Name: SATYAJIT GHANA CSE SEC-C Reg No: 17ETCS002159

Laboratory 4

Title of the Laboratory Exercise: Infix to Postfix

Introduction and Purpose of Experiment

Infix is a more human readable form of expression, but the operator priority needs to be checked properly for evaluating it, hence postfix and prefix notations are used that make it easier to evaluate an expression without worrying about the precedence of the operator, since the expression is now already organizes in the required form.

1. Aim and Objectives
   - To create a method to convert an infix expression to a postfix expression.

2. Pseudo Codes

```
1. Infix2Postfix
   1. Begin
   2. for (each token in the expression)
      a. if (token is a operand) output.add(token)
      b. else if (token == '(') push(token)
      c. else if (token == ')')
         i. while (!stack.isEmpty() && stack.peek != '(') output.add(stack.pop())
        ii. stack.pop()
      d. else
         i. while (!stack.isEmpty() && precedence(token) <= precedence(stack.peek())
            output.add(stack.pop())
        ii. stack.push(token)
   3. while (!stack.isEmpty()) output.add(stack.pop())
   4. print output
   5. End
```

3. Implementation in C

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

int infix2postfix(char *expr) {
    const char delim[] = " \n";

    Vector* output = newMinimalVector(string_compare, print_one_string);
    Stack* stack = newStack(strlen(expr));

    char* token = strtok(expr, delim);

    while (token != NULL) {
        if (isdigit(token[0]) || strlen(token) > 1) { /* operand was encountered */
            output -> add(output, token);
        } else if (strlen(token) == 1
            && token[0] == '(') {
            stack -> push(stack, token);
        } else if (strlen(token) == 1
            && token[0] == ')') {
            while (!stack -> isEmpty(stack) && *(char*)(stack -> peek(stack)) != '(' ) {
                output -> add(output, stack -> pop(stack));
```

```
                }
                stack -> pop(stack);
            } else { /* operator was encountered */
                while (!stack -> isEmpty(stack) && precedence(token[0]) <=
precedence(*(char*)(stack -> peek(stack)))) {
                    output -> add(output, stack -> pop(stack));
                }
                stack -> push(stack, new_string(token));
            }
            token = strtok(NULL, delim);
        }

        while (!stack -> isEmpty(stack)) {
            output -> add(output, stack -> pop(stack));
        }

        output -> print(output);

        return 1;
}
```

4. Presentation of Results

```
Enter an Infix Expression : 3 + 2 * 1 - 5
Given Infix Expression : 3 + 2 * 1 - 5
Postfix Expression : 3 2 1 * + 5 -
```

Explanation:

The Infix Expression is input as a string, which is a line, and read until a new line is encountered, this string is the infix expression that is to be converted into a postfix expression. The string expression is used as a formal argument for infix2postfix function.

The expression string is then tokenized using ‘ ’ as the delimiter, since the operator and the operands are separated using spaces. The tokenized input is then parsed using the infix-to-postfix pseudocode/algorithm. A Stack is created to store the operators, if the token is a operand it is directly added to the output, if there is an opening parenthesis then it is pushed to the stack, if a closing parenthesis is found then elements from the stack are popped until an opening parenthesis is found. For operators being encountered, there are added to the stack and any operator with greater than or equal to the precedence of the current token operator is popped and added to the output. The output is stored in a vector, which is then dispayed.