Expression Evaluation Algorithms

Homogeneous Brackets Validity

```
Input = Infix Expression
Output = 1 means valid 0 means invalid expression
Create a stack that can hold opening brackets in it
While (Scan the infixstring from left to left till the end)
{
        get next_character from infixstring
        if ( next_character is opening bracket '(' )
        {
                push next_character into stack
        }
        if ( next_character is closing bracket ')' )
        {
                if (stack is empty)
                        return 0
                stacktop = pop from stack
                if (stacktop is not opening bracket '(')
                {
                        Return 0
                }

        }

}
if (stack is empty)
        return 1
else
        return 0
```

****

Heterogeneous Brackets Validity

```
Input = Infix Expression
Output = 1 means valid 0 means invalid expression
Create a stack that can hold opening brackets in it
While (Scan the infixstring from left to left till the end)
{
        get next_character from infixstring
        if ( next_character is opening bracket '(' OR '{' OR '[')
        {
                push next_character into stack
        }
        if ( next_character is closing bracket ')' OR '}' OR ']')
        {
                if (stack is empty)
                        return 0

                stacktop = pop from stack
                if (stacktop and next_character are not matching pairs):
                      return 0
}
if (stack is empty)
        return 1
else
        return 0
```

****

Infix To Postfix Conversion

```
Input = infix string
Output = postfix string
Create a stack that can store operators in it

While (Scan the infix_string from left to right till the end)
{           get next_character from infix_string
            if (next_character is operand)
            {
                        Append next_character to postfix_string
            }
            else if (next_character is operator)
            {
                        while (stack is not empty AND precedence(stacktop) > precedence(next_character))
                        {
                                    pop the operator from stack and append it to postfix_string
                        }
                        if ( next_character is not ‘)’ )
                        {
                                    push next_character to stack
                        }
                        else if ( next_character is ‘)’ )
                        {
                                    while (stack is not empty AND stacktop != '(' )
                                    {
                                                pop from stack and append it to postfix_string
                                    }

                                    pop from stack                //it will pop ‘(‘ bracket
                        }
            }
}
while(stack is not empty)
{
            pop the operator from stack and append it to postfix_string
}
```

Some Rules about Precedence and about push and pop of operators in stack

- Closing bracket can never be pushed in stack.
- If operand then append in postfix string.
- If operator then push in stack.
- A low precedence operator can never on top of high precedence operator in the stack.
  E.g. if in stack, the operator are in this order from bottom to top
- + / ………this is right but
- / + ………this is wrong and
- / / ……..this is also wrong but what about this
- + / (+ ……….this is also right
- It means that this rules implements in the stack from the start of an ‘(‘ till the next ‘(‘ occurs.
- If operator is opening bracket then push it in stack but If operator is closing bracket ‘)’ then pop the stack until ‘(‘ not found in the stack.
- After popping all the operators until ‘(‘also pop that ‘(‘.

****

Postfix Expression Evaluation

```
Input = postfix string
Output = result of postfix expression
Create a stack that can store operands in it
While (Scan the postfixstring from left to right till the end)
{       get next_character from postfix_string
        if (next_character is operand)
        {
                push next_charcter into stack.
        }
        else if (next_character is operator)
        {
                operand2 = pop operand from stack
                operand1 = pop operand from stack
                Perform operation on the operand1 and operand2 depending on the operator.
                Push the result of this operation in stack. }
}
Pop the result of "Postfix Expression Evaluation" from stack.
```

****

Infix To Prefix Conversion

Part (a) using two stack

```
Input = infix string
Output = prefix string
Create a stack that can store operators in it called it as operatorstack
Create a stack that can store operands in it called it as operandstack
While (Scan the infix_string from left to right till the end)
{
        get next_character from infix_string
        if (next_character is operand)
        {
                push next_character into operandstack
        }
        else if (next_character is operator)
        {
                while (operatorstack is not empty AND precedence(stacktop) >  precedence(next_character) )
                {
                        operand2 = pop the operand from operandstack
                        operand1 = pop the operand from operandstack
                        operator = pop the operator from operatorstack
                        concatenate operator+operand1+operand2
                        push the concatenated result into operandstack
                }
                if ( next_character is not ')' )
                {
                        push next_character to operatorstack
                }
                else if ( next_character is ')' )
                {
                         while (stack is not empty AND stacktop != '(' )
                        {
                                  pop from stack and append it to postfix_string
                        }

                        pop from stack             //it will pop '(' bracket

                }
        }
}
while(operatorstack is not empty)
{
        operand2 = pop the operand from operandstack
        operand1 = pop the operand from operandstack
        operator = pop the operator from operatorstack
        concatenate operator+operand1+operand2
        push the concatenated result into operandstack
}
Pop the operandstack and store it in prefix string.
```

Some Rules about Precedence and about push and pop of operators in stack

- Precedence Rules are Same as for Postfix conversion
- Its advantage is that you get same string as you draw through pen and paper.
- e.g. try to find out prefix expression for "A$B*C-D+E/F/(G+H)"
- Manually it comes "-*$ABC+D//EF+GH"
- Applying Part (a) the same result will be produced
- But applying part (b) "-*$ABC+D/E/F+GH" is produced.

****

Part (b) using one stack

```
Input = Infix string
Output = Prefix string
Create a stack that can store operators in it
Reverse the infix string, While reversing, replace ( with ) and ) with (.

While (Scan reversed infix string from left to right till the end)
{       get next_character from infix_string
        if (next_character is operand)
        {
                Append next_character to prefix_string
        }
        else if (next_character is operator)
        {
                while (stack is not empty AND precedence(stacktop) > precedence(next_character))
                {
                        pop the operator from stack and append it to prefix_string
                }
                if ( next_character is not ')' )
                {
                        push next_character to stack
                }
                else if ( next_character is ')' )
                {
                        while (stack is not empty AND stacktop != '(' )
                        {
                                pop from stack and append it to prefix_string
                        }

                        pop from stack          //it will pop '(' bracket
                }
        }
}
while(stack is not empty)
{
        pop the operator from stack and append it to prefix_string
}
Reverse prefix_string → Final Answer
```

Some Rules about Precedence and about push and pop of operators in stack

- Opening bracket can never be pushed in stack.
- If operand then append in prefix string.
- If operator then push in stack.
- A low precedence operator can never on top of high precedence operator in the stack. E.g. if in stack the operator are in this order from bottom to top
- + / ………this is right but
- / + ………this is wrong and
- / / ……..this is also wrong but what about this
- + /) + ……….this is also right
- It means that this rules implements in the stack from the start of an ')' till the next ')' occurs.
- If operator is closing bracket then push it in stack but If operator is opening bracket '(' then pop the stack until ')' not found in the stack.
- After popping all the operators until ')' found, also pop that ')'.

Disadvantage:

▫ You will not get exactly the same prefix expression as you find out manually.

****

Prefix Expression Evaluation Prefix Evaluation for expression produced in part (b) of Infix into Prefix Conversion

```
Input = prefix string
Output = result of prefix expression
Create a stack that can store operands in it
While (Scan the prefixstring from right to left till the end)
{       get next_character from prefix_string
        if (next_character is operand)
        {
                push next_charcter into stack.
        }
        else if (next_character is operator)
        {
                operand1 = pop operand from stack
                operand2 = pop operand from stack
                Perform operation on the operand1 and operand2 depending on the operator.
                Push the result of this operation in stack.
        }
}
Pop the result of "Prefix Expression Evaluation" from stack.
```

****

Advantage of Evaluating Expressions which are in Postfix and Prefix form ▫

No nesting of brackets

Both prefix and postfix form expressions can be evaluating in one pass (reading the expression only once) rather than multiple passes in Infix Expression thus improving the time complexity.