# Data Structures & Algorithms
(PCC-CS 301)

Dr. Debashis Das
Associate Professor
Department of CSE
Techno India University, Kolkata

# Topics Covered

1. Application of Stack
   a. Conversion of Infix to Postfix expression
   b. Evaluation of Postfix expression

# Arithmetic Expression

- Different operations
  - Binary operations
    - Exponentiation
    - Multiplication , Division
    - Addition , Subtraction
  - Precedence

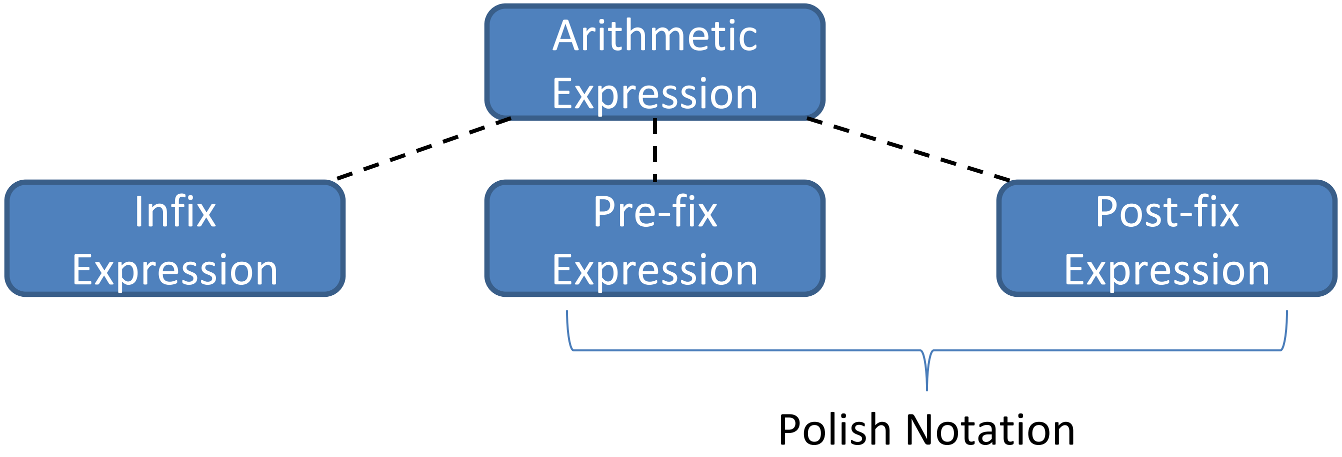
| Any operations inside parenthesis | Highest |
|---|---|
| Exponentiation | |
| Multiplication , Division | |
| Addition , Subtraction | Lowest |

  - Example

2^3+5*2^2-12/6 = 8+5*4-12/6 = 8+20-2 = 26

# Arithmetic Expression

- Representation



Arithmetic Expression

Infix Expression

Pre-fix Expression

Post-fix Expression

Polish Notation

- Infix expression
  - Operators should lie between two operands (for binary operations)

    Ex. - (A+B) / (C+D)

# Arithmetic Expression

- Representation
  - Polish notation (proposed by Jan Lukasiewicz)
    - Pre-fix expression
      - Any operator will be placed before its associated operands

        Ex. - /+AB+CD
    - Post-fix expression
      - An operator will be placed after its associated operands
      - Also termed as reverse polish notation
      - Used for arithmetic expression evaluation in computer

        Ex. - AB+CD+/

There will be no parenthesis used in Polish notations

# Arithmetic Expression

- Why Polish notation?
  - Arithmetic expression (infix)
    - It requires to express in proper parenthesization to evaluate

      (A+B)*C and A+(B*C) are different

    - Polish notation of expression
      - Can be evaluated without parenthesis
      - Useful in evaluating arithmetic expression in computer system
      - Stack data structure is sufficient to perform the job

      Postfix: AB+C* and ABC*+

# Polish Notation

- Infix to Prefix conversion (manually)

Infix Expression: A + B * (C - D) / E + F ^ G + H

Pre-fix conversion:

A + B * (C - D) / E + F ^ G + H
= A + B * [- C D] / E + F ^ G + H
= A + B * [- C D] / E + [^ F G] + H
= A + [* B – C D] / E + [^ F G] + H
=A + [/ * B – C D E] + [^ F G] + H
=[+ A / * B – C D E] + [^ F G] + H
=[+ + A / * B – C D E ^ F G] + H
=+ + + A / * B – C D E ^ F G H

+ + + A / * B – C D E ^ F G H

# Polish Notation

- Infix to Postfix conversion (manually)

Infix Expression: A + B * (C - D) / E + F ^ G + H

Pre-fix conversion:

A + B * (C - D) / E + F ^ G + H
= A + B * [C D -] / E + F ^ G + H
= A + B * [C D -] / E + [F G ^] + H
= A + [B C D - *] / E + [F G ^] + H
=A + [B C D - * E /] + [F G ^] + H
=[A B C D - * E / +] + [F G ^] + H
=[A B C D - * E / + F G ^ +] + H
= A B C D - * E / + F G ^ + H +

A B C D - * E / + F G ^ + H +

# Solving Arithmetic Expression

- Solve by computer (involves two steps )

  - Conversion of infix to postfix expression
  - Solve postfix expression

Both the methods require STACK to solve

# Solving Arithmetic Expression

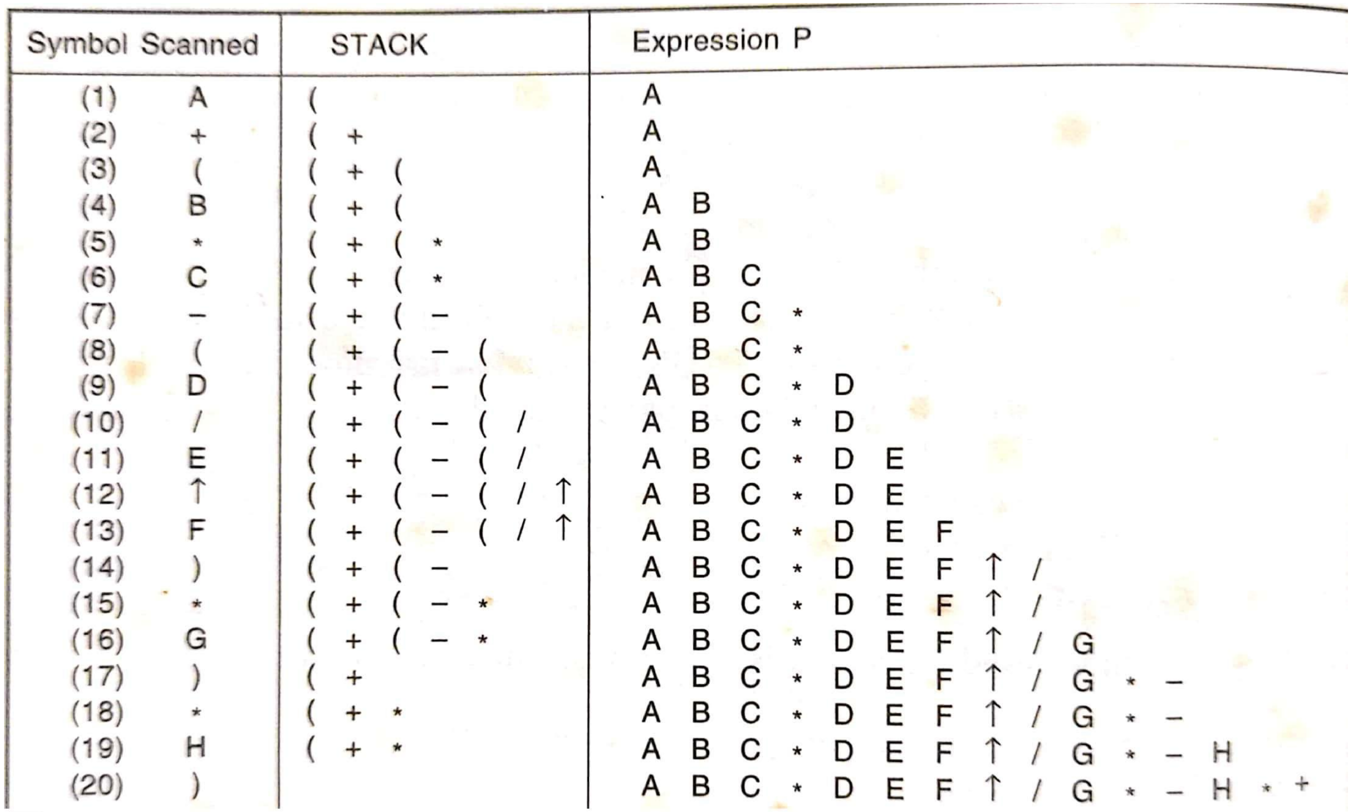
- Infix to Postfix conversion (algorithm)

```
Postfix ( Q )   // Q is the infix arithmetic expression
{
  step 1: Push "(" into the stack and add ")" to the end of Q
  step 2: Scan Q from left to right and repeat steps 3 to 6 for each element of Q
             until the Stack is empty
  step 3: If an operand is encountered, add it to P   // P is output postfix expression
  step 4: If a left parenthesis is encountered, push it into Stack
  step 5: if an operator is encountered then
            5.1. Repeatedly pop from Stack and add to P each operator which has same or
                 higher precedence than the encountered operator
            5.2. Push the encountered operator into Stack
  step 6: If a right parenthesis is encountered then
            6.1. Repeatedly pop from Stack and add to P each operator until a left
                 parenthesis is encountered
            6.2.  Remove the left parenthesis (but do not add to P )
  step 7: Exit
}
```

# Solving Arithmetic Expression

- Infix to Postfix conversion (mechanism)

Infix expression: A + ( B * C – ( D / E ^ F ) * G ) * H



| Symbol Scanned | | STACK | Expression P |
|---|---|---|---|
| (1) | A | ( | A |
| (2) | + | ( + | A |
| (3) | ( | ( + ( | A |
| (4) | B | ( + ( | A B |
| (5) | * | ( + ( * | A B |
| (6) | C | ( + ( * | A B C |
| (7) | − | ( + ( − | A B C * |
| (8) | ( | ( + ( − ( | A B C * |
| (9) | D | ( + ( − ( | A B C * D |
| (10) | / | ( + ( − ( / | A B C * D |
| (11) | E | ( + ( − ( / | A B C * D E |
| (12) | ↑ | ( + ( − ( / ↑ | A B C * D E |
| (13) | F | ( + ( − ( / ↑ | A B C * D E F |
| (14) | ) | ( + ( − | A B C * D E F ↑ / |
| (15) | * | ( + ( − * | A B C * D E F ↑ / |
| (16) | G | ( + ( − * | A B C * D E F ↑ / G |
| (17) | ) | ( + | A B C * D E F ↑ / G * − |
| (18) | * | ( + * | A B C * D E F ↑ / G * − |
| (19) | H | ( + * | A B C * D E F ↑ / G * − H |
| (20) | ) | | A B C * D E F ↑ / G * − H * + |

# Solving Arithmetic Expression

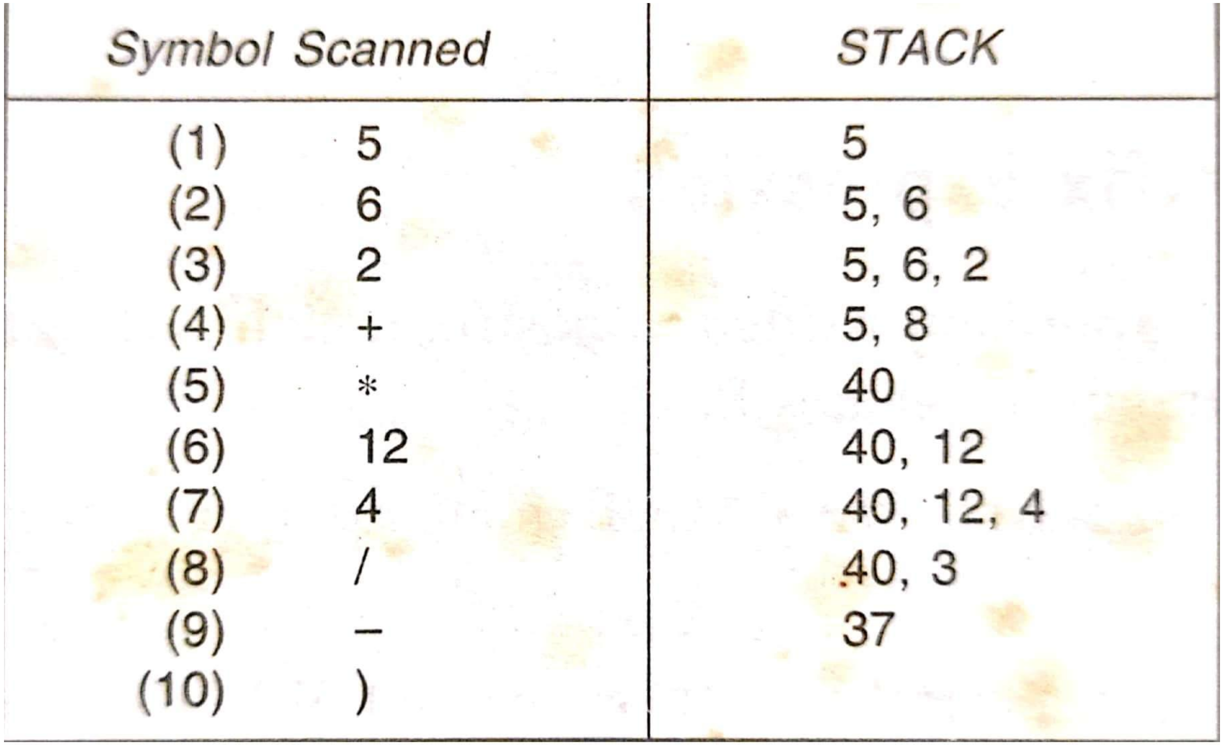
- Evaluation of Postfix expression (algorithm)

```
Evaluate_Postfix ( P )   // P is the post-fix expression
{
  step 1: Add ")" at the end of P
  step 2: Scan P from left to right and repeat steps 3 and 4 for each element of P
             until ")" is encountered
  step 3: If an operand is encountered, push it into Stack
  step 4: if an operator is encountered then
            5.1. Pop  two top elements from Stack, where A is the top element and B is the
                   next-to-top element
            5.2. Evaluate B (operator) A
            5.3. Push the result of back to Stack
  step 5: Set VALUE equal to the top element on Stack
  step 6: Exit
}
```

# Solving Arithmetic Expression

- Evaluation of Postfix expression (mechanism)

Post-fix expression: 5 6 2 + * 12 4 / -



| Symbol Scanned | | STACK |
|---|---|---|
| (1) | 5 | 5 |
| (2) | 6 | 5, 6 |
| (3) | 2 | 5, 6, 2 |
| (4) | + | 5, 8 |
| (5) | * | 40 |
| (6) | 12 | 40, 12 |
| (7) | 4 | 40, 12, 4 |
| (8) | / | 40, 3 |
| (9) | – | 37 |
| (10) | ) | |

Queries?