The Shunting Yard Algorithm

Edsger Dijkstra developed his "Shunting Yard" algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

In this algorithm, all operands are printed (or sent to output) when they are read. There are more complicated rules to handle operators and parentheses.

Examples

1. A * B + C becomes A B * C +

   The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.

   We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

```
A * B + C  -->  A B * C +

  Current Symbol     Operator Stack    Postfix String
1      A                                    A
2      *                  *                 A
3      B                  *                 A B
4      +                  +                 A B *  (pop and print * before pushing +)
5      C                  +                 A B * C
6                                           A B * C +
```

   The rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

2. A + B * C becomes A B C * +

   Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

```
A + B * C  -->  A B C * +

  Current Symbol     Operator Stack    Postfix String
1      A                                    A
2      +                  +                 A
3      B                  +                 A B
4      *                  + *               A B
```

```
  5      C                  + *              A B C
  6                                          A B C * +
```

In line 4, the '*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when the are both popped off in lines 6 and 7, their order will be reversed.

3. A * (B + C) becomes A B C + *

   A subexpression in parentheses must be done before the rest of the expression.

```
  A * ( B + C )  -->  A B C + *

    Current Symbol     Operator Stack    Postfix String
  1      A                                   A
  2      *                  *                A
  3      (                  * (              A B
  4      B                  * (              A B
  5      +                  * ( +            A B
  6      C                  * ( +            A B C
  7      )                  *                A B C +
  8                                          A B C + *
```

   Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

4. A - B + C becomes A B - C +

   When operators have the same precedence, we must consider association. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

```
  A - B + C  -->  A B - C +

    Current Symbol     Operator Stack    Postfix String
  1      A                                   A
  2      -                  -                A
  3      B                  -                A B
  4      +                  +                A B -
  5      C                  +                A B - C
  6                                          A B - C +
```

   In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5. A * B ^ C + D becomes A B C ^ * D +

   Here both the exponentiation and the multiplication must be done before the addition.

```
  A * B ^ C + D  -->  A B C ^ * D +

    Current Symbol     Operator Stack    Postfix String
  1      A                                   A
  2      *                  *                A
  3      B                  *                A B
  4      ^                  * ^              A B
  5      C                  * ^              A B C
  6      +                  +                A B C ^ *
  7      D                  +                A B C ^ * D
  8                                          A B C ^ * D +
```

   When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6. A * (B + C * D) + E becomes A B C D * + * E +

```
A * ( B + C * D ) + E  -->  A B C D * + * E +

  Current Symbol     Operator Stack    Postfix String
1      A                                  A
2      *                 *                A
3      (                 * (              A
4      B                 * (              A B
5      +                 * ( +            A B
6      C                 * ( +            A B C
7      *                 * ( + *          A B C
8      D                 * ( + *          A B C D
9      )                 *                A B C D * +
10     +                 +                A B C D * + *
11     E                 +                A B C D * + * E
12                                        A B C D * + * E +
```

Summary of the Rules

1. If the incoming symbols is an operand, print it..
2. If the incoming symbol is a left parenthesis, push it on the stack.
3. If the incoming symbol is a right parenthesis: discard the right parenthesis, pop and print the stack symbols until you see a left parenthesis. Pop the left parenthesis and discard it.
4. If the incoming symbol is an operator and the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
5. If the incoming symbol is an operator and has either higher precedence than the operator on the top of the stack, or has the same precedence as the operator on the top of the stack and is right associative -- push it on the stack.
6. If the incoming symbol is an operator and has either lower precedence than the operator on the top of the stack, or has the same precedence as the operator on the top of the stack and is left associative -- continue to pop the stack until this is not true. Then, push the incoming operator.
7. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Original text posted by Carol E. Wolf of Pace University; adapted by P. Oser