LINGI2132 - Languages and Translators
Building our own language
Part 2 - Semantic Analysis

de Voghel Brieuc - 5910 1600
Kneip Elisabeth - 2106 1700

Friday 2nd April, 2021

Abstract

The language we are implementing is named "Kneghel", after the contraction of our names.
In this report, we detail the changes in our language since its parser implementation and explain its features and limitations after having implemented the semantic analysis.

1 Changes in existing code

Our first modification was the testing of our example codes in a separate scripts. These tests allow us to separate unit tests and more complex, life-like examples.

In order to do the semantic analysis, we had to make a few changes in our code, particularly in our AST. First, we made most of the nodes' attributes public. They were originally private, and instead of writing numerous getters, we changed their accessibility.
Second, we added an abstract node : DeclarationNode. Indeed, we needed to differentiate a FunctionCall from a FunctionStatement for example.
Third, inspired by Uranium's sample code and the training session we had on semantic analysis, we added classes defining the scopes and the different types we use.

2 Features implemented in the semantic analysis

As decided during the grammar implementation, our language is dynamically typed. This allows for more flexibility to the programmer, but somewhat complicates the semantic analysis. In the following sections, we go through the different use cases and show what is and what is not allowed.

2.1 Using class as root

Kneghel uses the ClassNode as the root of any program file. Indeed, in our language, you cannot have a file without a class and you cannot have multiple classes in one file, similarly to Java. we can therefore not declare any variable or function before declaring a class because the scope would not exist.

2.2 Operations between different types

When writing the semantic analysis for our BinaryExpressionNode, we allowed our language to do arithmetic operations between doubles and integers. We simply verified that the types of the left and right children are one of the two, or that they are a argument in the scope of a function. As our language is dynamically typed, these cannot be determined beforehand.
A future improvement would be to include the arithmetic operations for strings and booleans as well. For the moment, any arithmetic operation (+, -, *, /, %) is only allowed between numbers.

On the other hand, boolean arithmetic (logical and, or, etc.) is only allowed for booleans. Here too, a future improvement could involve allowing to use integers and strings, holding boolean representations.

2.3 Remove and adding underscore to the scope

In our language, all functions need to return a value, and all function calls' return value must be assigned. An underscore is an identifier that we need to be able to reuse, allowing for the programmer to discard any non-pertinent return values. In other words, the value and the type of it associated to the underscore can change at any time. Usually, in a block, we are not be able to reassign a different type to a variable. However, we needed to do that for the underscore. Our solution is to remove the underscore from the scope every time we finish an assignment.

2.4 Identifiers as parameters

When calling a function, the parameters need to be an IdentifierNode. For the function Fibonacci, we need to declare even the integers used to initialise the method. It is a limitation we know we will have to work on later. However, we want to keep the idea to declare as much as possible before using it. This keeps us from having long lines of code and therefore assures us of having a well structured code.

2.5 Use of parameters inside functions

As the types of the variables used as parameters inside functions are not predetermined, we gave them the particular `Unkonwn` type. This type can be used in arithmetic operations and boolean logic, without triggering semantic errors. At runtime, a Kneghel program would fail if a parameter is given the wrong type when used for a particular case.

3 Testing

In the `SemanticAnalysisTest` file, we have written a large number of tests. Most of them are based on the root rule to be able to use the scope. Our tests are therefore always testing the semantic of a class. Some tests were possible with other rules when we were not declaring anything.

An important part was to test our code examples. We tested them first as unit tests to be able to debug our semantic, and that helped a lot. They now have their own testing script, `KneghelExamplesSemanticTests`. Unfortunately, we have a bug concerning our preimplemented functions such as print, int, makeArray, makeDict and len. We have not yet found a way to make their semantic work.