CSC 220 Data Structures

Lecture 6
Infix and Postfix

Concept of Infix and Postfix

Believe it or not, there are multiple ways to display and evaluate a simple mathematical expression. Take the following expression for example:

```
9 + 4 * 3
```

Each operator (+ and *) takes two operands (4 and 3 for *, the result of that with 9 for +). This is known as "infix notation". The "in" comes from the fact that the operators come "in" between the operands. I'm sure you have seen many expressions like this, but have you ever seen something like this:

```
9 4 3 * +
```

This is actually the same expression! The only difference is that the operators come after the operands. This is known as "postfix notation". The "post" comes from the fact that operators come after operands. What about this expression:

```
+ 9 * 4 3
```

Here the operators come before the operands. This is known as "prefix notation". For this course we will focus solely on infix and postfix notations, but it is interesting to note the existence of a prefix notation.

All of these different notations are just different ways of writing the same expression. In fact, when evaluated, all of them give us the same answer, 21. Two important questions seem to naturally arise from this:

1. Given an expression in infix notation, can we convert to postfix?
2. Given an expression in postfix form, can we evaluate it directly (i.e. without having to convert back into infix)?

The answer to both of these questions, is of course, yes! Let's examine the first question first.

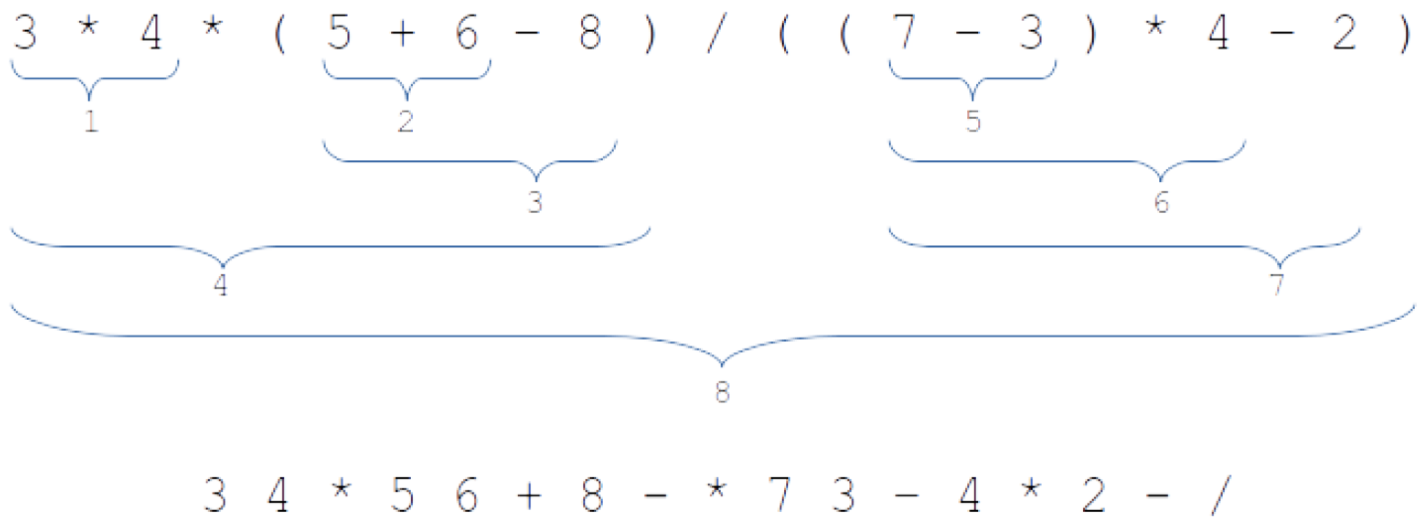
Converting Infix to Postfix

To get a general idea of how to convert, here are some examples of infix expressions and their equivalent postfix form:

Infix	Postfix
`5 + 2 * 9`	`5 2 9 * +`
`(5 + 2) * 9`	`5 2 + 9 *`
`9 * 3 + 2`	`9 3 * 2 +`
`(7 + 5) * 9 + 6`	`7 5 + 9 * 6 +`
`7 / (8 + (3 – 6)) * 4 * 5`	`7 8 3 6 – + / 4 * 5 *`
`3 * 4 * (5 + 6 – 8) / ((7 – 3) * 4 – 2)`	`3 4 * 5 6 + 8 – * 7 3 – 4 * 2 – /`

Note that the order of the operands **is the same** in both the infix form and the equivalent postfix form. One other important aspect to note here as you look at the different expressions, is that, while in infix notation we must define a certain "order of operations", in postfix notation this is actually not necessary. Let's briefly look at how to evaluate these expressions, just to give an overview before we dive in. Take, for instance, our first example in infix form, `5 + 2 * 9`. If we evaluate `5 + 2` first, we would get 7. If we then multiply this by 9, we would get 63. Is this correct? No, because we violated the natural order of operations. You see infix, while natural to us now, is actually quite messy. We have to attach extra rules to make it work. But what if we actually wanted the addition to come first? We must use parentheses, yet another rule to go by! Let's look at the same expression in postfix form, `5 2 9 * +`. The way to evaluate this is to read it left to right, and look for the first operator. We find the first operator after seeing a 5, 2 and 9. Now we apply that operator to the last two operands we saw (2 and 9). This gives us 18. We continue looking at the expression and notice a plus sign. Again we apply it to the last two operands (the 18 from before and the 5 we first saw). This gives us 23. There is only ONE way to evaluate this expression and we didn't need any order of operations nor parentheses. If we want to add first, we rearrange the expression to `5 2 + 9 *`. This would be equivalent to `(5 + 2) * 9`, which is shown in our second example. Even though infix seems natural, postfix is actually easier to deal with once you understand the process.

So, let's take a look at the answer to our first question, how do we convert from infix to postfix. To answer this, let's examine our most complicated expression from the examples above. Once you understand this, converting the other expressions should be simple.



The idea is to group the expression by the order in which it is evaluated. If we were to evaluate this infix form, we would evaluate 1 first (i.e. 3 * 4), then 2 (i.e. 5 + 6), then 3 (i.e. the result of 5 + 6 subtracted by 8), and so on until finally evaluating 8 (the result of the first part divided by the second part). For each part, to convert to postfix, we simply put the operator at the end. This means that 3 * 4 becomes 3 4 *. We then take 5 + 6 and convert it into 5 6 + (adding that to what we already have). The result of this is subtracted by 8, so again the subtraction operator goes at the end and the 8 goes before it. And so on.

It may take a little while to understand this, so here is some practice. Try to fill in the table below.

Infix	Postfix
`2 + 3`	
`5 + 6 – 7`	
`4 * 5 + 6`	
`4 + 5 * 6`	
`(4 + 5) * 6`	
`1 + 2 – (3 * 4) / 5`	
`(8 / 7) / (6 / 5)`	
`5 + (6 – (7 + 8) * 9)`	

Implementing Infix to Postfix Conversion

So, how do we do this conversion in code? How do we convert an infix expression into a postfix? We do it by using a… wait for it… QUEUE! We also need to use our friendly neighborhood STACK as well! Here are the details:

- We will use an infix queue that represents the arithmetic expression in infix form
- We will use a postfix queue to store our postfix expression as we create it from the infix queue
- We will use an operator stack to help with the conversion (i.e. help with rearranging our operators)

- We also need to define an infix priority which prioritizes operators and parentheses
    - This reflects the relative position of an operator in the arithmetic hierarchy
    - This is used to determine how long the operator waits in the operator stack before being enqueued in the postfix queue
    - This is the table we will use for our infix priority:

Token	(	^	*	/	+	-	default
Value	4	3	2	2	1	1	0

- Finally, we need to define a stack priority
    - This determines whether an operator waits in the operator stack or is enqueued into the postfix queue
    - This is the table we will use for our stack priority:

Token	^	*	/	+	-	default
Value	2	2	2	1	1	0

Here is the algorithm:

```
function infixToPostfix(infixString)
   // setting everything up
   postfixQueue ← empty queue
   operatorStack ← empty stack

   infixQueue ← infixString // enqueue each character into infixQueue

   // process the infixQueue
   while infixQueue is not empty
      // grab the next character
      token ← dequeue from infixQueue

      // process operands (i.e. digits)
      if isOperand(token) then
         enqueue token to postfixQueue

      // process parenthesis
      else if token is a right parenthesis then
         operator ← pop from operatorStack
         while operator is not a left parenthesis
            enqueue operator to postfixQueue
            operator ← pop from operatorStack
         end

      // process operators (other than right parenthesis)
      else
         // handle operators already on the operatorStack
         if operatorStack is not empty then
            operator ← peek from operatorStack

            // move operators with high priority into our postfixQueue
            while getStackPriority(operator) >= getInfixPriority(token)
               operator ← pop from operatorStack
               enqueue operator to postfixQueue

               // are there more operators on the operatorStack?
               if operatorStack is not empty then
                  operator ← peek from operatorStack // if so, grab it
               else
                  break // if not, exit out
               end
            end
         end

         // push our new operator
         push token to operatorStack
      end
   end
   // NOTE: this is outside of our while loop above
   // this will unload the operator stack onto our postfixQueue
   while operatorStack is not empty
      operator ← pop from operatorStack
      enqueue operator to postfixQueue
   end

   // transfer the postfixQueue contents into a string
   postfix ← postfixQueue // dequeue each value into a string

   return postfix // return that postfix string
end
```

Implementing Postfix Evaluation

To evaluate a postfix expression, we need to do the following:

```
function evalPostfix(postfixString)
   postfixQueue ← postfixString // enqueue each character into postfixQueue
   evalStack ← empty stack
   finalResult ← nothing

   // process the postfixQueue
   while postfixQueue is not empty
      token ← dequeue from postfixQueue

      // if token is a digit, throw it on the evalStack
      if isOperand(token) then
         push token to evalStack

      // otherwise, evaluate our sub-expression and
      // push the answer onto the evalStack
      else
         a ← pop from evalStack
         b ← pop from evalStack
         answer ← eval(token, b, a)
         push answer to evalStack
      end
   end

   // if our evalStack is not empty after processing everything, then
   // return the final answer stored at the top
   if evalStack is not empty then
      finalResult ← pop from evalStack
      return finalResult

   // otherwise report an error
   else
      return error value
   end
end
```

Let's practice this. Try to evaluate the postfix expressions in the table below:

Postfix	Result
`5 3 2 * +`	
`5 3 2 + *`	
`5 3 + 2 *`	
`6 9 4 - -`	
`1 2 3 4 + - +`	
`1 2 3 + 4 - +`	

Converting Postfix to Infix

Now that you know how to evaluate a postfix expression, you might find this useful for converting postfix back into infix. See if you can figure out how to fill in the table below. Remember that to alter the natural order of operations in infix notation, you must use parentheses.

Infix	Postfix
	`1 2 /`
	`1 2 3 / *`
	`7 8 * 6 +`
	`4 5 + 6 8 * +`
	`5 9 6 / + 4 2 - *`