COMP 2140 Assignment 3: Stacks and Postfix Expressions

Helen Cameron

Due: Wednesday 9 November 2016 at 10:00 p.m.

Important Announcement

Until you agree to the honesty declaration (see the instructions at the end of Assignment 1 under "Honesty Declaration"), you cannot hand in any assignments — that is, until you agree to the honesty declaration, you will not be able to see the assignment dropboxes in UM Learn.

How to Get Help

Your course instructor is helpful: Email me at `Helen.Cameron@cs.umanitoba.ca` or come to my office hours at 2:30–3:30 p.m. Mon/Wed/Fri or 4:00–4:30 p.m. Tues/Thurs in E2 477 EITC (or by appointment).

For email, please remember to put "[comp2140]" in the subject and use a meaningful subject, and to send from your UofM email account.

Course discussion groups on UM Learn: A discussion group for this assignment is available in the COMP 2140 course site on UM Learn (click on "Discussions" under "Communications"). Post questions and comments related to Assignment 3 there, and I will respond. Please do not post solutions, not even snippets of solutions there, or anywhere else. I strongly suggest that you read the assignment discussion group for helpful information.

Computer science help centre: The staff in the help centre can help you (but not give you assignment solutions!). See their website at `http://www.cs.umanitoba.ca/undergraduate/computer-science-help-centre.php` for location and hours. You can also email them at `helpctr@cs.umanitoba.ca`.

Programming Standards

When writing code for this course, follow the programming standards, available on this course's website on `UMLearn`. Failure to do so will result in the loss of marks.

Introduction to Infix, Postfix, and Prefix Expressions

Infix, postfix, and prefix expression are three different ways to write arithmetic expressions. They differ in where an operator is placed relative to its two operands:

Infix: The operator is between its operands: `A op B`. For example, 3+5. (You learned this type of expression in grade school.)

Prefix: The operator is immediately before its two operands: `op A B`. For example, + 3 5.

Postfix: The operator is immediately after its two operands: `A B op`. For example, 3 5 +. (You will use this type of expression in this assignment.)

All of the above example expressions represent adding 3 to 5.

Compilers and calculators use postfix expressions internally, translating from the infix expressions that we humans write. (The first calculators expected humans to enter expressions in postfix; see, for example, HP calculators of old: http://www.hpmuseum.org/rpn.htm.)

Here's an example of a more complex arithmetic expression written in each of the three ways:

Infix: $(5+3)+6\times(7-2)$

Prefix: $+ + 5\ 3 \times 6 - 7\ 2$

Postfix: $5\ 3 + 6\ 7\ 2 - \times +$

Note: for this assignment, the only operators we will consider are addition ($+$), subtraction ($-$), multiplication ($\times$), and division ($/$).

Example of translating from infix to postfix: Create the corresponding postfix expression for the following infix expression:

$$(17-3\times4)/5+6$$

First, notice that the operands are in the exact same order in all three types of expressions. So we know that the operands will be in the order 17 3 4 5 6. What remains to figure out is where the operators go in the postfix expression.

If you were computing $(17-3\times4)/5+6$, the first computation you would perform is 3×4. Since the two operands of $\times$ are 3 and 4, the $\times$ operator must be placed immediately after 3 4 in the postfix expression:

$$17\ 3\ 4 \times 5\ 6$$

The next computation you would perform after 3×4 is to compute $17-3\times4$. Since the two operands of the subtraction $-$ are 17 and 3 4 $\times$, the $-$ operator must be placed immediately after 17 3 4 $\times$ in the postfix expression:

$$17\ 3\ 4 \times -\ 5\ 6$$

Next, you would compute $(17-3\times4)/5$. Since the two operands of the division ($/$) are $(17-3\times4)$ and 5, the $/$ operator must be placed immediately after 17 3 4 $\times$ $-$ 5 in the postfix expression:

$$17\ 3\ 4 \times -\ 5\ /\ 6$$

Since the last computation you would perform is to add $(17-3\times4)/5$ and 6, the $+$ operator must be placed immediately after 17 3 4 $\times$ $-$ 5 / 6 in the postfix expression, giving the completed postfix expression:

$$17\ 3\ 4 \times -\ 5\ /\ 6\ +$$

Example of translating from postfix to infix: In this example, we are given the postfix expression and are asked to translate it to the equivalent infix expression:

$$1\ 3\ 2 \times +\ 6\ 14\ 3\ 4 + / - +$$

We know that the operands will have the same order in the infix expression that they have in the postfix expression; we just need to place operators correctly:

$$1\ 3\ 2\ 6\ 14\ 3\ 4$$

Looking at the operators in the postfix expression $1\ 3\ 2 \times +\ 6\ 14\ 3\ 4 + / - +$ from left to right:

- The first operator (from the left) is $\times$. It is immediately after 3 2 so those must be its two operands — it represents (3×2):

  $$1\ (3\times2) +\ 6\ 14\ 3\ 4 + / - +$$

  I'm putting parentheses around terms in the expression when I figure them out, because they will be operands of other operators in the expression. The parentheses help me keep track of what I have figured out and what I haven't.

- The next operator is $+$. It is immediately after 1 and (3×2), so those must be its operands — it represents $(1 + 3 \times 2)$:
$$(1 + 3 \times 2)\ 6\ 14\ 3\ 4 + / - +$$
Notice that I didn't keep the parentheses around 3×2. Since $\times$ has greater priority $+$, we don't need them. You could keep them, if you preferred.

- The next operator is $+$. It is immediately after 3 and 4, so those must be its operands — it represents $(3 + 4)$:
$$(1 + 3 \times 2)\ 6\ 14\ (3 + 4)\ / - +$$

- The next operator is $/$. It is immediately after 14 and $(3 + 4)$, so those must be its operands — it represents $14/(3 + 4)$:
$$(1 + 3 \times 2)\ 6\ (14/(3 + 4)) - +$$

- The next operator is $-$. It is immediately after 6 and $(14/(3 + 4))$, so those must be its operands — it represents $(6 - 14/(3 + 4))$:
$$(1 + 3 \times 2)\ (6 - 14/(3 + 4))\ +$$
Notice I didn't keep the parentheses around $14/(3 + 4)$. Since $/$ has greater priority than $-$, we don't need them. You could keep them, if you preferred.

- The last operator is $+$. It is immediately after $(1 + 3 \times 2)$ and $(6 - 14/(3 + 4))$, so those must be its operands — it represents $(1+3\times 2)+(6-14/(3+4))$. The completed infix expression is the following:
$$(1 + 3 \times 2) + (6 - 14/(3 + 4))$$

Important note: Any sequence of operations can be specified in **prefix** or **postfix** *without parentheses* and without the need for *operator priorities.* To write an expression in **infix**, you must know *operator priorities* and use *parentheses* to override those priorities when necessary.

Introduction to Evaluating a Postfix Expression

Explanation: How is a the value of a postfix expression computed? Postfix expressions are easy to evaluate using a stack of operands. Process the tokens (the operands and operators) of the postfix expression from left to right:

- Each operand is simply pushed onto the operand stack.
- For each operator, perform the operation on its two operands, which are the top two operands on the stack. So pop the two operands off the stack. Finally, push the result of the operation onto the stack.

At the end, there should be only one value left on the stack. The value of the expression is the only value on the stack.
Pseudo-code:

```
Stack operandStack = new Stack();
for each token in the expression from left to right {
  // (a token is an operand or operator)
  if ( the token is an operand )
    operandStack.push( token );
  else if ( the token is an operator ) {
    if ( the operand stack isn't empty ) {
      operand2 = operandStack.pop();
      if (the operand stack isn't empty ) {
        operand1 = operandStack.pop();
        result = performOperation( operand1, token, operand2 );
        operandStack.push( result );
      } else
        error: only one operand on the stack
    } else {
      error: no operands on the stack
    }
  } else
    error: token is neither an operand nor an operator
} // end for
if ( the operand stack isn't empty ) {
  finalResult = operandStack.pop();
  if ( the operand stack isn't empty ) {
    error: too many operands on the stack at the end
    finalResult = invalid
  }
} else {
  error: stack is empty at the end
  finalResult = invalid
}
return finalResult;
```

Notice the order in which the operands of an operator are popped from the operand stack: the second operand is popped first, then the first operand is popped. The stack reverses the order of the operands.
Example: Process 5 -3 + 6 7 2 $-$ $\times$ +

	Stack	**Resulting**
Token	**Operation**	**Stack**
	Create	$[\ \rangle$
5	Push	$[\ 5\ \rangle$
-3	Push	$[\ 5\ \ -3\ \rangle$
$+$	Pop, pop, push	$[\ 2\ \rangle$
6	Push	$[\ 2\ 6\ \rangle$
7	Push	$[\ 2\ 6\ 7\ \rangle$
2	Push	$[\ 2\ 6\ 7\ 2\ \rangle$
$-$	Pop, pop, push	$[\ 2\ 6\ 5\ \rangle$
$\times$	Pop, pop, push	$[\ 2\ 30\ \rangle$
$+$	Pop, pop, push	$[\ 32\ \rangle$

The value of the expression is 32. The postfix expression rewritten in infix is $(5 + -3) + 6 \times (7 - 2) = 2 + 6 \times 5 = 32$, so the algorithm does indeed produce the correct value.
Example: Process 13 2 5 + $-$ 3 / 1 $-$ 12 2 / $\times$

Token	Stack Operation	Resulting Stack
	Create	[⟩
13	Push	[13 ⟩
2	Push	[13 2 ⟩
5	Push	[13 2 5 ⟩
$+$	Pop, pop, push	[13 7 ⟩
$-$	Pop, pop, push	[6 ⟩
3	Push	[6 3 ⟩
/	Pop, pop, push	[2 ⟩
1	Push	[2 1 ⟩
$-$	Pop, pop, push	[1 ⟩
12	Push	[1 12 ⟩
2	Push	[1 12 2 ⟩
/	Pop, pop, push	[1 6 ⟩
$\times$	Pop, pop, push	[6 ⟩

The value of the expression is 6. The postfix expression rewritten in infix is $((13-(2+5))/3-1)\times(12/2) = ((13-7)/3-1)\times 6 = (6/3-1)\times 6 = (2-1)\times 6 = 1\times 6 = 6$, so the algorithm has again produced the correct result.

Assignment Question

Remember: You will need to read this assignment many times to understand all the details of the program you need to write.

Goal: To evaluate postfix expressions like a calculator.

Your task twofold. **First**, you are to write a program to read in postfix expressions and to compute their values (or to print an error message if the expression is not valid) — see the details below. **Second**, in the comments at the beginning of your program, you will give the translation of the following expressions to the desired form:

1. Translate infix expression $16 - 5 + 3 * 2$ to postfix.

2. Translate infix expression $(2 \times 5 - 6) \times ((2 + 18)/4)$ to postfix.

3. Translate postfix expression $2\ 7\ 2\ 3\ \times\ -\ \times$ to infix.

4. Translate postfix expression $2\ 3\ 4\ +\ \times\ 2\ /\ 5\ 1\ -\ -$ into infix.

Ensure that you leave blanks between the tokens of your expressions.

Program Organization

For this assignment, you must have at least the following classes:

- A `Stack` class that must be implemented as a linked list (see class notes). It must contain a constructor, as well as `push`, `pop`, `top`, and `isEmpty`, but no other `public` methods.

- An `Expression` class, which must store an expression as an array of tokens. This class must contains at least a constructor, a method that evaluates a postfix expression and returns its value, and a `toString()` method to permit the application class to print a stored expression.

- The application class, which reads in the postfix expressions and prints them out along with their values; see "Program Input" and "Program Output" below.

Program Input

Your application class (the class that contains `main`) must prompt for the name of the input file.

The input file contains on its first line an integer (and nothing else). That integer tells you how many postfix expressions the file contains. The remaining lines of the file contain postfix expressions, one per line. You should use a simple `for`-loop to read in the expressions.

Tokens in an input postfix expression are separated by blanks. Therefore, you can use the `String` method `split()` to split a line on whitespace to get the tokens of an expression for processing:

```
  String[] tokens = inputLine.trim().split( "\\s+" );
```

Note that "`*`" is used as the multiplication operator in the input files, instead of "×".

I will provide a small input file (`tinyInput3.txt`) and a larger input file (`largerInput3.txt`). Your program should work correctly on both of them.

Program Output

As well as the usual title and trailer messages, you should print out each postfix expression and its value with suitable titles.

Sample Output

```
Comp 2140  Assignment 3   Fall 2016
Evaluating postfix expressions.
Model Solution

Enter the postfix expressions file name (.txt files only):
tinyInput.txt

**********************************************

The postfix expressions and their values
========================================

Postfix expression: -3 4 +
Value of expression: 1

Postfix expression: 10 30 + 2 /
Value of expression: 20

Postfix expression: 16 6 - 8 3 - /
Value of expression: 2

Postfix expression: 16 16 8 / - 2 +
Value of expression: 16

Postfix expression: 3 -
ERROR: Operator - is missing its first operand (second operand is 3)
Expression erroneous, value cannot be computed.

Program ends.
```

Hand-in Instructions

Go to COMP2140 in UM Learn, then click “Dropbox” under “Assessments” at the top. You will find a dropbox folder called “Assignment 3”. (If you cannot see the assignment dropbox, follow the directions under “Honesty Declaration” on Assignment 1.) Click the link and follow the instructions. Please note the following:

- Submit ONE .java file only. The .java file must contain all the source code. The .java file must be named
`A3<your last name><your student id>.java` (e.g., `A3Cameron1234567.java`).
- Please do not submit anything else.
- You can submit as many times as you like, but only the most recent submission is kept.
- We only accept homework submissions via UM Learn. Please DO NOT try to email your homework to the instructor or TA or markers — it will not be accepted.
- We reserve the right to refuse to grade the homework or to deduct marks if you do not follow instructions.
- **Assignments become late immediately after the posted due date and time.** Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.