Humans usually write algebraic expressions like this:

\[ a + b \]

This is called **infix notation**, because the operator ("+") is inside the expression.

A problem is that we need parentheses or precedence rules to handle more complicated expressions:

\[ a + b \times c = (a + b) \times c \ ? \]
\[ = a + (b \times c) \ ? \]
Infix, postfix, and prefix notation

- There is no reason we can’t place the operator somewhere else.
- **infix** notation:
  
  a + b
- **prefix** notation:
  
  + a b
- **postfix** notation:
  
  a b +

Other names

- **Prefix notation** was introduced by the Polish logician Lukasiewicz, and is sometimes called “Polish notation”
- **Postfix notation** is sometimes called “reverse Polish notation” or RPN
  
  » Used on some calculators
  
  (the ones without ‘=’ signs)
Question: Why would anyone ever want to use anything so “unnatural,” when infix seems to work just fine?

Answer: With postfix and prefix notations, parentheses are no longer needed!

<table>
<thead>
<tr>
<th>infix</th>
<th>postfix</th>
<th>prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a + b) * c</td>
<td>a b + c *</td>
<td>* + a b c</td>
</tr>
<tr>
<td>a + (b * c)</td>
<td>a b c * +</td>
<td>+ a * b c</td>
</tr>
</tbody>
</table>

Converting from infix notation to postfix notation

Assume that your **infix** expression is of the form

\(<identifier> <operator> <identifier>\)

A postfix expression is created by rewriting this as

\(<identifier> <identifier> <operator>\)
Convert these infix expressions to postfix notation

- $x$
- $x + y$
- $(x + y) - z$
- $w * ((x + y) - z)$
- $(2 * a) / ((a + b) * (a - c))$

Convert these postfix expressions to infix notation

- $3 - r$
- $1 3 r - +$
- $s t * 1 3 r - + +$
- $v w x y z * - + *$
A stack-based algorithm to evaluate a postfix expression

for each character C in a given string, proceeding left to right
{
    if C is an operand
        push C onto stack
    else // C is an operator
        {
            pop item from stack, and store in Opr2
            pop item from stack, and store in Opr1
            result = Opr1 C Opr2, using C as an operator
            push result onto stack
        } 
}