Applications of Stacks

Based on the notes from David Fernandez-Baca and Steve Kautz

Bryn Mawr College
CS206 Intro to Data Structures

App1: Verifying Matched Parentheses

- Verifying whether an string of parentheses is well-formed.
  - "{[()]{[]}}()" -- well-formed
  - "{[]}[]()" -- not well-formed
- More precisely, a string γ of parentheses is well-formed if either γ is empty or γ has the form

  $(\alpha)\beta$ $[\alpha]\beta$ or $\{\alpha\}\beta$

  where α and β are themselves well-formed strings of parentheses. This kind of recursive definition lends itself naturally to a stack-based algorithm.

Verifying Matched Parentheses (cont.)

- Idea: to check a String like "{[(){[]}](){}", scan it character by character.
  - When you encounter a lefty— '{', '[', or '(' — push it onto the stack.
  - When you encounter a righty, pop its counterpart from atop the stack, and check that they match.
  - If there is a mismatch or exception, or if the stack is not empty when you reach the end of the string, the parentheses are not properly matched.
  - Detailed code is posted separately.

App2: Arithmetic Expressions

Infix notation:

- 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.

E.g., Because "-" has lower precedence than "*", the infix expression "3-4*5" is evaluated as "3-(4*5)", not as "(3-4)*5". If you want it to evaluate the second way, you need to parenthesize.

Arithmetic Expressions (cont.)

Postfix notation (a.k.a. Reverse Polish Notation (RPN))

- Operators follow their operands, e.g., adding three and four is written as "3 4 +" rather than "3+4".
- If there are multiple operations, the operator is given immediately after its second operand; so 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.

Advantage of Postfix Notation

The postfix notation obviates the need for parentheses that are required by infix.

- With infix notation, "3-4*5" can be written as "3-(4*5)", that means something quite different from "(3-4)*5".
- In postfix, the former is written as "3 4 5 * -", which unambiguously means "3 (4 5 *) –", and the latter is written as "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.

App2.1: Evaluating Postfix Expressions

- Interpreters of RPN are often stack-based.
- General idea:
  - Operands are pushed onto a stack, and when an operation is performed, its operands are popped from a stack and its result pushed back on.
  - At the end, the value of the postfix expression is on the top of the stack.
- Since all the needed stack operations take constant time, and the evaluation algorithm is quite simple, RPN expressions can be evaluated quickly and easily.

Evaluating Postfix Expressions (cont.)

We start with an empty stack and scan the postfix 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.
- When we reach the end of the expression, we pop off the single remaining from the stack. This is the value of the expression.

Evaluating Postfix Expressions (cont.)

We can encounter two kinds of errors:

- More than one operand is left on the stack at the end of the scan.
  This means that the expression had too many operands.
- There are fewer than two operands on the stack when scanning an operator.
  This means that the expression has too many operators.

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

Example: Do "7 11 – 2 * 3 + "on the board.

Detailed code is posted separately.

App2.2: Infix to Postfix Conversion

Evaluating infix expressions is non-trivial for several reasons:

- Operators have different precedences:

  ( ) > ^ > * = % = / > + = –
- Some operators are left associative: +, –, /, %
- One operator is right associative: ^

For example, expression 2^7^6 + (3 – 2 * 4) % 5 is evaluated as: 2^(7^6) + ((3 – (2 * 4)) % 5)

Algorithm – the Simple Case

We begin with the simpler case where all the operators are left associative and there are no parentheses. Also, for the time being, 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.

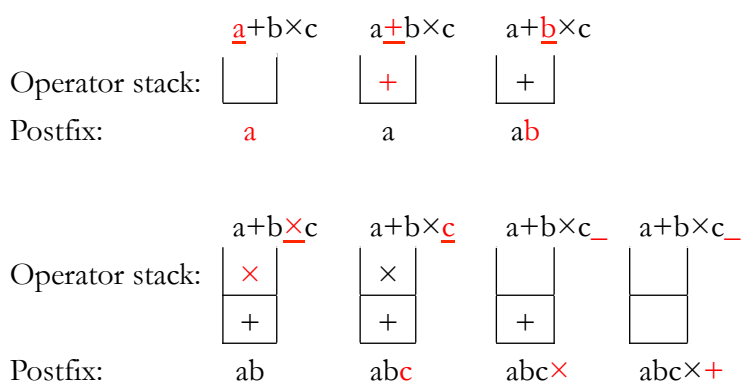
Algorithm – the Simple Case (cont.)

- If ρ is an operand, append it to the postfix expression.
- If ρ is an operator.

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

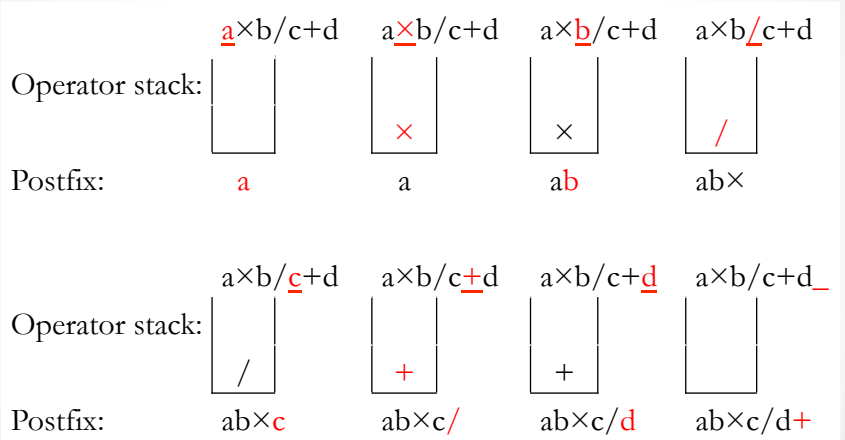
When there are no more items to scan, pop off the operators from the stack, appending them to the postfix as you do so.

Example: a+b×c



What would have happened if the input expression had been a×b+c? or a/b×c?

Example: a×b/c+d



Infix to Postfix Conversion: The General Case

The previous algorithm does not handle right-associative operators and parentheses. To fix this, we use a trick:
Each operator (including parentheses) will have two, possibly different, precedences, depending on whether it is being scanned as part of the input infix expression or it is already on the stack.

Symbol	Input Precedence	Stack Precedence
+ −	1	1
× / %	2	2
^	4	3
(	5	-1
)	0	0

The General Case

Now, in the algorithm, we replace the comparison

prec(ρ) ≤ prec(opStack.peek())

by

inputPrec(ρ) ≤ stackPrec(opStack.peek())

The revised precedence rules allow us to handle the right-associative "^" operator: Since the input precedence of "^" is higher than its stack precedence, if there are two successive "^" operators, they will both be pushed on the stack.

For example, consider the expression "a^b^c". This must be evaluated as "a^(b^c)", **not** as "(a^b)^c". That is, the postfix equivalent is "abc^^", **not** "ab^c^".

Algorithm – The General Case

As the expression is scanned, do the following:

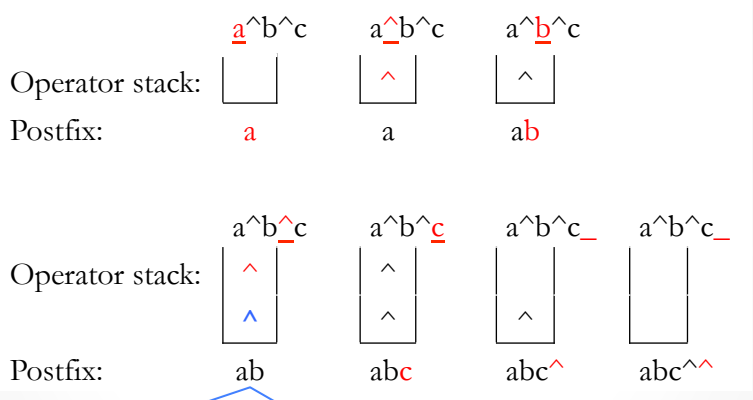
- If we encounter a new operand, append it to the postfix expression.
- If the next term, ρ, is an operator or a "(", do the following:

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

- If the input is ")", pop all operators from the stack until "(" and write them to the postfix string. Pop "(".

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

Example: "a^b^c"



i_prec(^) = 4 > 3 = s_prec(^) ➔ push ^, so it will be popped before ^.

Handling Parentheses

The trick also allows us to handle parentheses:

- The input precedence of a left parenthesis is 5, which is higher than that of any operator.
- Thus, all operators on the stack will remain there, and the "(" will be pushed on top of them.
- This makes sense, because a new subexpression is beginning, which has to be evaluated before all the operators on the stack. When a matching ")" is found, all operands on the stack down to the corresponding "(" will be popped.

Example: a×(b+c)