CS 228: Introduction to Data Structures
Lecture 24
Friday, October 21, 2016

Arithmetic Expressions

Infix notation is the notation for arithmetic and logical formula to which you are accustomed. In it, operators are written between the operands they act on; e.g., "2 + 2". Parentheses surrounding groups of operands and operators are used to indicate the intended order in which operations are to be performed. In the absence of parentheses, ***precedence rules*** determine the order of operations.

Example. The infix expression "3−4∗5" is evaluated as "3−(4∗5)", not as "(3−4)∗5", because "−" has lower precedence than "∗". If you want it to evaluate the second way, you need to parenthesize.

In ***postfix notation***, also known as ***reverse Polish notation***[1] (RPN), the operators follow their operands. For instance, to add three and four, one would write "3 4 +" rather than "3 + 4". If there are multiple operations, the operator is given immediately after its second operand; so

[1] Called "Polish" in reference to the nationality of logician Jan Łukasiewicz (1878–1956), who invented *prefix* notation.

the expression written "3 − 4 + 5" in infix notation would be written "3 4 − 5 +" in RPN: first subtract 4 from 3, then add 5 to that.

RPN obviates the need for the parentheses that are required by infix. While "3 − 4 ∗ 5" can also be written "3 − (4 ∗ 5)", that means something quite different from "(3 − 4) ∗ 5". In postfix, the former is written "3 4 5 ∗ −", which unambiguously means "3 (4 5 ∗) −", and which reduces to "3 20 −"; the latter is written "3 4 − 5 ∗", which unambiguously means "(3 4 −) 5 ∗". Postfix notation is easier to parse by computer than infix notation, but many programming languages use infix due to its familiarity.

Evaluating Postfix Expressions

Stacks and postfix expressions go hand-in-hand. To evaluate a postfix expression, start with an empty stack and scan the expression from left to right.

- If the next item is a number, push it onto the stack.
- If the next item is an operator, *Op*, do the following:
  - Pop two items *right* and *left* off the stack.

- Evaluate (*left Op right*) and push the value back onto the stack.

If there are fewer than two operands on the stack when scanning an operator, the expression must be invalid: there are more operators than operands. Now, suppose we have reached the end of the expression. If all went well (that is, if the expression is well-formed), then there is a single value in the stack: this must be the value of the expression. Otherwise, more than one operand remains on the stack. This means that the expression was invalid: it had too many operands.

Exercise. Try the above algorithm out on

```
7 11 − 2 ∗ 3 +
```

Note that, in infix, this expression would be written as

```
(7 − 11) ∗ 2 + 3.
```

Comment. As someone pointed out in class today, the latter infix expression is equivalent to the infix expression

```
3 + 2 ∗ (7 − 11)
```

which in postfix is

```
3 2 7 11 − ∗ +
```

Since the work is O(1) per "token" (i.e., operand or operator) in the expression, the total time complexity is O(n), where n is the number of items.

A simple implementation of the preceding algorithm is posted on Blackboard (`PosfixExample.java` and `ExpressionFormatException.java`). It assumes that expressions are space-delimited.

Evaluating Infix Expressions

To get a sense of why evaluating infix expressions is non-trivial, consider the following infix expression.

```
2^7^6 - (3 + 2 * 4) % 5 - 8
```

This expression is evaluated as:

```
(2^(7^6) - ((3 + (2 * 4)) % 5)) - 8
```

The equivalent postfix expression is

```
2 7 6 ^ ^ 3 2 4 * + 5 % - 8 -
```

It is evident from the preceding example that infix expressions are evaluated according to certain conventions:

- Operators have different ***precedences***:

  `( )  >  ^  >  *  =  %  =  /  >  +  =  –`

- Some operators are ***left associative***: `+, –, /, %`

  For example, 3 – 2 – 2 is evaluated as (3 – 2) – 2 (which equals –1), ***not*** as 3 – (2 – 2) (which equals 3).

- One operator is ***right associative***: `^`

  For example, 2 ^ 3 ^ 2 is evaluated as 2 ^ (3 ^ 2) (which equals 512), ***not*** as (2 ^ 3) ^ 2 (which equals 64).

Despite the complications of evaluating infix expressions, they are what most of us are familiar with, so programming languages must be equipped to handle them. One way to do this is to convert infix expressions into equivalent postfix expressions and then call the postfix expression evaluator. We'll see how to perform this conversion next.

Note. It is also possible to evaluate infix expressions within one scan, without converting them to postfix first, using two stacks, but we will not consider that here.

Infix to Postfix Conversion

The next application of stacks is an algorithm to convert an infix expression into its postfix equivalent. We begin with the simpler case where all the operators are left associative and there are no parentheses. Also, for the time being, we ignore the possibility of errors.

The algorithm uses an ***operator stack***, opStack, to temporarily store operators awaiting their right-hand-side operand, and to help manage the order of precedence.

The input infix expression is scanned from left to right. Suppose ρ is the item that is currently being scanned; ρ can be either an operand or an operator.

- If ρ is an operand, append it to the postfix expression.
- If ρ is an operator, we proceed as shown on the next page.

```
while !opStack.isEmpty() &&
        prec(ρ) ≤ prec(opStack.peek())
{
      σ = opStack.pop()
      append σ to postfix
}
opStack.push(ρ)
```

When there are no more items to scan, pop off the operators from the stack, appending them to the postfix as you do so.

Next time, we will give an example of this basic algorithm, and show how to extend it to a larger class of expressions.

Note. Powerpoint slides on postfix expressions and infix-to-postfix conversion are posted on Blackboard.