Lab 6 – CSCI 112

Information

- This lab is intended to be completed **individually.**
- The files must be submitted with the exact file name provided in this file. If the file names do not match you will receive **zero** points for that file.
- Before you submit, make sure that your code runs. Any code which does not run without errors will receive **zero** points.
- Do not share your work with anyone other than Professor Khan or the TAs. You may discuss algorithms, approaches, ideas, but **NOT** exact code.
- If you submit work after a second past the due date **WILL** be locked out from submission.

Review

Stacks

Stacks are a linear structure where both access and insertion are limited to one side. The top of the stack can be looked at, added to, or removed from. The data cannot be changed inside the stack.

Utils Module

As our programs become more complicated, with more files, it is good to store utility classes inside a module for organization. Inside the files to download is now a folder named **utils** which contains basic classes and containers. In addition to your array and node files, all stack implementations are inside this module.

linkedStack.py, arrayStack.py, abstractStack.py

The code for these three structures is provided in the book. Open the files and familiarize yourself with the implementations. Why does the **__iter__** use a different approach than we normally program in our collections?

Infix and Postfix Expressions

A typical mathematical expression you are used to seeing is written in **infix** notation. As a human, you then use your knowledge of the order of operations to determine which parts of the expression get evaluated first. For example, the following expression:

```
3 + 5 * 2
```

Despite the 3 plus 5 part occurring on the left, the order op operations indicates that 5 times 2 must happen first. Then 3 may be added to that result. **Postfix** notation removes the need to memorize these rules. Postfix is evaluated strictly on a left to right basis. When an operator is seen, it is applied to the two previous values and replaces the two numbers with the result. The above expression would then be written, in postfix:

```
3 5 2 * +
```

Calculation of postfix expressions is much easier for a computer to handle, and therefore if we can translate infix to postfix, then calculating the value of a mathematical expression is easier.

Assignment

Task 1 – Evaluators and Tokens

Open the `evaluator.py` file. This file defines an **Evaluator** class which will evaluate **postfix** expressions as we discussed in class. This class collaborates with the **Scanner** and **Token** classes to evaluate syntactically correct postfix expressions. Run the program and test out some postfix expressions utilizing *****, **+**, **/**, and **-**. Other operations do not work at this point.

Modify the **Evaluator** and **Token** classes to support two new operators, **%** for remainder (modulo) and **^** for exponentiation. Add new token types for **%** and **^**, modify `computeValue`, and test your program to see if it's working.

Task 2 – Translator

Open the `translator.py` file. This file defines a **Translator** class as discussed in lecture. This class collaborates with the **Scanner** and **Token** classes to convert syntactically correct **infix** expressions to **postfix** expressions. The code doesn't work yet as the logic for **translate** is missing. Following the algorithm discussed in class, program the translate method to convert infix to postfix expressions. Try programming it first **without** utilizing parentheses. **5 + 4 * 3** should translate to **5 4 3 * +.**

Once the translator works without parentheses, modify **Token** to have two new types for **LPAR** and **RPAR**, **(** and **)** respectively. Also note that these are not to be considered operators. Make sure to update the precedence for **%**, **^**, and the parentheses while you're in **Token**. Note here that while with normal operator precedence, **LPAR** and **RPAR** would have highest precedence, it is not the case in our program. **LPAR** should remain on the stack until an **RPAR** removes it. Test your program with several infix expressions including parentheses.

Once your translator works, you can open the `evaluatorApp.py` file to see it all work together. This file uses the `translator` and `evaluator` together to calculate the result of an infix mathematical expression. Test out your code with the `evaluatorApp.py` to verify that the translator and evaluator work together.

What To Turn In

Create a zip file named **Lab_6.zip**. Inside this zip archive should submit all the files from both tasks. Make sure your files have the correct names (including case), and extensions.