# LAB ASSIGNMENT -4

# 3-ADDRESS CODE & LLVM

**Name:** Sharan P **Reg.No:** 21BRS1582

## b)PROGRAM

**Aim**: Generate the TAC for the following arithmetic expression and represent them in Quadruples, Triples and Indirect triple structures

x=a+(b^c)*d-e/(f^(g*i%j));

## Code:

## LEX CODE:

```
%{
#include "y.tab.h"
%}
%%
[0-9]+? {yylval.sym=(char)yytext[0]; return NUMBER;}
[a-zA-Z]+? {yylval.sym=(char)yytext[0];return LETTER;}
\n {return 0;}
. {return yytext[0];}
%%

int yywrap() {
	return 0;
}
```

## YACC CODE:

```
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void ThreeAddressCode();
char AddToTable(char ,char, char);
int ind=0;
char temp = '1';
struct incod{
	char opd1;
```

```
        char opd2;
        char opr;
};
%}
%union
{
char sym;
}
%token <sym> LETTER NUMBER
%type <sym> expr
%left '+'
%left '*' '/'
%left '-'
%%
statement: LETTER '=' expr ';' {AddToTable((char)$1,(char)$3,'=');}
| expr ';'
;
expr:
expr '+' expr {$$ = AddToTable((char)$1,(char)$3,'+');}
| expr '-' expr {$$ = AddToTable((char)$1,(char)$3,'-');}
| expr '*' expr {$$ = AddToTable((char)$1,(char)$3,'*');}
| expr '/' expr {$$ = AddToTable((char)$1,(char)$3,'/');}
| expr '^' expr {$$ = AddToTable((char)$1,(char)$3,'^');}
| expr '%' expr {$$ = AddToTable((char)$1,(char)$3,'%');}
| '(' expr ')' {$$ = (char)$2;}
| NUMBER {$$ = (char)$1;}
| LETTER {$$ = (char)$1;}
|'-' expr {$$ = AddToTable((char)$2,(char)'\t','-');}
;
%%
yyerror(char *s){
        printf("%s",s);
        exit(0);
}

struct incod code[20];
char AddToTable(char opd1,char opd2,char opr){
        code[ind].opd1=opd1;
        code[ind].opd2=opd2;
        code[ind].opr=opr;
        ind++;
        return temp++;
}
void ThreeAddressCode(){
        int cnt =0;
        char temp = '1';
        printf("THREE ADDRESS CODE\n");
        while(cnt<ind){
                if(code[cnt].opr != '=') printf("t%c : = \t",temp++);
                if(isalpha(code[cnt].opd1)) printf(" %c\t",code[cnt].opd1);
                else if(code[cnt].opd1 >='1''&& code[cnt].opd1 <='9') printf("t%c\t",code[cnt].opd1);
```

```
                printf("%c\t",code[cnt].opr);
                if(isalpha(code[cnt].opd2))
                        printf(" %c\n",code[cnt].opd2);
                else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
                        printf("t%c\n",code[cnt].opd2);
                cnt++;
        }
}
void quadraple(){
        int cnt = 0;
        char temp ='1';
        printf("\nQUADRAPLE CODE\n");
        printf("Location \t op \t Arg1 \t Arg2 \t Result\n\n");
        while(cnt<ind){
                printf("(%c) \t",temp);
                printf(" %c\t",code[cnt].opr);
                if(code[cnt].opr == '='){
                        if(isalpha(code[cnt].opd2)) printf(" %c\t \t",code[cnt].opd2);
                        else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9') printf("t%c\t
\t",code[cnt].opd2);
                        printf(" %c\n",code[cnt].opd1);
                        cnt++;
                        continue;
                }
                if(isalpha(code[cnt].opd1)) printf(" %c\t",code[cnt].opd1);
                else if(code[cnt].opd1 >='1' && code[cnt].opd1 <='9') printf("t%c\t",code[cnt].opd1);
                if(isalpha(code[cnt].opd2)) printf(" %c\t",code[cnt].opd2);
                else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9') printf("t%c\t",code[cnt].opd2);
                else printf(" %c",code[cnt].opd2);
                printf("t%c\n",temp++);
                cnt++;
                temp++;
        }
}
void triple(){
        int cnt=0;
        char temp='1';
        printf("Location \t op \t Arg1 \t Arg2\n\n");
        while(cnt<ind){
                printf("(%c) \t",temp);
                printf(" %c\t",code[cnt].opr);
                if(code[cnt].opr == '='){
                        if(isalpha(code[cnt].opd2))
                        printf(" %c \t \t",code[cnt].opd2);
                        else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
printf("(%c)\n",code[cnt].opd2);
                        cnt++;
                        temp++;
                        continue;
                }
                if(isalpha(code[cnt].opd1)) printf(" %c \t",code[cnt].opd1);
```

```
                else if(code[cnt].opd1 >='1' && code[cnt].opd1 <='9')
printf("(%c)\t",code[cnt].opd1);
                if(isalpha(code[cnt].opd2)) printf(" %c \n",code[cnt].opd2);
                else if(code[cnt].opd2 >='1' && code[cnt].opd2 <='9')
printf("(%c)\n",code[cnt].opd2);
                else printf(" %c\n",code[cnt].opd2);
                cnt++;
                temp++;
        }
}

void indirecttriple(){
        printf("\nSTATEMENT\n");
        int cnt=0;
        while(cnt<ind){
                printf("(%d) \n",cnt+1);
                cnt++;
        }
        printf("\n\n");
        triple();
}

main(){
        printf("\nEnter the Expression : ");
        yyparse();
        ThreeAddressCode();
        quadraple();
        printf("\nTRIPLE CODE\n");
        triple();
        printf("\nINDIRECT TRIPLE CODE\n");
        indirecttriple();
}
```

## Output Screenshot:

```
sharan@Shar-Ubutu:~/CompilerDesign$ lex 4tac.l
sharan@Shar-Ubutu:~/CompilerDesign$ yacc -d 4tac.y
4tac.y: warning: 22 shift/reduce conflicts [-Wconflicts-sr]
sharan@Shar-Ubutu:~/CompilerDesign$ gcc lex.yy.c y.tab.c -w
sharan@Shar-Ubutu:~/CompilerDesign$ ./a.out

Enter the Expression : x=a+(b^c)*d-e/(f^(g*i%j));
THREE ADDRESS CODE
t1 : =   b      ^        c
t2 : =   d      -        e
t3 : =  *       t2
t4 : =   i      %        j
t5 : =   g      *       t4
t6 : =   f      ^       t5
t7 : =  /       t6
t8 : =   a      +       t7
 x      =       t8

QUADRAPLE CODE
Location         op      Arg1    Arg2    Result

(1)      ^       b       c       t1
(3)      -       d       e       t3
(5)      *       t1      t2      t5
(7)      %       i       j       t7
(9)      *       g       t4      t9
(;)      ^       f       t5      t;
(=)      /       t3      t6      t=
(?)      +       a       t7      t?
(A)      =       t8               x
```

```
TRIPLE CODE
Location         op      Arg1    Arg2

(1)      ^       b       c
(2)      -       d       e
(3)      *      (1)     (2)
(4)      %       i       j
(5)      *       g      (4)
(6)      ^       f      (5)
(7)      /      (3)     (6)
(8)      +       a      (7)
(9)      =      (8)

INDIRECT TRIPLE CODE

STATEMENT
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)


Location         op      Arg1    Arg2

(1)      ^       b       c
(2)      -       d       e
(3)      *      (1)     (2)
(4)      %       i       j
(5)      *       g      (4)
(6)      ^       f      (5)
(7)      /      (3)     (6)
(8)      +       a      (7)
(9)      =      (8)
sharan@Shar-Ubutu:~/CompilerDesign$
```

## Result:

We have generated the TAC for the given arithmetic expression and represent them in Quadruples, Triples and Indirect triple structures

**THE END OF PROGRAM**

# A STUDY ABOUT LLVM

## a)STUDY ABOUT LLVM:

The LLVM Project is a collection of modular and reusable compiler and tool-chain technologies. LLVM, standing for Low-Level Virtual Machine, is an open-source compiler infrastructure project renowned for its versatility and efficiency. It employs a platform-independent Intermediate Representation (IR), facilitating seamless translation between different programming languages and target architectures. LLVM's modular design enables developers to build customized compiler tool chains using its frontend and backend components. It boasts a robust optimization infrastructure, offering a suite of optimization passes for enhancing code performance and size. It supports Just-In-Time (JIT) compilation, enabling dynamic code generation and execution. LLVM has a wide range of extensive debugging and profiling tools. It provides a comprehensive ecosystem for compiler development.

**Some features of LLVM are:**

- Front-ends for C, C++, Objective-C, Fortran, etc. They support the ANSI-standard C and C++ languages. Additionally, many GCC extensions are supported.
- A stable implementation of the LLVM instruction set, which serves as both the online and offline code representation, together with assembly (ASCII) and bytecode (binary) readers and writers, and a verifier.
- A powerful pass-management system that automatically sequences passes (including analysis, transformation, and code-generation passes) based on their dependences, and pipelines them for efficiency.
- A wide range of global scalar optimizations.
- A link-time interprocedural optimization framework with a rich set of analyses and transformations, including sophisticated whole-program pointer analysis, call graph construction, and support for profile-guided optimizations.

- An easily retargetable code generator, which currently supports X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC, Alpha, CellSPU, MIPS, MSP430, SystemZ, WebAssembly and XCore.
- A Just-In-Time (JIT) code generation system, which currently supports X86, X86-64, ARM, AArch64, Mips, SystemZ, PowerPC, and PowerPC-64.
- Support for generating DWARF debugging information.
- A profiling system similar to gprof.
- A test framework with a number of benchmark codes and applications.
- APIs and debugging tools to simplify rapid development of LLVM components.

**Some strengths of LLVM are:**

1. LLVM uses a simple low-level language with strictly defined semantics.
2. It includes front-ends for C and C++. Front-ends for Java, Scheme, and other languages are in development.
3. It includes an aggressive optimizer, including scalar, interprocedural, profile-driven, and some simple loop optimizations.
4. It supports a life-long compilation model, including link-time, install-time, run-time, and offline optimization.
5. LLVM has full support for accurate garbage collection.
6. The LLVM code generator is relatively easy to retarget, and makes use of a powerful target description language.
7. LLVM has extensive documentation and has hosted many projects of various sorts.
8. Many third-party users have claimed that LLVM is easy to work with and develop for. For example, the (now removed) Stacker front-end was written in 4 days by someone who started knowing nothing about LLVM. Additionally, LLVM has tools to make development easier.
9. LLVM is under active development and is constantly being extended, enhanced and improved. See the status updates on the left bar to see the rate of development.
10. LLVM is freely available under an OSI-approved "Apache License Version 2.0" license.
11. LLVM is currently used by many commercial, non-profit or academic entities, who contribute many extensions and new features.

LLVM is and can be used for several projects . It tends to help people like a compiler researcher / developer interested in compile – time , link – time and run-time transformation for C and C++ programs. It also helps virtual machine researcher/developer that's interested in a portable, language independent instruction set and compilation framework. Other people that it might help are architecture researchers interested in compiler/ hardware techniques, security researchers interested in static analysis or instrumentation, instructor or developer that is interested in a system for quick prototyping of compiler transformation or even an end user who wants to get better performance out of their code.

LLVM being highly documented provides all the necessary data for the users in need of understanding. LLVM began in December 2000 as a collection of reusable libraries for low-level toolchain components like assemblers and compilers, designed to be compatible with Unix systems1. It's known for its unique internal architecture and tools like the Clang compiler.

Three-Phase Compiler Design: The traditional design of compilers includes three phases: the front end, the optimizer, and the back end2. LLVM follows this model, offering a retargetable and flexible architecture that supports multiple source languages and target architectures.

Over the past decade, LLVM has significantly influenced the landscape of language implementations, supporting a wide range of languages and replacing various special-purpose compilers. It's also used in new products like the OpenCL GPU programming language3. The LLVM Intermediate Representation (IR) is a key aspect of its design, serving as a language with well-defined semantics for mid-level analyses and transformations in the optimizer section of a compiler4.

LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, language-independent type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address

arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and setjmp/longjmp in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We can describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways:

- the size and effectiveness of the representation, including the type information it provides;
- compiler performance for several inter-procedural problems; and
- illustrative examples of the benefits LLVM provides for several challenging compiler problems.

Modern programming languages and software engineering principles are causing increasing problems for compiler systems. Traditional approaches, which use a simple compile-link-execute model, are unable to provide adequate application performance under the demands of the new conditions. Traditional approaches to inter-procedural and profile-driven compilation can provide the application performance needed, but require infeasible amounts of compilation time to build the application.

This thesis presents LLVM, a design and implementation of a compiler infrastructure which supports a unique multi-stage optimization system. This system is designed to support extensive inter-procedural and profile-driven optimizations, while being efficient enough for use in commercial compiler systems.

The LLVM virtual instruction set is the glue that holds the system together. It is a low-level representation, but with high-level type information. This provides the benefits of a low-level representation (compact representation, wide variety of available transformations, etc.) as well as providing high-level information to support aggressive inter-

procedural optimizations at link- and post-link time. In particular, this system is designed to support optimization in the field, both at run-time and during otherwise unused idle time on the machine. This thesis also describes an implementation of this compiler design, the LLVM compiler infrastructure, proving that the design is feasible. The LLVM compiler infrastructure is a maturing and efficient system, which we show is a good host for a variety of research.

**END OF STUDY**