LINGI2132 - Languages and Translators
Building our own language
Part 1 - Grammar and AST

DE VOGHEL Brieuc - 5910 1600
KNEIP Elisabeth - 2106 1700

Friday 12th March, 2021

Abstract

The language we are implementing is named "Kneghel", after the contraction of our names.
In this report, we detail the design choices of our language and explain its features and limitations.

1 Design choices

Kneghel is mainly inspired by the Java programming language. It also incorporates some functionalities from Python and is dynamically typed.

We like the verbosity it inherits from Java : this makes the language more clear and easier to parse. Keeping the same structure as the OOP style of Java (functions in classes, where the `main` function is the one evaluated), allows for a better integrity and understanding of the evaluation.

Also, we like our language to be explicit, meaning the programmer has more work to do, but only needs a basic understanding of the underlying working of Kneghel to be able to write a correct program. A beginner will thus have a less steep learning curve.

Short code-snippet giving an overview of what Kneghel looks like.

```
class MyClass {
    fun myFunction(a, b) {
        // this is a comment
        c = a[0] + b * -2
        print("Returning (a[0] + b) * 2")
        return c
    }

    /*
     * Comment (doc-string style)
     */
    fun main(args) {
        x = int(args[0])
        if x < 2 {
            myFunction(args[1], 2)
        } else {
            y = 0
            while y < 100 {
                y = myFunction(args[1], x)
            }
        }
    }
}
```

1.1 General structure

Every Kneghel file contains one class (like Java). Classes may have a `main` function, which will be evaluated when the program is run. Every other function must have a `return` statement.

Classes and functions are declared with the keywords `class` and `fun` respectively. Their declaration follows the Python style.

The main function accepts the parameter `args`, being array of strings, containing the parameters given to the program.

The rest is mainly straightforward when looking at the code snippet provided above.

1.2 Specificities

As we mentioned before, our language is dynamically typed. We do not need to indicate the type of a variable before using it. However, the different type names are used for parsing.
Java's way of parsing being not straightforward enough we chose to make specific functions for it, very much like Python. For example, to parse a string into an integer you would only have to call the function `int()` with the string as the parameter. We chose to implement doubles as well and the parsing function `double()` works exactly the same.

In case there were any error during the parsing, we needed error handling. Once again, we took inspiration from Python and Java and decided to raise exceptions when an error occurred.

Another choice we made was to not separate the initialisation and declaration of a variable. In doing so we created a counter-effect to the wordiness of our language. We tried to preserve a balance in order to avoid unnecessary long files when writing in Kneghel.

1.3 Features

One special feature we implemented is to accept these statements : 5 + -1 and 5 +- 1. They both add minus one to five. Negative numbers do not need to be in parenthesis.
However we do not accept 5 + +1. Positive numbers cannot have their sign in front.

2 Project Structure

Our project is divided in two main folders : the `main` and the `test` folder.

In the latter, there is the test file for our parser and the test file for our example code. The tests in a file can be run together -by running the whole class-, but they can also be run independently.

The `main` folder is a bit more complex. It contains three folders : `AST`, `examples`, `parser`.
The code for our parser is in the folder with the same name. In the `AST` folder, there is a large number of classes containing the different kinds of nodes our tree can have. We first implemented an `ASTNode` interface, then we developed two interfaces for the nodes: `ExpressionNode` and `StatementNode`. Those interfaces both extend the first one.
Each node implements either the `ExpressionNode` or the `StatementNode`. The difference is that the `ExpressionNode` is a value while the `StatementNode` is, like its name says, a statement.
In our parser, we only push on our AST tree objects that implement the `ASTNode`.

In the last folder, `examples`, there are a number of small programs written in Java and their translations in our language, Kneghel.

3 Testing

A series of unit tests provide comprehensive testing of all our different parsing rules. As rules combine into larger structures, the coherence between different rules is also maintained.
Tests for the example programs are yet to be developed more thoroughly, however they still provide robust testing for our complete parser.

4 Future Improvements

Our parser does more than asked. We implemented classes and doubles for example. However it is not complete. There are a number of things we could improve and implement in the future.
The first thing to do is to correct our parser. When running our tests with our example code, we observed some details than need improvements. Everything did not parse correctly together although separately, it does.

For example, one is to handle our errors correctly. Another is to make sure our reserved variable args contains the right parameters that were passed to the program. But we will handle that at runtime.
We also need to find more edge cases in order to make sure our parser is robust.

Still, we are satisfied with our accomplishment.