Data Structures

Infix to Postfix with Parentheses

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

Expressions with parentheses

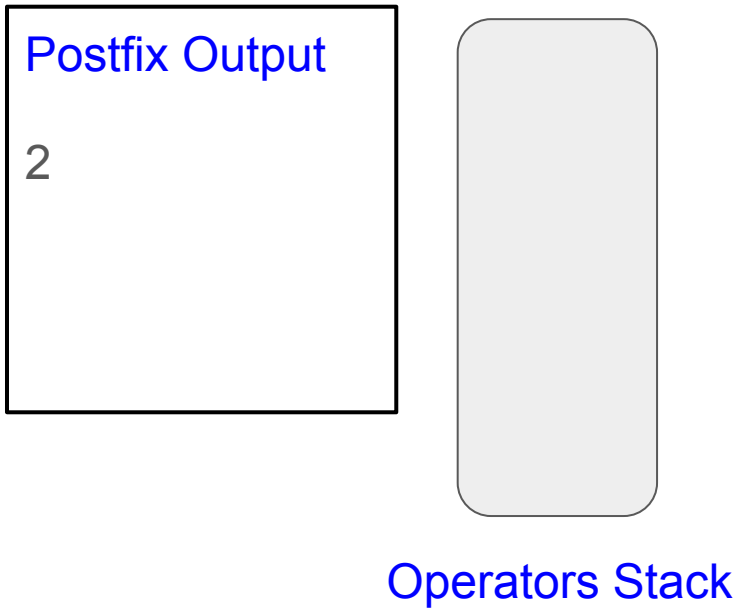
- For the expression 2 + 3 * 4 ⇒ 234*+ [which is the same as 2 + (3*4)]
- What about: (2+3) * 4
  - Observation: the expression inside the ( ) is **independent** of that outside
  - 2+3 ⇒? 23+
  - This is like A*4, where A is 23+
  - So, overall, A4* would be represented as: 23+4* (A=23+)
- 2+3-((5+2)*3)
  - A = 5+2 ⇒ 52+
  - B = A*3 ⇒ A3* ⇒ 52+3*
  - 2+3-B ⇒ 23+B- ⇒ 23+52+3*-
- So: can we independently call postfix conversions on these deeper ones first?
  - ~O(n^2)

Stack's Major Advantage

- We know stack is excellent with reversing tasks
- But it also work extremely well with sub-recursive tasks
- Can we change the stack code to simply consider the () in O(n)?
- The idea is simple
  - When you find (, just add it to the stack to indicate a sub-problem
  - Once an) is found (and assuming the input is accurate), pop everything from the stack until you find the corresponding (
  - This way, the same code can be applied to easily solve the sub-problem
  - The first) we meet represents the deepest sub-expression

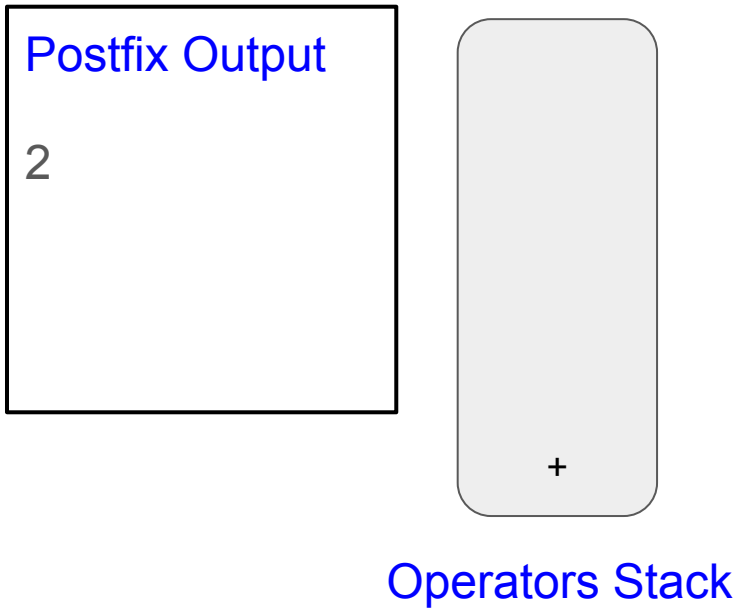
Parsing: **2**+3-((5+2)*3)

- Current Token 2
  - Digit
- Rule #1: if our token is a digit, simply add it to the output/postfix string immediately



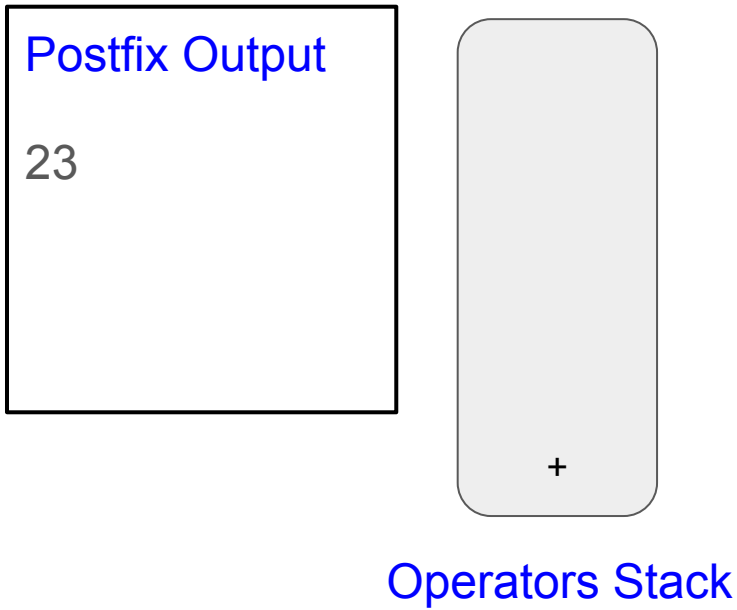
Parsing: 2+3-((5+2)*3)

- Current Token +
  - Operator
- Rule #2: if our token is an operator, and our stack is empty, simply push the operator into the stack



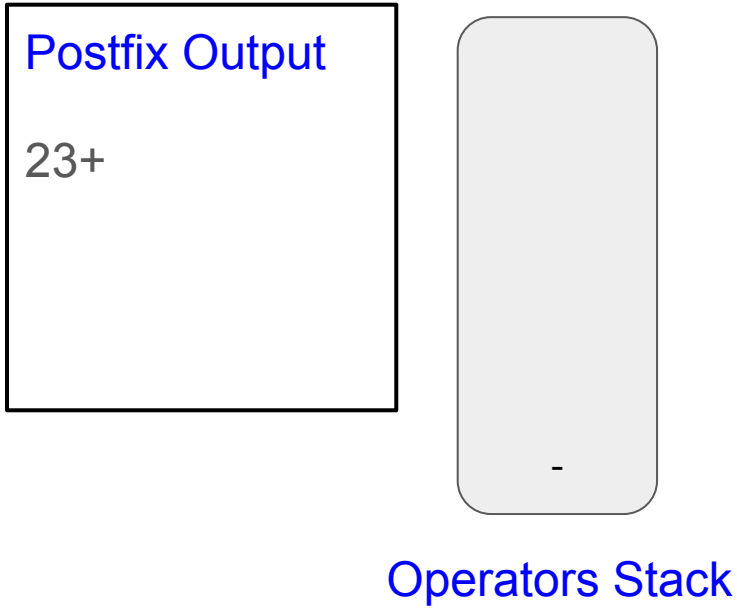
Parsing: 2+**3**-((5+2)*3)

- Current Token 3
  - Digit
- Rule #1: if our token is a digit, simply add it to the output/postfix string immediately



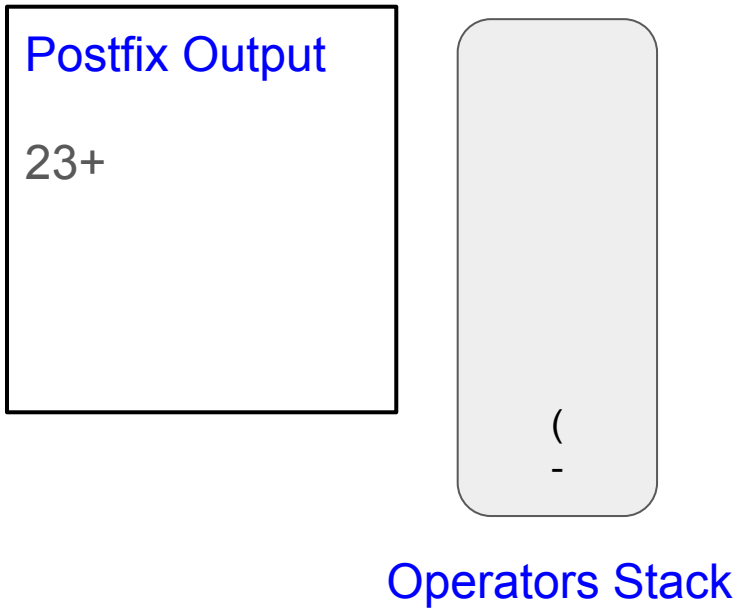
Parsing: 2+3-((5+2)*3)

- Current Token -
  - Operator
- Rule #4: : while the precedence of our current token is less than or equal to that of the top of the stack, pop the top of the stack, and add it to our postfix string
- After that has been processed, add the current token to the stack



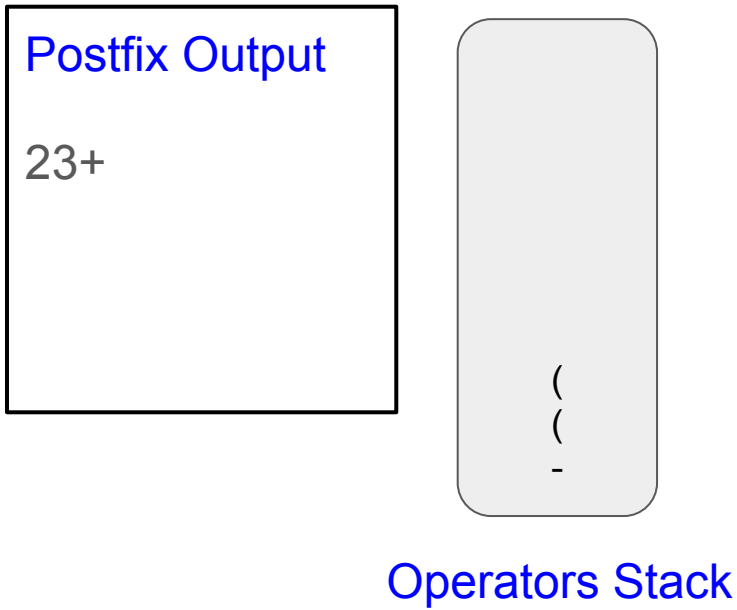
Parsing: 2+3-((5+2)*3)

- Current Token (
  - Operator
- Rule: if the token is a **(**, simply add it to our stack of operators
  - Don't concern yourself with its precedence
  - the (operator signals a new sub-problem



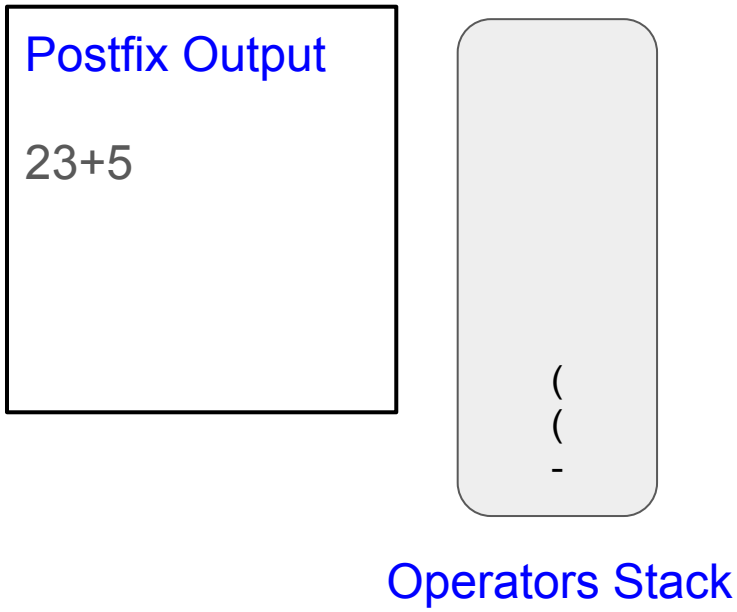
Parsing: 2+3-((5+2)*3)

- Current Token (
  - Operator
- Same as last slide



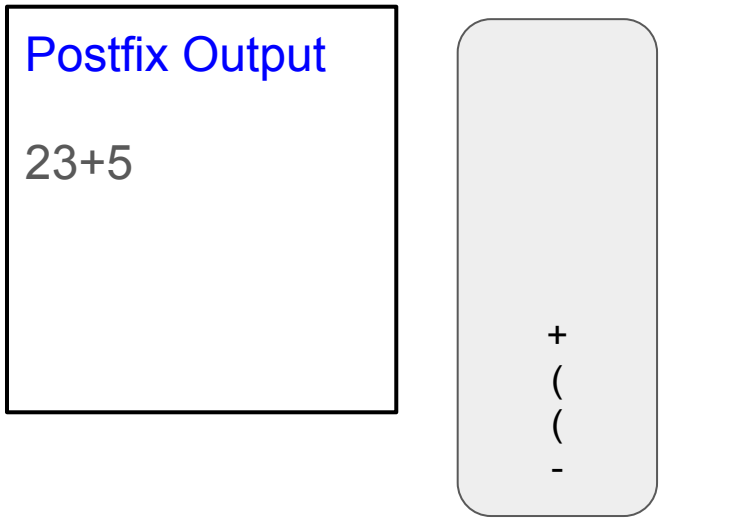
Parsing: 2+3-((**5**+2)*3)

- Current Token 5
  - Digit
- Rule #1: if our token is a digit, simply add it to the output/postfix string immediately



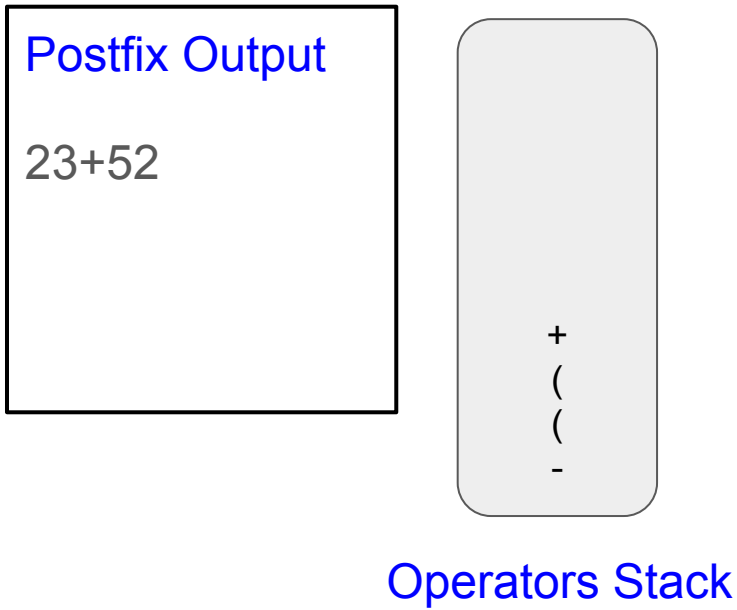
Parsing: 2+3-((5+2)*3)

- Current Token +
  - Operator
- Normally, if adding + to a non-empty stack, we would expect to pop the other operators, given the low precedence of +
- However, if the top of the stack is a (, we do not pop it. Simply place any operator (in this case, +) over any (in our stack
- Rule: if the top of the stack is a (, just add the operator to the stack



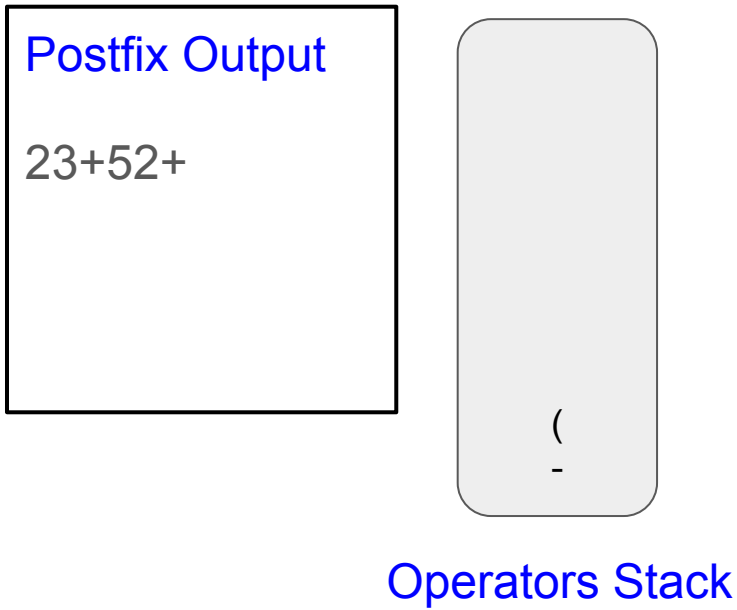
Parsing: 2+3-((5+**2**)*3)

- Current Token 2
  - Digit
- Rule #1: if our token is a digit, simply add it to the output/postfix string immediately



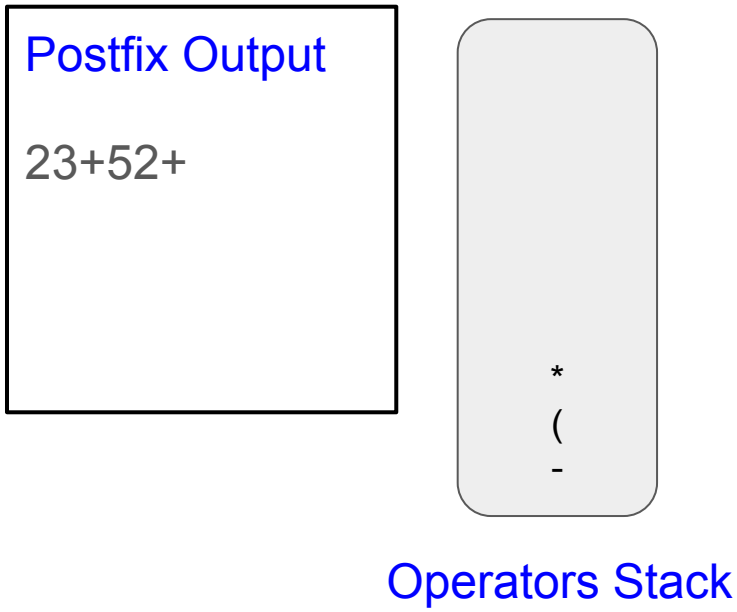
Parsing: 2+3-((5+2)*3)

- Current Token)
  - Digit
- Rule: if our token is), a sub-expression is complete.
  - Pop all operators that appear from the top of the stack until the corresponding (.



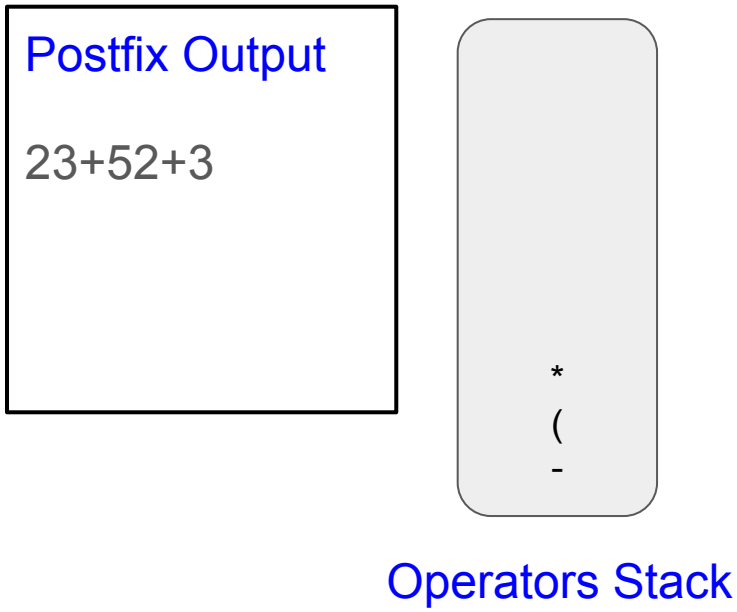
Parsing: 2+3-((5+2)*3)

- Current Token *
  - Operator
- Rule: if the top of the stack is a (, just add the new operators to the top of the stack



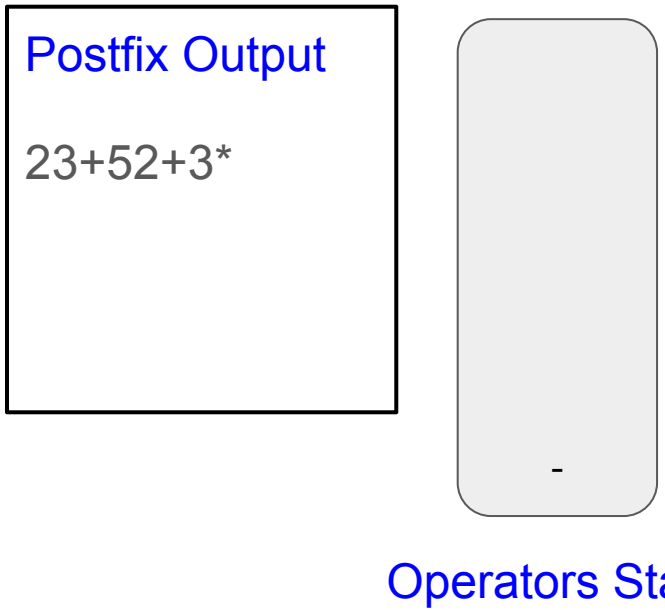
Parsing: 2+3-((5+2)*3)

- Current Token 3
  - Digit
- Rule #1: if our token is a digit, simply add it to the output/postfix string immediately



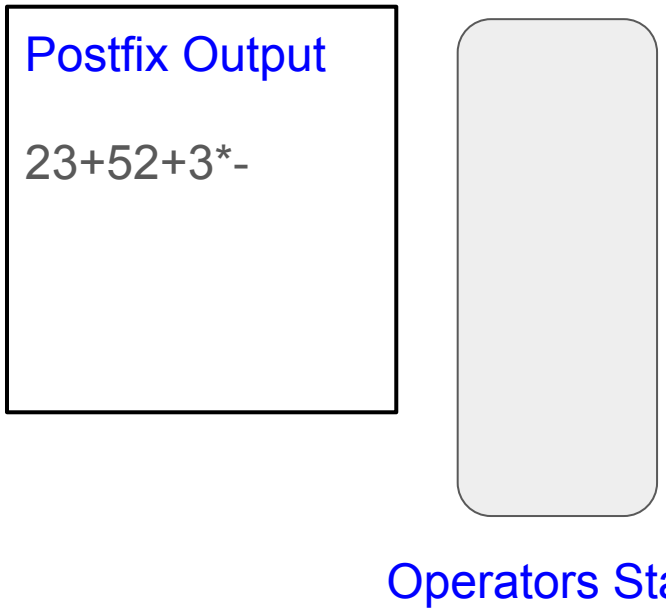
Parsing: 2+3-((5+2)*3)

- Current Token)
  - Digit
- Rule: if our token is), a sub-expression is complete.
  - Pop all operators that appear from the top of the stack until the corresponding (.



Parsing: 2+3-((5+2)*3)

- No current token!
- Rule #5: if we have finished iterating through our expression, pop each operator from the stack in order, adding it to the postfix
- The final expression 23+52+3*-
- Your turn: take 10 minutes to modify our previous code



```
def InfixToPostfix(infix):
    # Assume: no spaces, single digits, only + - * /
    operators = []
    postfix = ''
    def precedence(op):
        if op == '+' or op == '-':
            return 1
        if op == '*' or op == '/':
            return 2
        return 0            # ( is 0
    infix += '-'            # Whatever lowest priority: force stack got empty
    operators.append('#')   # Remove IsEmpty

    for char in infix:
        if char.isdigit():
            postfix += char
        elif char == '(':
            operators.append(char)
        elif char == ')':
            while operators[-1] != '(':
                postfix += operators[-1]
                operators.pop()
            operators.pop() # pop (
        else:
            while precedence(operators[-1]) >= precedence(char):
                postfix += operators[-1]
                operators.pop()
            operators.append(char)  # higher than any in the stack
    return postfix
```

Right to Left associativity

- So far we handled the basic operators: + - * /
  - All of them are L-R associative, meaning that for two operators of **equal** precedence, the left-most operator will be handled/treated first
- In terms of the algorithm, we learned that 2 cases add to the stack
  - Rule A: Stack precedence(top) > precedence(cur), e.g. * vs +
  - Rule B: Stack precedence(top) == precedence(cur), e.g. + vs + and + vs -
- What about an operator like ^?
  - In mathematics: 2^3^4 is evaluated 2^**(3^4)** NOT (2^3)^4
  - This is **right to left** precedence, that is the **most right** ^ is handled first
  - ^ has higher precedence than + - * /
    - Rule A applies e.g. ^ vs + and ^ vs *
    - However, *and this is the only difference*, rule B doesn't apply (don't pop from stack): *Homework*

M-M Conversions

- Given that we have 3 types, we can have multiple possible conversions
  - **Infix to postfix**, Infix to Prefix
  - Postfix to Infix, Postfix to prefix
  - Prefix to Infix, Prefix to Postfix
- Practically, **infix to postfix** is important to ease **evaluating** expressions
- Feel free to think about some of these conversions and how to code them

In reality

- For simplicity, we assumed constraints on expressions
  - From an education perspective, what we have covered teaching the basic tenets of conversions, albeit without considering ALL possible operators
- In practice, an expression could be as complicated as: (-25+5log(11! *5^3^12))
  - Observe how numbers can be **multiple digits** in length
  - Numbers might be **negative**: so - can be either **binary or unary**
    - *Unary operators have higher precedence than* ^ + - * /
  - Observe **functions,** such as log and factorial
  - It is more of *implementation skills* rather than other stack concepts
  - One challenge in parsing: we need something to facilitate separation of the input:
    - (-25 + 5 log (11 ! * 5 ^ 3 ^ 2))
    - We call every parsed item a **token**, and this is the most annoying part to extract

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."