CX1107 Data Structures and Algorithms **2020/21 Semester 2**

8: Advanced Data Structures

School of Computer Science and Engineering *Nanyang Technological University*

In this lecture, we will use the **transform-and-conquer** strategy to solve the algebraic expression evaluation problem.

8.1 Algebraic Expressions

A simple algebraic expression consists of operands and operators. $+$, $-$, $\times$ and $\div$ are the operators. We may call them as binary operators that work with two operands. Normally, we write the operator between its operands. The expression is known as **infix expression**. For example, an infix expression is written as follow:

$$a + b \times (c - d) \times e \div f$$

To evaluate the infix expression, we need to know its left-to-right association which the evaluation is from left to right and operator precedence rules which $\times$ and $\div$ have higher precedence than $+$ and $-$. If we also consider parentheses, the evaluation will be more complicated. Can you design an algorithm to evaluate an infix expression?

The direct approach will be very tedious and less efficient. We need to do multiple scanning on the expression to find the next operation. To be more efficient, we need to transform the infix expression into a **postfix expression** which the operators are written after its operands. It is also known as **reverse Polish expression** which was proposed in 1950s. For example, the postfix expression of "$2 \times (3 + 4)$" is "$2\ 3\ 4\ + \times$".

8.1.1 Evaluation A Postfix Expression

The conversion will be discussed in the next session. Let's assume that we have a postfix expression. If we would like to evaluate the expression, we first need to create an empty stack. The stack is used to store the operands. Since it is **postfix** expression, we will retrieve two operands from the stack when we read an operator. If the expression is an valid expression, the evaluation can be done in $\mathcal{O}(n)$.

1. Create an empty stack
2. Read the input of a postfix expression and tokenize the expression into operands and operators
3. Read tokens from left to right
4. If the token is an operand, push it into the stack
5. If the token is an operator, pop two operands from the stack
6. Evaluate it
7. Push the result back to the stack
8. Repeat the step 3 to 7 until all tokens are read

Algorithm 1 Evaluation Postfix Expression

```
function ExePost(String postfix)
    create a Stack S
    for each character c in postfix do
        if c is an operand then
            push(c, S)
        else
            operand1 ← pop(S)
            operand2 ← pop(S)
            result ← Evaluate(operand2, c, operand1)
            push(result, S)
```

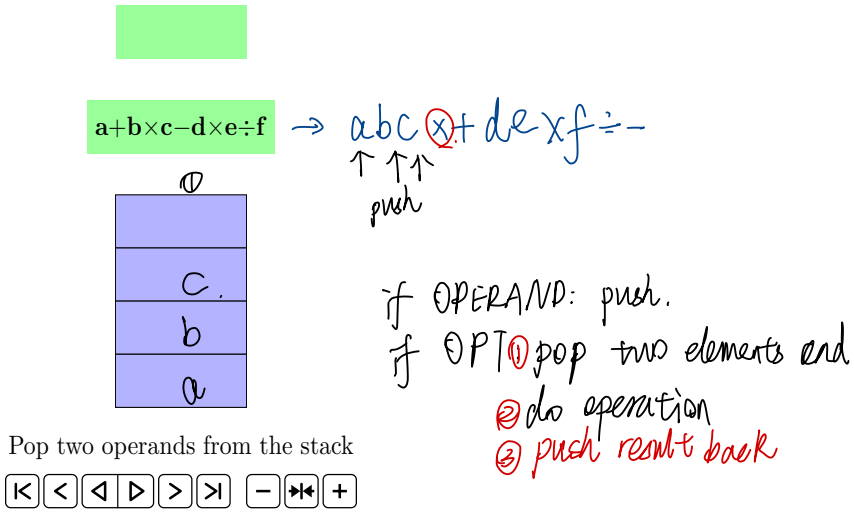

Figure 8.1: Evaluating a postfix expression of "$a\ b\ c\ \times\ +\ d\ e\ \times\ f\ \div\ -$"

8.1.2 Converting An Infix Expression to A Postfix Expression

From section 8.1.1, we can observe that evaluating postfix expressions instead of infix expressions can reduce the memory access and improve the computational efficiency. In this section, let's learn the conversion algorithm from an infix expression to a postfix expression. Infix expression is in the format of ⟨operand⟩⟨operator⟩⟨operand⟩. We need to 'move' the operator after its operands. It implies that we need a stack to store operators of the infix expression temporarily. When we read an operand, we can simply put it in the output. When we read an operator, we need to 'peek' the stack due to operator precedence rules.

Algorithm 2 Infix Expression to Postfix Expression

```
function In2Post(String infix, String postfix)
    create a Stack S
    for each character c in infix do
        if c is an operand then
            postfix ← c
        else if c = ')' then
            while peek(S) ≠ '(' do
                postfix ← pop(S)
            pop(S)
        else if c = '(' then
            push(c,S)
        else                                   ▷ c is an operator or left parenthesis
            while S ≠ empty && peek(S) != '(' && precedence of peek(S) ≥ precedence of c do
                postfix ← pop(S)
            push(c,S)
    while S is not empty do
        postfix ← pop(S)
```

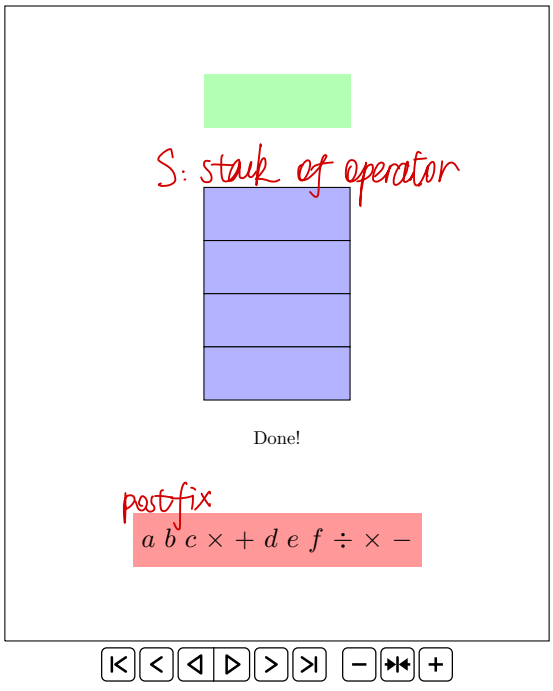

Figure 8.2: Convert an infix expression, $a + b \times c - d \times (e \div f)$, to a postfix expression

It is noted that prefix conversion is similar. Here I would like to leave it to you for practice. Please understand the fundamental concepts behind first. No point to blindly memorize the algorithm.