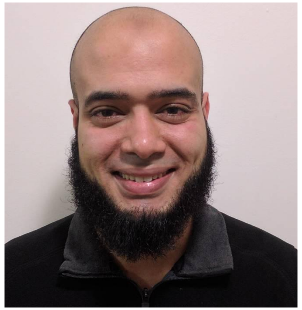
Data Structures

Infix to Postfix 1

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)

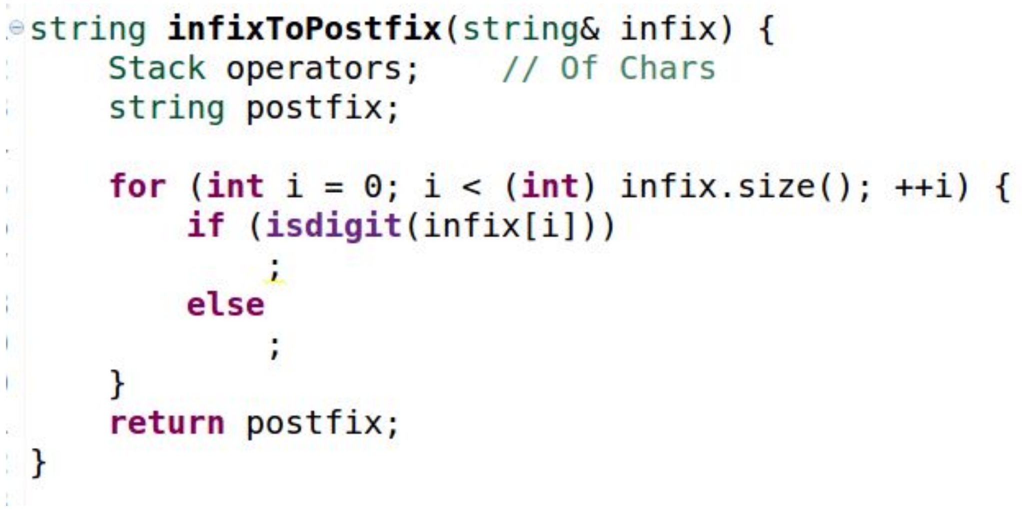


Infix to Postfix Conversion

- Task: Given an infix expression, convert to postfix expression
  - 1+2*3 ⇒ 123*+
- For simplicity, let's first consider these constraints
  - Input is a string **without** spaces. Output is a string
  - All numbers will be single digits and no sign. E.g. {0, 1, 2, ...9} but not -5 or +7
  - Operators are only: - + * / : observe all are left to right associativity
    - Remember: / * has higher precedence than + -
- Shunting-yard algorithm
  - The algorithm was invented by **Edsger Dijkstra** to do the conversion
  - We can both convert and evaluate using stacks
  - Parsed elements (numbers or operators) are called **tokens**

Infix to Postfix Algorithm

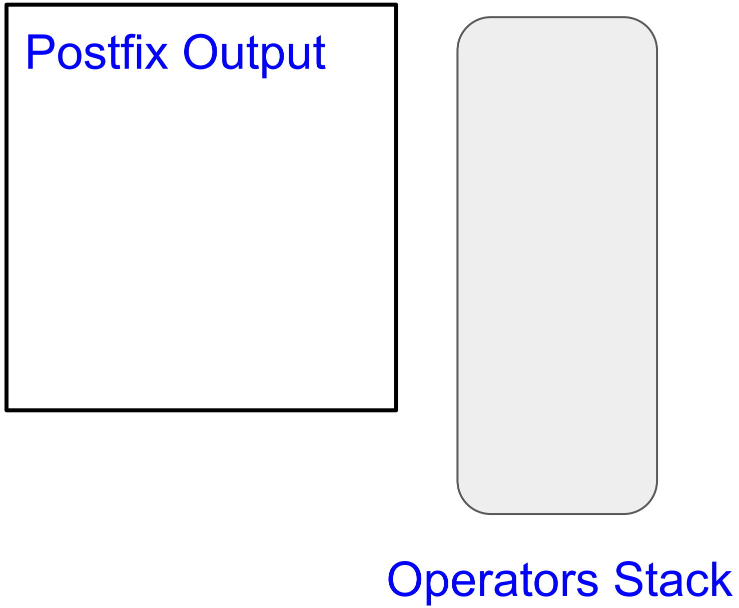
- We will maintain a string for the output and a stack of operators
  - So the stack will have only operators: + - * /
- We iterate on input, get next token
  - It is either a number (single digit) Or an operator



```
string infixToPostfix(string& infix) {
    Stack operators;    // Of Chars
    string postfix;

    for (int i = 0; i < (int) infix.size(); ++i) {
        if (isdigit(infix[i]))
            ;
        else
            ;
    }
    return postfix;
}
```

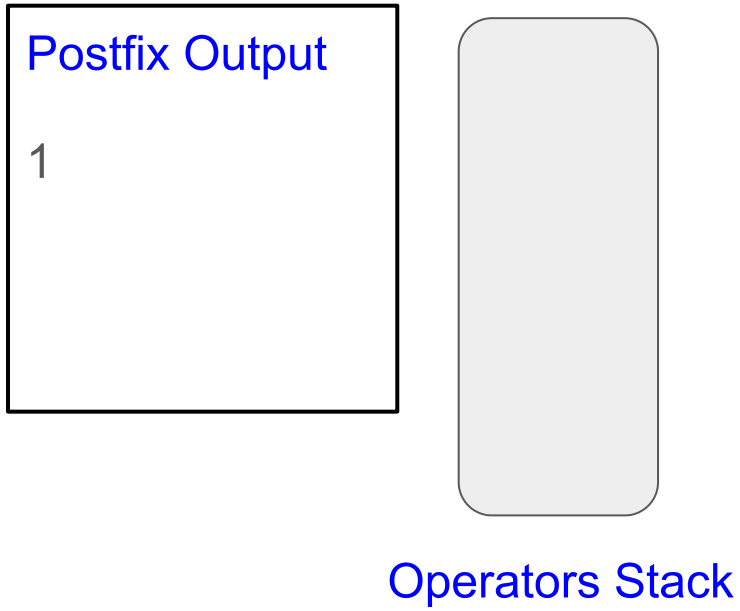
Parsing: 1+3*5-8/2

- Initially output postfix and stack empty
- Expected tokens are
  - 1
  - +
  - 3
  - *
  - 5
  - -
  - 8
  - /
  - 2
- Let’s iterate token by token



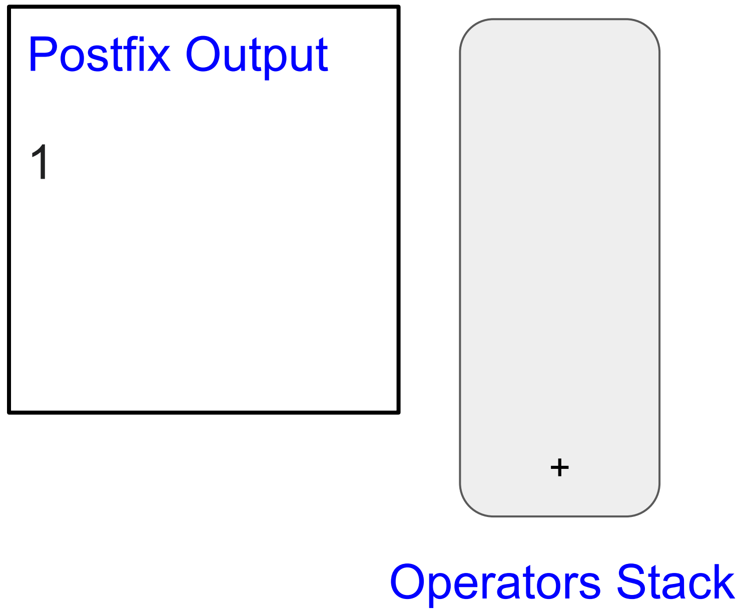
Parsing: 1+3*5-8/2

- Current Token 1
  - Digit
- Rule #1: If digit, add to output



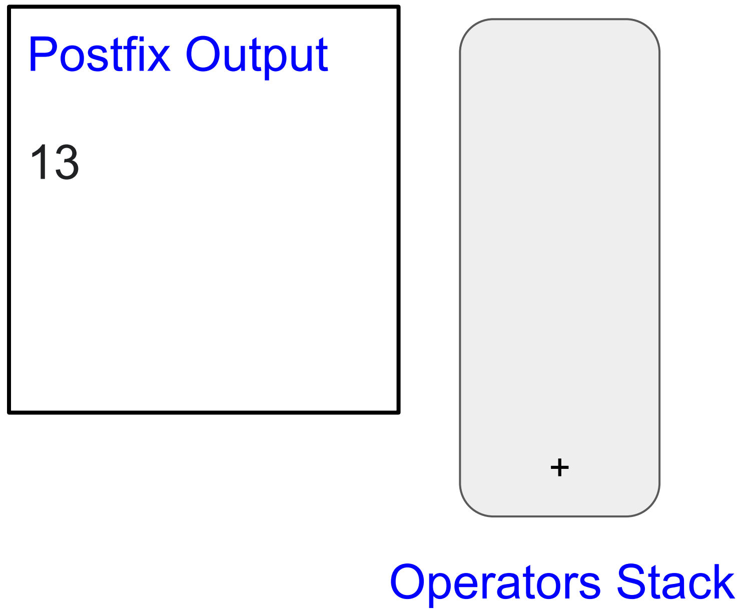
Parsing: 1**+**3*5-8/2

- Current Token +
  - Operator
- Rule #2: If operator and empty stack, push in the stack



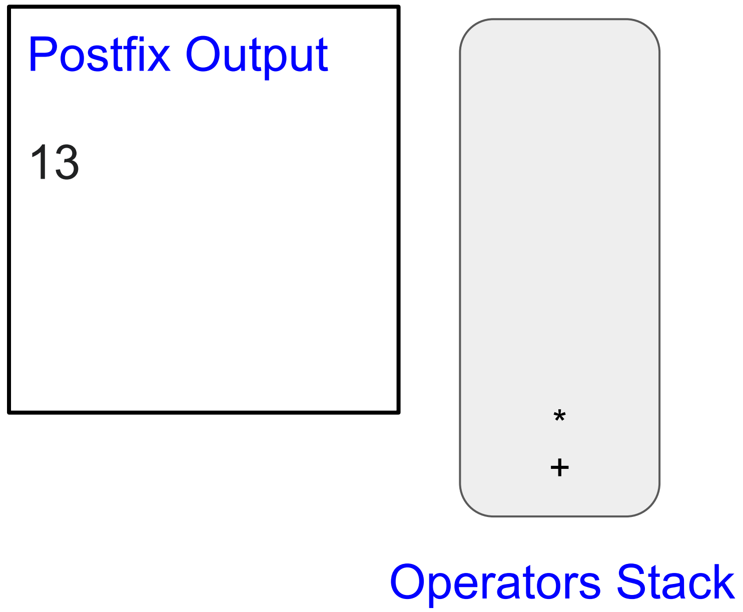
Parsing: 1+**3***5-8/2

- Current Token 3
  - Digit
- Rule #1: If digit, add to output



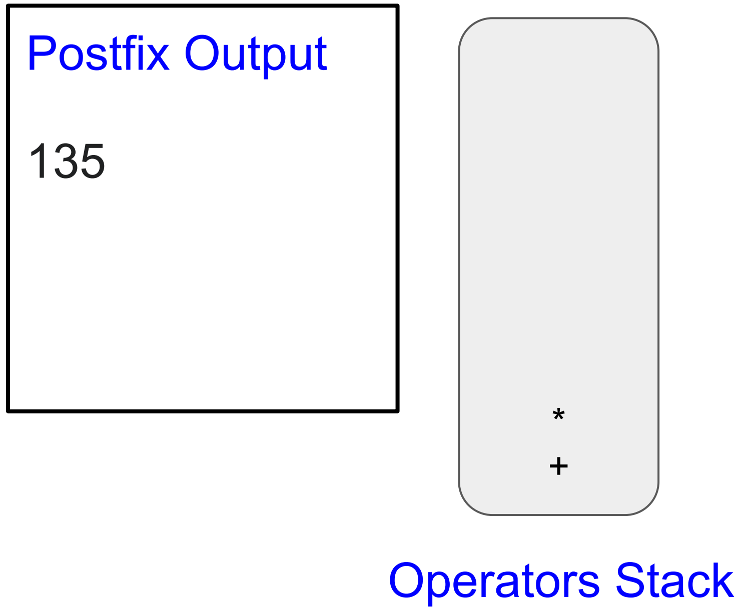
Parsing: 1+3*5-8/2

- Current Token *
  - Operator
- Rule #3: if the current operator **higher precedence** than top of stack, just add it to the stack



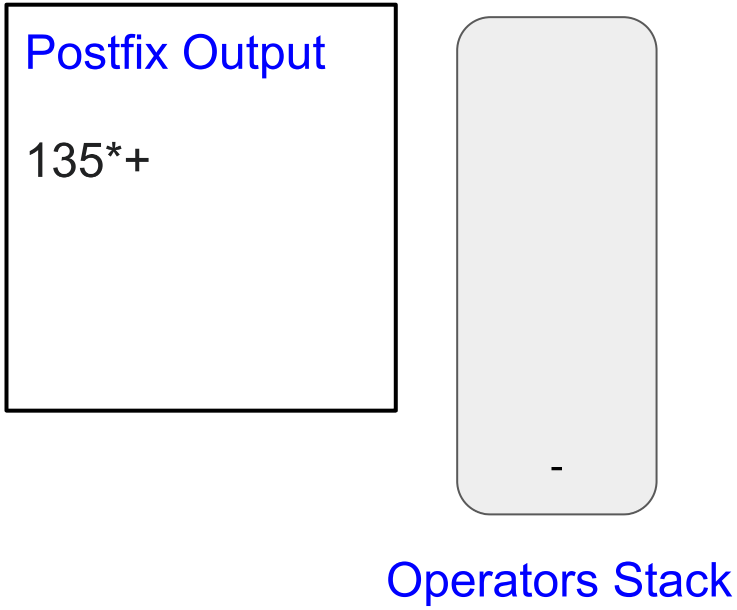
Parsing: 1+3***5**-8/2

- Current Token 5
  - Digit
- Rule #1: If digit, add to output



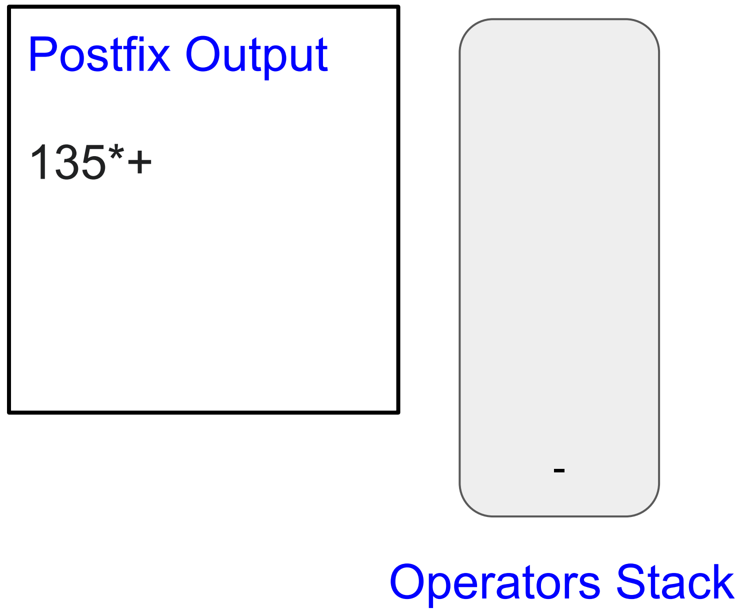
Parsing: 1+3*5-8/2

- Current Token -
  - Operator
- Rule #4: as long as precedence (cur) <= top, pop top and add to postfix
  - - vs * ? Smaller. Pop
  - - vs +? Equal. Pop
- Finally, add current token to the stack



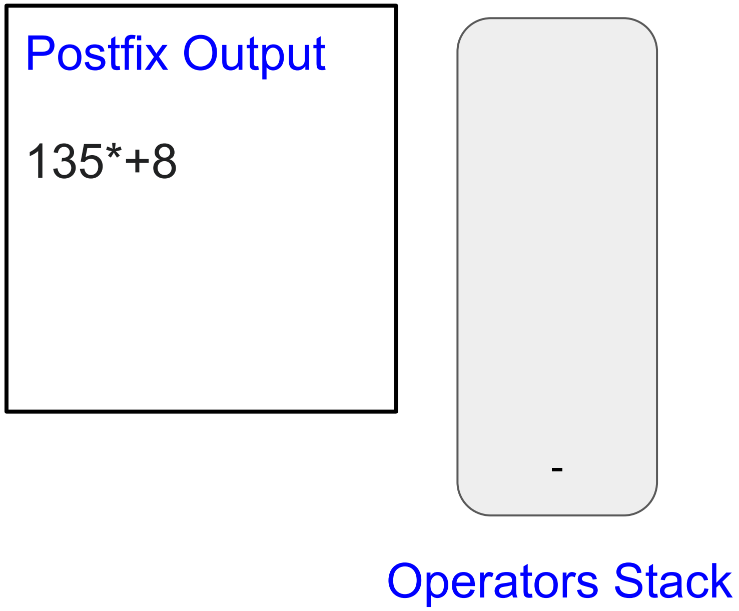
Parsing: 1+3*5-8/2

- Why popped *?
  - If it has **higher** precedence than current, then it must be applied before -
  - Now 3 and 5 will be multiplied: 3*5 = 15
- Why popped +?
  - If it has **equal** precedence to current and **left to right** associativity, then also it should be applied before -
  - Now 1 and 15 will be added: 1 + 15 = 16



Parsing: 1+3*5-**8**/2

- Current Token 8
  - Digit
- Rule #1: If digit, add to output



Parsing: 1+3*5-8/2

- Current Token /
  - Operator
- Rule #3: if the current operator **higher precedence** than top of stack, just add it

Postfix Output

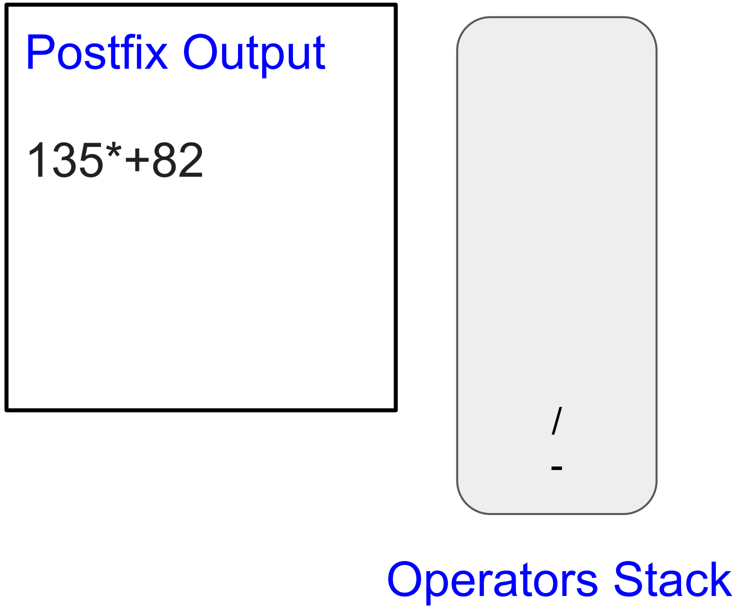
135*+8

/
-

Operators Stack

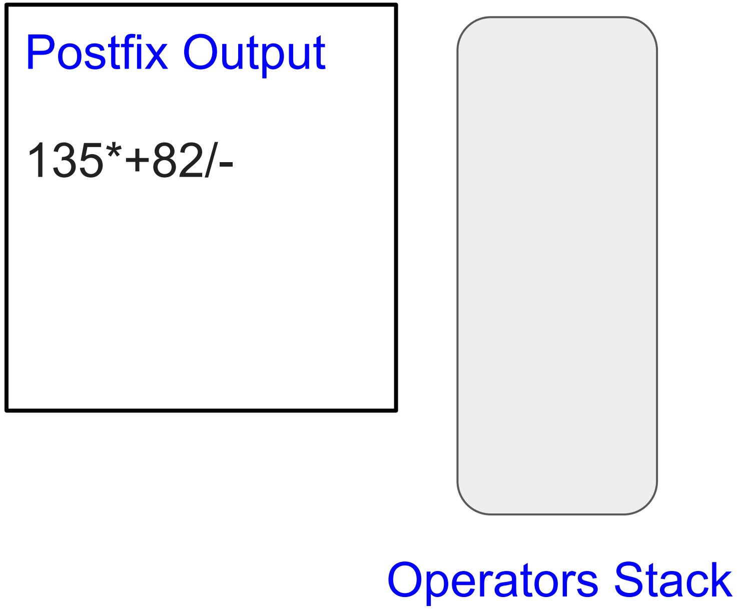
Parsing: 1+3*5-8/**2**

- Current Token 2
  - Digit
- Rule #1: If digit, add to output

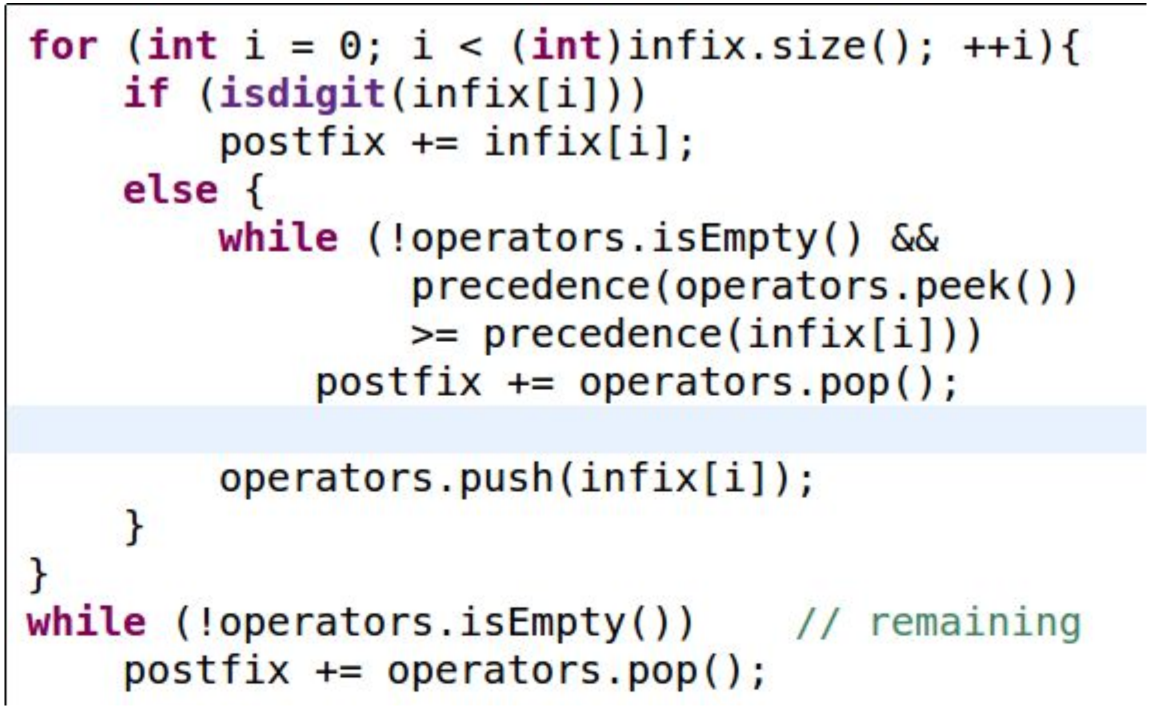


Parsing: 1+3*5-8/2

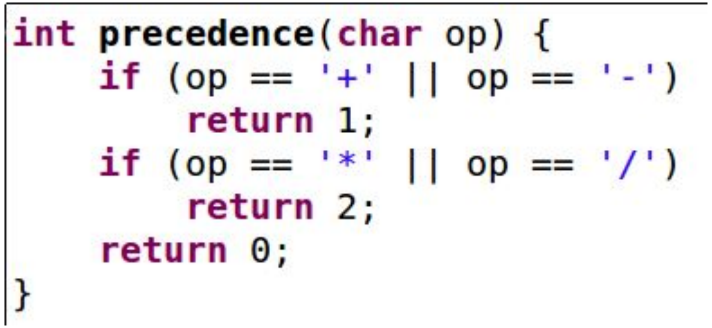
- Current Token NONE
- Rule #5: If finished, in order pop each item and add to postfix
- Final expression 135*+82/-
- Overall 5 simple rules
- Your turn: take 20 minutes coding it



Simplified Algorithm for Parsing: 1+3*5-8/2



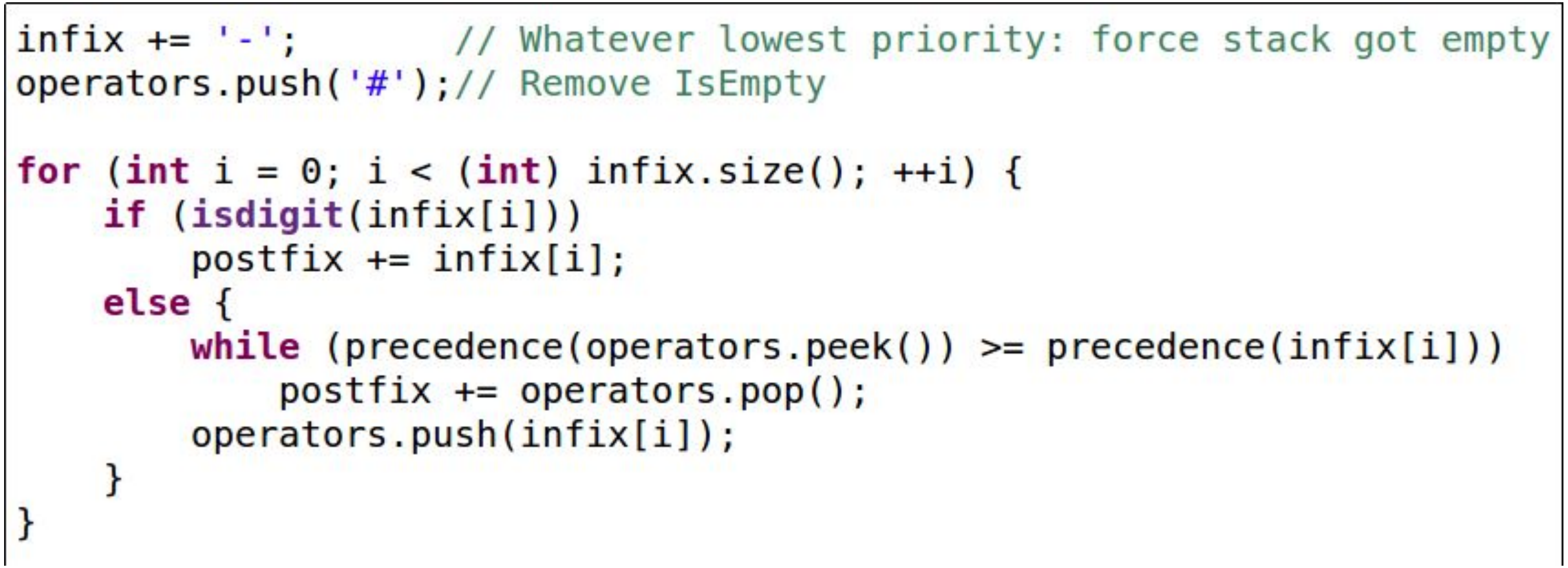
```
for (int i = 0; i < (int)infix.size(); ++i){
    if (isdigit(infix[i]))
        postfix += infix[i];
    else {
        while (!operators.isEmpty() &&
                precedence(operators.peek())
                >= precedence(infix[i]))
            postfix += operators.pop();

        operators.push(infix[i]);
    }
}
while (!operators.isEmpty())    // remaining
    postfix += operators.pop();
```



```
int precedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}
```

Smaller Code!

- Find 2 trivial changes that will result in:
  - Removing IsEmpty Check
  - Removing the last while loop!



```
infix += '-';        // Whatever lowest priority: force stack got empty
operators.push('#');// Remove IsEmpty

for (int i = 0; i < (int) infix.size(); ++i) {
    if (isdigit(infix[i]))
        postfix += infix[i];
    else {
        while (precedence(operators.peek()) >= precedence(infix[i]))
            postfix += operators.pop();
        operators.push(infix[i]);
    }
}
```

What is the time complexity?

- It seems we have a for loop, inside it a while loop
  - Intuitively this is O(n^2)
  - No. The devil in the details
- The maximum number of operators added in the stack is O(n)
  - And each will be removed once
  - So added once and removed once
- In fact, the code behaves like 2-3 parallel linear loops
  - E.g. 3N operations
  - Verify deeply and make sure you got it

Your Turn:

- Simulate on the code: 1+3*5-8/2
- By hand, convert 2+3*4-5*6+7 and compare with the algorithm output
- Think for 15 minutes: What if we have ( )
    - Recall, they have higher order
    - 2+(3*(4-5*2)*(9/3+6))
    - We know each expression () is independent on outsiders
        - Kind of a separate sub-problem!

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."