Lab 3: Tree interpretation

January 15, 2015

The goal of this Lab is to implement an ECMAScript interpreter that will evaluate the parse tree provided by an antlr4-based parser. The *Environment* class developed in Lab2 should be used directly in your project, while the *Function* class will probably need some modification.

1 Code base structure

During this lab, you will be mainly implementing the function of the *Interpreter/InterpreterVisitor.py* class. You should also have a look to the *ECMAScriptParser/ECMAScript.g4*, it is the grammar definition for ECMAScript, using the antlr4 syntax. It can be useful to look at it to check the syntax.

2 InterpreterVisitor

This is the class that you need to complete in this lab. At this point it should look like this:

```
class InterpreterVisitor(ECMAScriptVisitor):
  def __init__(self, input=None):
    self.environment = Environment()
    self.environment.defineVariable("console", Console())

  # Visit a parse tree produced by ECMAScriptParser#PropertyExpressionAssignment.
  def visitPropertyExpressionAssignment(self, ctx):
    raise Utils.UnimplementedVisitorException(ctx)

  # Visit a parse tree produced by ECMAScriptParser#getter.
  def visitGetter(self, ctx):
    raise Utils.UnimplementedVisitorException(ctx)

  ...
```

If you look in the code, you will see that some functions have already been implemented for you. So that the following ECMAScript code can be (almost) executed:

```
console.log(10)
```

Which is enough to pass the first part of the first test. Now if you start running the test suite you will get an exception that should lead you to which function to implement.

3 Test suite

3.1 Test suite

You can find the test suite in */home/TDDA69/Labs/Lab3/Tests*, each test case is made out of two files, one ECMAScript file and one text file. The ECMAScript is the code that need to be interpreted by your code and the text file is what should be outputed on the command line.

There are currently eight categories of tests:

1. *01_literals*: test for literal expression: float, booleans and string
2. *02_expressions*: tests for expression (addition, multiplication, logical...)
3. *03_variables*: tests for the creation and accessing variables
4. *04_statements*: tests for statements (if, loops...)
5. *05_arrays*: tests for arrays
6. *06_functions*: tests for creating and calling functions and lambdas
7. *07_normal_order*: test for the normal order
8. *08_objects*: tests for creating objects and associating properties

Most of those tests also include subtest.

3.2 Testing your interpreter

We provide a utility called *espytester.py* that can be run in the following way:

```
tdda69_lab3_tests dir_to_espy
```

Where *dir_to_espy* is the directory where you have put your code. So if you have followed the course instructions, it should be in *$HOME/espy* and the following command will run the test suite:

```
tdda69_lab3_tests $HOME/espy
```

Since running the full test suite can take a bit of time, you can run a single test with the following command:

```
tdda69_lab3_tests dir_to_espy [testname]
```

For instance:

```
tdda69_lab3_tests dir_to_espy 02_expressions/01_addition
```

4 Parse tree Context

If you look at the skeleton of the visitor, all functions takes a *ctx* argument, those classes are defined in *espy/ECMAScriptParser/ECMAScriptParser.py*, they all inherits from the class *ParserRuleContext* which is defined in the file *espy/antlr4/ParserRuleContext.py*.

The most important member variable of a *ctx* is the *children* (*ctx.children*), as the name implies, it contains the list of children in the parse tree of the context *ctx*. And the most important member function is the *accept* function which allow to evaluate the context.

There are two types of children:

- *antlr4.tree.Tree.TerminalNodeImpl* this is a terminal node in the parse tree, for instance, a keyword or a literal. This class is defined in the file *espy/antlr4/tree/Tree.py*. The most important member variable of this class is *symbol* of type *Token* which represent the token of that terminal node.

  The *Token* class is defined in the file *espy/antlr4/Token.py*, it has two important variables:

  - *type* which is a number representing the token. Those numbers are defined in the *ECMAScriptLexer* class (always use the constant defined in that class rather than the actual number, those number change when the parser is generated by *antlr4*)
  - *text* which is the source code text for the token. So for a keyword, you would get the text corresponding to that keyword, for a literal, it contains the value.

- an other *ParserRuleContext*, it can be an expression, a statement...

If we look at the following code:

```
if(test)
{
  dosomething();
} else {
  dosomethingelse();
}
```

The *ctx* variable passed to the function *visitIfStatement* has the following seven children:

1. *ctx.children[0]* is of type *antlr4.tree.Tree.TerminalNodeImpl* whose *type* is *ECMAScriptLexer.If* and *text* is "if"
2. *ctx.children[1]* is of type *antlr4.tree.Tree.TerminalNodeImpl* whose *type* is *ECMAScriptLexer.OpenBracket* and *text* is "("
3. *ctx.children[2]* is of type *ECMAScriptParser.ExpressionSequenceContext*, it is the test expression
4. *ctx.children[3]* is of type *antlr4.tree.Tree.TerminalNodeImpl* whose *type* is *ECMAScriptLexer.CloseBracket* and *text* is ")"
5. *ctx.children[4]* is of type *ECMAScriptParser.Statement*, it corresponds to the statement to execute if the test expression is true
6. *ctx.children[5]* is of type *antlr4.tree.Tree.TerminalNodeImpl* whose *type* is *ECMAScriptLexer.Else* and *text* is "else"
7. *ctx.children[6]* is of type *ECMAScriptParser.Statement*, it corresponds to the statement to execute if the test expression is false

In this example, to evaluate the if statement, the first step would be to compute the test expression value:

```
  test = ctx.children[2].accept(self)
```

This will then call the *visitExpressionSequence* function of *self* and should return the execution result of the test. Then depending on wether the result is true or not, different statement are executed:

```
1  if(test):
2    ctx.children[4].accept(self)
3  else:
4    ctx.children[6].accept(self)
```