Name	Roll No.	Section	DoP
Vidhan Dahatkar	166	B	

Practical - 8

Aim –

Develop a program for intermediate code generator to generate three address code using LEX & YACC.

Theory -

Intermediate Code Generation Using LEX and YACC

Intermediate code generation is an essential phase in the design of a compiler, where the source code is translated into a lower-level representation known as intermediate code. This representation serves as a bridge between high-level language and machine code, allowing for easier optimization and portability. One of the most commonly used forms of intermediate code is **Three Address Code (TAC)**, where each instruction contains at most three operands—typically two source operands and one destination. For example, a high-level expression like a = b + c * d would be broken down into a sequence such as t0 = c * d, t1 = b + t0, and a = t1.

To automate the process of generating such intermediate code, tools like **LEX** and **YACC** are used. LEX is a lexical analyzer generator that tokenizes the input stream into meaningful symbols such as identifiers, numbers, and operators. YACC, on the other hand, is a parser generator that takes the tokens from LEX and applies grammar rules to build the syntactic structure of the input. Together, LEX and YACC can be used to parse arithmetic expressions and generate corresponding three-address code by embedding semantic actions within grammar rules. This approach provides a systematic and efficient way to translate high-level expressions into intermediate code during compilation.

Program –

lexer.l

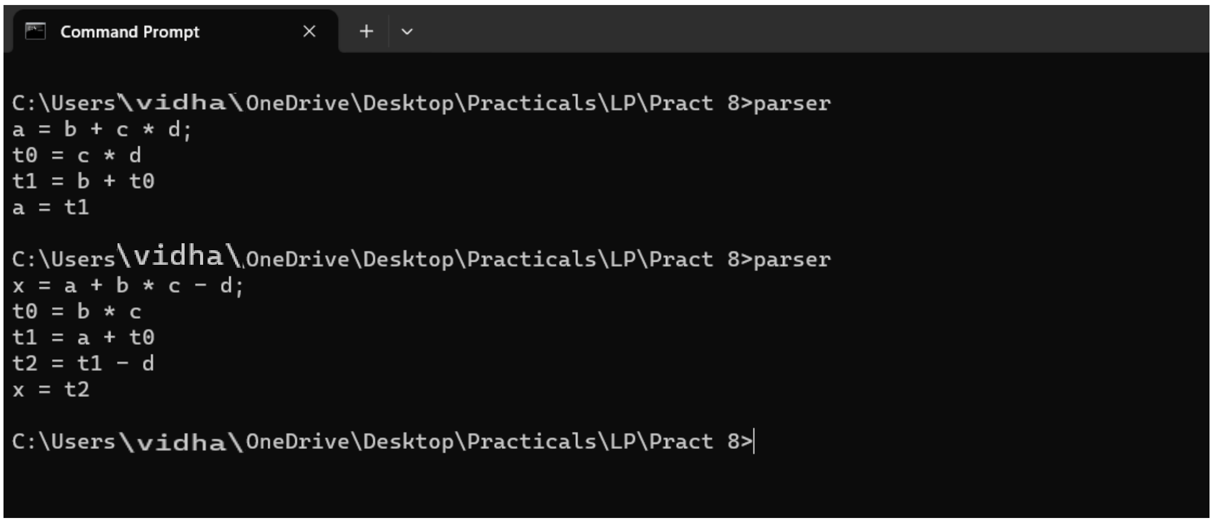
```
%{
#include "y.tab.h"
%}

%%
a   { return A; }
b   { return B; }
\n  { return EOL; }
.   { return INVALID; }
%%
```

parser.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int tempCount = 0;

char* newTemp() {
   char* temp = (char*)malloc(8);
   sprintf(temp, "t%d", tempCount++);
   return temp;
}

void yyerror(char* s) {
}
%}

%union {
   char* str;
}

%token <str> ID NUM
%token ASSIGN PLUS MINUS MUL DIV SEMI
%type <str> expr term factor

%start stmts

%%

stmts: stmts stmt
     | stmt
     ;

stmt: ID ASSIGN expr SEMI {
         printf("%s = %s\n", $1, $3);
      }
   | ID ASSIGN expr '\n' {
         printf("%s = %s\n", $1, $3);
      }
   ;

expr: expr PLUS term {
         char* temp = newTemp();
         printf("%s = %s + %s\n", temp, $1, $3);
```

```
            $$ = temp;
        }
    | expr MINUS term {
            char* temp = newTemp();
            printf("%s = %s - %s\n", temp, $1, $3);
            $$ = temp;
        }
    | term {
            $$ = $1;
        }
    ;

term: term MUL factor {
            char* temp = newTemp();
            printf("%s = %s * %s\n", temp, $1, $3);
            $$ = temp;
        }
    | term DIV factor {
            char* temp = newTemp();
            printf("%s = %s / %s\n", temp, $1, $3);
            $$ = temp;
        }
    | factor {
            $$ = $1;
        }
    ;

factor: ID {
            $$ = $1;
        }
    | NUM {
            $$ = $1;
        }
    ;

%%

int main() {
    yyparse();
    return 0;
}
```

Output -



Command Prompt

```
C:\Users\vidha\OneDrive\Desktop\Practicals\LP\Pract 8>parser
a = b + c * d;
t0 = c * d
t1 = b + t0
a = t1

C:\Users\vidha\OneDrive\Desktop\Practicals\LP\Pract 8>parser
x = a + b * c - d;
t0 = b * c
t1 = a + t0
t2 = t1 - d
x = t2

C:\Users\vidha\OneDrive\Desktop\Practicals\LP\Pract 8>
```

Conclusion –

In this practical, we learned how to use LEX and YACC to create an intermediate code generator that converts high-level expressions into Three Address Code (TAC). We understood the process of lexical analysis, parsing, and embedding semantic actions to generate intermediate representations. This helped us grasp how compilers break down complex expressions into simpler instructions for further processing. Overall, it demonstrated the importance of intermediate code in efficient and portable compiler design.