40

CS61B: Lecture 40 Friday, May 1, 2009

Expression Parsing (an application of stacks)

Arithmetic expressions can be written in prefix, infix, or postfix. Infix is the usual way of writing expressions that we're all familiar with: 3+4*7. You're also familiar with prefix notation from Scheme; the same expression is written +3*4*7. In postfix, it's written 34*7*+1.

In postfix, we write an operator after its operands: 4.7 *. Each operand may itself be any postfix expression. Hence, 1.2 + 3.4 + * is an expression whose operands are 1.2 + 3.4 + * is an expression whose operands are 1.2 + 3.4 + *. However, unlike in infix, parentheses are never necessary.

A postfix expression is easily evaluated by maintaining a stack of numbers. We read a postfix expression from left to right. When we encounter a number, we push it onto the stack. When we encounter an operator, we pop the top two numbers off the stack, perform the operation on them (using the first number popped as the $_$ second $_$ operand), and push the result onto the stack. Let's compute 2 7 5 - * 8 5 - *.

Here's code that performs one such operation. For brevity, the division operator and all error checking are omitted.

```
public static void operate(Stack numStack, char operator) throws
                                                        StackEmptyException {
 // Retrieve the operands from the stack.
 int operand2 = ((Integer) numStack.pop()).intValue();
 int operand1 = ((Integer) numStack.pop()).intValue();
 System.out.print(operator + " ");
 // Perform the operation and push the result on the number stack.
 switch (operator) {
 case '+':
                                                                 // Addition.
   numStack.push(new Integer(operand1 + operand2));
   break;
 case '-':
                                                               // Subtraction.
   numStack.push(new Integer(operand1 - operand2));
   break;
 case '*':
                                                           // Multiplication.
   numStack.push(new Integer(operand1 * operand2));
   break;
 case '^':
                                                           // Exponentiation.
   int result = 1;
   for (int i = 0; i < operand2; i++) {
     result = result * operand1;
   numStack.push(new Integer(result));
   break;
```

```
Here's code that reads a postfix expression (written on the command line).
prints it, evaluates it, and prints the result. For instance, if you type
  java ParsePostfix '73*2+'
The program responds
 7 3 * 2 +
              result:23
public class ParsePostfix {
 public static void main(String[] args) throws StackEmptyException {
   LinkedStack numStack = new LinkedStack();
                                                     // Create a number stack.
    // Process each character of the first command-line argument.
    for (int i = 0; i < args[0].length(); i++) {
     char c = args[0].charAt(i);
     if (Character.isDigit(c)) {
        // Print each digit; push each digit on number stack.
        System.out.print(c + " ");
       numStack.push(new Integer(Character.digit(c, 36)));
     } else if ((c == '+') || (c == '-') || (c == '*') || (c == '^')) {
        operate(numStack, c);
    System.out.println(" result:" + numStack.top());
 // Put the operate() method code here.
```

Converting Infix to Postfix

Infix expressions are more difficult to evaluate than postfix expressions, largely because of precedence rules. Exponentiation (^) has precedence over multiplication (*) and division (/), which have precendence over addition (+) and subtraction (-). Another complication is that, if two operators have the same precedence, they may be left-associative or right-associative. Addition, subtraction, multiplication, and division are left-associative: 2 - 3 - 5 = (2 - 3) - 5. Exponentiation is right-associative: $2 ^3 5 = 2$

We can convert infix to postfix with a simple algorithm that uses a stack to hold operators. The algorithm reads an expression from left to right.

- When we encounter a number, we immediately print it out.
- When we encounter an operator, we put it on a stack until an operator of lower or equal precedence appears (for exponentiation, only if it's lower), whereupon we pop it and print it.

When the string ends, we pop and print each operator left on the stack.

Why does this work? Consider three cases:

- If the infix expression is "3 * 2", the algorithm passes through the "3" and "2" while putting the "*" on the stack. The input ends at the "2", then we output the "*" from the stack, writing "3 2 *"
- Suppose the newest operator has lower precedence than the operator atop the stack; e.g., the string so far is "3 * 2 +". The operands of "*" are "3" and "2" regardless of what comes after the "+", so we write out the string "3 2 *", push the "+", and continue.
- Suppose the newest operator has higher precedence than the one atop the stack; e.g., the string so far is "3 * 2 ^". "3" is one of the operands of "*". The second operand begins with "2 ^", but is incomplete. We must keep the "*" on the stack until its second operand is complete.

Upon reaching the expression's end, we pop and print each remaining operator. Here's how we convert infix "3 * 5 + 4 ^ 5 ^ 6" to postfix "3 5 * 4 5 6 ^ ^ +".

```
The following main() method converts an infix expression to postfix and
simultaneously evaluates the postfix expression. It prints both the converted
postfix expression and the computed result. The algorithm uses two stacks:
one for numbers, and one for operators. (This code's shocking neglect of error
checking is not a good model for your own efforts. The reading of input is
also poor; only one-digit numbers can be input.)
public class ParseInfix {
 public static void main(String[] args) throws StackEmptyException {
   LinkedStack numStack = new LinkedStack();
                                                   // Create a number stack.
   LinkedStack opStack = new LinkedStack();
                                                   // Create an operand stack.
   // Process each character of the first command-line argument.
   for (int i = 0; i < args[0].length(); i++) {
     char c = args[0].charAt(i);
     if (Character.isDigit(c)) {
       // Print each digit; push each digit on number stack.
        System.out.print(c + " ");
       numStack.push(new Integer(Character.digit(c, 36)));
     } else if ((c == '+') || (c == '-') || (c == '*') || (c == '^')) {
       // Before pushing an operator onto the operand stack, check if
       // operators currently atop the stack have greater precedence.
       while (!opStack.isEmpty() && (c != '^') &&
              (precedence(((Character) opStack.top()).charValue()) >=
               precedence(c))) {
         // Perform the operation atop the operand stack.
         char operator = ((Character) opStack.pop()).charValue();
         operate(numStack, operator);
        opStack.push(new Character(c)); // Push new operation on operand stack.
   while (!opStack.isEmpty()) {
     char operator = ((Character) opStack.pop()).charValue();
     operate(numStack, operator);
   System.out.println(" result:" + numStack.top());
The algorithm requires us to assign each operator a number that indicates its
relative precedence. We could easily add more operators later.
 public static int precedence(char operator) {
   switch (operator) {
   case '^':
    return 3;
   case '*':
   case '/':
    return 2;
   case '+':
   case '-':
    return 1;
   default:
     return 0;
 // Put the operate() method code here.
```