**Fr. Conceicao Rodrigues College of Engineering**

**Department of Computer Engineering**

**Academic Term : Jan-May 2024 - 25**

**Class : T.E. (Computer - A)**

**Subject Name : System Programming and Compiler Construction**

**Subject Code : (CPC601)**

|  |  |
| --- | --- |
| **Practical No:** | 3 |
| **Title:** | To generate an Intermediate code. |
| **Date of Performance:** | 25/02/2025 |
| **Date of Submission:** | 21/03/2025 |
| **Roll No:** | 9913 |
| **Name of the Student:** | Mark Lopes |

**Evaluation:**

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| **1** | **Time Line (2)** |  |
| **2** | **Output(3)** |  |
| **3** | **Code optimization (2)** |  |
| **4** | **Postlab (3)** |  |

**Signature of the Teacher :**

**Experiment No 3**

**Aim** : To generate an Intermediate code.

**Description:**

Diagram

Description automatically generated

* 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 portion same for all the compilers.
* The second part of compiler, synthesis, is changed according to the target machine.
* It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.
* Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

* For example:
* a = b + c \* d;
* The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.
* r1 = c \* d;
* r2 = b + r1;
* a = r2
* r being used as registers in the target program.
* A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

### **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 | arg1 | arg2 | result |
| \* | c | d | r1 |
| + | b | r1 | r2 |
| + | r2 | r1 | r3 |
| = | r3 |  | a |

### **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 | arg1 | arg2 |
| \* | 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.

### **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.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_LEN 100

typedef enum { ID, NUM, PLUS, MINUS, MULT, DIV, ASSIGN, LBRACKET, RBRACKET, LPAREN, RPAREN, END, ERROR } TokenType;

typedef struct {

    TokenType type;

    char lexeme[MAX\_LEN];

} Token;

typedef struct {

    char op[10], arg1[MAX\_LEN], arg2[MAX\_LEN], result[MAX\_LEN];

} Quadruple;

typedef struct {

    char op[10], arg1[MAX\_LEN], arg2[MAX\_LEN];

} Triple;

char \*input;

int pos = 0, tempVar = 0, quadIdx = 0, tripleIdx = 0;

Token token;

Quadruple quads[MAX\_LEN];

Triple triples[MAX\_LEN];

char\* create\_temp() {

    char\* temp = (char\*)malloc(10);

    sprintf(temp, "t%d", tempVar++);

    return temp;

}

void fetch\_token() {

    while (input[pos] && isspace(input[pos])) pos++;

    if (!input[pos]) { token.type = END; strcpy(token.lexeme, "EOF"); return; }

    if (isalpha(input[pos])) {

        int i = 0;

        while (isalnum(input[pos])) token.lexeme[i++] = input[pos++];

        token.lexeme[i] = '\0';

        token.type = ID;

    } else if (isdigit(input[pos])) {

        int i = 0;

        while (isdigit(input[pos])) token.lexeme[i++] = input[pos++];

        token.lexeme[i] = '\0';

        token.type = NUM;

    } else {

        switch (input[pos++]) {

            case '+': token.type = PLUS; strcpy(token.lexeme, "+"); break;

            case '-': token.type = MINUS; strcpy(token.lexeme, "-"); break;

            case '\*': token.type = MULT; strcpy(token.lexeme, "\*"); break;

            case '/': token.type = DIV; strcpy(token.lexeme, "/"); break;

            case '=': token.type = ASSIGN; strcpy(token.lexeme, "="); break;

            case '[': token.type = LBRACKET; strcpy(token.lexeme, "["); break;

            case ']': token.type = RBRACKET; strcpy(token.lexeme, "]"); break;

            case '(': token.type = LPAREN; strcpy(token.lexeme, "("); break;

            case ')': token.type = RPAREN; strcpy(token.lexeme, ")"); break;

            default: token.type = ERROR; token.lexeme[0] = input[pos - 1]; token.lexeme[1] = '\0';

        }

    }

}

void generate\_quad(char\* op, char\* arg1, char\* arg2, char\* result) {

    strcpy(quads[quadIdx].op, op);

    strcpy(quads[quadIdx].arg1, arg1);

    strcpy(quads[quadIdx].arg2, arg2);

    strcpy(quads[quadIdx].result, result);

    quadIdx++;

}

void generate\_triple(char\* op, char\* arg1, char\* arg2) {

    strcpy(triples[tripleIdx].op, op);

    strcpy(triples[tripleIdx].arg1, arg1);

    strcpy(triples[tripleIdx].arg2, arg2);

    tripleIdx++;

}

char\* parse\_expression();

char\* parse\_factor() {

    char \*res;

    if (token.type == ID) {

        res = strdup(token.lexeme);

        fetch\_token();

        if (token.type == LBRACKET) {

            fetch\_token();

            char\* index = parse\_expression();

            if (token.type != RBRACKET) { printf("Syntax Error: Expected ']'\n"); exit(1); }

            fetch\_token();

            char\* temp = create\_temp();

            generate\_quad("[]", res, index, temp);

            generate\_triple("[]", res, index);

            free(res);

            free(index);

            return temp;

        }

        return res;

    } else if (token.type == NUM) {

        res = strdup(token.lexeme);

        fetch\_token();

        return res;

    } else if (token.type == LPAREN) {

        fetch\_token();

        res = parse\_expression();

        if (token.type != RPAREN) { printf("Syntax Error: Expected ')'\n"); exit(1); }

        fetch\_token();

        return res;

    }

    printf("Syntax Error: Unexpected token %s\n", token.lexeme);

    exit(1);

}

char\* parse\_term() {

    char\* left = parse\_factor();

    while (token.type == MULT || token.type == DIV) {

        TokenType op = token.type;

        fetch\_token();

        char\* right = parse\_factor();

        char\* temp = create\_temp();

        generate\_quad(op == MULT ? "\*" : "/", left, right, temp);

        generate\_triple(op == MULT ? "\*" : "/", left, right);

        free(left);

        free(right);

        left = temp;

    }

    return left;

}

char\* parse\_expression() {

    char\* left = parse\_term();

    while (token.type == PLUS || token.type == MINUS) {

        TokenType op = token.type;

        fetch\_token();

        char\* right = parse\_term();

        char\* temp = create\_temp();

        generate\_quad(op == PLUS ? "+" : "-", left, right, temp);

        generate\_triple(op == PLUS ? "+" : "-", left, right);

        free(left);

        free(right);

        left = temp;

    }

    return left;

}

int main() {

    char buffer[MAX\_LEN];

    printf("Enter an arithmetic expression: ");

    fgets(buffer, sizeof(buffer), stdin);

    buffer[strcspn(buffer, "\n")] = '\0';

    input = buffer;

    pos = tempVar = quadIdx = tripleIdx = 0;

    fetch\_token();

    char\* left = parse\_factor();

    if (token.type != ASSIGN) { printf("Syntax Error: Expected '='\n"); exit(1); }

    fetch\_token();

    char\* right = parse\_expression();

    generate\_quad("=", right, "", left);

    generate\_triple("=", right, "");

    free(left);

    free(right);

    printf("\nThree-Address Code:\n");

    for (int i = 0; i < quadIdx; i++) {

        printf("%s = %s %s %s\n", quads[i].result, quads[i].arg1, quads[i].op, quads[i].arg2);

    }

    printf("\nTriples:\n");

    printf("| Index | Op  | Arg1  | Arg2  |\n");

    printf("|-------|-----|-------|-------|\n");

    for (int i = 0; i < tripleIdx; i++) {

        printf("| %5d | %-3s | %-5s | %-5s |\n", i, triples[i].op, triples[i].arg1, triples[i].arg2);

    }

    printf("\nQuadruples:\n");

    printf("| Index | Op  | Arg1  | Arg2  | Result |\n");

    printf("|-------|-----|-------|-------|--------|\n");

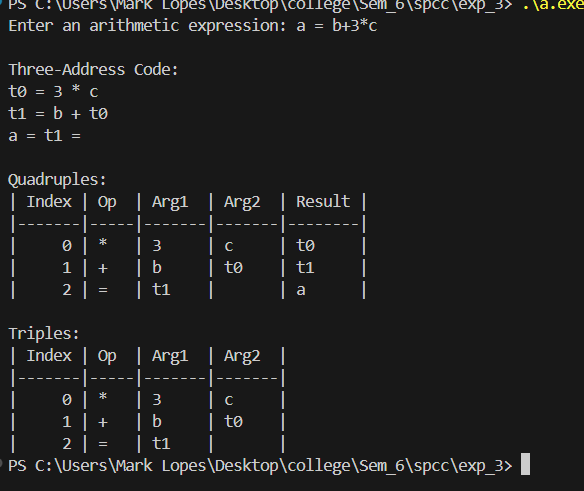
    for (int i = 0; i < quadIdx; i++) {

        printf("| %5d | %-3s | %-5s | %-5s | %-6s |\n", i, quads[i].op, quads[i].arg1, quads[i].arg2, quads[i].result);

    }

    return 1;

}

**OUTPUT:**

**Post Lab Question**

1. **Write the intermediate code generated for ----**

**while ( a<b ) do**

**If ( c< d) then**

**X= y+z**

**Else**

**X= y-z**

1. **Write the intermediate code generated for ----**

**switch E**

**Begin**

**case V1 : S1**

**case V2 : S2**

**….**

**default: Sn**

**end**

