Experiment No.7
Implementation of Intermediate Code Generation (ICG)
Date of Performance: 17-03-2025
Date of Submission: 24-03-2025

Aim: Implementation of Intermediate Code Generation (ICG)

Objective: To develop an ability to apply and develop Intermediate Code representation techniques in Compiler construction.

Theory:

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, and then the back end of the compiler uses this intermediate code to generate the target code.

The benefits of using machine independent intermediate code are:

- Because of the machine independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
- Retargeting is facilitated.
- It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.\
- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portions same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.

The following are commonly used intermediate code representation:

1. **Postfix Notation –**

The ordinary (infix) way of writing the sum of a and b is with operator in the middle: a + b
The postfix notation for the same expression places the operator at the right end as ab +. In

general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Example – The postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + *ab -+.

2. Three-AddressCode

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references, but it is still called three address statement.

Example – The three-address code for the expression a + b * c + d:

T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d

T 1, T 2, T 3 are temporary variables.

3. Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

4. Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

5. Indirect Triples

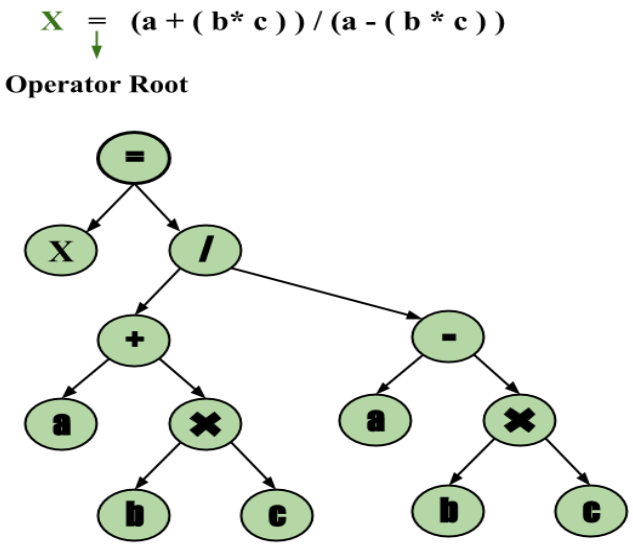
This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

6. Syntax Tree

Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Example –
x = (a + b * c) / (a – b * c)



Note: students need to implement at least two ICG for given input statement.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>


#define MAX 100


char stack[MAX][10];
```

```
int top = -1;
int tempCount = 1;

// Push to stack
void push(char* str) {
    strcpy(stack[++top], str);
}

// Pop from stack
char* pop() {
    if (top == -1) return "";
    return stack[top--];
}

// Check if character is operator
int isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}

// Return precedence of operators
int precedence(char op) {
    if (op == '*' || op == '/') return 2;
    if (op == '+' || op == '-') return 1;
    return 0;
}
```

```
// Convert infix to postfix
void infixToPostfix(char infix[], char postfix[]) {
    char opStack[MAX];
    int topOp = -1, i = 0, j = 0;
    char ch;

    while ((ch = infix[i++]) != '\0') {
        if (isspace(ch)) continue;

        if (isalnum(ch)) {
            postfix[j++] = ch;
        } else if (ch == '(') {
            opStack[++topOp] = ch;
        } else if (ch == ')') {
            while (topOp != -1 && opStack[topOp] != '(')
                postfix[j++] = opStack[topOp--];
            topOp--; // pop '('
        } else if (isOperator(ch)) {
            while (topOp != -1 && precedence(opStack[topOp]) >= precedence(ch))
                postfix[j++] = opStack[topOp--];
            opStack[++topOp] = ch;
        }
    }
    while (topOp != -1)
        postfix[j++] = opStack[topOp--];
    postfix[j] = '\0';
```

```
}

// Generate Three Address Code from postfix expression
void generateTAC(char postfix[], char resultVar) {
    int i = 0;
    char ch;
    char op1[10], op2[10], res[10];

    while ((ch = postfix[i++]) != '\0') {
        if (isalnum(ch)) {
            char str[2];
            str[0] = ch;
            str[1] = '\0';
            push(str);
        } else if (isOperator(ch)) {
            strcpy(op2, pop());
            strcpy(op1, pop());

            sprintf(res, "t%d", tempCount++);
            printf("%s = %s %c %s\n", res, op1, ch, op2);
            push(res);
        }
    }

    // Final result assignment
    printf("%c = %s\n", resultVar, pop());
```

```
}

int main() {
    char input[MAX], expr[MAX], postfix[MAX];
    char resultVar;

    printf("Enter an arithmetic expression (e.g., a = b + c * d): ");
    fgets(input, MAX, stdin);

    // Remove newline character
    input[strcspn(input, "\n")] = 0;

    // Extract LHS (result variable) and RHS expression
    int i = 0, j = 0;
    while (input[i] != '=' && input[i] != '\0') {
        if (!isspace(input[i])) {
            resultVar = input[i];
        }
        i++;
    }

    i++; // Skip '='
    while (input[i] != '\0') {
        if (!isspace(input[i])) {
            expr[j++] = input[i];
        }
```

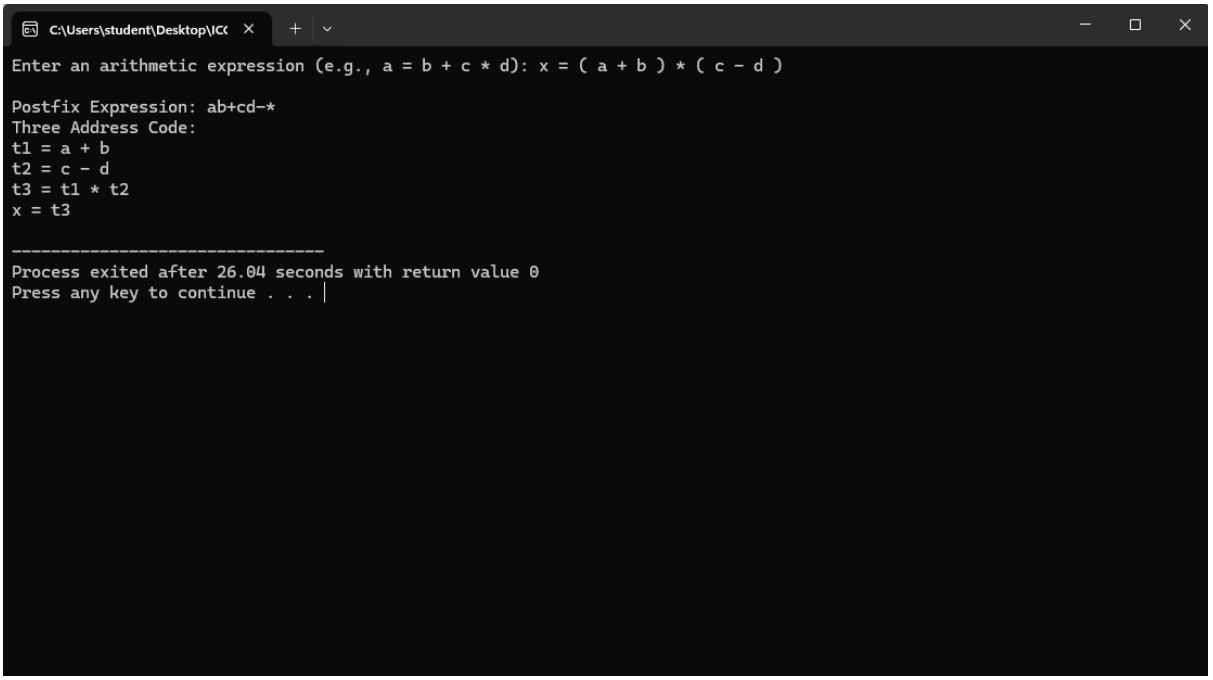
```
        i++;
    }
    expr[j] = '\0';

    infixToPostfix(expr, postfix);

    printf("\nPostfix Expression: %s\n", postfix);
    printf("Three Address Code:\n");
    generateTAC(postfix, resultVar);

    return 0;
}
```

Output:



Conclusion:

→ Intermediate Code Generation (ICG) plays a vital role in compiler construction by serving as a bridge between high-level language and machine code. The intermediate representation generated for the given input statement effectively breaks down complex expressions into simpler Three Address Code (TAC) form, making it easier to analyze and optimize. It clearly demonstrates how arithmetic expressions are translated into step-by-step instructions using temporary variables. This representation enhances portability, simplifies optimization, and allows better error detection during the compilation process, thus validating the effectiveness of ICG techniques.