Compiler Design

Week - 10

Topic : Implementing Intermediate Code Prefix and Postfix

Aim:
Implementation of intermediate code generation (Postfix & Prefix)

Algorithm:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
    1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
    2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until the infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Code :

Prefix :

```cpp
#include<bits/stdc++.h>
using namespace std;

bool isOperator(char i){
    if(i == '+' || i == '-' ||i == '/' ||i == '*'){
        return true;
    }
    return false;
}
void GetAnswer(stack<int>&store,char op){
    int fst = store.top();
    store.pop();
    int sc = store.top();
    store.pop();
    if(op == '+'){
        fst = fst + sc;
    }else if(op == '-'){
        fst = fst - sc;
    }else if(op == '*'){
        fst = fst * sc;
    }else if(op == '/'){
        fst = fst / sc;
    }
    store.push(fst);
}
int main(){
    string s;
    getline(cin,s);
    stack<int> store;
    int a = 0,j = 0;
    char op = ' ';
    bool chr = false,digit = true;
    for(int i = s.size() - 1; i >= 0; i--){
        if(s[i] == ' ' && digit){
            store.push(a);
            a = 0;
            j = 0;
        }else{
            if(!isOperator(s[i]) && s[i] != ' '){
                a = a + (s[i] - '0') * pow(10,j);
                j++;
                digit  = true;
            }else{
                if(isOperator(s[i])){
                    op = s[i];
                    digit = false;
                    GetAnswer(store,op);
                }
            }
        }
    }
    if(!store.empty()){
        cout << store.top() << endl;
    }
    return 0;
}
```

Postfix:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Stack
{
    int top;
    unsigned capacity;
    int *array;
};
struct Stack *createStack(unsigned capacity)
{
    struct Stack *stack = (struct Stack *)
        malloc(sizeof(struct Stack));
    if (!stack)
        return NULL;
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int *)malloc(stack->capacity *
                                 sizeof(int));
    return stack;
}
int isEmpty(struct Stack *stack)
{
    return stack->top == -1;
}
char peek(struct Stack *stack)
{
    return stack->array[stack->top];
}
char pop(struct Stack *stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return '$';
}
void push(struct Stack *stack, char op)
{
    stack->array[++stack->top] = op;
}
int isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') ||
           (ch >= 'A' && ch <= 'Z');
}
int Prec(char ch)
{
    switch (ch)
    {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    case '^':
        return 3;
    }
    return -1;
```

```
}
int infixToPostfix(char *exp)
{
    int i, k;
    struct Stack *stack = createStack(strlen(exp));
    if (!stack)
        return -1;
    for (i = 0, k = -1; exp[i]; ++i)
    {
        if (isOperand(exp[i]))
            exp[++k] = exp[i];
        else if (exp[i] == '(')
            push(stack, exp[i]);
        else if (exp[i] == ')')
        {
            while (!isEmpty(stack) && peek(stack) != '(')
                exp[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return -1;
            else
                pop(stack);
        }
        else
        {
            while (!isEmpty(stack) &&
                   Prec(exp[i]) <= Prec(peek(stack)))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }
    }
    while (!isEmpty(stack))
        exp[++k] = pop(stack);
    exp[++k] = '\0';
    printf("%s\n", exp);
    return 0;
}
int main()
{
    char exp[] = ("(A+B)*(C+D)");
    infixToPostfix(exp);
    return 0;
}
```

Prefix :

```
#include <bits/stdc++.h>
using namespace std;
bool isOperator(char c){
    return (!isalpha(c) && !isdigit(c));
}
int getPriority(char C){
    if (C == '-' || C == '+')
        return 1;
    else if (C == '*' || C == '/')
        return 2;
    else if (C == '^')
        return 3;
    return 0;
}
```

```
string infixToPostfix(string infix){
    infix = '(' + infix + ')';
    int l = infix.size();
    stack<char> char_stack;
    string output;
    for (int i = 0; i < l; i++){
        if (isalpha(infix[i]) || isdigit(infix[i]))
            output += infix[i];
        else if (infix[i] == '(')
            char_stack.push('(');
        else if (infix[i] == ')'){
            while (char_stack.top() != '('){
                output += char_stack.top();
                char_stack.pop();
            }
            char_stack.pop();
        }
        else{
            if (isOperator(char_stack.top())){
                if (infix[i] == '^'){
                    while (getPriority(infix[i]) <= getPriority(char_stack.top())){
                        output += char_stack.top();
                        char_stack.pop();
                    }
                }
                else{
                    while (getPriority(infix[i]) < getPriority(char_stack.top())){
                        output += char_stack.top();
                        char_stack.pop();
                    }
                }
                char_stack.push(infix[i]);
            }
        }
    }
    while (!char_stack.empty()){
        output += char_stack.top();
        char_stack.pop();
    }
    return output;
}
string ToPrefix(string infix){
    int l = infix.size();
    reverse(infix.begin(), infix.end());
    for (int i = 0; i < l; i++){
        if (infix[i] == '('){
            infix[i] = ')';
            i++;
        }
        else if (infix[i] == ')'){
            infix[i] = '(';
            i++;
        }
    }
    string prefix = infixToPostfix(infix);
    reverse(prefix.begin(), prefix.end());
    return prefix;
}
int main()
{
    string s = ("(A+B)*(C+D)");
```

```
        cout << ToPrefix(s) << std::endl;
        return 0;
    }
```

Output :

```
27:week10 - (master)$ gpp EvaluatePrefixExpression.cpp
28:week10 - (master)$ ./a.out
*+AB+CD
29:week10 - (master)$ gpp EvaluatePostfixExpression.cpp
30:week10 - (master)$ ./a.out
AB+CD+*
31:week10 - (master)$
```

Result :

Implementation of Intermediate code generation(Prefix & Postfix) has been done successfully.