Troy Cope

Concepts of Programming Languages

CS4308, Section 03

Deliverable 3, Interpreter

April 17th, 2022

Sharon Perry

Project Deliverable is 100% completed and working as designed.

As with part one and two, the Scanner and the Parser, Java was the language of choice for the third deliverable, the Interpreter. My implementation of an interpreter for the language Julia was built around the structure that the parser produced. It is important to note that several parser files were updated to implement the interpreter with full encapsulation. This also included separating them into classes that way direct typecasting was available for certain methods such as evaluateArith(). This also came with an implementation of an abstract class [InTree] which allowed for indirect typecasting to use a getChildren() method.

Onto the creation and process of making the interpreter. The interpreter itself started with a drastic and defining choice: either build it around the way the parser was already doing things via implementing a tree with multiple branches or simplify the tree into a linear architecture which would make implementation easy. As expected, I choose the former. To begin this implementation, it was necessary to build a standard object loop. This is the first occurrence of InTree casting, where I downcast to keep a continuous stream of the lowest level children possible. This effectively uses a depth first algorithm to ensure that instructions are being used properly. With the base loop created, the next step is to implement each object. To do this I use a switch statement and the variable director. With a bit of preprocessing this runs smoothly and calls each function case based on their lowest level class. This is where the second part of my design comes in, which is more efficient but also a more individualized approach. I use the lowest level classes (leaves in a tree for instance) to pass back their parameters using returnArgs(). By passing back arguments it allows them to be processed directly via their parents. This also contributed to my other goal, which was to use as little data storage as possible, therefore the large majority of the program is intended to be used from Strings. Maintaining Strings was a personal choice and a challenge. There is a singular exception to this, from the assignStatement, which could be changed by pushing the parameters by one each. I chose not to make this change for clarity reasons.

Once the data is passed upwards and the individual cases are called, the interpreter then focused on each case. The cases worth mentioning are the assignStatement block, ifBlock and whileBlock. The assignStatement case uses a Hashmap<String, Integer> to map keys which act as the RAM of the Julia program. It also calls the evalateArith() method which is nearly identical to the evaluateBoolean() call. These are shown with their own comments in the code and extensively explained. The repeatBlock is a copy of the whileBlock but used swapped parameters and therefore is not mentioned.

The architecturally interesting cases are the whileBlock and the ifBlock. The idea was to address these two blocks with the multiple-branch tree approach in mind. Therefore, there are extensive comments in the code showing why I choose what. I will recap these comments briefly. The ifBlock uses a slimming of children to simplify the program, making it more linear in approach instead of the numerical linearization (meaning that the program executes branch cases in the order they are called via a depth first algorithm) the program initially has. The whileBlock is an unorthodox approach in context with the uses of the ifBlock. Instead of using the architecture to its advantage, the whileBlock handles multiple blocks within its case. To do this, it replicated the base loop which the entire program runs off, and recursively calls the conditional while block until the while condition is satisfied. Once the condition is satisfied, since the declarations are direct copies and bear no new children, the program resumes as normal. The rest of the program resolves itself, and therefore once the loop condition is fulfilled the interpreter ends.

Input and Output Capture

Test1, input:

Graphical user interface, text

Description automatically generated

Test 1, output:



Test2, input:

Text

Description automatically generated

Test2, output:

Text

Description automatically generated  
NOTE: If the unknown token “=+” is accounted for in the parser, then the output is 3.

Test3, input:

Text

Description automatically generated

Test3, output:

