CSCI 200 Lab 4—Evaluating infix arithmetic expressions

Please work with your current pair partner on this lab. There will be new pairs for the next lab.

Preliminaries

In this lab you will use stacks and queues to implement a number of programs which, combined, evaluate infix integer arithmetic expressions such as 21 + 4 * 5.

You'll first implement a tokenizer program that takes a string representing a complete infix expression and breaks it into discrete pieces called tokens. (In the expression 21 + 4 * 5, the tokens are 21, +, 4, *, and 5) Once we have a sequence of tokens, we can go on to convert the sequence into a form that a computer can evaluate more easily and then perform the evaluation.

Here is an example of what we're doing. Consider the previous infix expression: 21 + 4 * 5. It's called an infix expression because the operators are placed between the operands: + between 21 and 4 and * between 4 and 5. The tokenizer separates the expression into the tokens 21, +, 4, *, and 5, and places them in a queue in that order.

Next, an infix-to-postfix converter program reorders the tokens so that the expression is in postfix order with the operators following the operands. For our example, this process results in the expression: 21 4 5 * + (meaning that the multiplication (*) of 4 by 5 should occur before the addition (+) of 21 and the result of the multiplication).

If the infix expression contains parentheses, the conversion process eliminates them since postfix expressions don't need them for evaluation.

Finally, a postfix-expression evaluator program uses a stack to evaluate the expression:

```
   21 4 5 * +
= 21 20 +
= 41
```

Note that the order of operators in the postfix expression and the order of operations in the evaluation process follow the same order that we would use to evaluate the original infix expression.

Take some time to work through an exercise individually using pencil and paper; if you can’t do this by hand then you SHOULD NOT even TRY to program it. Start by producing the list of tokens the tokenizer program should give for the input expression 344+6*90/3 + (9-3)%5. Using this list of tokens, determine the reordered list of tokens the infix-to-postfix program should produce as its output. (Remember that it removes parentheses) Finally, use this second list of tokens to determine the value that the postfix-expression evaluator program produces; you should get 525. Once you have completed the exercise on your own, compare your solution with your pair partner’s, and correct any problems in your solutions.

*Show your solution to the exercise to the lab TA or instructor before you proceed.*
Part 1: Building and Testing a Token class hierarchy

Rather than just using strings to represent tokens, we will classify the tokens into a class hierarchy of token types. The root of the hierarchy will be a Token interface. There are two basic types of tokens: arithmetic operators like + and * and operands like 21 and 4; in addition, there are two special token types: left parentheses and right parentheses. The UML diagram below shows the complete hierarchy, as we will implement it.

Start by moving the expressions package from this lab folder into your JavaPackages folder (but keep the classes InfixPostfixDriver, TokenizerTest, InfixPostfixTest and PostfixEvaluatorTest in your lab folder).

Implement the above hierarchy inside this expressions package as you did for the worker hierarchy, but note that this hierarchy is really much simpler, even if it does contain more classes. The complete Token interface is given. Most of the methods will just return a constant value or the value of an instance variable. Implement each method at the highest possible level in the hierarchy that doesn't require using if or switch statements; declare methods in abstract classes as abstract if you don’t implement them there. (There’s no need to re-declare abstract methods already declared in a super-class or super-interface.) Please note the following:

Operator classes:
- getPrecedence: this method returns a precedence of 1 for the Plus and Minus classes, 2 for Times, Divide and Remainder (saying that operators *, / and % have higher precedence over + and -). Declare constants for the precedence levels in the Operator abstract class.
- evaluate: this method performs the appropriate operation on the values of the two operands and then returns a new IntegerLiteral operand constructed on the result of the operation.
**Operand classes:**

- The abstract class `Operand` contains only abstract methods; it is present to allow for more kinds of operands in the future.
- The `IntegerLiteral` class should have an `int` instance variable to hold the value of the literal.
- One `IntegerLiteral` class constructor should take a `String` parameter and throw an `IllegalArgumentException` if the string can't be parsed into an `int` (using `Integer.parseInt()`).
- A second constructor should take an `int` parameter.

**Leaf classes** (the classes across the bottom of the diagram and the two parenthesis classes):

- Override the `toString` method for each of the leaf classes. For example, the `Divide` class's `toString` method should return "/", while for `IntegerLiteral` it should return a `String` representing the integer value returned by its `getValue` method.

Include appropriate Javadoc comments for each class. Include a package statement in all these class files, and make sure you save all classes in the `expressions` package inside your `JavaPackages` folder.

This lab requires the use of the `stack` and `queue` data structures. We will use the `zhstructures` versions of both: the interfaces `ZHStack` and `ZHQueue` with the classes `ZHArrayStack` and `ZHArrayQueue`. All these interfaces and classes are generic over the class parameter `<ElementType>`.

**Important note about casting:** There will be places where you need to treat a token specifically as a particular kind of token; for instance, you may need to call the `getValue` method of an `Operand` token. If the variable type you're using is more general than the actual type of the token (probably `Token`) you may get a compiler error telling you that an object of that type (`Token`) doesn't have a method like that (`getValue`). If you know that the object is in fact a more specific subtype (`Operand`), you can cast it to the subtype and the compiler error will change to a warning about an unchecked cast. You'll probably need to do that several times in this lab.

Be sure that all code in this part of the lab compiles without errors before you go on. You may show this part to the TA or lab instructor at this point.
Part 2: A Tokenizer for expressions

Complete the supplied Tokenizer class inside the expressions package using the following guidelines.

The makeToken method gets a string that should represent one of the tokens in the hierarchy above. It determines which type of token it is, calls the appropriate token constructor and returns a new token object (e.g., new Plus() is created and returned for the string "+"). You may want to use a switch/case statement defined over the first character of the parameter string to determine the type of token. All tokens except integer literals come from strings of a single character, so the method should throw an exception if the string length is not one unless the first character is a digit. (We're not allowing negative literals in this lab; use (0 − 5) to get −5.) The method should also throw an exception if the string doesn't match any of the token types. Recall that the method call Character.isDigit(c), where c is a char, returns true if c is a digit.

The parseString method gets a string and scans it for tokens. Since there may or may not be whitespace between adjacent tokens, it is necessary to write the tokenizer by hand rather than using Java’s Scanner or StringTokenizer classes. Here's a pseudo-code version of the algorithm:

```
algorithm parseString( tokensStr : String ) : ZHQueue<Token>
set queue ← a new empty ZHArrayQueue<Token>
set i ← 0
while i < the length of tokensStr do
  if the character at location i in tokensStr is an operator
    or a parenthesis then
    call makeToken on a substring of tokensStr
    consisting of that character
    add the resulting token to the queue
    increment i
  else if character at location i in tokensStr is a digit then
    loop to find the next position j ≤ tokensStr.length
    that is not a digit
    call makeToken on a substring of tokensStr
    from location i to j-1 inclusive
    add the resulting token to the queue
    set i ← j
  else if character at location i in tokensStr is whitespace then
    increment i
  else [it's an illegal character at the start of a token]
    throw an IllegalArgumentException
end if
end while
return the queue
```

Note: you can use the Character.isDigit method to test for digits and the Character.isWhitespace method to test for whitespace. You can use the substring method to get substrings, but remember that the character at the terminal position in the substring method is not included in the result. (e.g., str.substring(i, j) returns a string with the characters of str in positions i to j-1.)
When you have completed the *Tokenizer* class so that it compiles without errors, use the provided *TokenizerTest.java* JUnit test class (available in your lab folder) to test that the classes you have implemented for this lab so far work and interact properly. This test class is complete and tests the *Tokenizer* class, which uses all the *Token* classes.
Part 3: Converting an infix expression to a postfix expression

In this part, you'll implement a class with a static method that takes a queue of tokens in infix order and returns a queue of tokens representing the same expression in postfix order. You'll then use a JUnit test class and a driver program to test your conversion.

Create an *InfixToPostfix* class inside your *expressions* package in your *JavaPackages* folder. Include the following three methods:

```java
// takes an infix queue and returns the equivalent postfix queue
public static ZHQueue<Token> convert(ZHQueue<Token> infixQueue);

private static void processRightParenthesis(ZHStack<Token> opStack,
                                          ZHQueue<Token> postfixQueue);

private static void processOperator(ZHStack<Token> opStack,
                                    ZHQueue<Token> postfixQueue,
                                    Operator op);
```

The *convert* method is the only public method of the *InfixToPostfix* class. It is the method that performs the conversion. It works by processing tokens from the *infixQueue* one-by-one, moving operands (*IntegerLiteral* tokens, in this case) directly into a queue called *postfixQueue* and pushing operators and left parentheses onto a stack called *opStack*.

Operators require special handling, since we move them to *postfixQueue only* after their second operand, and we must account for operator precedence before we push the operator onto *opStack*. When we encounter a right parenthesis, we have to process any operators pushed onto the stack since the last left parenthesis. The pseudo-code algorithm for this method is as follows:
algorithm convert(infixQueue : ZHQueue<Token>) : ZHQueue<Token>
if infixQueue is empty then
    throw a new IllegalArgumentException with the message
    "Empty infix expression"
end if
set postfixQueue ← a new empty ZHArrayQueue<Token>
set opStack ← a new empty ZHArrayStack<Token>
while infixQueue is not empty do
    dequeue nextTok from infixQueue
    if nextTok is an operand (i.e. nextTok instanceof Operand) then
        enqueue nextTok to postfixQueue
    else if nextTok is a left parenthesis then
        push nextTok onto opStack
    else if nextTok is a right parenthesis then
        call processRightParenthesis
    else [ nextTok is an operator ]
        call processOperator
    end if
end while
while opStack is not empty do
    pop nextOp from opStack
    if nextOp is a left parenthesis then
        throw a new IllegalArgumentException with the message
        "Unmatched left parenthesis"
    end if
    enqueue nextOp to postfixQueue
end while
return postfixQueue

To make the code for the convert method easier to implement and understand, we broke out two parts of the algorithm into separate sub-algorithms: processRightParenthesis and processOperator. We call these methods are called when we encounter a right parenthesis or an operator in the infix expression queue, respectively. Since they only make sense when in the context of the convert method, we make them private methods.
When we encounter a right parenthesis, all we have to do is pop all the operands from \textit{opStack} and enqueue them into \textit{postfixQueue} until we find the matching left parenthesis on the stack; if we reach the bottom of the stack before finding a matching left parenthesis, the right parenthesis must be unmatched and we throw an exception. The algorithm for the \textit{processRightParenthesis} method is as follows:

\begin{algorithm}
\textbf{processRightParenthesis}(opStack : ZHStack<Token>, postfixQueue : ZHQueue<Token>) : void
\begin{algorithmic}
\State \textbf{while} opStack is not empty \textbf{and} top of opStack is not a left parenthesis \textbf{do}
\State \hspace{1em} pop an element from opStack and enqueue it to postfixQueue
\EndWhile
\If{opStack is empty}
\State throw an IllegalArgumentException with the message \textit{"Unmatched right parenthesis"}
\EndIf
\Endalgorithmic
\end{algorithm}

When we encounter an operator, we need to push it onto \textit{opStack}. First, however, we need to check any operators already on the stack; as long as there is an operator on the top of the stack with greater or equal precedence to the new operator, we need to pop that operator and enqueue it to \textit{postfixQueue}. Once we reach the bottom of the stack or find a left parenthesis on the top of the stack, we can push the new operator and return. The algorithm for the \textit{processOperator} method is below:

\begin{algorithm}
\textbf{processOperator}(opStack : ZHStack<Token>, postfixQueue : ZHQueue<Token>, op : Operator) : void
\begin{algorithmic}
\If{opStack is not empty}
\State set topTok \leftarrow top element of opStack \texttt{[} peek without popping \texttt{]}
\EndIf
\While{opStack is not empty \textbf{and} topTok is not a left parenthesis \textbf{and} precedence(op) \leq precedence(topTok)}
\State pop element from opStack and enqueue it to postfixQueue
\If{opStack is not empty}
\State topTok \leftarrow top element of opStack \texttt{[} again, peek without popping \texttt{]}
\EndIf
\EndWhile
\Endalgorithmic
\end{algorithm}

When you are done implementing this algorithm, use the provided \texttt{InfixPostfixTest.java} test program (in your lab folder) to test your \texttt{InfixToPostfix} program. Additionally, the \texttt{InfixPostfixDriver.java} program tests the conversion; it requests the user to input a string representing an infix arithmetic expression, calls \texttt{Tokenizer.parseString} on the input, calls \texttt{InfixToPostfix.convert} on the tokenizer output queue, and then displays the resulting postfix queue using the iterator. You may use this program to test your \texttt{InfixToPostfix} class more thoroughly. Verify that it correctly converts well-formed infix expressions to the equivalent postfix expression and that it throws an exception on mismatched parentheses. You
should predict your expected outcome for each test and then check whether you got the expected result.
Part 4: Evaluating a postfix expression

In this part, you'll implement a class with a static method that takes a queue of tokens in postfix order and returns an int that is the value of the expression. You'll then use a JUnit test class and a driver program to test the evaluation.

Create a PostfixEvaluator class inside your expressions package in your JavaPackages folder with the following method:

```java
public static int evaluate(ZHQueue<Token> postfixQueue);
```

This evaluate method works by processing tokens from the postfixQueue, again one-by-one, in this case pushing operands (not operators) onto a stack. Whenever we encounter an operator, there should be two operands for that operator already on the stack. We pop the two operands (note that the right operand would pop before the left operand because we're using a stack), perform the operation, and push the result onto the stack. When the whole queue is processed, the stack should contain a single IntegerLiteral representing the value of the expression. The algorithm for evaluate is as follows:

```
algorithm evaluate(postfixQueue : ZHQueue<Token>) : int
if postfixQueue is empty then
    throw an IllegalArgumentException with the message
    "Empty postfix expression"
end if
set opStack ← an empty ZHArrayStack<Operand>
while postfixQueue is not empty do
    dequeue nextTok from postfixQueue
    if nextTok is an operand then
        push nextTok onto opStack
    else [ nextTok is an operator ]
        if opStack is empty then
            throw an IllegalArgumentException with the message
            "Operator with no operands"
        end if
        pop rightOperand from opStack
        if opStack is empty then
            throw an IllegalArgumentException with the message
            "Operator with only one operand"
        end if
        pop leftOperand from opStack
        call evaluate(leftOperand, rightOperand) on the operator nextTok
        push the result back onto opStack
    end if
end while
pop result from opStack
if opStack is not empty then
    throw an IllegalArgumentException with the message
    "Too many operands"
end if
return result
```
To test your work, use the PostfixEvaluatorTest.java test program. Again, you should also use the provided InfixPostfixDriver.java program to test your code more thoroughly. The driver program calls the PostfixEvaluator.evaluate method and displays the result but first you need to uncomment the last two lines in the driver program. You should keep the display of the postfix expressions for completeness and to aid in error checking. Test your program on a wide variety of infix expressions (both well-formed and ill-formed) to verify that it correctly evaluates the expressions.