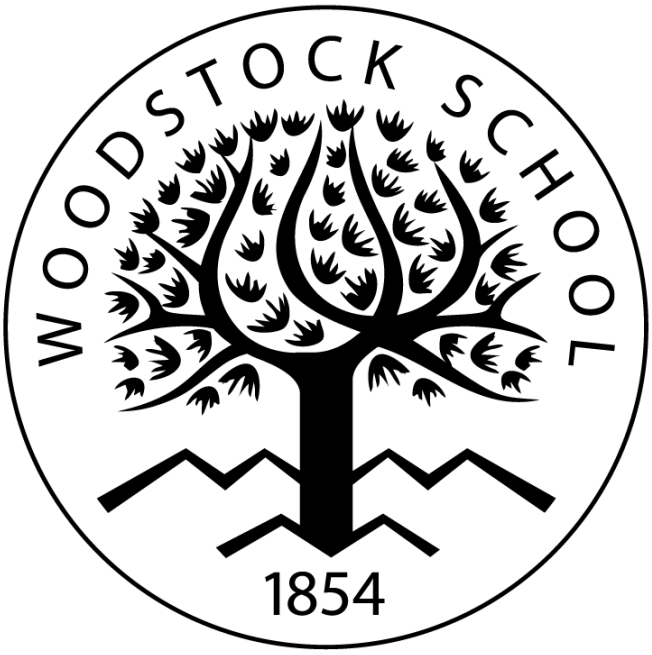
Calculator Lab

AP Computer Science A



Name: _____

Contents

Background

Modern calculators and even many search engines have become adept at reading strings of mathematical symbols and being able to calculate the value of the given expression. Type into a Google search bar the expression: `3 + 4 * 8 - 16 / 32` and Google will correctly return the value `34.5`. In this lab, you will be implementing methods that will take in strings of characters, like the example above, and return the correctly calculated value of the expression.

Before we begin, we first need to explore a number of different ways that have been developed to write a given mathematical expression. Each method differs in the abilities of humans and computers to read and process them. In fact, our tasks will primarily hinge on teaching a computer to read and process the most "computer-readable" version and then convert the others to that version.

Note that for the purposes of this lab, only the four basic, binary arithmetic operators ($+$, $-$, $\times$, $\div$) are going to be considered; however, all of the concepts explored here can be extended to handle more complex mathematical expressions and operations.

Three Different Notations

Infix Notation

Infix notation is the notation you are likely familiar with. When using infix notation, the binary operators are placed *between* the numbers to be operated on (called its *operands*). Here are a few examples:

$$5 + 7 \qquad 12 + 6 - 3 \qquad 3 + 4 \times 8 - 16 \div 32$$

Note that the order of operations (*PEMDAS*/*BODMAS*) dictates how complex expressions are evaluated.

Prefix Notation/Polish Notation

First described by Polish logician Jan Łukasiewicz in 1924, *prefix notation* places each operator *before* its operands. Here are the same examples as above, this time in prefix notation:

$$+\,5\,7 \qquad -\,+\,12\,6\,3 \qquad -\,+\,3\,\times\,4\,8\,\div\,16\,32$$

Prefix notation has the benefit of allowing for complex expressions to follow an explicit order of operations *without parenthesis or brackets*. A properly written expression can be evaluated from left-to-right. In this way, the order of operations dictates how an expression is written, rather than how it is evaluated.

Note that for non-commutative operations ($-$ or $\div$), the order of the operands follows their order in the written expression. That is: $\div\ 16\ 32$ will always evaluate to 0.5 and never to 2.

Postfix Notation/Reverse-Polish Notation

Although prefix notation was first described in the 1920's, *postfix notation* was only introduced in the 1950's. It was later described by famed Edsger Dijkstra as a way of evaluating mathematical expressions requiring fewer calls to computer memory. Unsurprisingly, prefix notation places each operator *after* its operands. Here, again, are the same exmaples as above, this time in postfix notation:

$$5\,7\,+ \qquad 12\,6\,+\,3\,- \qquad 3\,4\,8\,\times\,+\,16\,32\,\div\,-$$

As with prefix notation, the order of the operands for non-commutative operations follows their order in the written expression. Thus, $16\ 32\ \div$ evaluates to 0.5.

Although infix notation is most likely easiest for us to read, it is actually fairly difficult for a computer to process correctly, particularly once additional operators and symbols, such as parenthesis or brackets, are added into it. Because of this, Activity #1 will ask you to first implement a postfix notation calculator.

Applications

Question #1: Although parenthesis are not explicitly required by prefix or postfix notation, they do help for human-readability, and so that you can better understand how these expressions are evaluated. Consider the following:

$$- + 12\,6\,3 \rightarrow -(+12\,6)\,3$$

The parenthesis make it more clear that the $+$ operator uses the two immediate operands that follow it. It is then clearer that the $-$ operator has operands: $(+\,12\,6)$ and 3, or 18 and 3.

Add parenthesis to each of the following prefix or postfix expressions.

$+ \times 3\,2 - 6\,4$ $+\,5\,7 - 3 \div 10\,5$ $7\,4 - 6\,5 + \times$ $3\,4\,8 \times + 16\,32 \div -$

Question #2: Evaluate each of the expressions in Question #1.

Question #3: Convert the following infix expression to a prefix and a postfix one.

$$5 \times 3 + 6 + (4 - 2)$$

Prefix: **Postfix:**

Question #4: Why do you think a computer has an easier time handling prefix and postfix notations instead of infix notation?

Activity #1

Introduction

As it was noted in the background, postfix notation has been suggested as desirable for the evaluation of mathematical expressions within a computer system due to lower memory requirements than prefix notation. The reason for this is simple: when processing a string of characters representing a mathematical expression in postfix, once an operator is encountered it can be immediately evaluated. This is because all of its operands will already have been processed and stored in memory.

For example, an expression like this: `2 3 4 + -`, will have the `3` and `4` in memory as soon as the operator, `+` is encountered. The result, `7`, can then be calculated and stored in anticipation of future operators. Compare that to the prefix expression: `- + 3 4 2`. Here, the computer must process and store the `-` operator before it can be evaluated because its operands are not yet in memory, then process and store the `+` operator because its operands, too, are not in memory. Although alternative methods for processing prefix notation have been developed, during the 1960's the benefits of postfix notation were clear.

Because postfix notation has been the dominant form of writing expressions for computer evaluation since then, implementing a method for processing it will be your first exercise in this Activity. You will then process prefix notation, hopefully using some of the techniques learned while working on the `evalPostfix()` method. Each of these methods will rely heavily on the stack data structure for storing and accessing processed information.

Exercises

1. Implement the `evalPostfix()` method, which will take as a parameter a `String` holding a mathematical expression in postfix notation and return the result of evaluating that expression.
   Note: Remember that, for our purposes, $\times \rightarrow *$ and $\div \rightarrow /$.
   Hint: Use a stack to hold the operands.

2. Implement the `evalPrefix()` method, which will take as a parameter a `String` holding a mathematical expression in prefix notation and return the result of evaluating that expression.
   Hint: You may want to consider evaluating the expression backwards (right to left).

Questions

Question #5: Explain how you determined whether or not a given character/characters was an operation or an operand.

Question #6: Why is evaluating an expression in prefix notation backwards easier?

Activity #2

Introduction

In this activity, you will be implementing a method for processing and evaluating expressions in infix notation. Remember, for the purposes of this lab, we are only considering the four basic operators ($+, -, \times, \div$) and no parenthesis or brackets. Even so, infix notation has proven so challenging for computers to process that it is normally converted to either prefix or postfix notation first, then processed using evaluation methods for the given notation.

The Shunting-Yard Algorithm

In addition to encouraging the use of postfix notation, Edsger Dijkstra also invented an algorithm for converting from infix notation to postfix notation. Here is a simplified version of that algorithm, in order to handle only the operators selected for this exercise. Note that the term *token* is used to represent a small part of the expression being processed which could be either an operator or operand.

1. Create a postfix string and operator stack.
2. While there are tokens to read:
   (a) Read the next token.
   (b) If the token is a number, append it to the postfix string.
   (c) If the token is an operator:
      i. While there is an operator on top of the operator stack:
         A. If the current token has *less* precedence than the operator on top of the stack, pop the operator off the stack and append it to the postfix string.
      ii. Push the token on top of the operator stack.
3. Return the postfix string.

This algorithm was named the "shunting-yard algorithm" due to the way the algorithm rearranges symbols in a similar fashion to how a shunting-yard might rearrange rail cars.

Exercises

1. Implement the `infix2postfix()` method, which will take as a parameter a `String` in infix notation and return a `String` in postfix notation.
   Hint: Use the shunting-yard algorithm!
2. Implement the `evalInfix()` method, which will take as a parameter a `String` holding a mathematical expression in infix notation and return the result of evaluating that expression.

Questions

Question #7: Why do you think Dijkstra offered an algorithm from converting from infix to postfix notation after advocating for postfix notation's use in computing?

Question #8: Explain how you might evaluate an expression in infix notation without first converting it to postfix notation. Do you believe the conversion process is more or less costly than the method you developed? Explain why.

Final Analysis

Question #9: Why is it important for computer scientists and programmers to consider methods for expressing information that differs from the way humans might express it?

Question #10: During the late 1960's, Hewlett Packard began designing and producing lines of engineering and financial calculators that used postfix notation. Even today, Hewlett Packard offers a (diminished) line of calculators using postfix notation. Why do you think Hewlett Packard continues to produce a few models using this notation? Why do you think few other calculators are produced using postfix notation?

Question #11: What part of the implementation of `evalPostfix()`, `evalPrefix()`, or `evalInfix()` did you find most challenging? How did you overcome this challenge?

Question #12: What new programming techniques or knowledge did you learn as a result of this lab?

Template Class & Test Cases

```
/**
 * Calculator Lab (Template Class and Test Cases)
 * This is the template class and test cases for the Calculator Lab.
 * Written for the Woodstock School in Mussoorie, Uttarakhand, India.
 *
 * @author Jeffrey Santos
 * @version 1.0
 */

public class Calculator {
  public static void main(String[] args) {
    //  Test cases for evalPostfix:
    System.out.println(evalPostfix("3 4 10 + -"));          // Output: -11
    System.out.println(evalPostfix("5 6 * 2 8 - +"));       // Output: 24
    System.out.println(evalPostfix("10 2 / 4 5 * 3 + +"));  // Output: 28

    //  Test cases for evalPrefix:
    System.out.println(evalPrefix("* + 2 5 - 3 7"));        // Output: -28
    System.out.println(evalPrefix("- - - 4 5 8 3"));        // Output: -12
    System.out.println(evalPrefix("+ * 2 5 / 6 2"));        // Output: 13

    //  Test cases for infix2postfix:
    //   Output: 2 3 4 * + 8 -
    System.out.println(infix2postfix("2 + 3 * 4 - 8"));
    //   Output: 3 7 - 8 2 / +
    System.out.println(infix2postfix("3 - 7 + 8 / 2"));
    //   Output: 2 10 * 5 3 * 2 / +
    System.out.println(infix2postfix("2 * 10 + 5 * 3 / 2"));

    //  Test cases for evalInfix:
    System.out.println(evalInfix("2 + 3 * 4 - 8"));       // Output: 6
    System.out.println(evalInfix("3 - 7 + 8 / 2"));       // Output: 0
    System.out.println(evalInfix("2 * 10 + 5 * 3 / 2"));  // Output: 27.5
  }

  /**
   * Evaluates a given postfix notation expression and returns the result.
   *
   * @param expression
   *  Preconditions:  expression is well-formed in postfix notation
   *                  each token is separated by a single blank space
   * @return  The value of the expression after evaluation.
   */
  public static double evalPostfix(String expression) {
    //  To be implemented in Activity #1, Exercise 1
  }

  /**
   * Evaluates a given prefix notation expression and returns the result.
   *
   * @param expression
   *  Precondition: expression is well-formed in prefix notation
   *                each token is separated by a single blank space
   * @return  The value of the expression after evaluation.
   */
  public static double evalPrefix(String expression) {
    //  To be implemented in Activity #1, Exercise 2
  }

  /**
   * Converts a string from infix notation to postfix notation.
   *
   * @param expression
```

```
    *  Precondition: expression is well-formed in infix notation
    *                each token is separated by a single blank space
    * @return  An equivalent expression in postfix notation.
    */
   public static String infix2postfix(String expression) {
     //  To be implemented in Activity #2, Exercise 1
   }

   /**
    * Evaluates a given infix notation expression and returns the result.
    *
    * @param expression
    *  Precondition: expression is well-formed in infix notation
    *                each token is separated by a single blank space
    * @return  The value of the expression after evaluation.
    */
   public static double evalInfix(String expression) {
     //  To be implemented in Activity #2, Exercise 2
   }
}
```