Evaluation of Expressions

### Arithmetic Expressions

- $(a + b) * (c + d) + e - f/g*h + 3.25$

- Expressions comprise three kinds of entities.
  - Operators (+, -, /, *).
  - Operands (a, b, c, d, e, f, g, h, 3.25, $(a + b)$, $(c + d)$, etc.).
  - Delimiters (, ).

### Operator Degree

- Number of operands that the operator requires.
- Binary operator requires two operands.
  - $a + b$
  - $c / d$
  - $e - f$
- Unary operator requires one operand.
  - $+ g$
  - $- h$

### Infix Form

- Normal way to write an expression.
- Binary operators come in between their left and right operands.
  - $a * b$
  - $(a + b) * c$
  - $(a * b / c)$
  - $(a + b) * (c + d) + e - f/g*h + 3.25$
Operator Priorities

• How do you figure out the operands of an operator?
  
  \( a + b \times c \)  
  \( a \times b + c / d \)

• This is done by assigning operator priorities.
  
  \( \text{priority}(\times) = \text{priority}(\div) > \text{priority}(+) = \text{priority}(-) \)

• When an operand lies between two operators, the operand associates with the operator that has higher priority.

Evaluation Expression in C++

• When evaluating operations of the same priorities, it follows the direction from left to right.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unary minus, !</td>
</tr>
<tr>
<td>2</td>
<td>*, /, %</td>
</tr>
<tr>
<td>3</td>
<td>+, -</td>
</tr>
<tr>
<td>4</td>
<td>&lt;, &lt;=, &gt;=, &gt;</td>
</tr>
<tr>
<td>5</td>
<td>== (equal), !=</td>
</tr>
<tr>
<td>6</td>
<td>&amp;&amp; (and)</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

• C++ treats
  
  – Nonzero as true
  – zero as false
  – !3&&5 +1 \rightarrow 0

In Class Exercise

• \( x=6, y=5 \)

• \( 10+x\times5/y+1 \)

• \( (x>=5)\&\&y<10 \)

• \( !x>10+!y \)

Tie Breaker

• When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.

\( a + b - c \)

\( a \times b / c / d \)
**Delimiters**

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.
  
  \[(a + b) \times (c - d) / (e - f)\]

**Infix Expression Is Hard To Parse**

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- So it is easier for a computer to evaluate expressions that are in these forms.

**Postfix Form**

- The postfix form of a variable or constant is the same as its infix form.
  
  a, b, 3.25
- The relative order of operands is the same in infix and postfix forms.
- Operators come immediately after the postfix form of their operands.
  
  Infix = \(a + b\)  \(\rightarrow\) Postfix = \(ab+\)

**Postfix Examples**

- Infix = \(a + b \times c\)
  
  Postfix = \(a b c * +\)
- Infix = \(a \times b + c\)
  
  Postfix = \(a b * c +\)
- Infix = \((a + b) \times (c - d) / (e + f)\)
  
  Postfix = \(a b + c d - * e f + /\)
Unary Operators

- Replace with new symbols.
  - \( + a \Rightarrow a @ \)
  - \( + a + b \Rightarrow a @ b + \)
  - \( - a \Rightarrow a ? \)
  - \( - a - b \Rightarrow a ? b - \)

Postfix Notation

Expressions are converted into Postfix notation before compiler can accept and process them.

\[ X = \frac{A}{B} - C + D \cdot E - A \cdot C \]

<table>
<thead>
<tr>
<th>Infix</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A / B - C + D \cdot E - A \cdot C )</td>
<td>( A B / C - D E * + A C * - ) (Operators come in-between operands)</td>
</tr>
<tr>
<td>( A B / C - D E * + A C * - )</td>
<td>( A B / C - D E * + A C * - ) (Operators come after operands)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 = A / B )</td>
<td>( T_1 C - D E * + A C * - )</td>
</tr>
<tr>
<td>( T_2 = T_1 - C )</td>
<td>( T_2 D E * + A C * - )</td>
</tr>
<tr>
<td>( T_3 = D * E )</td>
<td>( T_2 T_3 + A C * - )</td>
</tr>
<tr>
<td>( T_4 = T_2 + T_3 )</td>
<td>( T_4 A C * - )</td>
</tr>
<tr>
<td>( T_5 = A * C )</td>
<td>( T_4 T_5 - )</td>
</tr>
<tr>
<td>( T_6 = T_4 - T_5 )</td>
<td>( T_6 )</td>
</tr>
</tbody>
</table>

Postfix Evaluation

- Scan postfix expression from left to right pushing operands on to a stack.
- When an operator is encountered, pop as many operands as this operator needs; evaluate the operator; push the result on to the stack.
- This works because, in postfix, operators come immediately after their operands.

Postfix Evaluation

- \( (a + b) \cdot (c - d) / (e + f) \)
- \( a b + c d - * e f + / \)
- \( a b + c d - * e f + / \)
- \( a b + c d - * e f + / \)
Postfix Evaluation

• \((a + b) \times (c - d) / (e + f)\)
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /

stack

\(d\)

\(c\)

\((a + b)\)

Postfix Evaluation

• \((a + b) \times (c - d) / (e + f)\)
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /

\(e\)

\(f\)

\((a + b)\times(c - d)\)

stack

Postfix Evaluation

• \((a + b) \times (c - d) / (e + f)\)
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /
• \(a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /

stack

\((c - d)\)

\((a + b)\)
Infix to Postfix

- The order of the operands in both form is the same.
- An algorithm for producing postfix from infix:
  1. Fully parenthesize the expression.
  2. Move all operators so that they replace their corresponding right parentheses.
  3. Delete all parentheses.

Infix to Postfix

- For example: A/B-C+D*E-A*C
  1. Fully parenthesize the expression.
     (((((A/B)-C)+(D*E))-(A*C))
  2. Move all operators so that they replace their corresponding right parentheses.
     ((((AB)/C)-(DE*)+)(AC*))-
  3. Delete all parentheses.
     AB/C-DE*+AC*- 

In Class Exercise

- Write the postfix form:
  A&&B+C*D

Infix to Postfix

- We scan an expression for the first time, we can form the postfix by immediately passing any operands to the output.
- For example: A+B*C
  => ABC*+

<table>
<thead>
<tr>
<th>Next token</th>
<th>Stack</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Empty</td>
<td>None</td>
</tr>
<tr>
<td>A</td>
<td>Empty</td>
<td>A</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>+</td>
<td>AB</td>
</tr>
<tr>
<td>*</td>
<td>+*</td>
<td>AB</td>
</tr>
<tr>
<td>C</td>
<td>+*</td>
<td>ABC</td>
</tr>
</tbody>
</table>

Since * has higher priority, we should stack *.
**Infix to Postfix**

- Example: A*(B+C)/D => ABC+*D/

- When we get ‘)’, we want to unstack down to the corresponding ‘(’ and then delete the left and right parentheses.

<table>
<thead>
<tr>
<th>Next token</th>
<th>Stack</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Empty</td>
<td>None</td>
</tr>
<tr>
<td>A</td>
<td>Empty</td>
<td>A</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>A</td>
</tr>
<tr>
<td>(</td>
<td>*(</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>*(</td>
<td>AB</td>
</tr>
<tr>
<td>+</td>
<td>*(+</td>
<td>AB</td>
</tr>
<tr>
<td>C</td>
<td>*(+</td>
<td>ABC</td>
</tr>
<tr>
<td>)</td>
<td>*</td>
<td>ABC+</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>ABC+*</td>
</tr>
<tr>
<td>D</td>
<td>/</td>
<td>ABC+*D</td>
</tr>
<tr>
<td>Done</td>
<td>Empty</td>
<td>ABC+*D/</td>
</tr>
</tbody>
</table>

**Analysis of Postfix**

- The function makes only a left-to-right pass across the input.
- The complexity of Postfix is $\Theta(n)$, where $n$ is the number of tokens in the expression.
  - The time spent on each operands is $O(1)$.
  - Each operator is stacked and unstacked at most once.
  - Hence, the time spent on each operator is also $O(1)$.

- These examples motivate a priority-based scheme for stacking and unstacking operators.
- When the left parenthesis ‘(’ is not in the stack, it behaves as an operator with high priority.
- Whereas once ‘(’ gets in, it behaves as one with low priority (no operator other than the matching right parenthesis should cause it to get unstacked).
- Two priorities for operators: isp (in-stack priority) and icp (in-coming priority).
- The isp and icp of all operators in Figure 3.15 in p 160 remain unchanged.
- We assume that isp(‘)’ = 8 (the lowest), icp(‘)’ = 0 (the highest), and isp(#) = 8 (# → the last token).
Prefix Form

- The prefix form of a variable or constant is the same as its infix form.
  - $a, b, 3.25$
- The relative order of operands is the same in infix and prefix forms.
- Operators come immediately before the prefix form of their operands.
  - Infix = $a + b$
  - Postfix = $ab+$
  - Prefix = $+ab$

Prefix Examples

- Infix = $a + b * c$
  - Prefix = $+a * b c$
- Infix = $a * b + c$
  - Prefix = $+a b c$
- Infix = $(a + b) * (c - d) / (e + f)$
  - Prefix = $/+a b - c d+e f$

Prefix Notation

Expressions are converted into Prefix notation before compiler can accept and process them.

$$X = A / B - C + D * E - A * C$$

Infix => $A / B - C + D * E - A * C$
Prefix => $- + - / A B C * D E * A C$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1 = A * C$</td>
<td>$A / B - C + D * E - T_1$</td>
</tr>
<tr>
<td>$T_2 = D * E$</td>
<td>$A / B - C + T_2 - T_1$</td>
</tr>
<tr>
<td>$T_3 = A / B$</td>
<td>$T_3 - C + T_2 - T_1$</td>
</tr>
<tr>
<td>$T_4 = T_3 - C$</td>
<td>$T_4 + T_2 - T_1$</td>
</tr>
<tr>
<td>$T_5 = T_4 + T_2$</td>
<td>$T_5 - T_1$</td>
</tr>
<tr>
<td>$T_6 = T_5 - T_1$</td>
<td>$T_6$</td>
</tr>
</tbody>
</table>

Prefix Evaluation

- Scan prefix expression from right to left pushing operands on to a stack.
- When an operator is encountered, pop as many operands as this operator needs; evaluate the operator; push the result on to the stack.
- This works because, in prefix, operators come immediately before their operands.
Prefix Evaluation

• (a + b) * (c – d) / (e + f)
• / * + a b - c d + e f

• / * + a b - c d + e f
• / * + a b - c d + e f
• / * + a b - c d + e f

stack

Prefix Evaluation

• (a + b) * (c – d) / (e + f)
• / * + a b - c d + e f

• / * + a b - c d + e f
• / * + a b - c d + e f
• / * + a b - c d + e f

stack

Prefix Evaluation

• (a + b) * (c – d) / (e + f)
• / * + a b - c d + e f

• / * + a b - c d + e f
• / * + a b - c d + e f
• / * + a b - c d + e f

stack

Prefix Evaluation

• (a + b) * (c – d) / (e + f)
• / * + a b - c d + e f

• / * + a b - c d + e f
• / * + a b - c d + e f
• / * + a b - c d + e f

stack
Prefix Evaluation

• \((a + b) \times (c - d) / (e + f)\)
• \(/ + \ a \ b - c \ d + e \ f\)

Infix to Prefix

• The order of the operands in both form is the same.

An algorithm for producing prefix from infix:
1. Fully parenthesize the expression.
2. Move all operators so that they replace their corresponding left parentheses.
3. Delete all parentheses.

Infix to Prefix

• For example: \(A/B-C+D*E-A*C\)
  1. Fully parenthesize the expression.
     \(((A/B)-C)+(D*E)-(A*C))\)
  2. Move all operators so that they replace their corresponding left parentheses.
     \((-+(-/AB)C)(DE))(AC))\)
  3. Delete all parentheses.
     \(+/-ABC*DE*AC\)

In Class Exercise

• Write the prefix form:
  \(A&&B+C*D\)
### Infix to Prefix

- We reverse an expression at first.
- Create empty reversed prefix String by passing any operands to the output.
- We can form the prefix by immediately reverse again the reversed prefix String.
- For example: A+B*C
  - reverse: C*B+ => +A*B*C

### Example: A*(B+C)*D

- Reverse: D *)C+B("A => **A+BCD

- When we get ‘(’, we want to unstack down to the corresponding ‘)’ and then delete the left and right parentheses.

### Infix to Prefix

<table>
<thead>
<tr>
<th>Next token</th>
<th>Stack</th>
<th>Reverse S</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Empty</td>
<td>None</td>
</tr>
<tr>
<td>C</td>
<td>Empty</td>
<td>C</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>*</td>
<td>CB</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>CB*</td>
</tr>
<tr>
<td>A</td>
<td>+</td>
<td>CB*A+</td>
</tr>
</tbody>
</table>

Since * has higher priority, we should pop *, then push +.

### Result rule of priorities:

- Operators are taken out of the stack as long as their isp is numerically less than the icp of the new operator.
- Not the same as Infix to Postfix

---

<table>
<thead>
<tr>
<th>Next token</th>
<th>Stack</th>
<th>Reverse S</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Empty</td>
<td>None</td>
</tr>
<tr>
<td>D</td>
<td>Empty</td>
<td>D</td>
</tr>
<tr>
<td>*)</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>DC</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>DCB+</td>
</tr>
<tr>
<td>*</td>
<td>**</td>
<td>DCB+A</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>DCB+A**</td>
</tr>
</tbody>
</table>

*Note: The isp and icp of all operators in Figure 3.15 in p 160 remain unchanged.*

- We assume that isp(')') = 8 (the lowest), icp(')') = 0 (the highest), and isp('#') = 8 (# -> the last token)
Analysis of Prefix

• The function makes only a left-to-right pass across the input (reversed prefix String).
• The complexity of Postfix is \( \Theta(n) \), where \( n \) is the number of tokens in the expression.
  – The time spent on each operands is \( O(1) \).
  – Each operator is stacked and unstacked at most once.
  – Hence, the time spent on each operator is also \( O(1) \)

Homework

• Sec. 3.7 Exercise 3 (Page 166)
  – Convert infix expressions to prefix expressions