CSCI 204 – Introduction to Computer Science II

Lab 6: Stack ADT

1. Objectives

In this lab, you will practice the following:

- Learn about the Stack ADT
- Implement the Stack ADT using an array
- Use a Stack to convert expressions from infix form to postfix form
- Use a Stack to compute the answer to postfix expressions
- Review and use Python exceptions

2. Getting started

Make a directory for this lab `lab06` under your `csci204` directory and copy all files from the course Linux directory `~csci204/2017-fall/student/labs/lab06/`. You should get these two files.

```
infix2postfix.py  stack.py
```

3. A simple calculator using stacks

Stacks have many uses. In the earlier calculators, they were using stacks to process numbers and math operators. You are accustomed to seeing math expressions such as `3 + 4` in **infix** format where the operator (+) is in between the operands (3 and 4). This format is easy for the human user: for example, if you see an expression `3 + 4 * 5`, you know that first 4 should be multiplied by 5, and then 3 should be added to the result. It is harder for a computer that reads input from left to right to follow the same steps. It first reads 3, then reads "+" sign. When it encounters 4, it does not know whether any operation of higher precedence is coming, so it may attempt to add 3 and 4 right away. The solution is to use a special format called **postfix** notation where the operator comes after the operands so `3 + 4` becomes `3 4 +`. The calculator software can then scan the input from left to right and figure out the proper order of operations to produce a final result.

In this lab, you will implement a **Stack ADT** as an **array**. You will test your stack on the provided infix to postfix program. Then you will extend the program so that it will handle expressions with parentheses. Then you will extend the program to compute answers for arithmetic expressions.

4. Arrays and Python

The built-in Python list is built **using an array**. It provides a lot of list functionality not present in the basic array. We will be using just the basic array (as you did with a few previous labs).

This is very similar to what you did in the list lab. Note that because we are using an array, you'll need to keep track of both ***`size`*** (how many items are in the `Stack`) and ***`capacity`*** (the maximum number of elements that you use to store Stack items).

Refer to the given program `stack.py` to find out the methods you need to implement. Test the work of your stack operations before moving on to the next part. In order to test this, write a collection of tests in a file named `teststack.py` that performs various tests. The `main()` method should push and pop a number of items and test `is_empty()` and `top()` between each push and pop. Have the `main()` method print an error message when any test comes back with unexpected results. Here is an example to get you started.

```
x = MyStack()
if not x.is_empty():
   print("Starting stack should have been empty but isn't")
x.push('q')
y = x.top()
if not y == 'q':
    print("Top of stack should be q but was", x.top())
if not y == x.pop():
    print("The pop() method has problem(s)")
```

5. Stack

Your first task is to complete the Stack ADT.

Task 1: Complete the Stack ADT in `stack.py` so that your stack is implemented using an **array**.

1. Write a constructor to create an empty stack with initial capacity of 2 and size of 0. The elements in the array should be initialized as None.
2. Write a `__expand__()` method which doubles the capacity of the array when full.
3. Write a `push()` method that add an item to the top of the stack.
4. Write a `pop()` method that removes the item at the top and returns it.
5. Write a `top()` method to return the item at the top without removing it.
6. Write an `is_empty()` method that returns `True` if the stack is empty, `False` otherwise.

6. Operator precedence and associativity

To understand the conversion from infix to postfix, you first need to understand operator **precedence** and **associativity**.

Is 3 + 4 * 5 equal to 35 or 23?

Did the *order of the operators* dictate your answer or did *which operators they were* dictate your answer? Hopefully, you agree that *which* operators were more important than what order they appear in the expression. **Precedence** tells us which operator is more important and gets computed first. In 3 + 4 * 5, the * has higher precedence than + so 4*5 = 20 gets computed and then 3 + 20 = 23 gets computed. Similarly, 3 * 4 + 5 means 3 * 4 = 12 gets computed and then 12 + 5 = 17 gets computed.

Is 5 - 4 -3 equal to -2 or 4?

The operators are the same here (and have the same precedence) so did you do the *left operator first* or the *right operator first*? (that is, 4-3 first or 5-4 first?) **Associativity** tells us which

operator is more important when both operators have the same precedence. In 5 – 4 - 3, both operators have the same precedence so we compute the answer for the left operator first (left associativity). The expression 5 - 4 – 3 means 5 – 4 = 1 and then 1 - 3 = -2.

All of the standard math operators (+, -, *, /) use left associativity. * and / have equal precedence to each other and have higher precedence than + and – (which have equal precedence to each other). The precedence and associativity of the operators will need to be preserved when we translate infix to postfix.

Task 2: Which operator has the greatest precedence in each of the following expressions? (Answer with the name of the operator or "same" if they all have the same precedence.) Save your answers in a file named *lab.txt*.

1. 3 + 4 + 5
2. 3 – 2 - 7
3. 8 * 4 – 2
4. 4 – 2 / 5
5. 5 / 3 * 9
6. 6 * 2 + 1
7. 7 + 3 – 2
8. 3 + 8 * 5
9. 6 + 2 * 5 - 8

7. Algorithm to convert prefix to postfix

Here is an algorithm to convert an infix expression to a postfix one.

```
Initialize an output string to be empty
Empty the stack in use
For each token in the input string   # the token can be a number or an operator
   If it's a number, add it to the output string
   Otherwise  # it must be an operator
      While (the top of the stack is an operator with greater
               or equal precedence to this operator)
           Pop the stack
           Add the popped operator to the output string
      Push this token on the Stack.

While there are things on the stack
   Pop the stack
   Add the popped item to the output
```

Example: 3 * 4 + 2

Input	Stack afterwards	Output afterwards
3	top → None	3
*	top → *	3
4	top → *	3 4
+	top → +	3 4 *
2	top → +	3 4 * 2
	top → None	3 4 * 2 +

Example: 3 + 4 * 2

Input	Stack afterwards	Output afterwards
3	top → None	3
+	top → +	3
4	top → +	3 4
*	top → * +	3 4
2	top → * +	3 4 2
	top → None	3 4 2 * +

*To check if an item is a certain type you might want to look up the `isinstance()` or `type()` functions. You can also consider using the `isnumeric()` method.

Task 3: What postfix expression is equivalent to the given infix expression? Save your answers in *lab.txt*.

a) 3 + 4
b) 3 – 2 - 7
c) 8 * 4 – 2
d) 4 – 2 / 5
e) 5 / 3 * 9
f) 6 * 2 + 1
g) 7 + 3 – 2
h) 3 + 8 * 5
i) 6 + 2 * 5 - 8

The file `infix2postfix.py` contains a complete program for converting an expression from infix to postfix notation. The input accepts any single digit 0...9 as values and the operators +, -, *, and / with **no** spaces between them. (You will be asked to revise the program to allow the expression to contain any number of white space characters and numbers of more digits.) Try to run the program `infix2postfix.py` on all the expressions you just manually translated into postfix to see what the program does.

8. Allow parentheses in the expressions

Your next task is to change the program so that it will handle input strings with parentheses. Parentheses have the highest precedence and trump all of the math operators. Due to the layout of postfix format, parentheses are never necessary in postfix.

9. Algorithm to convert prefix with optional parentheses to postfix

Here is an algorithm that can take care of the expressions with optional parentheses. The underlined portions are added to the original algorithm in Section 7 to handle parentheses.

```
For each number and operator in the input
    If it's a number, add it to the output
    Else if it's a left parenthesis, push it onto the stack.
    Else if it's a right parenthesis
        While the stack is not empty and the top of the stack is not a left parenthesis
            Pop the stack
            Add the popped operator to the output
        If the top of the stack was a left parenthesis, pop it
        Else there is a mismatched right parenthesis so throw an exception
    Otherwise # it must be an operator
        While (the top of the stack is not a left parenthesis
                and the top of the stack is an operator with greater
                or equal precedence to this operator)
            Pop the stack
            Add the popped operator to the output
        Push this operator on the Stack.
    While there are things on the stack
        Pop the stack
        If the popped item is an operator, add it to the output
        Else there is a mismatched left parenthesis so throw an exception
```

Example: 3 * (4 + 2)

Input	Stack afterwards	Output afterwards
3	top → None	3
*	top → *	3
(	top → (*	3
4	top → (*	3 4
+	top → + (*	3 4
2	top → + (*	3 4 2
)	top → *	3 4 2 +
	top → None	3 4 2 + *

Task 4: What postfix expression is equivalent to the given infix expression? Save your answers in *lab.txt*.

a) (3)
b) (3 + 4)
c) 3 – (2 - 7)
d) 8 * (4 – 2)
e) 4 – (2 / 5)
f) (5 / 3) * 9
g) (6 * 2 + 1)
h) 7 + (6 - 3) / 2
i) 7 * (6 – 3) / 2
j) (3 + 8) * 5
k) 6 * (2 + 5) – 8
l) (4 + 2) * 7 / 3
m) (6 + 2) * (9 + 1)

Task 5: Revise the code in `infix2postfix.py` so that it can translate expressions with parentheses from infix to postfix.

The program examines each character in the expression. If the character is a variable it does one thing, and if the character is an operator, it does something else. You will need to add other possibilities for the characters the program sees. You will add processing for a left parenthesis and processing for a right parenthesis.

Handle a left parenthesis
Put this code in its own method. If the character being processed is a left parenthesis, just push it onto the stack. Note that the stack may now contain left parentheses in addition to operators. You will need to make other changes later on because of this.

Handle a right parenthesis
Put this code in its own method. When your program sees a right parenthesis, pop operators off the stack and add them to the output until you reach the matching left parenthesis. When you reach the left parenthesis, remove it from the stack too (don't output it). If the stack goes empty while you are looking for the left parenthesis, it means the left parenthesis is missing and you should raise a `ParensMismatchException`.

Handle operators
The processing for an operator will be similar to what was done before. Previously, the program would pop and print operators until the stack became empty or the top of the stack has an operator of higher or equal precedence than the current operator. Now there is an additional condition that will stop this loop, the loop must also end if a left parenthesis is found on top of the stack. Don't remove that left parenthesis.

Emptying the stack
At the end of the `translate()` method, the program removes any remaining operators from the stack and add them to the output string. If there were too few right parentheses in the expression, there will be an unmatched left parenthesis on the stack now. If you see one, you should raise a `ParensMismatchException`.

Parentheses mismatch and illegal expression exceptions
If a parentheses mismatch happens during translation, the program raises a `ParensMismatchException`.

If a bad character (not a number, operator, or parenthesis) is seen, the program raises an `IllegalExpressionException`. Write the `ParensMismatchException` class so that it inherits from `Exception`. Then catch the two kinds of exceptions in the `main()` method and print an error message for each one.

Check your results
Make sure all of the following examples work. Run your program with each of the expressions from **Task 4** as well as the following expressions.

Helpful notes: you don't have to type these expressions over and over again to feed them to the program. Instead, you can save these expressions in a text file, e.g., *input.txt*, and redirect the input to the program from a terminal window by

```
python infix2postfix.py < input.txt
```

Infix (input)	**Postfix (output)**
(((3)))	3
((3)	error
(3))	error

10. Algorithm to compute the answer to a postfix expression

Here is an algorithm that computes the value of a postfix expression.

```
For each number and operator in the input
    If it's a number, push it onto the stack
    Otherwise # it must be an operator
        Pop the stack and store the number as the right operand
        Pop the stack again and store the number as the left operand
        Compute the value of "left operator right"
        Push the result onto the stack
Pop the stack and return the result
```

Example: 3 5 1 - *

Input	**Stack afterwards**
3	top → 3
5	top → 5 3
1	top → 1 5 3
-	top → 4 3
*	head → 12

So far the `translate()` method in `infix2postfix.py` can only process expressions of limited format, i.e., expressions with single digit as value and no space in between values and operators. This restriction makes writing of the program a bit easier, but really limits the functionality of the program. To relax this limitation a little bit, we will allow any valid numerical values (e.g., values with multiple digits) in our input. For example, 123 + 34 or (345 + 567) * 4 / 2. However we still impose the limit such that the numbers and operators are separated by at least one white space. (Think about how to process the expression if no white space is required. It is definitely doable, but a bit more complicated.)

Task 6: Revise the method `translate()` such that the program is able to handle expressions with values that are more than one digit. Note that in this case, we require each token in the input be separated by at least one white space.

Try your revised program with the following input.

3 + 4 * 5
(3 + 4) * 5 + 6
(123 - 456) * 3
456 - 100 / (4 + 6)

Helpful note: How do you know what is the correct result for each of these expressions? You'd say "Use a calculator, of course!" You are correct. You can use the calculator that comes with any of the operating systems such as Windows or Linux, or you can use the one on your smartphone, or any one over the internet. However you can also use the Linux utility program called *bc* for basic calculation. Use the example shown in the following screen capture as a guidance, you can compute any value of any expressions. (The *bc* program doesn't need the white spaces in between tokens.) Use *Control-d* to quit the *bc* program.

```
[host]$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
3+4*5
23
456-100/(4+6)
446
^d
```

Task 7: Write a function in `infix2postfix.py` which computes the answer to a postfix expression. Name this function `evaluateExpression()` which takes a postfix expression as the parameter and returns the computed value of the expression.

Task 8: Create a file named `testinfix2postfix.py` that contains a `main()` function. Call your `evaluateExpression()` function from the `main()` function with the expressions listed in Task 6 and print the answers. Make sure you test all the expressions in this handout. Save the result in `lab.txt`.

11.Submission

Remove all unnecessary files and make a zip file from the `lab05` directory. Submit this zip file to Moodle. Make sure you include everything such that we can run your program directly without other files.