GSL Documentation

Release 0.0.1

Clemens Koza

Feb 18, 2019

Contents

GSL is inspired by, but completely independent of, iMatix' GSL (Generator scripting language). It is a Python-based tool for model-oriented programming, which you can read about here, again borrowing from iMatix.

In contrast to iMatix' GSL, this tool does not use its own scripting language but Python; with Python 3.6's f-strings, it is very appropriate for code generation tasks. GSL's focus lies on reading the source models and making them available in a convenient form, and providing utilities that are useful during code generation, especially for output handling and string manipulation.

CHAPTER 1

Python code generation examples

Actual code generation in Python is pretty straightforward and can be done with or without GSL. We will describe here some guidelines that can help keep code generation code more readable.

Generally speaking, a code generator will contain code in two languages: the generator language (here, Python 3.6+), and the target language (in our examples Java). This makes the code inherently more difficult to read, calling for good tools and conventions. In Python, target language code will appear in string literals. Consider this simple code generator:

```
from collections import namedtuple

Class = namedtuple('Class', ('name', 'members',))
Field = namedtuple('Field', ('name',))
Method = namedtuple('Method', ('name',))

model = Class("HelloWorld", [Field("foo"), Method("bar")])

def class_declaration(model):
    print(f"public class {model.name} {{")
    for member in model.members:
        if isinstance(member, Field):
            print(f"")
            print(f"    private int {member.name};")
        elif isinstance(member, Method):
            print(f"")
            print(f"    public void {member.name}() {{")
            print(f"        // TODO")
            print(f"    }}")
    print(f"}}")

class_declaration(model)
```

Which outputs:

```
public class HelloWorld {

    private int foo;

    public void bar() {
        // TODO
    }
}
```

The generator code is nicely readable, but target code readability suffers from the different indentation levels. Separate `print` statements don't hurt, but depend on one's taste. We advocate for the following style:

```
from collections import namedtuple

Class = namedtuple('Class', ('name', 'members',))
Field = namedtuple('Field', ('name',))
Method = namedtuple('Method', ('name',))

model = Class("HelloWorld", [Field("foo"), Method("bar")])

def class_declaration(model):
    print(f"""\
public class {model.name} {{""")
    for member in model.members:
        if isinstance(member, Field):
            print(f"""\

    private int {member.name};""")
        elif isinstance(member, Method):
            print(f"""\

    public void {member.name}() {{
        // TODO
    }}""")
    print(f"""\
}}""")

class_declaration(model)
```

This code has a different readability tradeoff. Generator code indentation is interrupted by target code, but in return each target code line's indentation is directly apparent. The \ at the start of each literal ignores the following line break, allowing the first line of the literal to start at the same level as the rest.

1.1 Adding GSL

Directly printing strings takes us some flexibility: no postprocessing of generated code, no choice for the output location (at least in this simple form). But it is, of course, very simple. The `yield` keyword gives us back that flexibility, with almost no readability tradeoff.

We will also replace `namedtuple` with GSL's `pseudo_tuple`. Pseudo tuples are not really Python tuples; they serve the same purpose as named tuples, but are modifiable and can hold additional data. This can come in handy if we want to enrich the in-memory data structure of our model with inferred information.

```
from gsl import pseudo_tuple, lines, printlines

```

(continues on next page)

(continued from previous page)

```
Class = pseudo_tuple('Class', ('name', 'members',))
Field = pseudo_tuple('Field', ('name',))
Method = pseudo_tuple('Method', ('name',))

model = Class("HelloWorld", [Field("foo"), Method("bar")])

def class_declaration(model):
    yield from lines(f"""\
public class {model.name} {{""")
    for member in model.members:
        if isinstance(member, Field):
            yield from lines(f"""\

    private int {member.name};""")
        elif isinstance(member, Method):
            yield from lines(f"""\

    public void {member.name}() {{
        // TODO
    }}""")
        yield from lines(f"""\
}}""")

with open("HelloWorld.java", "w") as f:
    printlines(class_declaration(model), file=f)
```

If you're not familiar with `yield`, here's the (simplified) basics: it turns the function into a "generator", meaning it returns multiple values in a stream. The function is run on-demand to the next `yield` statement whenever a value is needed, which is generally good. It might lead to confusion if the generator function has side effects, so try to avoid them.

`yield from` is a variant that, instead of returning one value, returns all values from an iterable (such as another generator). In other words: `yield from lines(...)` means that the next few values will come from the `lines` generator. Specifically, it splits the string into individual lines. Why not just yield the combination of lines, just as we printed multiline strings in the previous version? Well, we're doing all this to gain flexibility, and one thing we could use that for is prefixing each line with additional indentation or comment them out. That is easy if we yield individual lines, not so much if we return multiline strings.

If you're not sold on all that, take a look at this function and think about how to do the same with `print` or without splitting lines:

```
def commented(code, block=False):
    if block:
        yield "/*"
    for line in code:
        yield (" * " if block else "// ") + line
    if block:
        yield " */"
```

CHAPTER 2

YAML

GSL provides a thin wrapper around the ruamel.yaml YAML library:

```
from gsl import pseudo_tuple, lines, printlines
from gsl.yaml import YAML

Class = pseudo_tuple('Class', ('name', 'members',))
Field = pseudo_tuple('Field', ('name',))
Method = pseudo_tuple('Method', ('name',))

yaml = YAML(typ='safe')
yaml.register_class(Class)
yaml.register_class(Field)
yaml.register_class(Method)
model = yaml.load("""\
- !Class
  name: HelloWorld
  members:
  - !Field
    name: foo
  - !Method
    name: bar
""")

# def class_declaration

for class_model in model:
    with open(f"{class_model.name}.java", "w") as f:
        printlines(class_declaration(class_model), file=f)
```

Tags like `!Class` allow us to get the exact same model as before. One caveat is that the classes to be generated need to be modifiable; `namedtuple` wouldn't have worked here.

CHAPTER 3

ANTLR

Often, a model based on YAML (or any other markup language) is enough to describe a model concisely, but there are cases where a DSL (domain specific language) boosts expresiveness immensely. In these cases, ANTLR can be used to parse the DSL, and GSL to process the parse tree.

Let's first write a grammar that lets us express this simple model:

```
class HelloWorld {
    field foo;
    method bar;
}
```

We will not go into detail about writing grammars here and simply give it:

```
grammar SimpleClass;

model: classDef* EOF;

classDef: 'class' name=IDENTIFIER '{' (fieldDef | methodDef)* '}';
fieldDef: 'field' name=IDENTIFIER ';';
methodDef: 'method' name=IDENTIFIER ';';

IDENTIFIER: [_a-zA-Z][a-zA-Z0-9]*;

WS: [ \t]+ -> channel(HIDDEN);
```

Then, generate lexer and parser from this grammar:

```
antlr4 -Dlanguage=Python3 -visitor -no-listener SimpleClass.g4
```

And finally, write a simple program that parses our model:

```
from gsl.antlr import Antlr

from SimpleClassLexer import SimpleClassLexer
```

(continues on next page)

(continued from previous page)

```
from SimpleClassParser import SimpleClassParser

antlr = Antlr(SimpleClassLexer, SimpleClassParser)
p = antlr.parser(antlr.input_stream("""\
class HelloWorld {
    field foo;
    method bar;
}
"""))
model = p.model()

print(antlr.to_string(model))
```

The `Antlr` class provides us with convenience methods for using our grammar. As a first step, we simply print the parse tree to better understand what's happening here:

```
(model (classDef class HelloWorld { (fieldDef field foo ;) (methodDef method bar ;) }
↪) <EOF>)
```

The parse tree contains all tokens consumed, including `class`, `method`, `<EOF>` etc., that were important for parsing but don't add anything to the model. We can already generate code from this model:

```
from gsl import lines, printlines

# ...

def class_declaration(model):
    yield from lines(f"""\
public class {model.IDENTIFIER()} {{""")
    for member in model.getChildren():
        if isinstance(member, SimpleClassParser.FieldDefContext):
            yield from lines(f"""\

    private int {member.IDENTIFIER()};""")
        elif isinstance(member, SimpleClassParser.MethodDefContext):
            yield from lines(f"""\

    public void {member.IDENTIFIER()}() {{
        // TODO
    }}""")
    yield from lines(f"""\
}}""")

for class_model in model.classDef():
    with open(f"{class_model.IDENTIFIER()}.java", "w") as f:
        printlines(class_declaration(class_model), file=f)
```

This code has a slight downside: `getChildren()` returns all children, not only the fields and methods we're interested in. We filter out the other children inside the loop, yes, but being able to get specific kinds of children would be good in its own right.

3.1 Parse tree transformation with visitors

We could use `fieldDef()` and `methodDef()` to get fields and methods separately, but then we lose their relative order. Up until now, we also managed to have our model represented the same way in memory; the code here is specific to what our grammar looks like.

What we really want is a model that reflects our needs: ignore semicolons and list fields and methods together. ANTLR provides visitors for doing parse tree transformations, and GSL adds its own APIs and DSL (g4v) for making it as seamless as possible.

Let's create a visitor to process our parse tree. . .

```
from gsl import pseudo_tuple, file, output
from gsl.antlr import Antlr, ParseTreeVisitor

# ...

Class = pseudo_tuple('Class', ('name', 'members',))
Field = pseudo_tuple('Field', ('name',))
Method = pseudo_tuple('Method', ('name',))

class SimpleClassVisitor(ParseTreeVisitor):
    def visitModel(self, ctx):
        return self.visitNodes(self.get_children(ctx, SimpleClassParser.
↪ClassDefContext))

    def visitClassDef(self, ctx):
        return Class(
            self.visitNode(ctx.IDENTIFIER()),
            self.visitNodes(self.get_children(ctx, SimpleClassParser.FieldDefContext,␣
↪SimpleClassParser.MethodDefContext)),
        )

    def visitFieldDef(self, ctx):
        return Field(self.visitNode(ctx.IDENTIFIER()))

    def visitMethodDef(self, ctx):
        return Method(self.visitNode(ctx.IDENTIFIER()))

model = p.model().accept(SimpleClassVisitor())

print(model)
```

. . . and take a look at the result:

```
[Class(name='HelloWorld', members=[Field(name='foo'), Method(name='bar')])]
```

Exactly what we had before! Before we integrate this into the rest of our code, let's automate the creation of this visitor:

3.2 ANTLR 4 Visitors DSL (g4v)

A visitor for creating a concise data structure like above is fairly straight-forward, and in many cases can be created automatically. Even if some parts do not follow strict patterns, subclassing allows one to use a generated visitor as the baseline.

Using g4v, the visitor from the previous section can be defined as follows:

```
visitor SimpleClassVisitor for grammar SimpleClass;

```

(continues on next page)

(continued from previous page)

```
model = classDef*;
classDef = Class(name=IDENTIFIER, members=(fieldDef | methodDef)*);
fieldDef = Field(name=IDENTIFIER);
methodDef = Method(name=IDENTIFIER);
```

Then, create the visitor from this (overwriting the file created by ANTLR:

```
g4v SimpleClassVisitor.g4v
```

And finally, the full code generator:

```
from gsl import lines, printlines
from gsl.antlr import Antlr

from SimpleClassLexer import SimpleClassLexer
from SimpleClassParser import SimpleClassParser
from SimpleClassVisitor import SimpleClassVisitor, Class, Field, Method

antlr = Antlr(SimpleClassLexer, SimpleClassParser)
p = antlr.parser(antlr.input_stream("""\
class HelloWorld {
    field foo;
    method bar;
}
"""))
model = p.model().accept(SimpleClassVisitor())

def class_declaration(model):
    yield from lines(f"""\
public class {model.name} {{""")
    for member in model.members:
        if isinstance(member, Field):
            yield from lines(f"""\

    private int {member.name};""")
        elif isinstance(member, Method):
            yield from lines(f"""\

    public void {member.name}() {{
        // TODO
    }}""")
    yield from lines(f"""\
}}""")

for class_model in model:
    with open(f"{class_model.name}.java", "w") as f:
        printlines(class_declaration(class_model), file=f)
```

CHAPTER 4

Structuring complex code generators

Up until now, we have looked at how to write a code generator for one kind of code. In reality, the same model may be used to generate many different sources, such as accompanying SQL scripts, or simply the same code in different target languages.

Apart from separating the code into different Python modules, it also helps to differentiate between diffferent kinds of code generator functions and employ a naming convention. Here is a suggestion:

- `something_str(...)`: these functions return a code snippet as a single string
- `something_code(...)`: these functions yield multiple lines of code (not necessarily a whole file)
- `generate_something_code(...)`: these functions actually write generated code to one or more output files

A module would for example have a `generate_code()` function that generates all relevant files:

```
def generate_code(model):
    for foo in model.foos:
        generate_foo_code(foo)
    for bar in model.bars:
        generate_bar_code(bar)
```

Each individual code generator would print the code to a file, but to use `yield`, we need a separate code function that we would call immediately. It makes sense to use a nested function for this, and just call it `code` as it yields "the" code, not just some, for foo:

```
def generate_foo_code(foo):
    def code():
        yield "foo:"
        for snippet in foo.snippets:
            yield from foo_snippet_code(snippet)

    with open(foo_filename(foo)) as f:
        printlines(code(foo), file=f)
```

Hiding actually opening and writing the file below the possibly long `code()` function is ugly, so we provide the additional `print_to` decorator that runs the function:

```
def generate_foo_code(foo):
    @print_to(foo_filename(foo)):
    def code():
        yield "foo:"
        for snippet in foo.snippets:
            yield from foo_snippet_code(snippet)
```

A thing to watch out for here is that `print_to` does not transform the method, it executes it once with context! Think of `print_to` as a pseudo content manager, and the `code` function as a `with` block that allows `yield`.

4.1 Allowing persistent code customization

Often, it is only feasible to generate a scaffolding for the desired code, leaving some parts to be implemented by hand. GSL supports that using the `generate` decorator. While using `print_to` can be read as a one-shot command - "print to this file" -, `generate` is better viewed as a stateful description of a file's content - "this is how to get the file's contents".

`generate` looks for a few specific patterns in both the existing file (if any) and the generated code, syntactically similar to XML tags:

- `<GSL customizable: label>` and `<default GSL customizable: label>` open a customizable block;
- `</GSL customizable: label>` closes a customizable block;
- `<GSL customizable: label />` and `<default GSL customizable: label />` mark an empty customizable block.

Before going into details, let's look at some examples. Imagine we created the following code:

```
public class HelloWorld {

    private int foo;

    public HelloWorld() {
        // <default GSL customizable: constructor />
    }

    public void bar() {
        // <default GSL customizable: method-bar>
        // TODO
        // </GSL customizable: method-bar>
    }
}
```

The constructor has an empty block labelled `constructor`, and `bar()` has a block containing a `// TODO`. If we modified this, ordinarily, that change would be overwritten when regenerating code. To make a persistent change, we remove the `default` and add our code. For illustration purposes, let's say `bar()` should be a no-op; we can make the block empty for this:

```
public class HelloWorld {

    private int foo;

```

(continues on next page)

(continued from previous page)

```
    public HelloWorld() {
        // <GSL customizable: constructor>
        this.foo = 42;
        // </GSL customizable: constructor>
    }

    public void bar() {
        // <GSL customizable: method-bar />
    }
}
```

Both blocks are not `default`, so their content - or their emptiness - is preserved.

We have not only modified the blocks' contents, but also the block markers. What happens when we, for example, change our code generator to use `/* ... */` instead of `// ...`?

```
public class HelloWorld {

    private int foo;

    public HelloWorld() {
        /* <GSL customizable: constructor> */
        this.foo = 42;
        /* </GSL customizable: constructor> */
    }

    public void bar() {
        /* <GSL customizable: method-bar /> */
    }
}
```

Exactly what you'd probably expect. Everything outside the pattern is taken from the code generator, the pattern and block remain unmodified. Note that `constructor` didn't have a closing mark in the generated code; the opening marks' formatting is duplicated here. Likewise, the `method-bar` closing mark's formatting is ignored because an empty tag is used.

Now for the details:

- A mark's label may consist of letters, digits, underscores and hyphens.
- Mark patterns are matched case- and whitespace-sensitively.
- Generated code can only contain default blocks; only the target file can have non-default blocks.
- Blocks may not be nested, must be closed, and must have a unique label in the file. This is true for file contents, and for the generated code that should be written to the file. An error in the file contents will prevent the code generation; an error in the generated code will abort the code generation at that line. Note: This may mean customized code is lost! Always use code versioning for files that contain hand-written code.
- Blocks that are default or don't exist in the target file are not preserved; their code is generated as without this feature.
- A nondefault block's content or emptiness is preserved. However, if that block's label does not appear in the generated code, the block will not appear in the target file.
- The lines that contain a block's marks are not part of the block and are not preserved. Only the block's name, whether the block is default, and whether the block is empty is preserved about the marks. Anything outside the mark pattern is taken from the code generator, not from the target file.

CHAPTER 5

Learn More

CHAPTER 6

TODOs