Course Project Report

Deliverable # 3

11/26/21

Sarah Thomas

sthom317@students.kennesaw.edu

Kennesaw State University

College of Computing and Software Engineering

Department of Computer Science

CS 4308, Concepts of Programming Languages, W01

Report Content

Purpose:

The goal of this course project is to test our understanding of lexical analysis by developing a language processor for a subset of the BASIC language. We will be developing and implementing a scanner, a parser, and interpreter as part of our development process. For the third deliverable of this project, we were to develop and implement a complete interpreter that also works with a scanner and parser that we previously built. The job of the interpreter is to take in a statement whether from other code or user input, convert it to machine code and execute it. .

Solution:

To build and implement my interpreter, I created a new Python file, **'interpreterLexicalAnalysis.py'** , within the same folder as the files used to build and implement my scanner and parser. These previous files included **'main.py**', **'scannerLexicalAnalysis'**, and **'parserLexicalAnalysis'**. **'main.py'** included script that received input from a user, which was later changed to receive input from a text file named **'parserTest.txt'**. **'scannerLexicalAnalysis.py'** includes all the code to build my scanner and **'parserLexicalAnalysis.py'** contains all the code to build my parser. The first step to building my interpreter was to import both the scanner and parser Python files into the interpreter Python file. This would allow access to the class **Token**, which held all the token information, as well as the **Scanner** class and **Parser** class, which allow both the scanner and parser to work in tandem with the interpreter. After importing all necessary information, I defined 2 classes within **'interpreterLexicalAnalysis.py'** : **Number** and **Interpreter**. The **Number** class held functions that would allow mathematical operations to be performed and executed on the nodes passed in to the interpreter by the parser. The **Interpreter** class contained functions that kept track of whether nodes were visited or not during the program's execution and housed a function to run the program. The definitions of each function per class are given below:

<u>Number class:</u>

- **set_position()**: This method takes in a start position of a node and an end position of a node, initializes these values, and then returns the positions of the node
- **added():** This method is called to add numbers. It first checks to see if there is are number nodes available and if there are, it adds the values together and returns the sum.

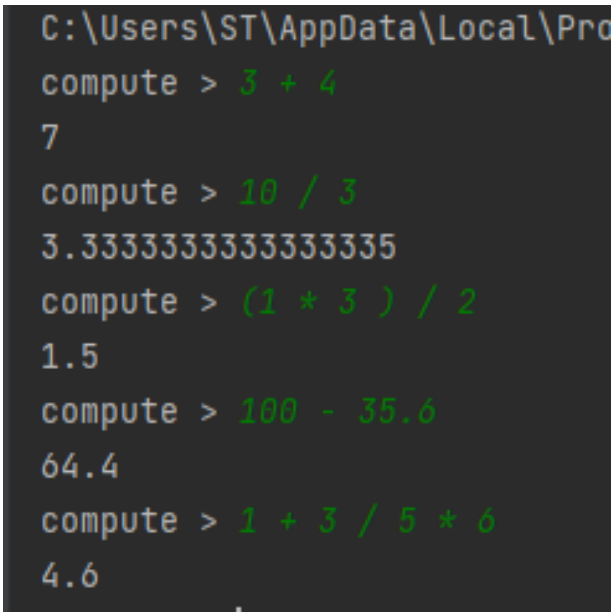
- **subtracted():** This method is called to subtract numbers. It first checks to see if there are two number nodes available and if there are, it subtracts the second value from the first value and returns the difference.
- **multiplied():** This method is called to multiply numbers together. It first checks to see if there is are number nodes available and if there are, it multiplies the values together and returns the product of the two values.
- **divided():** This method is called to divide numbers. It first checks to see if there is are number nodes available and if there are, the first value is divided by the second value and the result is returned.

Interpreter class:

- **visitNode():** this method takes in the node values, assesses them for what type of node they are (number, binary operation, unary operation) and calls a specific 'visit node' function based on the type of node found
- **no_visit():** this method is called when there is not a visit method defined
- **visit_NodeNum():** if a number node is found, this method is called and an instance of the **Number** class is created as well as the positions of the nodes are set
- **visit_OpNode()**: if a binary operator node is found, this method is called and the operator token of the node is checked to determine what mathematical function from the **Number** class needs to be called. Then the two values are operated on based on the called function and position to find the result. The result is returned.
- **visit_UnOpNode():** If a unary operator node is found, this function is called. It multiplies the node by –1 by calling the **multiplied** function and then returns the node. This is when a negative number is evaluated.
- run(): this method runs the entire program

Once "**interpreterLexicalAnalysis.py**" is successfully built, I imported it to the **'main.py'** file to be compiled and called **run()** to execute the program. The program will take in user input, create an instance, and use it to call the **tokenizer()** function from the scanner which is then stored in a variable called **tok**. Then the program takes the tokens that were outputted from the scanner as input for the parser. The **parse_tokens()** function was then called to parse all the given tokens and then used to build a tree. The **visitNode()** function from the interpreter is called and the parsed tokens are passed in as nodes. Finally, the result's value is returned.

Results:


C:\Users\ST\AppData\Local\Pro
compute > 3 + 4
7
compute > 10 / 3
3.3333333333333335
compute > (1 * 3) / 2
1.5
compute > 100 - 35.6
64.4
compute > 1 + 3 / 5 * 6
4.6

References

Sebesta, Robert W. *Concepts of Programming Languages - 10th Edition*. Pearson Addison Wesley, 2012.