CSCD 326 Data Structures I
Infix Expressions
Infix Expressions

- Binary operators appear between operands:
  \[ W - X / Y - Z \]

- Order of evaluation is determined by:
  - precedence rules
  - parentheses
  - association (L to R or R to L)

- Direct evaluation is very difficult for a program reading left to right.
Postfix Expressions

- Binary operators appear after both operands:
  - $X + Y$ in postfix form is: $X Y +$
- Order of operations is completely determined by the expression
  - no parentheses or precedence required
  - for example:
    - $A * B - C$ becomes $A B * C -$
Consider: \( A \ B \ * \ C \ - \)

- When evaluating the postfix expression the result of \( A \ B \ * \) becomes the first operand for \( C \ - \)
- \( A \ * \ ( B \ - \ C ) \) becomes \( A \ B \ C \ - \ * \)
- Here the result of \( B \ C \ - \) becomes the second operand for \( A \) operand -
Postfix Evaluation with a Stack

- Evaluation of postfix expressions makes use of an operand stack.
- Algorithm Pseudocode:
  - push all operands on stack until an operator is encountered
  - when an operator is encountered pop the last two operands off the stack and apply the operation - order of operands is important
  - push the result of the operation back on the stack to be used as an operand later
  - when end of expression is reached one operand remains on the stack - it is the final result
Infix to Postfix Conversion

Since postfix expressions are easily evaluated, the easiest way to evaluate infix is to convert from infix to postfix and then evaluate.

This conversion uses an operator stack since low precedence operators must be saved and applied after high precedence ones.
Conversion Algorithm

stack.push(EOL marker)  //optional – can check for empty stack
for (each character ch in the infix expression) {
  switch (ch) {
    case operand:
      //append ch to output postfix expression
      postfixExpr = postfixExpr + ch;
    case '(':
      stack.push(ch);  // push ch onto operator stack
    case ')':
      while (top of stack is not '(' ) {
        postfixExpr = postfixExpr + stack.pop();  // pop and append expr.
      }
      stack.pop();
    case operator:
      while (!stack.isEmpty() && inputPrec(ch) <= stackPrec(stack.top())) {
        postfixExpr = postfixExpr + stack.pop();
      }
      stack.push(ch);
  }  // end for
  while (!stack.isEmpty() {  // pop all remaining operators off stack and append
    postfixExpr = postfixExpr + stack.pop();
  }
}
Conversion Example - 1

\[ W - X / Y + Z \]

- `-` is pushed onto operator stack
- `/` has higher precedence than `-` on stack - it gets pushed
- `+` has lower precedence than `/` on stack - pop `/` and output - then `-` is on top of stack - since subtraction associates left to right - pop `-` off stack and output - push `+` onto stack
- when end of expression is reached pop remaining operator and output
- resulting expression is: \[ W X Y /- Z + \]
Conversion Example - 2

'^' represents exponentiation in some languages

```
3
2
2
```

or

```
2 ^2 ^3
```

Evaluated left to right this is $4^3 = 64$

Evaluated right to left this is $2^8 = 512$ - this is the correct evaluation order
Conversion Example - 2 (2)

2 ^1 2 ^2 3:

- push '^1'
- when '^2' is encountered do not pop the stack but push '^2' - due to R to L associativity.
- at end of expression, pop all remaining operators and output
- final expression is:

2 2 3 ^2 ^1
## Precedence Order for algorithm

<table>
<thead>
<tr>
<th>Token</th>
<th>EOL</th>
<th>(</th>
<th>)</th>
<th>^</th>
<th>/</th>
<th>*</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>-1</td>
<td>0</td>
<td>99</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Example Expression

A * ( B -C)

Token:

Output:

Stack:

<table>
<thead>
<tr>
<th>Token</th>
<th>EOL</th>
<th>(</th>
<th>)</th>
<th>^</th>
<th>/</th>
<th>*</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Stack</td>
<td>-1</td>
<td>0</td>
<td>99</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Example Expression (2)

A * B -C

Token:

Output:

Stack:

<table>
<thead>
<tr>
<th>Token</th>
<th>EOL</th>
<th>(</th>
<th>)</th>
<th>^</th>
<th>/</th>
<th>*</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Stack</td>
<td>-1</td>
<td>0</td>
<td>99</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Pencil Method

- Allows a paper algorithm check
  - Fully parenthesize expression
  - Move each operator to replace its closest right parenthesis
  - Remove left parentheses

- A * B - C
- (A * B) - C
- (( A * B ) - C)
Example Expression (3)

\[ A + B - ( C + D \times E ) \]

Token:

Output:

Stack:

<table>
<thead>
<tr>
<th>Token</th>
<th>EOL</th>
<th>(</th>
<th>)</th>
<th>^</th>
<th>/</th>
<th>*</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Stack</td>
<td>-1</td>
<td>0</td>
<td>99</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Example Expression (4)

A + B * C - D / E

Token:

Output:

Stack:

<table>
<thead>
<tr>
<th>Token</th>
<th>EOL</th>
<th>(</th>
<th>)</th>
<th>^</th>
<th>/</th>
<th>*</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Stack</td>
<td>-1</td>
<td>0</td>
<td>99</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Example Expression (5)

\[ A + ( B + C \times ( D - E \times F )) \times G \]

Token:

Output:

Stack: