PAC I - Recitation

Wednesday, Nov. 29, 2017
Today we’ll cover...

- Review Stacks
- Introduce Lab10
  - Postfix vs Infix
  - Converting to Postfix
  - Evaluating Postfix
Stacks

- Stacks are a data-structure that hold data in a “last-in, first-out” (LIFO) fashion
- Imagine a pile of papers on your desk
  - you “push” one on top of the pile and then “pop” the top paper off of your pile. You can only directly access that top element at any given time.
Stacks

public interface Stack<E> {

    /**
     * Returns the number of elements in the stack.
     * @return number of elements in the stack
     */
    int size();

    /**
     * Tests whether the stack is empty.
     * @return true if the stack is empty, false otherwise
     */
    boolean isEmpty();

    /**
     * Inserts an element at the top of the stack.
     * @param e   the element to be inserted
     */
    void push(E e);

    /**
     * Returns, but does not remove, the element at the
     * top of the stack.
     * @return top element in the stack (or null if empty)
     */
    E top();

    /**
     * Removes and returns the top element from the
     * stack.
     * @return element removed (or null if empty)
     */
    E pop();
}

Data Structures and Algorithms in Java, Sixth Edition
Goodrich, Tamaassia, Goldwasser
Postfix Notation (aka Reverse Polish Notation)

- A mathematical notation where operators follow their operands:
  o  3 4 +

- The notation we’re all used to seeing is called “infix notation”:
  o  3 + 4

- Similarly, there is “prefix notation” where operators precede their operands:
  o  + 3 4
Postfix Notation (aka Reverse Polish Notation)

- The name “Reverse Polish Notation” gets its name from its creator, Polish logician Jan Łukasiewicz.
- The notation was independently created by others, most notably the Dutch computer scientist, Edsger Dijkstra, whose “shunting yard algorithm” will be utilized for converting infix expressions into postfix notation.
Dijkstra’s “Shunting Yard Algorithm”

while there are tokens to be read:
    read a token
    if the token is a operand, append it to the output string
    elif the token is an operator, then:
        while there is an operator at the top of the stack with >= precedence:
            pop operators from the operator stack, append to output string
            push the read operator onto the stack
    elif the token is a left parenthesis (i.e. "("), then:
        push it onto the stack.
    elif the token is a right parenthesis (i.e. ")"), then:
        while the operator at the top of the stack is not a left paren:
            pop operators from the stack, append to output string.
            pop the left paren from the stack.
        /* if the stack runs out without finding a left paren, then there are mismatched parentheses. */
    if there are no more tokens to read:
        while there are still operators on the stack:
            /* if operator on top of stack is a paren, then there are mismatched parentheses. */
            pop the operator; append to output string.
exit

Converting Infix to Postfix

Infix expression: \((4+8)\times(6-5)\)

1) push ( on the stack
Converting Infix to Postfix

Infix expression: \((4+8)*(6-5)\)

1) push ( on the stack

2) append 4 to output string

STACK

OUTPUT STRING: 4
Converting Infix to Postfix

Infix expression: \((4+8)*(6-5)\)

1) push ( on the stack
2) append 4 to output string
3) push + on the stack

OUTPUT STRING: 4
Converting Infix to Postfix

Infix expression: \((4+8)\times(6-5)\)

1) push ( on the stack
2) append 4 to output string
3) push + on the stack
4) append 8 to output string

OUTPUT STRING: 4 8
Converting Infix to Postfix

Infix expression: \((4+8)\times6\)

1) push ( on the stack
2) append 4 to output string
3) push + on the stack
4) append 8 to output string
5) found right paren; pop from stack until left paren is found; append each operator to output string

STACK

OUTPUT STRING: 4 8 +
Converting Infix to Postfix

Infix expression: \((4+8)\times 6\)

1) push ( on the stack

2) append 4 to output string

3) push + on the stack

4) append 8 to output string

5) found right paren; pop from stack until left paren is found; append each operator to output string

6) push * to the stack

```
OUTPUT STRING: 4 8 +
```
Converting Infix to Postfix

Infix expression: (4+8) * 6

1) push ( on the stack
2) append 4 to output string
3) push + on the stack
4) append 8 to output string
5) found right paren; pop from stack until left paren is found; append each operator to output string
6) push * to the stack
7) append 6 to output string

OUTPUT STRING: 4 8 + 6

STACK
Converting Infix to Postfix

Infix expression: \((4+8) \times 6\)

1) push \( ( \) on the stack
2) append 4 to output string
3) push + on the stack
4) append 8 to output string
5) found right paren; pop from stack until left paren is found; append each operator to output string
6) push \( \times \) to the stack
7) append 6 to output string
8) no more tokens remain; pop from stack until empty

OUTPUT STRING: \( 4 \ 8 \ + \ 6 \ \times \)
Evaluating a PostFix Expression

The Algorithm:

while there are tokens in the postfix expression:
   if token is an operand:
      push it to the stack

   if token is an operator:
      op1 = pop operand from the stack
      op2 = pop operand from the stack
      result = evaluate(op1, op2, operator)
      push result back to the stack

pop final result from stack
Evaluating a PostFix Expression

This time we’ll push operands to the stack instead of operators.

Given the postfix expression: 3 4 5 + *

1) Push 3 to the stack

3
STACK
Evaluating a PostFix Expression

This time we’ll push operands to the stack instead of operators.

Given the postfix expression: 3 4 5 + *

1) Push 3 to the stack

2) Push 4 to the stack
Evaluating a PostFix Expression

This time we’ll push operands to the stack instead of operators.

Given the postfix expression: 3 4 5 + *

1) Push 3 to the stack
2) Push 4 to the stack
3) Push 5 to the stack

STACK

5
4
3
Evaluating a PostFix Expression

This time we’ll push operands to the stack instead of operators.

Given the postfix expression: $3 \ 4 \ 5 \ + \ *$

1) Push 3 to the stack
2) Push 4 to the stack
3) Push 5 to the stack
4) Encountered operator, pop top two operands and perform operation: $4 + 5 = 9$ Push 9 to the stack
Evaluating a PostFix Expression

This time we’ll push operands to the stack instead of operators.

Given the postfix expression: 3 4 5 + *

1) Push 3 to the stack
2) Push 4 to the stack
3) Push 5 to the stack
4) Encountered operator, pop top two operands and perform operation: 4 + 5 = 9 Push 9 to the stack
5) Encountered operator, pop top two operands and perform operation: 3 * 9 = 27 Push 27 to the stack

STACK

27
Evaluating a PostFix Expression

This time we’ll push operands to the stack instead of operators.

Given the postfix expression: 3 4 5 + *

1) Push 3 to the stack

2) Push 4 to the stack

3) Push 5 to the stack

4) Encountered operator, pop top two operands and perform operation: $4 + 5 = 9$ Push 9 to the stack

5) Encountered operator, pop top two operands and perform operation: $3 \times 9 = 27$ Push 27 to the stack

6) No more characters in string, the final result is at the top of the stack

STACK

27
Lab10 Postfix Calculator

- Create a Converter and a Calculator class that will take infix expressions as input, convert them to postfix, evaluate the postfix expression and the print the result (see assignment spec for further details)

- Use a Linked or Array Stack for implementing the algorithm. Try to write this yourself rather than copy the provided examples!

- You will be provided with some code to aid in the parsing of expressions to tokenize operators and operands

- Due Friday, 12/15 (two days after the final)