2 - Focus on Nested Subprograms

- There is already a rule to count lines in a subprogram (`maximum_subprogram_lines`)
- We want to focus on nested subprograms
  - We need to see if any of the subprogram parents is a subprogram
  - Create a helper function to search the subprogram parents for `SubpBody`

```
fun enclosing_subp(n) =
   from n through parent select first SubpBody
```

- Now add this function to our list comprehension as a filter

```
@unit_check
fun long_nested_subprograms(unit) =
    [
        {message: "subprogram name: " &
                  img(subp.p_defining_name().p_fully_qualified_name()),
         loc: subp}
        for subp in
            (from unit.root select BasicDecl(p_is_subprogram(): true))
            if enclosing_subp(subp) != null
    ]
```

Run rule to see behavior

```
advanced.adb:4:07: rule violation: subprogram name: "Advanced.Visible1.Nested1"
advanced.adb:18:07: rule violation: subprogram name: "Advanced.Visible1.Nested2"
advanced.adb:27:07: rule violation: subprogram name: "Advanced.Visible2.Nested1"
advanced.adb:28:10: rule violation: subprogram name: "Advanced.Visible2.Nested1.Nesteda"
advanced.adb:32:10: rule violation: subprogram name: "Advanced.Visible2.Nested1.Nestedb"
...
```

- Down to 24 'violations'
- None of the top-level subprograms are listed

Step 3 - Count the Lines in a Subprogram

- We want to count the lines of code in our nested subprogram
- Helper functions in `metrics.lkql` do a lot of those calculations
  - File resides in GNATCHECK installation directory
  - We want `logical_slocs`
- Update our query to only include nested subprogram with more than 5 lines of code
  - Don't forget to add the external reference to `metrics`

Step 3 - Count the Lines in a Subprogram

- We want to count the lines of code in our nested subprogram
- Helper functions in `metrics.lkql` do a lot of those calculations
  - File resides in GNATCHECK installation directory
  - We want `logical_slocs`
- Update our query to only include nested subprogram with more than 5 lines of code
  - Don't forget to add the external reference to `metrics`

```
if enclosing_subp(subp) != null and metrics.logical_slocs(subp) > 5
```

- This is the only logic line that needs to be updated
  - But we need to add `import metrics` to the top of the file

Run rule to see behavior

Step 3 - Count the Lines in a Subprogram

- We want to count the lines of code in our nested subprogram
- Helper functions in `metrics.lkql` do a lot of those calculations
  - File resides in GNATCHECK installation directory
  - We want `logical_slocs`
- Update our query to only include nested subprogram with more than 5 lines of code
  - Don't forget to add the external reference to `metrics`

```
if enclosing_subp(subp) != null and metrics.logical_slocs(subp) > 5
```

- This is the only logic line that needs to be updated
  - But we need to add `import metrics` to the top of the file

Run rule to see behavior

```
advanced.adb:4:07: rule violation: subprogram name: "Advanced.Visible1.Nested1"
advanced.adb:27:07: rule violation: subprogram name: "Advanced.Visible2.Nested1"
advanced.adb:32:10: rule violation: subprogram name: "Advanced.Visible2.Nested1.Nestedb"
advanced.adb:45:07: rule violation: subprogram name: "Advanced.Visible2.Nested2"
...
```

- Down to 11 'violations'
- Note that higher level subprograms are more likely to get flagged
  - Because their lines of code include those in the nested subprograms

Step 4 - Why Only in Subprograms?

- Our rule only checks subprograms nested in other subprograms
  - But what about subprograms in protected objects? or tasks?
- Delivered helper file `stdlib.lkql` has a lot of these queries
  - `has_local_scope` expands our subprogram parent into something more broad
- Update the rule to use the `stdlib` query and run the rule

Step 4 - Why Only in Subprograms?

- Our rule only checks subprograms nested in other subprograms
  - But what about subprograms in protected objects? or tasks?
- Delivered helper file `stdlib.lkql` has a lot of these queries
  - `has_local_scope` expands our subprogram parent into something more broad
- Update the rule to use the `stdlib` query and run the rule

```
if stdlib.has_local_scope(subp) and metrics.logical_slocs(subp) > 5
```

- Don't forget to add `import stdlib` to the top of the file
- (You could also remove `enclosing_subp`, but it's not necessary)

```
...
advanced.adb:97:10: rule violation: subprogram name: "Advanced.Visible3.Nested2.Nesteda"
advanced.adb:109:10: rule violation: subprogram name: "Advanced.Visible3.Nested2.Nestedb"
advanced.adb:134:07: rule violation: subprogram name: "Advanced.Protected_Object.Set"
```

- Notice the subprogram inside `Protected_Object` is now included

Extra Credit - Different Values for Different Depths

- What if we wanted to enforce smaller limits for deeply nested subprograms?
- Now we need a function to count the depth of the subprogram
  - If a parent is a subprogram, add one to the count
- We can use the `reduce` function to count the appropriate parents
  - Use list comprehension to build a list of matching nodes
    - Append **1** to the list if the node matches
  - Use a function to accumulate the list elements
  - Set the initial value of the `reduce` accumulator to 0

Implement Different Values for Different Depths

- Create a function to add two numbers

Implement Different Values for Different Depths

- Create a function to add two numbers

```
fun add(x, y) = x + y
```

- Create a function that uses `reduce` to count appropriate parents

Implement Different Values for Different Depths

- Create a function to add two numbers

```
fun add(x, y) = x + y
```

- Create a function that uses `reduce` to count appropriate parents

```
@memoized
fun count_depth(n) =
    reduce(
        [1 for p in n.parents(true) if p is SubpBody],
        add,
        0
    )
```

We use `@memoized` because we may use this function multiple times on the same node

- Update rule to use different line counts for different depths
  - Try 10 for depth 2, 8 for depth 3, and 6 for depth 4

Implement Different Values for Different Depths

- Create a function to add two numbers

```
fun add(x, y) = x + y
```

- Create a function that uses `reduce` to count appropriate parents

```
@memoized
fun count_depth(n) =
    reduce(
        [1 for p in n.parents(true) if p is SubpBody],
        add,
        0
    )
```

We use `@memoized` because we may use this function multiple times on the same node

- Update rule to use different line counts for different depths
  - Try 10 for depth 2, 8 for depth 3, and 6 for depth 4

```
if stdlib.has_local_scope(subp) and
    ( (count_depth(subp) == 4 and metrics.logical_slocs(subp) > 6) or
      (count_depth(subp) == 3 and metrics.logical_slocs(subp) > 8) or
      (count_depth(subp) == 2 and metrics.logical_slocs(subp) > 10))
```

Summary

What We Covered

- **Expressions In-Depth**
  - Fields and properties
  - Safe function calls
  - Importing from other modules
- **Selectors In-Depth**
  - Building data from recursive traversal of nodes
- **LKQL API**
  - Libadalang as part of LKQL
  - Standard library functions and rules