# Langkit

source code analyzers for the masses

Pierre-Marie de Rodat Raphaël Amiard

Software Engineers at AdaCore



Langkit: A meta compiler

### High level vision

A DSL to implement language analysis front ends (parsing & more).

Front ends generated by Langkit could be the basis for:

- compilers
- debuggers (e.g. expression evaluator)
- interactive code browsers
- static analyzers
- automatic code refactoring tools



## Original use case: Libadalang

- The Ada ecosystem lacks a good library to create code-aware tools
- Several partial analyzers in various tools (e.g. GPS, the main IDE)
- Libadalang = Langkit-generated library for Ada-aware tools



# The DSL

AdaCore

### **Syntax**

- Python-based DSL for now
- Uses the Python syntax to create data structures that are then compiled
- Will have its own syntax one day!



## **DSL Episode 1: Lexing**

```
from langkit.lexer import LexerToken, WithText, WithSymbol

class MyTokens(LexerToken):
    Def = WithText()
    Identifier = WithSymbol()
# ...

my_lexer = Lexer(MyTokens)
my_lexer.add_rules(
    (Literal('def'), MyTokens.Def),
    (Pattern(r'[a-zA-Z][a-zA-Z0-9]*',
    MyTokens.Identifier)),
    # ...
)
```

- Define a list of token kinds
- Provide regexp-based scanning rules to produce them



### **DSL Episode 2: Tree**

```
from langkit.dsl import ASTNode, Field, abstract

@abstract
class RootNode(ASTNode):
    pass

class Name(RootNode):
    token_node = True

class Def(RootNode):
    name = Field()

# ...
```

- Define tree nodes the parser can produce
- Uses a Python class hierarchy to describe the generated node hierarchy
- Nodes can be abstract



## **DSL Episode 3: Parsing**

```
from langkit.parsers import Grammar, List

g = Grammar('main_rule')
g.add_rules(
    main_rule=List(g.def_rule),
    name=Name(MyTokens.Identifier),
    def_rule=Def('def', g.name),
    # ...
)
```

- Recursive descent parser combinators (sequences, lists, optional parts, alternatives, ...), based on Packrat
- Compiling the grammar:
  - Infers node types Field if type not specified
  - Checks consistency otherwise



## **DSL Episode 4: Scoping**

```
class Function(RootNode):
    local vars = Field()
    env spec = EnvSpec(
        add_env() # Associate an env to this node
class VariableDeclaration(RootNode):
    name = Field()
    env spec = EnvSpec(
        add_to_env(
            mappings=New(T.env assoc, key=Self.name.symbol, val=Self)
            # In the current env, add a mapping from the name of this var to
            # the node itself.
```

- Foundation for semantic analysis
- Create name/nodes mappings: lexical environments



## **DSL Episode 5: Semantic analysis**

```
from langkit.expressions import langkit_property

class VariableReference(FooNode):
    name = Field()

    @langkit_property(public=True)
    def var_decl():
        return Self.node_env.get_first(Self.name)
```

- Create methods on nodes (called "properties")
- Public properties: user API for semantic analysis
- Private ones: implementation detail, hidden from users
- Functional programming language



### **DSL Episode 6: Logic DSL**

- Hard problems in semantic analysis:
  - Overload resolution
  - Type inference
- Require non local knowledge

```
def f1 (i : int) -> int;
def f1 (f : float) -> int;

def f2 (c : char) -> int;
def f2 (c : char) -> char;

f1 (1);
f1 (0.3);
f2 (2);
f1 (f2 (f2 ('C')));
```



## **DSL Episode 6: Logic DSL**

```
f1 (f2 (f2 ('C')));
A.args[0].type = char
A.returns.type = B.args[0].type
B.returns.type = C.args[0].type
A is in [f2 (c : char) \rightarrow int,
          f2 (c : char) \rightarrow char
B is in [f2 (c : char) -> int,
         f2 (c : char) -> charl
C is in [f1 (i : int) -> int.
          f1 (f : float) -> float]
```

#### Solution:

```
A = f1 (i : int) -> int
B = f2 (c : char) -> int
C = f2 (c : char) -> char
```

The generated libraries

# Base library: Ada (W00T!)

### Requirements for the target language:

- Fast
- Low level enough
- Memory management agnostic (no GC)
- Easy to bind to C and other languages

#### Candidates

■ C, C++, Ada, Rust, ...

#### Chosen one: Ada

Since the project is developed at AdaCore: no surprises:)



## Bindings to other languages

### Automatically generated C bindings

So that it is very easy to generate bindings to any languages the users wants

#### First class citizen Python bindings

- Python is the de facto scripting language of the Langkit ecosystem
- Everything possible in Ada is possible in Python

### Easy to generate bindings to new languages

- No need for external bindings generators
- $\blacksquare \ \ \, \text{Knowledge about data types, functions, memory management -> Langkit}$



## Crafted for incremental analysis

- Reloading happens a lot in IDE: performance required
- Avoid big recomputations for common operations
- No need to recompute *everything* when reloading one source file:
  - Keep source file-specific data as much isolated as possible
  - Reduced update process when removing/reloading source files



### Syntactic analysis

- Small Ada syntactic analyzer
- Can be adapted to another language trivially

Detected 4 serious bugs in AdaCore's own code



### **Unparser**

- Create a new source file only from the tree (not using original source information)
- Can also be used to create sources from completely synthetic trees
- Uses the grammar and the AST definition (no additional code needed)



## Rewriting (work in progress)

#### Source code:

```
procedure Main is
begin
   Put_Line ("Hello world");
end Main;
```

#### Let's rewrite:

```
call = unit.root.findall(lal.CallExpr) # Find the call
diff = ctx.start_rewriting() # Start a rewriting
param_diff = diff.get_node(call.f_suffix[0]) # Get the param of the call
# Replace the expression of the parameter with a new node
param_diff.f_expr = lal.rewriting.StringLiteral('"Bye world"')
diff.apply()
```

#### Result:

```
procedure Main is
begin
   Put_Line ("Bye world");
end Main;
```



Generic tools shipping with the libraries

#### **Small tools**

### ./playground

- Command line tool based on IPython
- Allow interactive exploration of the tree/API in general

## ./parse

- Allow inspection of the AST
- Dump lexical environments



# **Code indenter (prototype)**

Provide a declarative data structure for indentation rules

```
block_rule = field_rules(constant_increment=3)

indent_map = {
    lal.PackageDecl: Indent(
        field_rules=indent_fields(
            public_part=block_rule, private_part=block_rule
        )
    ),
    # ...
}
```

• Get auto indentation on tab in your favorite editor

## Syntax highlighter (not done)

- Auto generation of syntax highlighter
- Highlight keywords by default
- Custom rules to highlight more complex syntax based rules
- Automatic support in your editor



# Language Server Protocol? (not done)

- Tentative plan: automatically generate basic LSP support from the plug-in
- We have a Neovim plug-in already doing for Ada:
  - Indentation
  - Go to definition
  - Tree editing and exploration
- In the future: more editors, more languages?



### **Existing Langkit-based libraries & prototypes**

- Ada https://github.com/AdaCore/libadalang
- Python https: //github.com/AdaCore/langkit/tree/master/contrib/python
- JSON
- GPR files (AdaCore's project description language)
   https://github.com/AdaCore/gpr
- KConfig (Linux kernel configuration description language)
   https://github.com/Fabien-Chouteau/libkconfiglang



#### Conclusion

- Sources are on GitHub: https://github.com/AdaCore/langkit
- Tutorial, too: https: //github.com/AdaCore/langkit/blob/master/doc/tutorial.rst
- Still work in progress: APIs are moving and "doc is the code" (no separate documentation document)
- We gladly accept issues and pull requests, but our priority right now is Libadalang

