CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #8

John Ridgway

February 24, 2015

1 Postfix Arithmetic

Arithmetic Notations

- You are used to infix notation for arithmetic expressions, e.g., $2 \times (3 + 7)$.
- Alternative notations exist; we could use a functional-style notation: $\times(2, +(3, 7))$, where we put the operators in as functions.
- But we don't really need the parentheses.
- In $\times$ 2 + 3 7, we see the $\times$ and go looking for its two operands; we immediately find one, 2, but then we see the +, so we know that we must compute the sub-expression beginning with the + before doing the multiplication.
- This is called *prefix* notation, and is also known as *polish* notation.

Postfix Notation

- Another possibility is *postfix* notation, also referred to as *reverse polish* notation. In postfix notation we write the operations last, so $2 \times (3 + 7)$ would be written as 2 3 7 + $\times$.
- A single operand (usually a number) is a legal postfix expression.
- You can make a legal postfix expression by stringing together (usually two) postfix expressions and an operator that takes that many expressions. Some examples: 5, 7, 5 7 +, 5 7 $\times$, 5 7 4 + $\times$.

Postfix Arithmetic Question 1

What is the postfix form of the infix expression $(7 \times 6) - 42$?

A. $7\ \times 6\ 42-$

B. $7\ 6\ \times\ 42\ -$

C. $\times\ 7\ 6\ 42\ -$

D. $7\ 6\ 42\ \times\ -$

Evaluating Postfix Expressions

- The rules for evaluating postfix expressions are very simple:
- Scan from left to right until you find an operator. Replace the operator and the immediately preceding operands with the result of applying the operator to the operands.
- Repeat the above until there is only one item left; that is the answer.

Converting Postfix to Infix

- We can use almost the same rules for converting from postfix to infix notation.
- Scan from left to right until you find an operator. Replace the operator and the immediately preceding operands with a string consisting of a left-parenthesis, the first operand (which may be a string), the operator, the second operand, and a right parenthesis.

Postfix Arithmetic Question

What is the infix form of the postfix expression $3\ 5\ +\ 7\ \times\ 4\ 2\ +\ \times$?

A. $(3 + 5) \times (7 + (4 \times 2))$

B. $3 \times (5 + 7) \times (4 + 2)$

C. $(((3 + 5) \times 7) \times 4) + 2$

D. $((3 + 5) \times 7) \times (4 + 2)$

The Stack Evaluation Algorithm

- Alternatively, we can evaluate postfix expressions using a stack, whose entries each hold one of our numerical values (an operand or the result of an operator applied to operands).
- When we see an operand, we push it onto the stack, and when we see an operator, we pop two values off of the stack (remembering them!), apply the operator to them, and push the result back onto the stack.
- At the end, if we have exactly one value on the stack, that is our answer.

Evaluating With a Stack

- If we ever empty the stack, or if we finish with more than one value on it, the postfix expression was not valid. Thus our algorithm also tests a candidate postfix expression for validity.
- In the example 3 4 5 + 2 × 3 4 × − + we push 3, push 4, push 5, pop 4 and 5 and add them to make 9, push 2, pop 9 and 2 and multiply them to make 18, push 3, push 4, pop 3 and 4 and multiply them to make 12, pop 18 and 12 and subtract 12 from 18 to make 6, and finally pop 3 and 6 and add them to make 9.

Error Conditions

- Just like our parenthesized expressions, postfix expressions can go wrong in two ways, by having too many operators too quickly or by having not enough operators when the expression ends.
- In the first case the stack will be emptied, and in the second case more than one value will be left on the stack at the end. DJW use a bounded stack, which adds the additional risk of stack overflow.
- We'll define a new class of exceptions called PostfixException to represent both of these conditions, counting on our error messages to tell the user what went wrong.
- We can choose to guard against any stack exceptions, or to enclose them in try blocks and catch them. DJW choose the latter.

Start of PostfixEvaluator

```
public class PostfixEvaluator {
  public static int evaluate(String expr) {
  BSI<Integer> stack=new AS<Integer>(50);
  int value, operand1, operand2;
  int result = 0;
  String op;
  Scanner tokenizer = new Scanner(expr);
```

Body of PostfixEvaluator

```
while (tokenizer.hasNext()) {
   if (tokenizer.hasNextInt()) {
      value = tokenizer.nextInt();
      if (stack.isFull()) {
        throw new PE("stack overflow"); }
      stack.push(value);}
   else {op = tokenizer.next();
      operand2 = stack.top(); stack.pop();
      operand1 = stack.top(); stack.pop();
      // result = result of op on
      // operand1 and operand2
      stack.push(result); } }
```

End of PostfixEvaluator

```
if (stack.isEmpty()) {
  throw new PE("stack underflow"); }
result = stack.top(); stack.pop();
if (!stack.isEmpty()) {
  throw new PE("too many operands"); }
return result; } }
```

The PFixConsole Class

- DJW have a class that takes and evaluates expressions from the console, catching and reporting errors instead of crashing.

Toward Infix Evaluation

- Building an evaluator for ordinary infix expressions would require us to deal with both parentheses and the hierarchy of operations.
- The usual method to handle infix is to actually translate the infix expression into the correct equivalent postfix expression, and use a stack-based evaluator to handle the result.
- This translation (see, for example, `http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix`), uses a second stack to hold operators that are not yet ready to appear in the postfix translation.

Toward Infix Evaluation (continued)

- Operands may be output as soon as they are seen. Any operator must be held until its second argument has been output, but some must be held longer due to precedence.
- The case without parentheses is simpler.
- If we see an operator with lower precedence than the one on the top of the stack, we output it. Otherwise we push it.
- We empty the stack to the output after we have read all the characters.
- Handling parentheses as well is more complicated but can be done using the same operator stack.