Level: Hard

Given an arithmetic expression with *,/,- & + operators and single digit numbers, evaluate it and return the result.

For example,
1 + 2 / 1 + 3 * 2 ==> 9

Questions to Clarify:
Q. Can the expression have parentheses - '()'?
A. No, it will only have those 4 operators of single digit numbers.

Q. Can we assume the input to be an array of characters?
e.g, ['1','+','2','\','1'..]
A. Yes that's fine.

Q. When we divide two numbers, is the result an integer or can it be a fraction?
E.g, ½ is 0 or 0.5?
A. Depends on your language. Here, we will use integer arithmetic, so ½ will be 0.

Q. Can we assume that the expression is valid?
A. Yes, assume that the expression is valid.

Q. If the expression is empty, can I return 0?
A. Yes.

Solution:
In arithmetic expression evaluation, we use 2 stacks - an operator stack and an operand stack. Before that, we need to look at operator precedence. Multiplication (*) and Division(/) have higher precedence than + and -. The precedence is as follows:

```
Precedence 2 -> / , *  (/ and * have same precedence)
Precedence 1 -> + , -
```

Back to using 2 stacks - operator stack and operand stack. We iterate through the expression. If we encounter a number, we put it on the operand stack. If we encounter an operator, we put it on the operator stack. In our earlier example, let's say we have gone through `1,+,2,/,1`. Our stacks look as follows:

```
Operand Stack:  1 -> 2 -> 1          <-- top of stack
Operator Stack: + -> /               <-- top of stack
```

Now, we encounter a +. If we see a higher or equal precedence on top of the stack, we evaluate it before inserting this operator. So, we pop `/` from the operator stack, pop `1` and `2` from the operand stack and evaluate '`2 / 1`'. The result is `2`, which we pop back into the operand stack. Now the stacks are as follows:

```
Operand Stack:  1 -> 2              <-- top of stack
Operator Stack: +                   <-- top of stack
```

We look at the top of the operator stack again. There is an operator that has equal precedence as +, so we evaluate the top operator again. The stack looks as follows now:

```
Operand Stack:  3                   <-- top of stack
Operator Stack:                     <-- top of stack
```

Now, there is no operator with higher precedence than +, so you can push the + operator. The stack will look as follows:

```
Operand Stack:  3                 <-- top of stack
Operator Stack: +                 <-- top of stack
```

After we've gone through the expression, we end up with the following:

```
Operand Stack:  3 -> 3 -> 2       <-- top of stack
Operator Stack: + -> *            <-- top of stack
```

After the loop, we can pop each operator, evaluate it and push the result back.
We do this until there is nothing left on the operator stack.
After evaluating the *,

```
Operand Stack:  3 -> 6          <-- top of stack
Operator Stack: +                <-- top of stack
```

After evaluating the + operator, there is only one element left in the operand stack: 9. The operator stack is empty. At the end, there should only be one number left, and that is the result.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a few
lines quickly to plan out your solution. Your actual solution should be in a real
language and use good syntax.)

Helper Functions that you may not have to implement:
process() function - evaluates the operator on top of the operator stack and
                     pushes the result onto the operand stack.

precedence() function - returns the precedence of an operator

Main function:
init operand, operator stack
loop through character ch in expression:
    if ch is operand:
          push into operand stack
```

```
    if ch is operator:
         while precedence(top of operator stack) >= precedence(ch)
               run process() function to evaluate the top operator
         push ch to operator stack

while operator stack is not empty
   run process() function

At the end result should be the only number on the operand stack.
```

Test Cases:
Edge Cases: Empty expression, single number in expression
Base Cases: Single operation (`1+2`, `1*2`)
Regular Cases: Multiple operators

Time Complexity: O(n)

Space Complexity: O(n) because we store a copy of the operator/operands in the stacks

```
public static int evaluate(char[] expression) {
    if (expression == null || expression.length == 0)
        return 0;

    Stack<Integer> operand = new Stack<>();
    Stack<Character> operator = new Stack<>();
    for (char ch : expression) {
        if (isOperand(ch))
            operand.push(ch-'0');
        else if (isOperator(ch)) {
            while (!operator.isEmpty()
                      && precedence(operator.peek()) >= precedence(ch)) {
                process(operator, operand);
            }
            operator.push(ch);
        }
    }

    while (!operator.isEmpty()) {
        process(operator, operand);
    }

    return operand.pop();
}

/*
 * Helper functions. Ask the interviewer if they want you to implement
 * these.
```

```
 */

private static boolean isOperand(char ch) {
    return (ch >= '0') && (ch <= '9');
}


private static boolean isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}

private static int precedence(char ch) {
    switch(ch) {
        case '/':
        case '*': return 2;
        case '+':
        case '-': return 1;
        default: throw new IllegalArgumentException("Invalid operator: " + ch);
    }
}

private static void process(Stack<Character> operator, Stack<Integer> operand) {
    int num2 = operand.pop();
    int num1 = operand.pop();

    char op = operator.pop();

    int result = 0;

    switch(op) {
        case '/': result = num1 / num2;
            break;
        case '*': result = num1 * num2;
            break;
        case '+': result = num1 + num2;
            break;
        case '-': result = num1 - num2;
            break;
    }
    operand.push(result);
}
```

Level: Hard

Given an arithmetic expression with *,/,- & + operators and single digit numbers, evaluate it and return the result.

The expression can also contain parenthesis.

For example,
1 + 2 / 1 + 3 * 2 ==> 9
1 + (1 + 3) * 2 ==> 9
1 + 2 / (1 + 3) * 2 ⇒ 1

Solution:

As shown in the video, we add a few lines to the previous solution. The added lines are in bold:

```
public static int evaluateWithParenthesis(char[] expression) {
    if (expression == null || expression.length == 0)
        return 0;

    Stack<Integer> operand = new Stack<>();
    Stack<Character> operator = new Stack<>();
    for (char ch : expression) {
        if (isOperand(ch))
            operand.push(ch-'0');
        else if (isOperator(ch)) {
            while (!operator.isEmpty()
                            && precedence(operator.peek()) >= precedence(ch)) {
                process(operator, operand);
            }
            operator.push(ch);
        } else if (ch == '(') {
            operator.push(ch);
        } else if (ch == ')') {
            while (operator.peek() != '(') {
                process(operator, operand);
            }
            operator.pop();
        }
    }

    while (!operator.isEmpty()) {
        process(operator, operand);
    }

    return operand.pop();
```

```
}

/*
 * Helper functions. Ask interviewer if they want you to implement.
 */

private static boolean isOperand(char ch) {
    return (ch >= '0') && (ch <= '9');
}


private static boolean isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}

private static int precedence(char ch) {
    switch(ch) {
        case '/':
        case '*': return 2;
        case '+':
        case '-': return 1;
        case '(':
        case ')': return 0;
        default: throw new IllegalArgumentException("Invalid operator: " + ch);
    }
}

private static void process(Stack<Character> operator, Stack<Integer> operand) {
    int num2 = operand.pop();
    int num1 = operand.pop();

    char op = operator.pop();

    int result = 0;

    switch(op) {
        case '/': result = num1 / num2;
            break;
        case '*': result = num1 * num2;
            break;
        case '+': result = num1 + num2;
            break;
        case '-': result = num1 - num2;
            break;
    }
    operand.push(result);
}
```