LINGI2132 : Final report

Group M
Maciej Piotrowski
Thibault Wafflard

We encountered some problems with the submission on Inginious. In particular, Inginious didn't pass some tests where we used some \r\n and wanted it to be just \n. So, for the submission on Inginious we changed it but that causes some tests to fail on Intelij IDE. You need to add a \r before \n in a few places in the test files.

1.General remarks

For this final submission, we were asked to write a walk-tree interpreter for our language, write a comprehensive list of automated tests and finish the project. Since our semantic analysis from the last submission wasn't completely finished, we first finished this part before diving into the interpreter part. Then, for the interpreter, we decided to make use of the code sample for the interpreter of the "Sigh" language (available on the teacher's github). We adapted it so that it can match and work with the specification of our language (TM). It was needed since our language works differently and is dynamically typed (Sigh is statically typed). We also made some tests to check the correct working of our language. Once we were done with the interpreter, we checked some last few bugs and once everything was up and running we decided to add some extra features.

2.Required features

- **Literals:** (*Integer*) 5, *(String)* "Hello", *(Boolean)* true, *(Float)* 2.0
- **Comments:**
  (Line comment) ##String
  (Block comment) #*String*#
- **Variable definitions:** *(Dynamic)*
  let x = 10
- **Simple arithmetic:** *(Sum)* 5+5, *(Difference)* 2-1, *(Times)* 3*1, *(Division)* 4/2, *(Modulo)* 3%2
- **Priority of operations:** 1+2*3 ⇔ 1+(2*3)
- **Conditions:** *(If)* if(true) {return (1)}, *(If and else)* if(x==5 && true) {return (5)} else { let y =2; return (y)}
- **Loops:** *(While)* let i = 0
  while (i < 3) { print(i+" "); i = i + 1 }
- **Function definition:**
  def sum_pair (pair){
  return (pair.a + pair.b)
  }
- **Array and map access:**
  [1].get(0)
  map.get("KeyString")
- **Function call:**
  fib(b, a+b, N-1)
- **Printing to stdout and errors:**
  print("FizzBuzz")
  rprint("Arithmetic Exception")
- **Strings parse to int:**
  parseInt("5")
- **Passing parameters to program:**
  java /TMcompiler.java "path_to_file.tm" arg1, ..., argN -walker [optional]

3.Extra supported features

- **Generic types:**
  struct Leaf {name = "Leaf"; value = anInt }
  struct Root {name = "Root"; value = anInt; left = new Leaf(); right = new Leaf() }
  def equals (nodeA, nodeB){ return (nodeA.value == nodeB.value) }
  equals() will work with both types of structures
- **Final variables:**
  pinned x = anInt ; x = 2; ---> AssertionError because x was pinned (final)
- **Object orientations:** struct Leaf {name = "Leaf"; value = anInt }
  We didn't implement object orientation as is, but we implemented structures that are working very well and could be used like a sort of object orientation
- **Better handling for the errors/exceptions:**
  In our language, it's not possible to create new errors/exceptions but we are handling (normally) all the errors and displaying a String describing the encountered error
- **Exponent:** 5**2

4.Features semantic

- It's mandatory, when declaring a variable, to assign a value to it directly.
- We have some default values for some types: anInt, aFloat, aString. We can use them to represent respectively 0, 0.0 and an empty String (“”)
- Each return statement needs to have the result that it return between parentheses and there are functions without any return statement and they are of type Void
- Our language allows if/while conditions like: while(true)
- We have one put and one get methods that we can apply to both, maps and arrays, with the same grammar
- If we want to put an value in an array at an index that exceeds the array's size, our language will automatically resize the array to fit the query
- Our “put” method returns a boolean: true if it puts a value in an array at an index that is less or equal the array size or in a map with a key that wasn't there already, false if it the put resize the array or if the key exists already in the map
- It's possible to create empty arrays, arrays of a certain length given in arg and also it's possible to create an array containing already values that we want
- Our language is able to already use functions that we are defining afterwards and manage recursive functions
- If we want to compare any types (arrays, structures, etc.), we can use “==” if they have the same content

5.Observations

- The put/get methods for arrays and maps:
  First of all, we have only 1 method put and 1 method get for both arrays and maps. The difference in the GET is that for an array we will put an Int as an arg for the index and for the map we will put a String as an arg for the matching key. For the PUT method, we will give an index (int) and a value for the array and an key (string) and value for the map.

The second important observation is that our PUT method updates the array. For example, if we have an array of size 5 and want to put an element at the position 6, it will update the array so that it will have a size of 6 now.
The last thing about the PUT method is that it returns a boolean. If everything goes fine it will return true but if it puts an element at an index higher than the array size or with an already existing key, it will return false.

- Structures:
  When defining a new structure, we can either assign directly some values at the structure variables that will be reused by each of the structure's instances or we also have some default values just to indicate which type the variable will be. We can either create a new instance with some new values given in parameters or a new instance without parameters that will therefore take the values defined in the structure definition.
  An extra point is that we can define functions so that we force it to take into parameters a certain type of struct by doing for example: def name(a:Person), we will force the function name to consider only Person type struct as args.

- Primitive types:
  All the primitive types are managed from the semantics but the errors related to the "free" variables (NotYetType) are only managed in the interpreter afterwards. That's why we have 2 places where we are catching some basic exceptions, in 2 different files (related to semantics and interpreter). We left it like this, so that we are able to catch some easier errors earlier.

- The NotYetType is one of the most important features of our language. It allows the language to be very permissive, mostly during the interpretation of a function. Indeed, ast.ParameterNode (used as parameters function) are often Not Yet Typed. This means that the use of these variables are very dynamic. For example : In two different instances of the same function, the compound operation could be very different thanks to the dynamic type. Also, two different structures can have the same attributes name, and so a function can use this similitude to interact with different structures like it was the same.

- The UtilStatic contains a lot of useful functions that we use.

- It's possible to test entire files just by creating them, writing the code to test and putting the file in the main directory. You can run it by writing:
  java /File_Name.java "path_to_file.tm" arg1, ..., argN -walker [optional]
  The -walker is optional and is there to print the entire AST.

- If no particular file is given, the TMCompiler test will launch the tests on our TM_code_samples.tm which contains all major code samples that we are testing.

6.Tests

A big part of the project are the tests. Indeed we did a lot of tests to check each part of our language. We did a lot of separate tests (that you can find in the tests directory) and also a test that takes a file with some code in our language and tests it entirely. We passed all of them. We have a separate test to check each part of the project: grammar, semantic and interpreter. The tests were an important and very useful task because it helped us not just to detect that we have errors but to see where they come from. Finally, after all the corrections, the tests assure us that all the 3 parts of the project are working well all together.

7.Conclusion

Our final version isn't perfect and doesn't manage a lot of extra features but we are very happy how far we come. The beginnings were hard because it's a project that covers a lot of new concepts that we didn't see ever before and we were a bit lost. We were using Autumn's library which doesn't provide a lot of documentation besides the one on the github repository (which is very complete but quite complex). That's a reason why we fell behind a little bit but we managed to catch up and to progress. The github repository with the Sigh language helped us a lot because there were examples of grammar, semantic and interpreter that showed us how it can be done and helped us to understand much better the working of Autumn. Once we understood it, it was a lot easier to write things for our language. Our language is very dynamic and has a lot of nice and fun features but unfortunately, we were missing time to develop it more and add some amazing extra features. In general, the project was very interesting and we learned a lot of things but we feel that we could do much better if we had more time or if we had "unlocked" earlier.