CongoCC for newcomers

- CongoCC and Parsers
- A Closer Look at Parsers

CongoCC and Parsers

What is CongoCC

CongoCC a Java program and its main functionality is a "parser-generator". That is, CongoCC generates "parsers". A parser is a program that takes in as input a text file, parses the text file, and — in typical usage — generates a data structure that represents the structure of the text file. The parsers generated by CongoCC can be executed separately, without any dependencies on CongoCC.

What are parsers?

Think about a simple "hello world" Java program, one that looks like the following (from https://introcs.cs.princeton.edu/java/11hello/HelloWorld.java.html):

```
/*************************************************************************
 *  Compilation:  javac HelloWorld.java
 *  Execution:    java HelloWorld
 *
 *  Prints "Hello, World". By tradition, this is everyone's first program.
 *
 *  % java HelloWorld
 *  Hello, World
 *
 *  These 17 lines of text are comments.  They are not part of the program;
 *  they serve to remind us about its properties.  The first two lines tell
 *  us what to type to compile and test the program.  The next line describes
 *  the purpose of the program.  The next few lines give a sample execution
 *  of the program and the resulting output.  We will always include such
 *  lines in our programs and encourage you to do the same.
 *
 *************************************************************************/

public class HelloWorld {

    public static void main(String[] args) {
```

```
        // Prints "Hello, World" in the terminal window.
        System.out.println("Hello, World");
    }
}
```

It is easy enough to compile and run this:

```
$ javac HelloWorld.java
$ java HelloWorld
Hello, World
```

The Java compiler takes in the `HelloWorld.java` file and produces a compiled `HelloWorld.class` file, which we can run using the Java program. Somewhere early in the compilation process, the compiler has to be able to understand the structure of the Java source file (in this example, `HelloWorld.java`). It needs this structure to know what to compile. The above source file has the following structure:

1. There is a comment block at the top.
2. There is a class named `HelloWorld`.
3. The class named `HelloWorld` has a single method named `main`.
4. The method `main` in the class "`HelloWorld` is of type `public static void` and takes in one argument called `args`. The argument is of type `String[]`.
5. The method `main` in the class `HelloWorld` has a comment and it calls the method `System.out.println` with the argument `"Hello, World"`.

The part of the compiler that is responsible for parsing the input source file and producing a structured representation of the source file is called the "parser". This structured representation is called a "Syntax Tree", "Abstract Syntax Tree", or "Concrete Syntax Tree". For now, it doesn't matter too much what we call them. Generally speaking, parsers are language-specific. A single parser only understands the rules of a single language. For example, the parser in the Java compiler will only understand the Java language and will not understand Python, C++, or Rust.

What does CongoCC do?

CongoCC generates parsers given an input "grammar". A grammar (sometimes called "formal grammar") describes the rules of a language. For example, the Java programming language grammar has rules that look like these:

```
ClassOrInterfaceDeclaration:
    {Modifier} (ClassDeclaration | InterfaceDeclaration)
```

```
ClassDeclaration:
    NormalClassDeclaration
    EnumDeclaration

InterfaceDeclaration:
    NormalInterfaceDeclaration
    AnnotationTypeDeclaration

NormalClassDeclaration:
    class Identifier [TypeParameters]
                                [extends Type] [implements TypeList] ClassBody

EnumDeclaration:
    enum Identifier [implements TypeList] EnumBody

NormalInterfaceDeclaration:
    interface Identifier [TypeParameters] [extends TypeList] InterfaceBody

AnnotationTypeDeclaration:
    @ interface Identifier AnnotationTypeBody
```

These rules describe how a programmar will declare classes within a Java source file. In programming languages, a grammar is necessary because it captures the precise syntactic rules of the language. For example, in the above rules, under `NormalClassDeclaration`, we're told that when defining a regular class in Java, we *must* use the `class` keyword (e.g.: `public class HelloWorld`).

CongoCC has a flexible format with which you can express the grammar of programming languages such as C#, Python, and Java (or even your own constructed language!). Once you express the grammar in CongoCC's format, you can pass it to CongoCC to generate a parser that can parse text files following those rules.

CongoCC's Parser programs

As described above, the parsers generated by CongoCC are programs themselves. Note however that CongoCC does not produce a compiled parser program. Instead, CongoCC produces the source code of a parser program, which can then be independently compiled and used. As of writing, CongoCC is powerful enough to generate a parser source code in C#, Python, or Java. In other words, you can use CongoCC to generate a Python parser program that is capable of parsing C# source files! Support for more languages in CongoCC is planned.

When would I use a parser?

Parsers solve a very specific problem: capturing the structure of a text file according to some rules. Any time you need such structure, you need a parser. The need for parsers is immediate when you look at problems within the domain of programming languages:

- If you are writing a compiler, you need a parser.
- If you are writing a tool that verifies the source code written by people for various properties, you need a parser. For example, the Python linting tool Ruff needs a parser (https://github.com/astral-sh/ruff/tree/main/crates/ruff_python_ast/src)
- If you are developing an IDE like PyCharm, and you want to provide a functionality where users can rename a function `foo` to `bar` and have the rest of Python project be automatically updated so that all previous calls to `foo()` are renamed to `bar()`, you need a parser.

But let's say you're never going to do anything with programming languages. Even so, parsers can be useful! Imagine you're writing a system for businesses and you want to provide them with a way of expressing different kinds of business rules, such as: "if my current stock of printing paper drops below 10 units, automatically order more from supplier A, B, or C, after taking into consideration my budget, the prices from the suppliers, and their shipping speed."

In such instances, you need some kind of a DSL (Domain Specific Language). It could even be some custom JSON-based format, e.g.:

```
{
  "if": {
    "ItemType": "PaperStock",
    "ConditionType": "LessThanEqual",
    "ConditionValue": 10
  },
  "then": {
          "DoSomethingHere".
  }
}
```

When the user of your system provides you with some text written in your custom DSL, you will need to reason about it so that your system can do the appropriate things.

Your first option is to write adhoc logic in your program:

```
json = <input json from user>
condition_to_check = None
then_clause = None
```

```
for stanza, attributes in json.items():
    if stanza == "if":
        item_type = attributes.get("ItemType")
        condition_type = attributes.get("ConditionType")
        condition_value = attributes.get("ConditionValue")
        if item_type is None or condition_type is None or condition_value is None:
            raise RuntimeError("The rules are not well-formed. Missing either ItemType,
ConditionType, or ConditionValue.")

        condition_to_check = (item_type, condition_type, condition_value)
    elif stanza == "then":
       then_clause = produce_then_clause_from_attributes(attributes)

if condition_to_check is not None:
    if then_clause is None:
         raise RuntimeError("The if-condition is malformed. A "then" clause is not provided")

    if run_condition(condition_to_check) is True:
        then_clause.execute()
```

As you can see, this is very error prone and time consuming. Each time you add new features your DSL, you will need to do a lot of work to maintain this adhoc code.

The second option is to make use of parsers:

```
json = <input json from user>
parsed_tree, error_message = parse_tree(json)
if parsed_tree is None:
    raise RuntimeError(f"The specified business rules are not well-formed. Encountered the
following errors: " + error_message)

for node in parsed_tree:
    if node.type == NodeType.IF:
        if run_condition(node.condition) is True:
            execute(node.then_child_node)
```

With a parser, you have cleaner code because you no longer need to parse anything yourself and you don't have to deal with all the error cases. You can instead focus on your business logic. With parsers, you have another two options: either write the parser yourself or use CongoCC. If you want to spend more time actually working on your business logic, the second option is the correct option :) You can express your DSL rules in the CongoCC format, generate a parser, and use the parser in your system without needing CongoCC again until the next time you change your DSL rules.

A Closer Look at Parsers

Parsers in detail

The previous section described parsers at a high-level and described when they may be used. In this section, we're going to look at parsers in closer look. The concepts discussed here are foundational to how CongoCC works and is required if you want to work on CongoCC or do advanced things with it.

Tokenizers and Lexers

Imagine you're writing a simple program to count the number of words in an input english-language string, like "The quick brown fox jumps over the lazy dogs". Your program would probably do something like this: `count_words = len(input_string.split())`. This works because the rules in english are simple: characters in a sentence not separated by a space are part of the same word and words are separated by a space. In this program, the sentence is split into "tokens" using `input_string.split()` and the number of tokens is the number of words.

A tokenizer takes as input a text file that follows the rule of some language and breaks the text file into individual tokens. It's more complicated than just splitting the text file by space because what constitutes a "token" is different in different languages. Take the following line of code:

```
string some_var = "hello world";
```

As you can see, there are 5 tokens:

- `string`
- `some_var`
- `=`
- `"hello world"`
- `;`

Unlike the rules of english, it's not enough to just on whitespace (e.g.: the space in `"hello world"` belongs to the token) and characters next to each other can be part of different tokens (e.g.: the `;` after `"hello world"`)

In our string counting program, we only cared about the *number* of words. In more advanced programs like parsers and compilers, we need to understand how the tokens actually relate to each other and to do that, we need to what the tokens represent. Following the earlier example, it's more useful if the tokenizer can look at the line of the code and produce the following output:

- Token: `string`, TokenType:
- `some_var`
- `=`
- `"hello world"`
- `;`