LAB ASSIGNMENT -4

LLVM AND 3-ADDRESS CODE

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

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 Just-In-Time (JIT) code generation system, which currently supports X86, X86-64, ARM, AArch64, Mips, SystemZ, PowerPC, and PowerPC-64.

Some strengths of LLVM are:

- LLVM uses a simple low-level language with strictly defined semantics.
- It includes front-ends for C and C++. Front-ends for Java, Scheme, and other languages are in development.
- It includes an aggressive optimizer, including scalar, interprocedural, profile-driven, and some simple loop optimizations.
- It supports a life-long compilation model, including link-time, install-time, run-time, and offline optimization.
- LLVM has full support for accurate garbage collection.
- The LLVM code generator is relatively easy to retarget, and makes use of a powerful target description language.
- LLVM has extensive documentation and has hosted many projects of various sorts.

---

b)PROGRAM

Aim: To write a program in Lex/ Yacc compiler to generate three address code for an expression X= a+b*c using Syntax Directed Translations.

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;
}
```

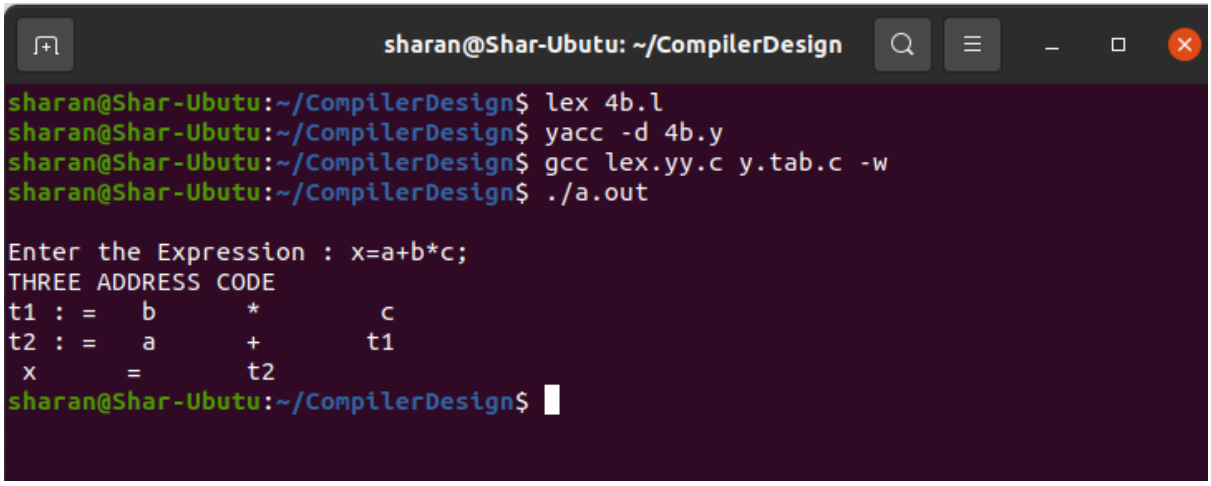
YACC CODE:

```
%{
#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 ')' {$$ = (char)$2;}
| NUMBER {$$ = (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++;
        }
}
main(){
        printf("\nEnter the Expression : ");
        yyparse();
        ThreeAddressCode();
}
```

Output Screenshot:



Result:

We have created a program in Lex/ Yacc compiler to generate three address code for an expression X= a+b*c using Syntax Directed Translations.

END