

Lab:07

Infix to Postfix Conversion

Theory:

Infix expression: The expression of the form “a operator b” ($a + b$) i.e., when an operator is in-between every pair of operands.

Postfix expression: The expression of the form “a b operator” ($ab+$) i.e., When every pair of operands is followed by an operator.

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.

Consider the expression: $a + b * c + d$

- The compiler first scans the expression to evaluate the expression $b * c$, then again scans the expression to add a to it.
- The result is then added to d after another scan.

The repeated scanning makes it very inefficient. Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix (or prefix) form before evaluation.

The corresponding expression in postfix form is $abc*+d+$. The postfix expressions can be evaluated easily using a stack.

Algorithm for infix to postfix conversion

1. Scan the infix expression **from left to right**.
2. Initialize an empty stack
3. If the scanned character is an operand, put it in the postfix expression.
4. Otherwise, do the following

- If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a ‘(’, then push the current operator onto the stack.
 - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.
5. If the scanned character is a ‘(’, push it to the stack.
 6. If the scanned character is a ‘)’, pop the stack and output it until a ‘(’ is encountered, and discard both the parenthesis.
 7. Repeat steps **3-6** until the infix expression is scanned.
 8. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
 9. Finally, print the postfix expression.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

char stk[50];
int top = -1;

int isEmpty()
{
    return top == -1;
}

int isFull()
{
    return top == MAX - 1;
}

char peek()    {  if
(isEmpty())
return -1;    return
stk[top];
}

char pop()
{
    if (isEmpty())
```

```

        return -1;

    char ch = stk[top];
    top--;
    return(ch);
}

void push(char oper)
{
    if (isFull())
        printf("Stack Full!!!!");
    else {
        top++;
        stk[top] = oper;
    }
}

int checkIfOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

int precedence(char ch)
{
    switch (ch)
    {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    case '^':
        return 3;
    }
    return -1;
}

void convertInfixToPostfix(char* expression)
{

```

```
int i, j = 0;  
char postfix[MAX];
```

```

    for (i = 0; expression[i]; ++i)
    {
        if (checkIfOperand(expression[i]))
            postfix[j++] = expression[i];

        else if (expression[i] == '(')
            push(expression[i]);

        else if (expression[i] == ')')
        {
            while (!isEmpty() && peek() != '(')
                postfix[j++] = pop();
            if (!isEmpty() && peek() != '(')
                return; // Invalid expression
            else
                pop();
        }
        else
        {
            while (!isEmpty() && precedence(expression[i]) <=
                precedence(peek()))
                postfix[j++] = pop();
            push(expression[i]);
        }
        while (!isEmpty())
            postfix[j++] = pop();

        postfix[j] = '\\0';    // Null-terminate the postfix expression

        printf("Postfix Expression: %s\\n", postfix);
    }

int main()
{
    char expression[50];
    printf("Enter the infix expression:");
    scanf("%s", expression);
    convertInfixToPostfix(expression);
    return 0;
}

```

Discussion and Conclusion

In this lab report on Infix to Postfix conversion in Data Structures and Algorithms (DSA), we implemented and analyzed the conversion process using stacks. The study highlighted the limitations of infix notation in expression evaluation and the advantages of postfix notation, which eliminates the need for parentheses and follows a straightforward evaluation order. By using the stack data structure, we efficiently converted infix expressions to postfix form while maintaining operator precedence and associativity. The results demonstrated that stack-based conversion is both systematic and efficient. In conclusion, mastering this technique is essential for applications like expression evaluation in compilers and calculators.

Biraj Pandey

ACE079BCT023