CS 228: Introduction to Data Structures
Lecture 25
Monday, October 24, 2016

Infix to Postfix Conversion (Continued)

As mentioned last time, the infix-to-postfix conversion algorithm uses an ***operator stack***, opStack, to temporarily store operators awaiting their right operand, and to help manage the order of precedence. Let us continue with the simpler case where all the operators are left associative and there are no parentheses.

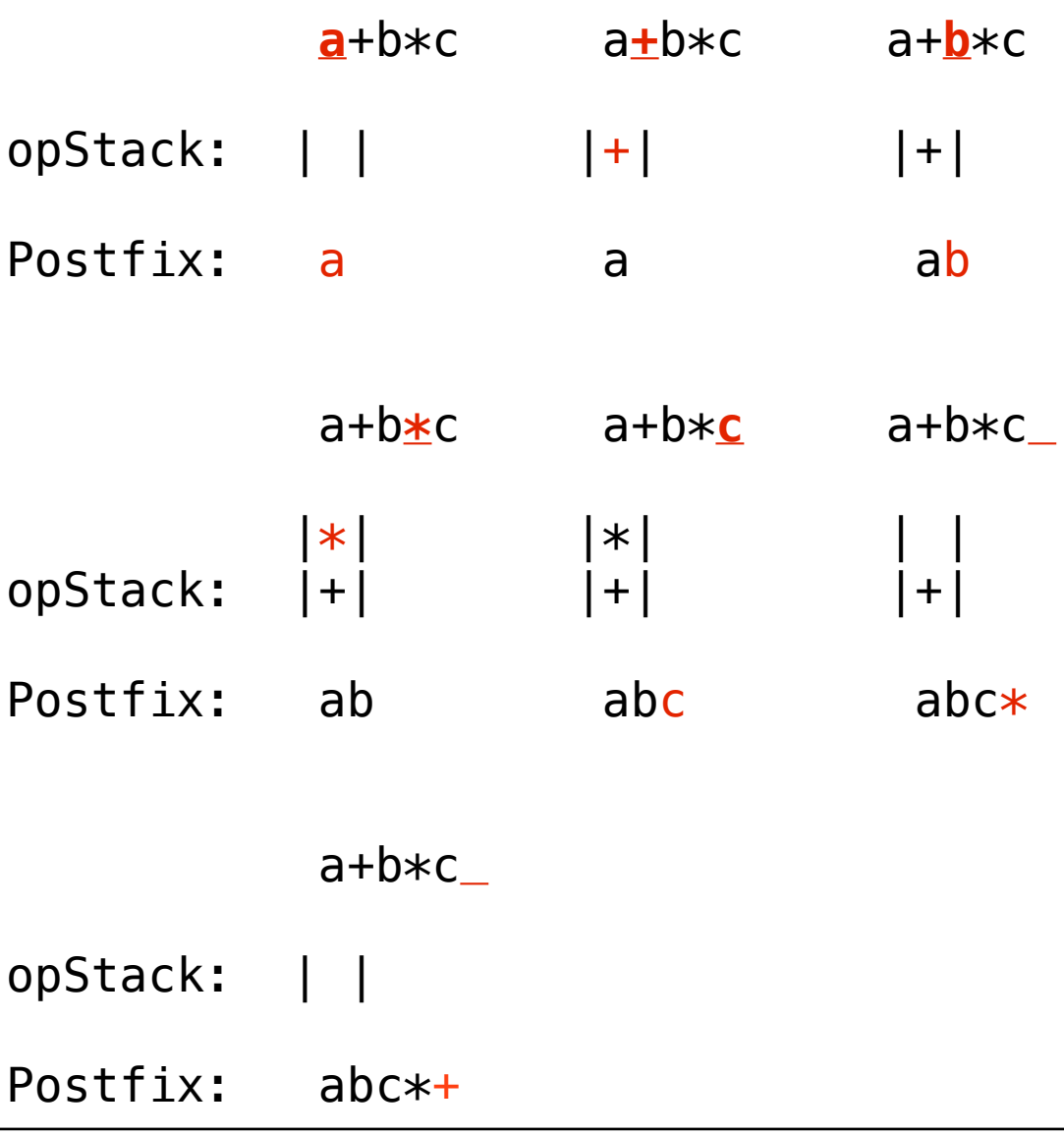
The input infix expression is scanned from left to right. Suppose ρ is the item that is currently being scanned; ρ can be either an operand or an operator.

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

```
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. Suppose the input infix string is a+b∗c.



Exercise. What happens if the input expression is a∗b+c? a/b∗c?

The next example that shows how the basic algorithm handles left associativity and precedence.

Example. Suppose the input infix string is a*b/c+d.

```
            a*b/c+d      a*b/c+d      a*b/c+d
opStack:   | |          |*|          |*|
Postfix:    a            a            ab

            a*b/c+d      a*b/c+d      a*b/c+d
opStack:   |/|          |/|          |+|
Postfix:    ab*          ab*c         ab*c/

            a*b/c+d      a*b/c+d_
opStack:   |+|          | |
Postfix:    ab*c/d       ab*c/d+
```

The General Case

The algorithm we just saw 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

Now, in the algorithm, we replace the comparison

$$\text{prec}(\rho) \leq \text{prec}(\text{opStack.peek}())$$

by

$$\textbf{inputPrec}(\rho) \leq \textbf{stackPrec}(\text{opStack.peek}())$$

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.

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^".

```
            a^b^c       a^b^c       a^b^c

opStack:   | |         |^|         |^|

Postfix:    a           a           ab


            a^b^c       a^b^c       a^b^c_

           |^|         |^|
opStack:   |^|         |^|         |^|

Postfix:    ab          abc         abc^
            v
            v
```

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

```
            a^b^c_

opStack:   | |

Postfix:    abc^^
```

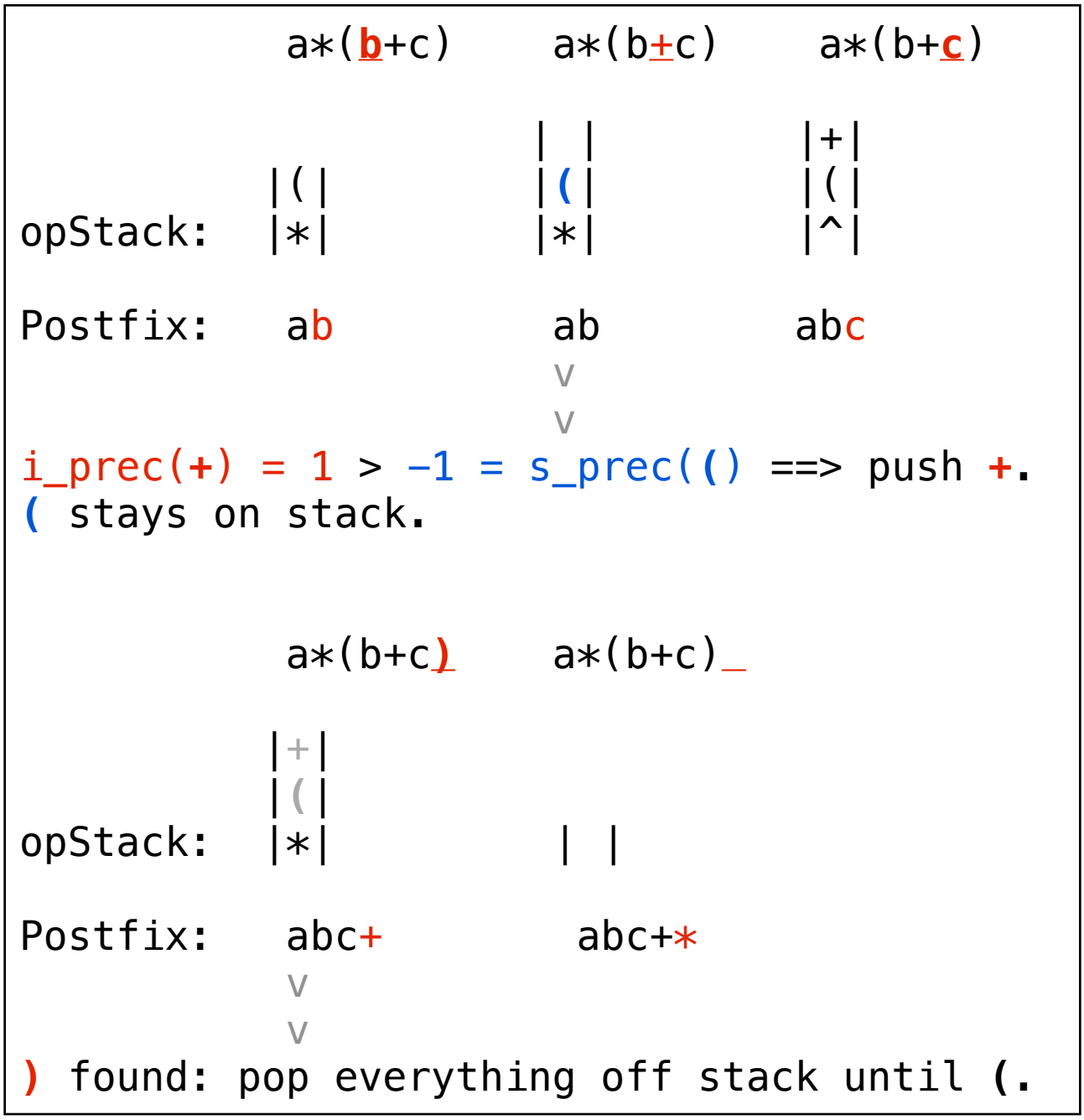
The trick also allows us to handle parentheses: The input precedence of a left parenthesis is 5, which is higher than that of any operator. Thus, all operators on the stack will remain there, and the "(" will be pushed on top of them. This makes sense, because a new subexpression is beginning, which 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)     a*(b+c)     a*(b+c)

                                    |(|
opStack:   | |          |*|         |*|

Postfix:    a           a           a
                                    v
                                    v
i_prec(() = 5 > 2 = s_prec(*) ==> push (.
Now s_prec(() = -1.
```



Algorithm

As the input infix 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.

Exercise. Apply the algorithm to "3 * (4 – 2 ^ 5) + 6".

Time Complexity

Suppose the infix string has *n* operators and operands. There are *O*(*n*) operators, and, when

processing a single operator ρ, the conversion algorithm may perform opStack.pop() $O(n)$ times. This appears to imply that the total time is $O(n \cdot n) = O(n^2)$, but this estimate is too pessimistic.

Fact. *The worst-case time for the conversion is* $O(n)$.

To verify this, let's count the number of write, push, and pop operations.

- Every operator or operand that is not "(" or ")" is written to the postfix string. So, there are $O(n)$ writes.

- Every operator that is not ")" gets pushed onto the stack. So, there are $O(n)$ pushes.

- #pops ≤ #pushes. So, there are $O(n)$ pops.

Since each write, push, or pop takes $O(1)$ time, the total time complexity is $O(n)$

Error Checking (Not covered in class)

Every term in an expression, whether it is an operator or an operand, has a ***rank***. The full table is:

1	for any operand
-1	for +, −, ∗, /, %, ^
0	for (,)

Ranks can help us to check the validity of an expression. The ***cumulative rank*** of an expression is the sum of the ranks of individual terms.

	2 ^ 7 ^ 6 + (3 − 2 ∗ 4) % 5
Cumulative rank	1 0 1 0 1 0 0 1 0 1 0 1 1 0 1

The cumulative rank after each symbol is always 0 or 1. If it falls outside this range during a scan, we report an error. Also, the cumulative rank for an expression as a whole must be 1, since there must be exactly one more operand than operator. If the cumulative rank of an expression is not 1, the expression is invalid. For instance, the cumulative rank of the (invalid) expression "2 4 + 3" is 2.