

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Academic Year: 2023 – 2024 (ODD SEM)**

**LAB MANUAL – R-2021**

**III CSE - SEMESTER V**

**CS3501 – COMPILER DESIGN LABORATORY**

**PREPARED BY**

**Dr.P.KRISHNAVENI**

**ASP/CSE**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TITLE OF THE EXPERIMENT** | **Page NO** |
| 1.a | Program to Recognize a Few Patterns in C. | 4 |
| 1.b | Implementation of Symbol Table using C. | 10 |
| 2 | Implementation of a Lexical Analyzer using Lex | 13 |
| 3.a | Program to Recognize a Valid Arithmetic Expression using YACC. | 17 |
| 3.b | Program to Recognize a Valid Variable using YACC. | 20 |
| 3.c | Program to Recognize a Valid Control Structure Syntax of C Language using YACC. | 23 |
| 3.d | Implementation of a Calculator using LEX and YACC. | 26 |
| 4 | Intermediate Code Generation using Lex & YACC. | 30 |
| 5 | Implementation of Type Checking using C. | 36 |
| 6 | Implementation of Code Optimization for a Simple C Program. | 39 |
| 7 | Implementation of Back-end of the Compiler. | 44 |

**CS3501 COMPILER DESIGN LABORATORY L T P C**

0 0 4 2

**COURSE OBJECTIVES:**

• To learn the various parsing techniques.  
• To understand intermediate code generation and run-time environment.  
• To learn to implement the front-end of the compiler.  
• To learn to implement code generator.

• To learn to implement code optimization.

**LIST OF EXPERIMENTS:**

1. Using the LEX tool, Develop a lexical analyzer to recognize a few patterns in C. (Ex.  
 identifiers, constants, comments, operators etc.). Create a symbol table, while recognizing  
 identifiers.  
2. Implement a Lexical Analyzer using LEX Tool  
3. Generate YACC specification for a few syntactic categories.  
 a. Program to recognize a valid arithmetic expression that uses operator +, -, \* and /.  
 b. Program to recognize a valid variable which starts with a letter followed by any

number of letters or digits.  
 c. Program to recognize a valid control structures syntax of C language (For loop,  
 while loop, if-else, if-else-if, switch-case, etc.).  
 d. Implementation of calculator using LEX and YACC  
4. Generate three address code for a simple program using LEX and YACC.  
5. Implement type checking using Lex and Yacc.  
6. Implement simple code optimization techniques (Constant folding, Strength reduction and  
 Algebraic transformation)  
7. Implement back-end of the compiler for which the three address code is given as input and  
 the 8086 assembly language code is produced as output.

**COURSE OUTCOMES:**  
CO1: Understand the techniques in different phases of a compiler.  
CO2: Design a lexical analyser for a sample language and learn to use the LEX tool.  
CO3: Apply different parsing algorithms to develop a parser and learn to use YACC tool  
CO4: Understand semantics rules (SDT), intermediate code generation and run-time

environment.

CO5: Implement code generation and apply code optimization techniques.

**Ex: 1.a**

**Program to Recognize a Few Patterns in C**

**Aim**

To write a C program to develop a lexical analyzer to recognize a few patterns in C.

**Algorithm**

Step1: Start the program.

Step2: Read a simple C language program as input string.

Step3: Display the input string.

Step4: Separate the tokens of the string.

Step5: Display each token with its type.

Step6: Also display whether the token is valid or invalid.

Step7: Stop the program.

**Source Code**

//Develop a lexical analyzer to recognize a few patterns in C.

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER.

bool isDelimiter(char ch)

{

if (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

ch == '[' || ch == ']' || ch == '{' || ch == '}')

return (true);

return (false);

}

// Returns 'true' if the character is an OPERATOR.

bool isOperator(char ch)

{

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '>' || ch == '<' ||

ch == '=')

return (true);

return (false);

}

// Returns 'true' if the string is a VALID IDENTIFIER.

bool validIdentifier(char\* str)

{

if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

str[0] == '3' || str[0] == '4' || str[0] == '5' ||

str[0] == '6' || str[0] == '7' || str[0] == '8' ||

str[0] == '9' || isDelimiter(str[0]) == true)

return (false);

return (true);

}

// Returns 'true' if the string is a KEYWORD.

bool isKeyword(char\* str)

{

if (!strcmp(str, "if") || !strcmp(str, "else") ||

!strcmp(str, "while") || !strcmp(str, "do") ||

!strcmp(str, "break") ||

!strcmp(str, "continue") || !strcmp(str, "int")

|| !strcmp(str, "double") || !strcmp(str, "float")

|| !strcmp(str, "return") || !strcmp(str, "char")

|| !strcmp(str, "case") || !strcmp(str, "char")

|| !strcmp(str, "sizeof") || !strcmp(str, "long")

|| !strcmp(str, "short") || !strcmp(str, "typedef")

|| !strcmp(str, "switch") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "static")

|| !strcmp(str, "struct") || !strcmp(str, "goto"))

return (true);

return (false);

}

// Returns 'true' if the string is an INTEGER.

bool isInteger(char\* str)

{

int i, len = strlen(str);

if (len == 0)

return (false);

for (i = 0; i < len; i++) {

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' || (str[i] == '-' && i > 0))

return (false);

}

return (true);

}

// Returns 'true' if the string is a REAL NUMBER.

bool isRealNumber(char\* str)

{

int i, len = strlen(str);

bool hasDecimal = false;

if (len == 0)

return (false);

for (i = 0; i < len; i++) {

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' && str[i] != '.' ||

(str[i] == '-' && i > 0))

return (false);

if (str[i] == '.')

hasDecimal = true;

}

return (hasDecimal);

}

// Extracts the SUBSTRING.

char\* subString(char\* str, int left, int right)

{

int i;

char\* subStr = (char\*)malloc(

sizeof(char) \* (right - left + 2));

for (i = left; i <= right; i++)

subStr[i - left] = str[i];

subStr[right - left + 1] = '\0';

return (subStr);

}

// Parsing the input STRING.

void parse(char\* str)

{

int left = 0, right = 0;

int len = strlen(str);

while (right <= len && left <= right) {

if (isDelimiter(str[right]) == false)

right++;

if (isDelimiter(str[right]) == true && left == right) {

if (isOperator(str[right]) == true)

printf("'%c' IS AN OPERATOR\n", str[right]);

right++;

left = right;

} else if (isDelimiter(str[right]) == true && left != right

|| (right == len && left != right)) {

char\* subStr = subString(str, left, right - 1);

if (isKeyword(subStr) == true)

printf("'%s' IS A KEYWORD\n", subStr);

else if (isInteger(subStr) == true)

printf("'%s' IS AN INTEGER\n", subStr);

else if (isRealNumber(subStr) == true)

printf("'%s' IS A REAL NUMBER\n", subStr);

else if (validIdentifier(subStr) == true

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS A VALID IDENTIFIER\n", subStr);

else if (validIdentifier(subStr) == false

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

left = right;

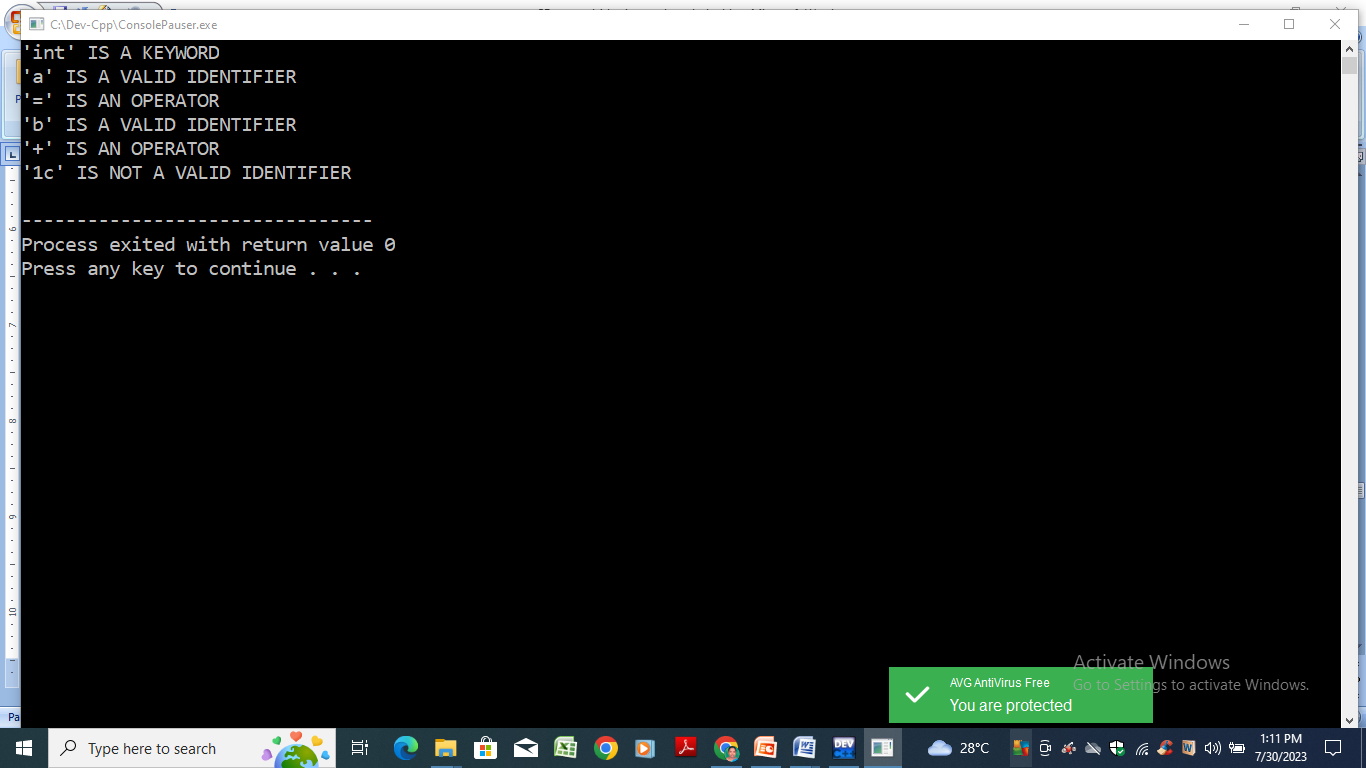
}

}

return;

}

**OUTPUT**

****

**RESULT**

Thus the program for developing a lexical analyzer to recognize a few patterns in C has been executed successfully.

**Ex: 1.b**

**Implementation of Symbol Table using C**

**Aim**

To write a program for implementing symbol table using C.

**Algorithm**

Step 1: Start the program.

Step 2: Read an expression in C.

Step 3: Create an entry in the symbol table for each valid identifier.

Step 4: Display each identifier with its type.

Step 5: Stop the program.

**Source Code**

//Implementation of symbol table

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

#include<stdlib.h>

#include<string.h>

#include<math.h>

void main()

{

int i=0,j=0,x=0,n;

void \*p,\*add[5];

char ch,srch,b[15],d[15],c;

printf("Expression terminated by $:");

while((c=getchar())!='$')

{

b[i]=c;

i++;

}

n=i-1;

printf("Given Expression:");

i=0;

while(i<=n)

{

printf("%c",b[i]);

i++;

}

printf("\n Symbol Table\n");

printf("Symbol \t addr \t type");

while(j<=n)

{

c=b[j];

if(isalpha(toascii(c)))

{

p=malloc(c);

add[x]=p;

d[x]=c;

printf("\n%c \t %d \t identifier\n",c,p);

x++;

j++;

}

else

{

ch=c;

if(ch=='+'||ch=='-'||ch=='\*'||ch=='=')

{

p=malloc(ch);

add[x]=p;

d[x]=ch;

printf("\n %c \t %d \t operator\n",ch,p);

x++;

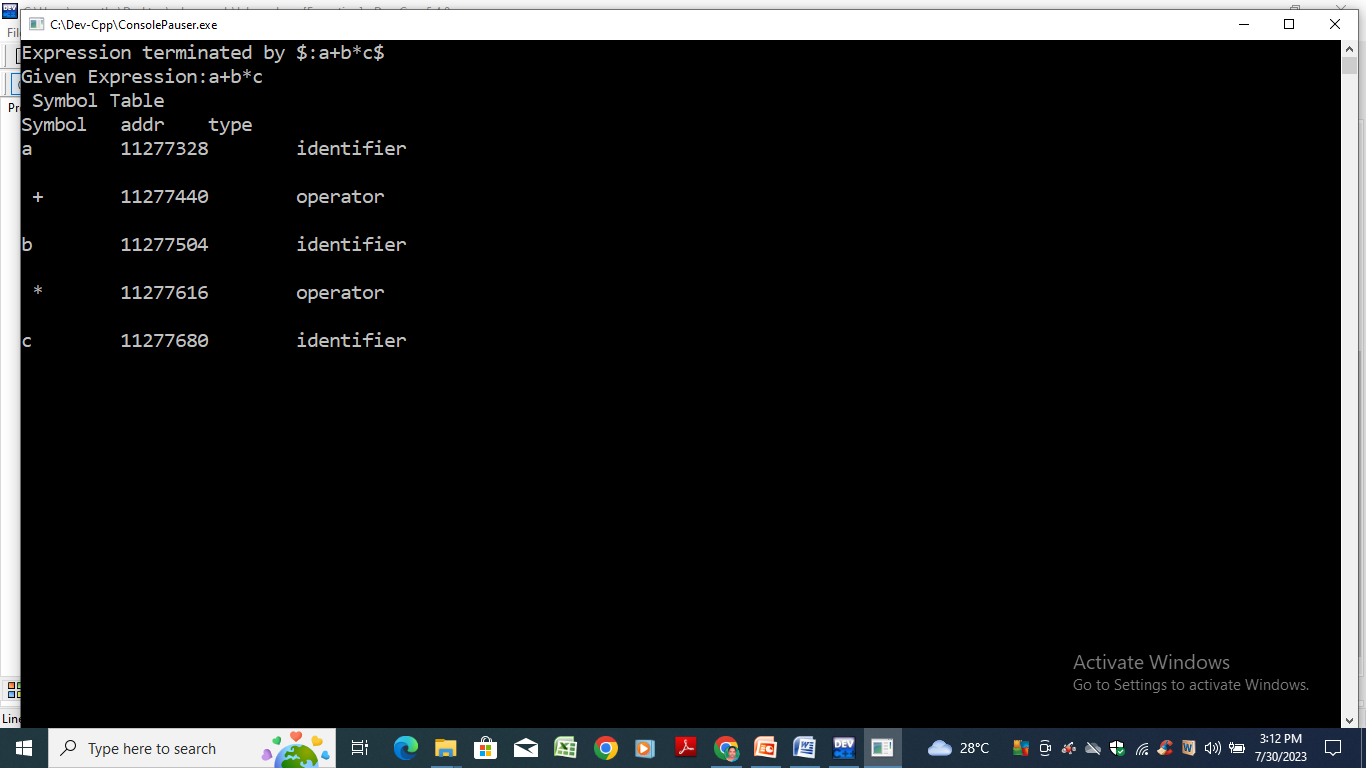
j++;

}}}

getch();

}

**OUTPUT**

****

**RESULT**

Thus the program for symbol table has been executed successfully.

**Ex: 2**

**Implementation of a Lexical Analyzer using LEX**

**Aim**

To write a program for implementing a Lexical Analyzer using LEX tool.

**Procedure**

1. A LEX program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%. The format is as follows: definitions %% rules %% user\_subroutines
2. In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. An identifier is defined such that the first letter of an identifier is alphabet and the remaining letters are alphanumeric.
3. In rules section, the left column contains the pattern to be recognized in an input file named yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.
4. Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.
5. When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.
6. In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.
7. The LEX command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the gcc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.
8. This LEX program reads a C file as input and displays the tokens of the input file as output.

**Source Code**

/\*Implementation of Lexical Analyzer using LEX tool – lexical.l \*/

%{

int COMMENT=0;

%}

identifier [a-zA-Z][a-zA-Z0-9]\*

%%

#.\* {printf("\n%s is a preprocessor directive",yytext);}

int |

float |

char |

double |

while |

for |

struct |

typedef |

do |

if |

break |

continue |

void |

switch |

return |

else |

goto {printf("\n\t%s is a keyword",yytext);}

"/\*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}

{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}

\{ {if(!COMMENT)printf("\n BLOCK BEGINS");}

\} {if(!COMMENT)printf("BLOCK ENDS ");}

{identifier}(\[[0-9]\*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}

\".\*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}

[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}

\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}

\( ECHO;

= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}

\<= |

\>= |

\< |

== |

\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}

%%

int main(int argc, char \*\*argv)

{

FILE \*file;

file=fopen("input.txt","r");

if(!file)

{

printf("could not open the file");

exit(0);

}

yyin=file;

yylex();

printf("\n");

return(0);

}

int yywrap()

{

return(1);

}

**INPUT**

//input.txt

#include<stdio.h>

#include<conio.h>

void main()

{

int a,b,c;

a=1;

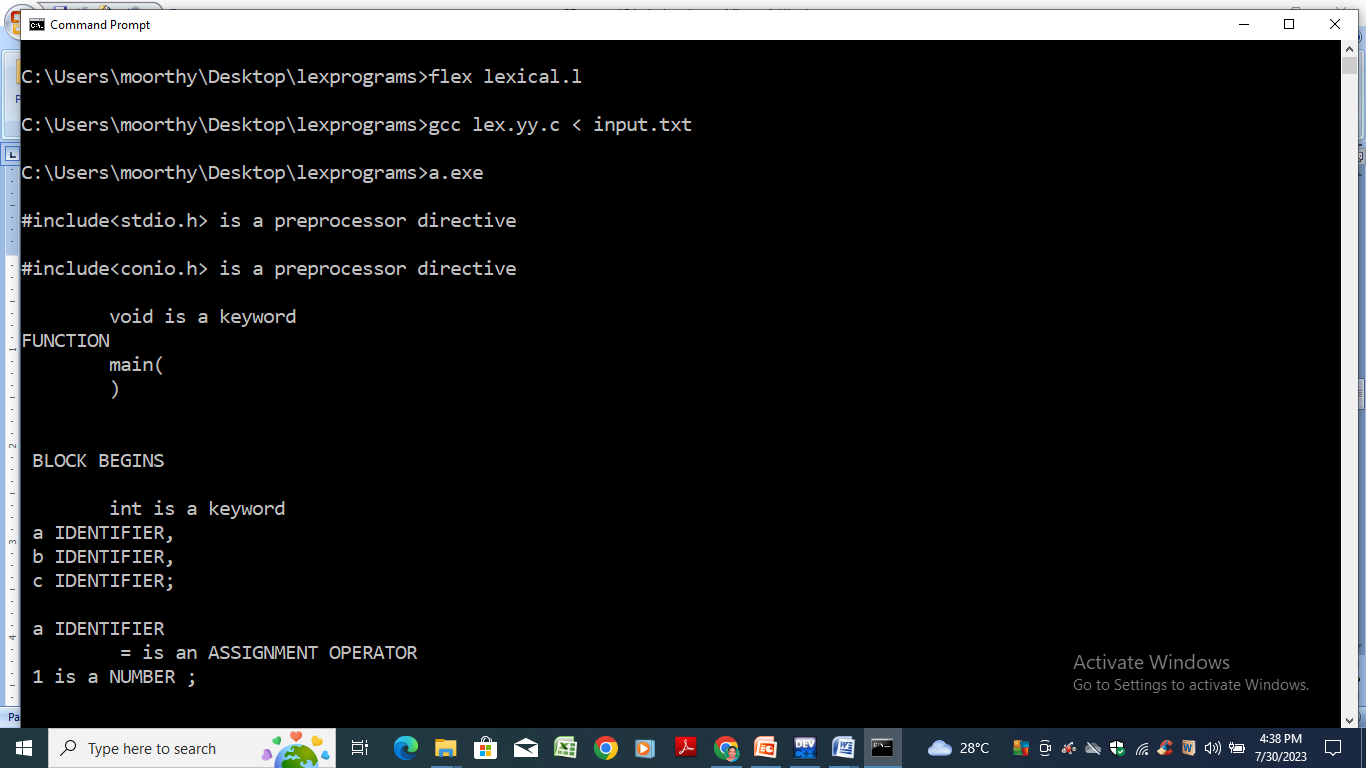
b=2;

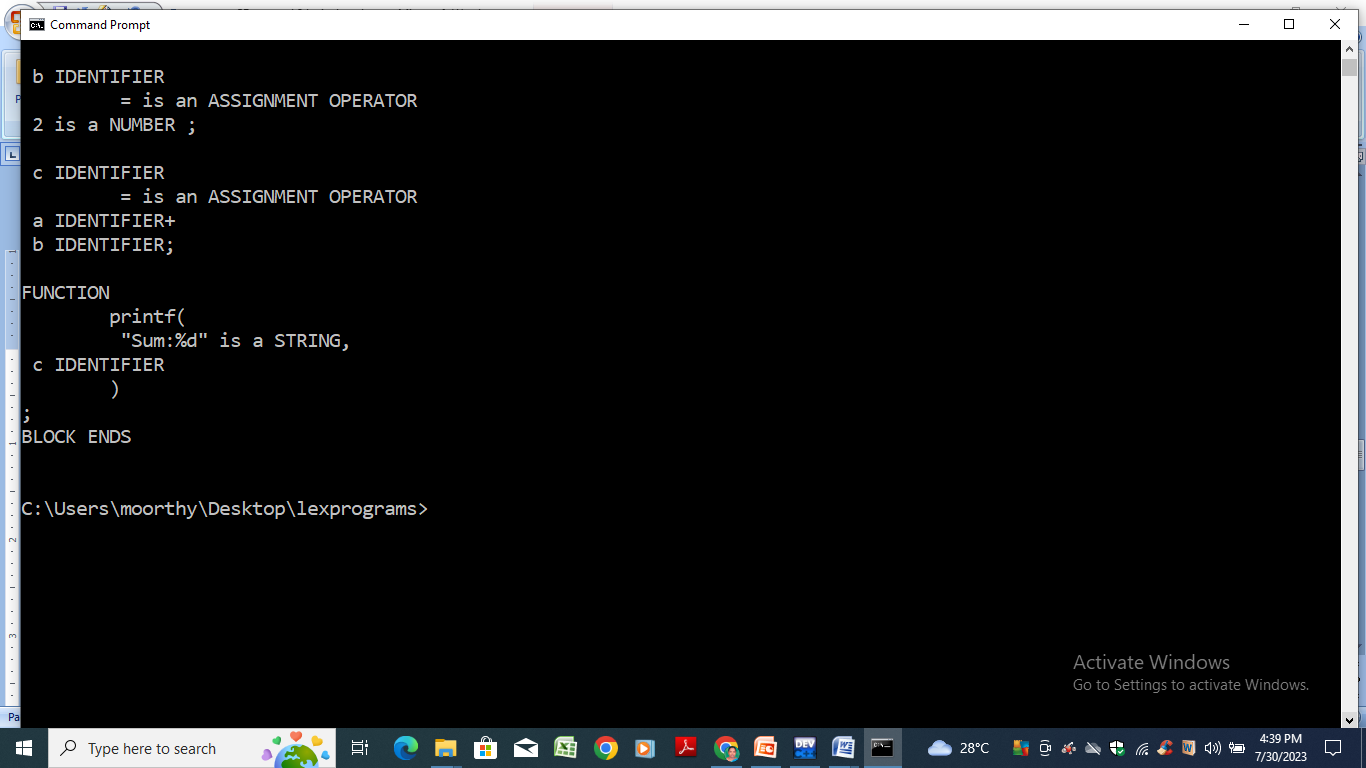
c=a+b;

printf("Sum:%d",c);

}

**OUTPUT**

****

****

**RESULT**

Thus the program for implementation of a Lexical Analyzer using LEX tool has been executed successfully.

**Ex: 3.a**

**Program to Recognize a Valid Arithmetic Expression using YACC**

**Aim**

To write a C program to recognize a valid arithmetic expression that uses operator +, -, \* and /.

**Algorithm**

Step1: Start the program.

Step2: Read an expression.

Step3: Check the validity of the given expression according to the rule using yacc.

Step4: Using expression rule print the result of the given values.

Step5: Stop the program.

**Source Code**

/\* LEX Code arith.l \*/

%{

#include "y.tab.h"

%}

%%

[a-zA-Z\_][a-zA-Z\_0-9]\* return id;

[0-9]+(\.[0-9]\*)? return num;

[+/\*] return op;

. return yytext[0];

\n return 0;

%%

int yywrap()

{

return 1;

}

/\* YACC Code arith.y \*/

%{

#include<stdio.h>

int valid=1;

%}

%token num id op

%%

start : id '=' s ';'

s : id x

| num x

| '-' num x

| '(' s ')' x

;

x : op s

| '-' s

|

;

%%

int yyerror()

{

valid=0;

printf("\nInvalid expression!\n");

return 0;

}

int main()

{

printf("\nEnter the expression:\n");

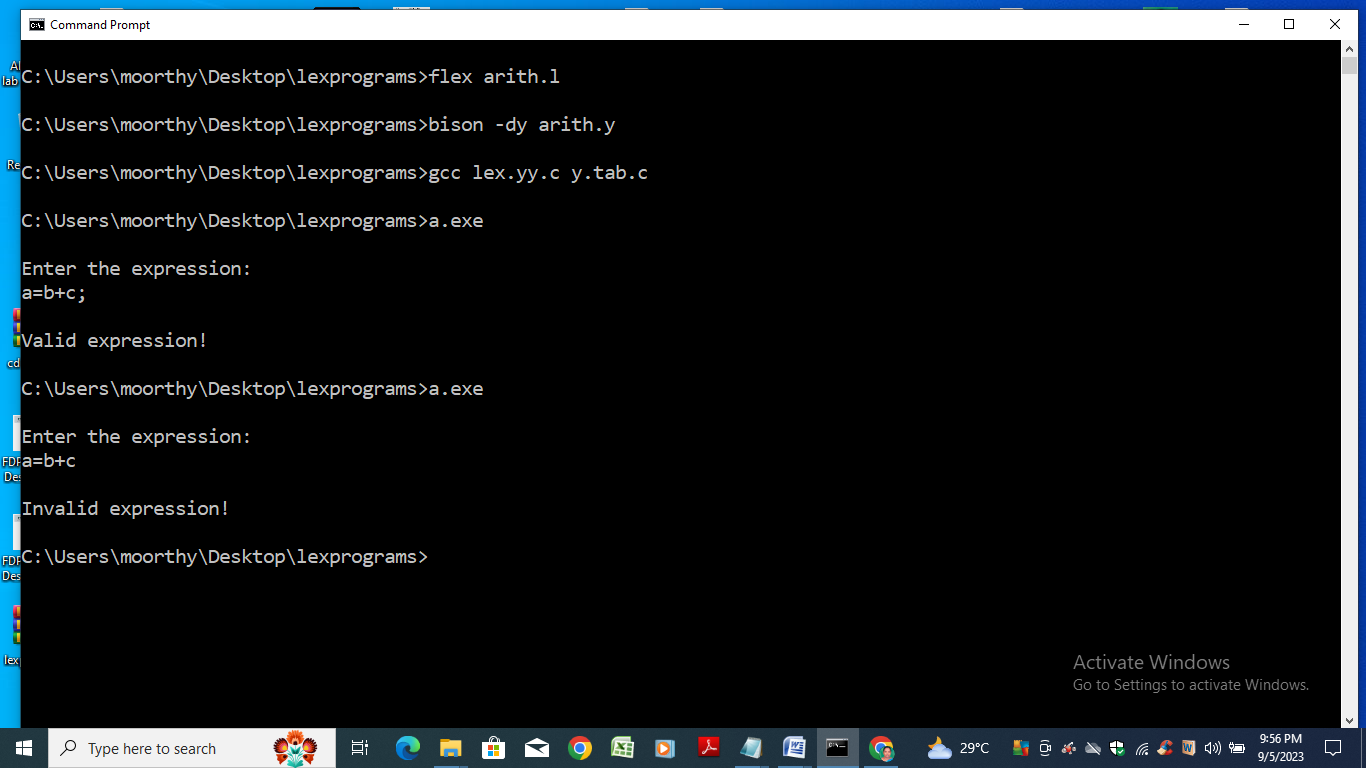
yyparse();

if(valid)

{ printf("\nValid expression!\n"); }

}

**Output**

****

**RESULT**

Thus the program to recognize a valid arithmetic expression using YACC has been executed successfully.

**Ex: 3.b**

**Program to Recognize a Valid Variable using YACC**

**Aim**

To write a C program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

**Algorithm**

Step1: Start the program.

Step2: Read the name of a variable.

Step3: Check the validity of the given variable according to the rule using YACC.

Step4: Display the result.

Step5: Stop the program.

**Source Code**

/\*Lex part: vartest.l \*/

%{

#include "y.tab.h"

%}

%%

[a-zA-Z\_][a-zA-Z\_0-9]\* return letter;

[0-9] return digit;

. return yytext[0];

\n return 0;

%%

int yywrap()

{

return 1;

}

/\* yacc part: vartest.y\*/

%{

#include<stdio.h>

int valid=1;

%}

%token digit letter

%%

start : letter s

s : letter s

| digit s

|

;

%%

int yyerror()

{

printf("\n Its not an identifier!\n");

valid=0;

return 0;

}

int main()

{

printf("\n Enter a name to test for identifier ");

yyparse();

if(valid)

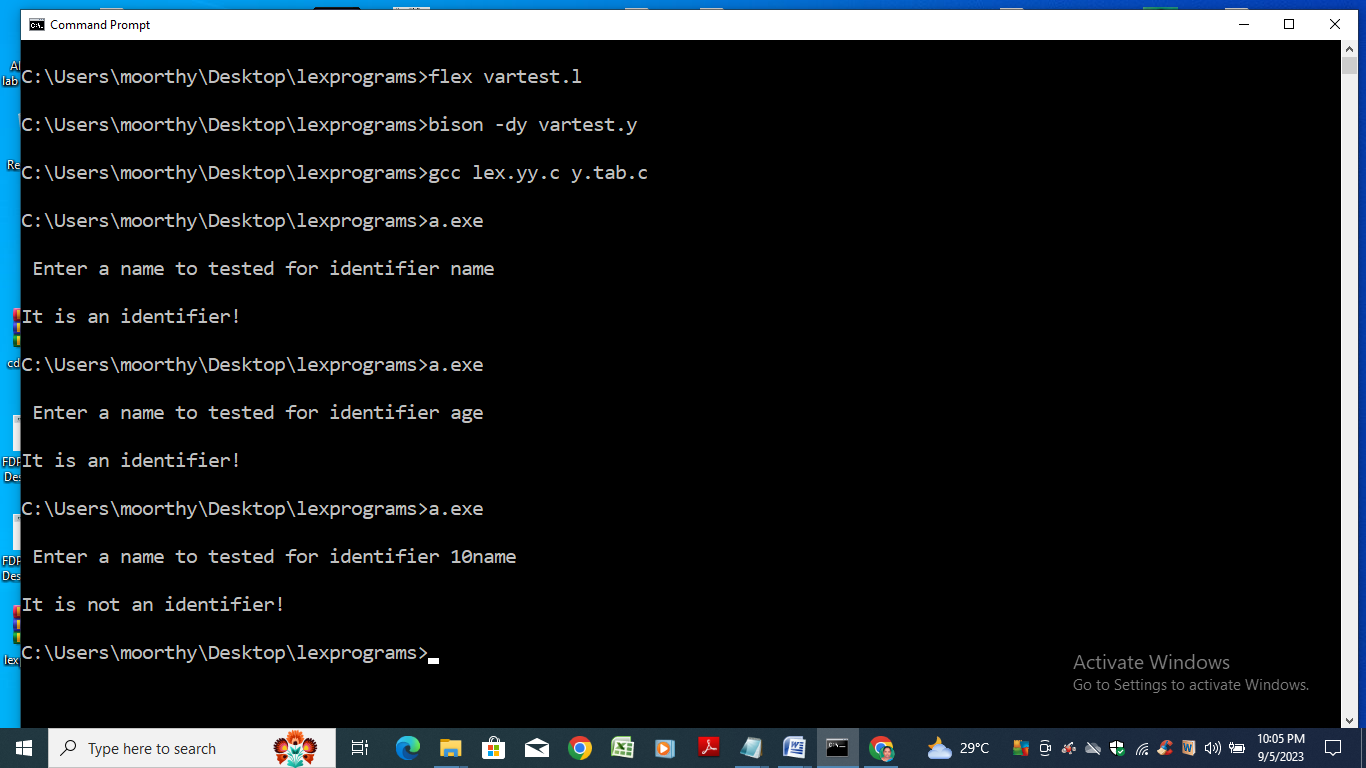
{

printf("\nIt is an identifier!\n");

}

}

**Output**

****

**RESULT**

Thus the program to recognize a valid variable using YACC has been executed successfully.

**Ex: 3.c**

**Program to Recognize a Valid Control Structure Syntax of C Language using YACC**

**Aim**

To write a C program to recognize avalid control structure syntax of C language using YACC.

**Algorithm**

Step1: Start the program.

Step2: Read a valid control structure.

Step3: Checking the validity of the given control structure according to the rule using YACC.

Step4: Display the result.

Step5: Stop the program.

**Source Code**

/\* YACC PART: iff.y\*/

%{

#include<stdio.h>

#include<stdlib.h>

int count=0;

%}

%token IF RELOP S NUMBER ID NL

%%

stmt: if\_stmt NL {printf("No. of nested if statements=%d\n",count);exit(0);}

;

if\_stmt : IF'('cond')''{'if\_stmt'}' {count++;}

|S

;

cond: x RELOP x

;

x:ID | NUMBER

;

%%

int yyerror(char \*msg)

{

printf("the statement is invalid\n");

exit(0);

}

main()

{

printf("enter the statement\n");

yyparse();

}

/\*LEX PART: iff.l\*/

%option noyywrap

%{

#include "y.tab.h"

%}

%%

"if" {return IF;}

[sS][0-9]\* {return S;}

"<"|">"|"=="|"<="|">="|"!=" {return RELOP;}

[0-9]+ {return NUMBER;}

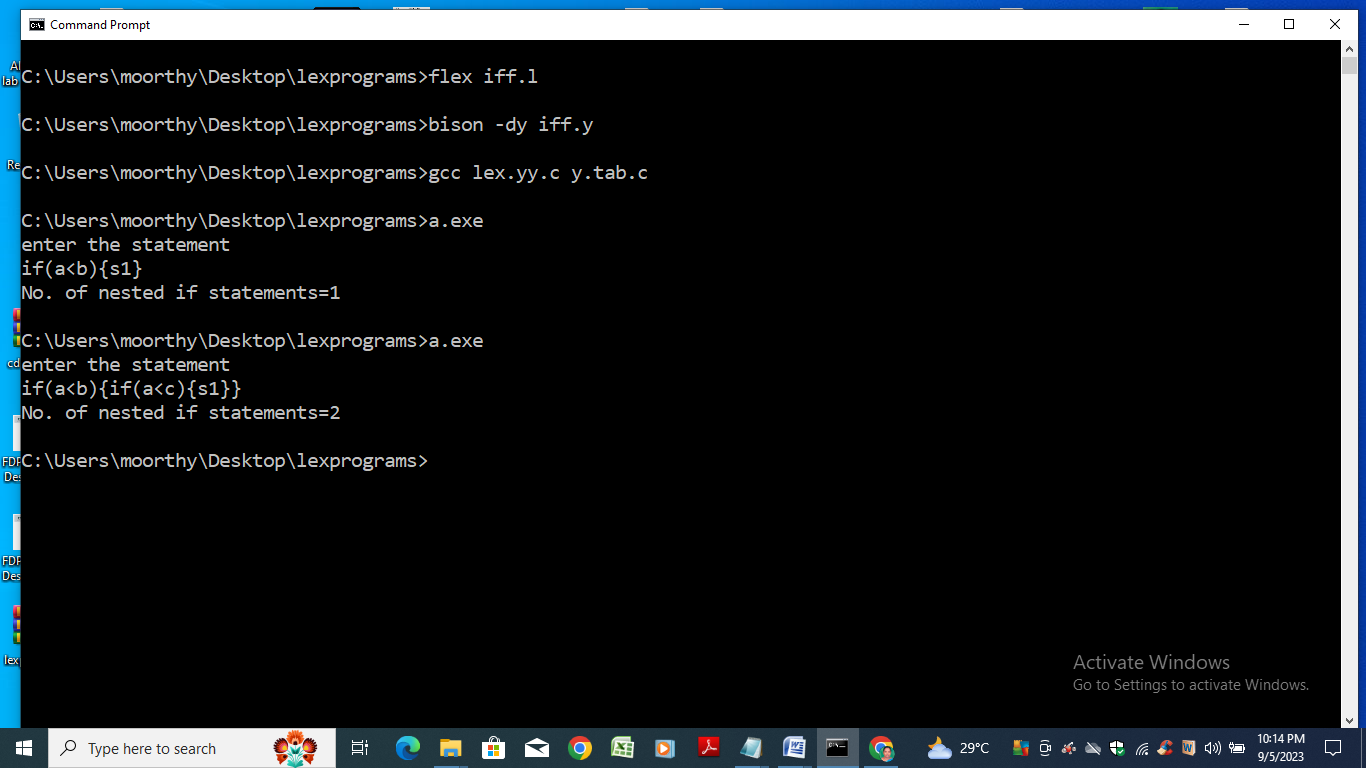
[a-z][a-zA-Z0-9\_]\* {return ID;}

\n {return NL;}

. {return yytext[0];}

%%

**Output**

****

**RESULT**

Thus the program to recognize a valid control structure syntax of C Language using YACC has been executed successfully.

.

**Ex: 3.d**

**Implementation of a Calculator using LEX and YACC**

**Aim**

To write a program for implementing a calculator using LEX and YACC.

**Algorithm**

Step1: A Yacc source program has three part: Declarations %% translation rules %% C routines

Step2: Declarations Section: This section contains entries that:

i. Include standard I/O header file.

ii. Define global variables.

iii. Define the list rule as the place to start processing.

iv. Define the tokens used by the parser.

v. Define the operators and their precedence.

Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of

a grammar production and the associated semantic action.

Step4: Programs Section: The programs section contains the following subroutines. Because,

these subroutines are included in this file, it is not necessary to use the yacc library when

process this file.

Step5: Main- The required main program that calls the yyparse subroutine to start the program.

Step6: yyerror(s) -This error-handling subroutine only prints a syntax error message.

Step7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs.

The calc.l file contains include statements for standard input and output, as programmer

file information if we use the -d flag with the yacc command. The y.tab.h file contains

definitions for the tokens that the parser program uses.

Step8: calci.l contains the rules to generate these tokens from the input stream.

**Source Code**

/\* lex code calci.l \*/

%{

/\* Definition section \*/

#include<stdio.h>

#include "y.tab.h"

extern int yylval;

%}

/\* Rule Section \*/

%%

[0-9]+ {

yylval=atoi(yytext);

return NUMBER;

}

[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

/\* Yacc code : calci.y \*/

%{

/\* Definition section \*/

#include<stdio.h>

int flag=0;

%}

%token NUMBER

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

/\* Rule Section \*/

%%

ArithmeticExpression:

E{

printf("\nResult=%d\n", $$);

return 0;

};

E:E'+'E {$$=$1+$3;}

|E'-'E {$$=$1-$3;}

|E'\*'E {$$=$1\*$3;}

|E'/'E {$$=$1/$3;}

|E'%'E {$$=$1%$3;}

|'('E')' {$$=$2;}

| NUMBER {$$=$1;}

;

%%

//driver code

void main()

{

printf("\nEnter Any Arithmetic Expression which can have operations Add, Sub, Mul, Div, Modulus and Round brackets:\n");

yyparse();

if(flag==0)

printf("\nEntered arithmetic expression is Valid\n\n");

}

void yyerror()

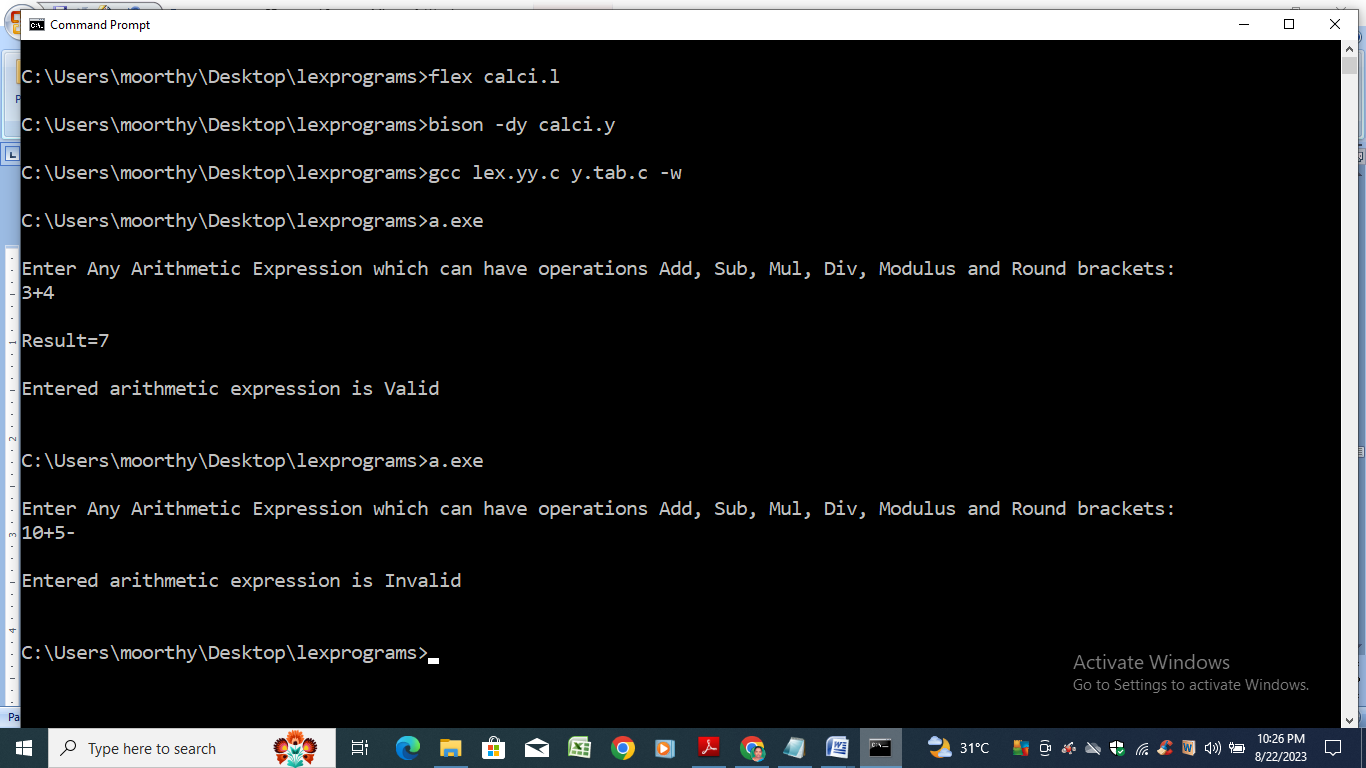
{

printf("\nEntered arithmetic expression is Invalid\n\n");

flag=1;

}

**Output**

****

**RESULT**

Thus the program to implement a calculator using LEX and YACC has been executed.

**Ex: 4**

**Intermediate Code Generation using LEX and YACC**

**Aim**

To write a program to generate intermediate code for a given expression using LEX and YACC.

.

**Algorithm**

Step1: Start the program.

Step2: Read the expression as input.

Step3: Define the functions for performing intermediate code generation.

Step4: Call the functions and generate three address code.

Step5: Display the three address code.

Step6: Stop the program.

**Source Code**

//Lex

%{

#include"y.tab.h"

extern char yyval;

%}

%%

[0-9]+ {yylval.symbol=(char)(yytext[0]);return NUMBER;}

[a-z] {yylval.symbol= (char)(yytext[0]);return LETTER;}

. {return yytext[0];}

\n {return 0;}

%%

//YACC

%{

#include"y.tab.h"

#include<stdio.h>

char addtotable(char,char,char);

int index1=0;

char temp = 'A'-1;

struct expr{

char operand1;

char operand2;

char operator;

char result;

};

%}

%union{

char symbol;

}

%left '+' '-'

%left '/' '\*'

%token <symbol> LETTER NUMBER

%type <symbol> exp

%%

statement: LETTER '=' exp ';' {addtotable((char)$1,(char)$3,'=');};

exp: exp '+' exp {$$ = addtotable((char)$1,(char)$3,'+');}

|exp '-' exp {$$ = addtotable((char)$1,(char)$3,'-');}

|exp '/' exp {$$ = addtotable((char)$1,(char)$3,'/');}

|exp '\*' exp {$$ = addtotable((char)$1,(char)$3,'\*');}

|'(' exp ')' {$$= (char)$2;}

|NUMBER {$$ = (char)$1;}

|LETTER {(char)$1;};

%%

struct expr arr[20];

void yyerror(char \*s)

{

printf("Errror %s",s);

}

char addtotable(char a, char b, char o)

{

temp++;

arr[index1].operand1 =a;

arr[index1].operand2 = b;

arr[index1].operator = o;

arr[index1].result=temp;

index1++;

return temp;

}

void threeAdd()

{

int i=0;

char temp='A';

while(i<index1)

{

printf("%c:=\t",arr[i].result);

printf("%c\t",arr[i].operand1);

printf("%c\t",arr[i].operator);

printf("%c\t",arr[i].operand2);

i++;

temp++;

printf("\n");

}

}

void fouradd()

{

int i=0;

char temp='A';

while(i<index1){

printf("%c\t",arr[i].operator);

printf("%c\t",arr[i].operand1);

printf("%c\t",arr[i].operand2);

printf("%c",arr[i].result);

i++;

temp++;

printf("\n");

}

}

int find(char l)

{

int i;

for(i=0;i<index1;i++)

if(arr[i].result==l) break;

return i;

}

void triple()

{

int i=0;

char temp='A';

while(i<index1)

{

printf("%c\t",arr[i].operator);

if(!isupper(arr[i].operand1))

printf("%c\t",arr[i].operand1);

else

{

printf("pointer");

printf("%d\t",find(arr[i].operand1));

}

if(!isupper(arr[i].operand2))

printf("%c\t",arr[i].operand2);

else

{

printf("pointer");

printf("%d\t",find(arr[i].operand2));

}

i++;

temp++;

printf("\n");

}

}

int yywrap()

{

return 1;

}

int main()

{

printf("Enter the expression: ");

yyparse();

threeAdd();

printf("\n");

fouradd();

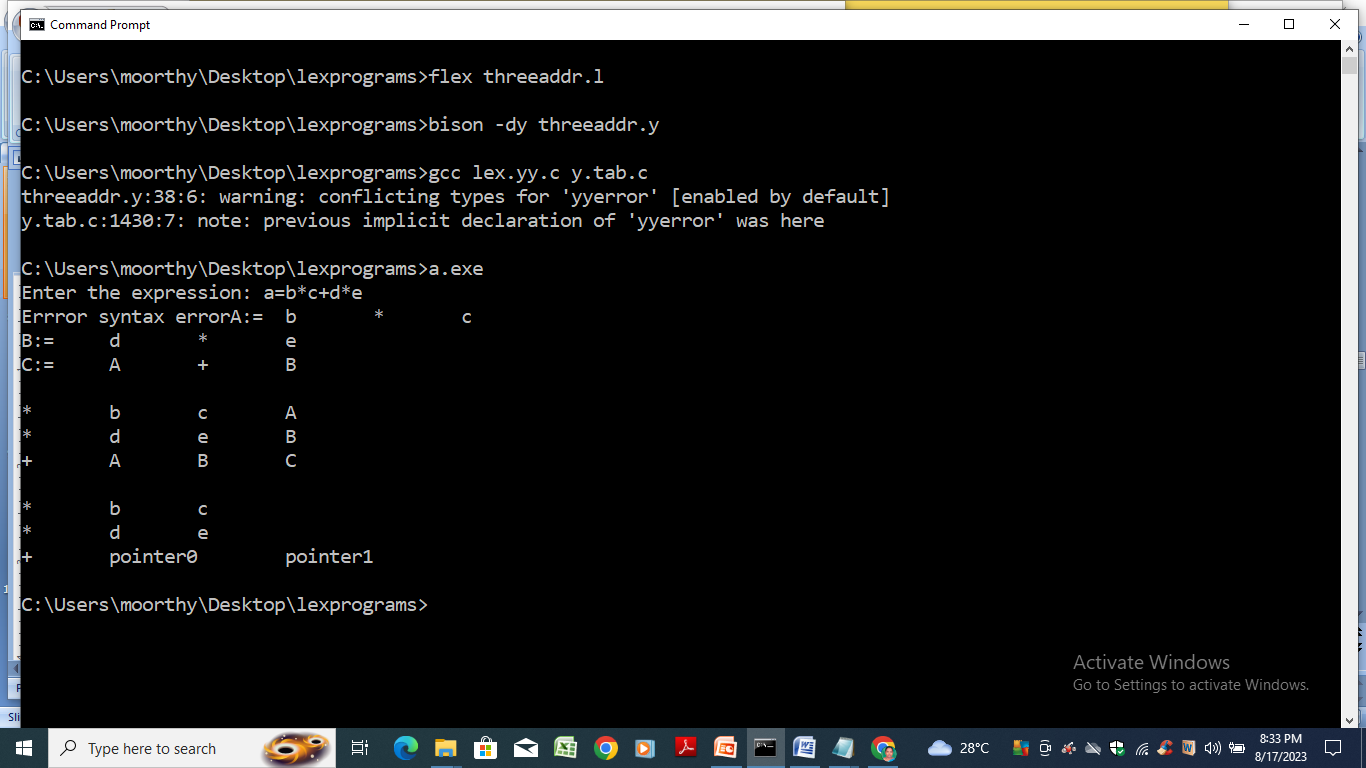
printf("\n");

triple();

return 0;

}

**OUTPUT**



**RESULT**

Thus the program to generate intermediate code using LEX and YACC has been executed successfully.

**Ex: 5**

**Implementation of Type Checking Using C**

**Aim**

To write a C program to develop a lexical analyzer to recognize a few patterns in C.

**Algorithm**

Step0: Start the program.

Step1: Track the global scope type information (e.g. classes and their members).

Step2: Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types

upwards.

Step3: If type found correct, do the operation.

Step4: Type mismatches, semantic error will be notified

Step5: Stop the program.

**Source Code**

/\* To implement type checking

Type checking is the process of verifying that the types of operands in a program are compatible with the operator being used and that the program is free from type errors before the code is executed.

\*/

#include<stdio.h>

#include<stdlib.h>

int main()

{

int n,i,k,flag=0;

char vari[15],typ[15],b[15],c;

printf("Enter the number of variables:");

scanf(" %d",&n);

for(i=0;i<n;i++)

{

printf("Enter the variable[%d]:",i);

scanf(" %c",&vari[i]);

printf("Enter the variable-type[%d](float-f,int-i):",i);

scanf(" %c",&typ[i]);

if(typ[i]=='f')

flag=1;

}

printf("Enter the Expression(end with $):");

i=0;

getchar();

while((c=getchar())!='$')

{

b[i]=c;

i++; }

k=i;

for(i=0;i<k;i++)

{

if(b[i]=='/')

{

flag=1;

break; } }

for(i=0;i<n;i++)

{

if(b[0]==vari[i])

{

if(flag==1)

{

if(typ[i]=='f')

{ printf("\nthe datatype is correctly defined..!\n");

break; }

else

{ printf("Identifier %c must be a float type..!\n",vari[i]);

break; } }

else

{ printf("\nthe datatype is correctly defined..!\n");

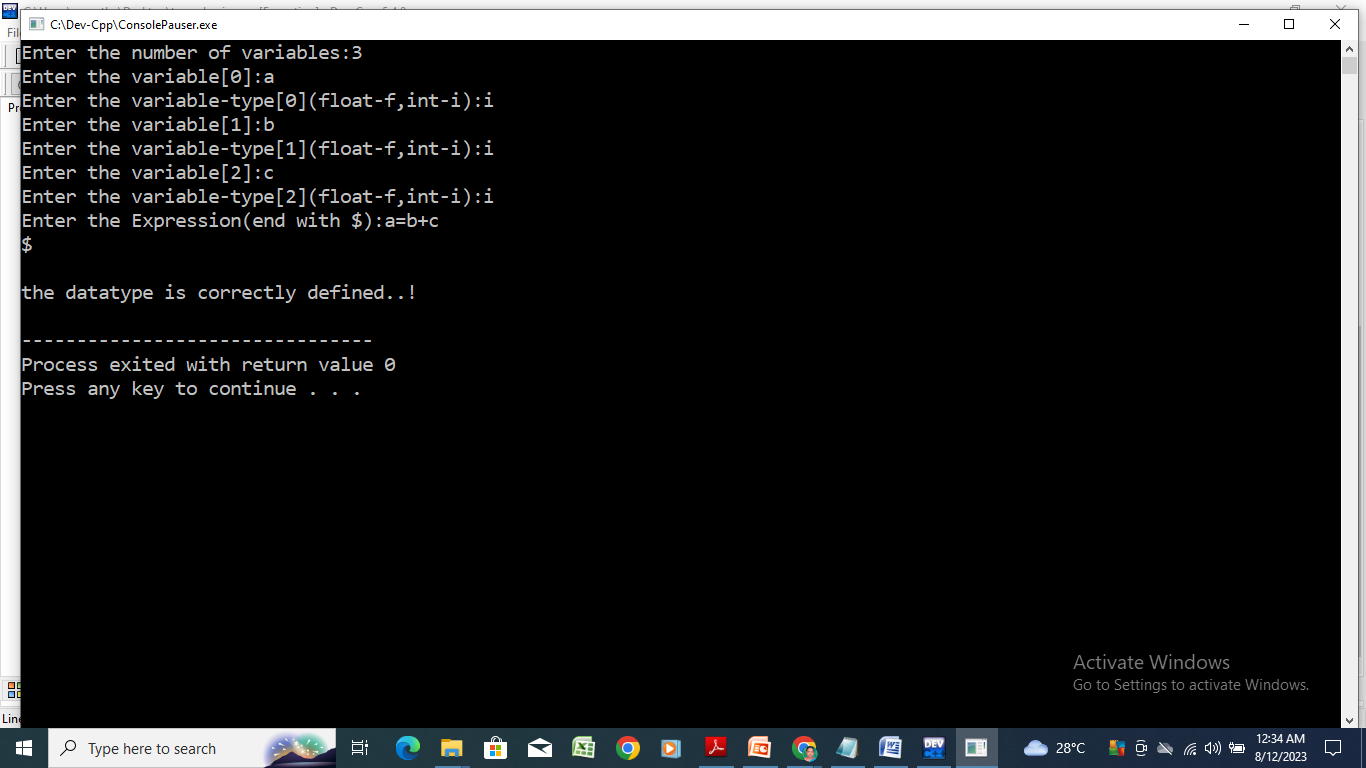
break; } }

}

return 0;

}

**OUTPUT:**

****

**Ex: 6**

**Implementation of Code Optimization for a Simple C Program**

**Aim**

To write a program for performing code optimization.

**Algorithm**

Step1: Step Start the program.

Step2: Read the three address codes from left to right. // **Common Sub expression elimination**

Step3: Store the first three address code in a string.

Step4: Compare the string with the other expressions.

Step5: If there is a match, remove the expression.

Step6: Read the next three address code from left to right and store in a string. Perform steps 4

to 6 for each three address code. //**Dead code elimination**

Step7: Read the first three address code from left to right and get the operands.

Step8: Check each operand is used in any other three address code.

Step9: If the operands are not used, then eliminate the complete three address code.

Step10: Read the next three address code from left to right and get the operands. Go to step 8. If

there is no next three address code, go to step 11.

Step11: Display the optimized code.

Step12: Stop the program.

**Source Code**

//Code Optimization Technique

#include<stdio.h>

#include<string.h>

struct op

{

char l;

char r[20];

}

op[10],pr[10];

void main()

{

int a,i,k,j,n,z=0,m,q;

char \*p,\*l;

char temp,t;

char \*tem;

printf("Enter the Number of Values:");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("left: ");

scanf(" %c",&op[i].l);

printf("right: ");

scanf(" %s",&op[i].r);

}

printf("Intermediate Code\n") ;

for(i=0;i<n;i++)

{

printf("%c=",op[i].l);

printf("%s\n",op[i].r);

}

for(i=0;i<n-1;i++)

{

temp=op[i].l;

for(j=0;j<n;j++)

{

p=strchr(op[j].r,temp);

if(p)

{

pr[z].l=op[i].l;

strcpy(pr[z].r,op[i].

r);

z++;

}

}

}

pr[z].l=op[n-1].l;

strcpy(pr[z].r,op[n-1].r);

z++;

printf("\nAfter Dead Code Elimination\n");

for(k=0;k<z;k++)

{

printf("%c\t=",pr[k].l);

printf("%s\n",pr[k].r);

}

for(m=0;m<z;m++)

{

tem=pr[m].r;

for(j=m+1;j<z;j++)

{

p=strstr(tem,pr[j].r);

if(p)

{

t=pr[j].l;

pr[j].l=pr[m].l;

for(i=0;i<z;i++)

{

l=strchr(pr[i].r,t) ;

if(l)

{

a=l-pr[i].r;

printf("pos: %d\n",a);

pr[i].r[a]=pr[m].l;

}}}}}

printf("Eliminate Common Expression\n");

for(i=0;i<z;i++)

{

printf("%c\t=",pr[i].l);

printf("%s\n",pr[i].r);

}

for(i=0;i<z;i++)

{

for(j=i+1;j<z;j++)

{

q=strcmp(pr[i].r,pr[j].r);

if((pr[i].l==pr[j].l)&&!q)

{

pr[i].l='\0';

}

}

}

printf("Optimized Code\n");

for(i=0;i<z;i++)

{

if(pr[i].l!='\0')

{

printf("%c=",pr[i].l);

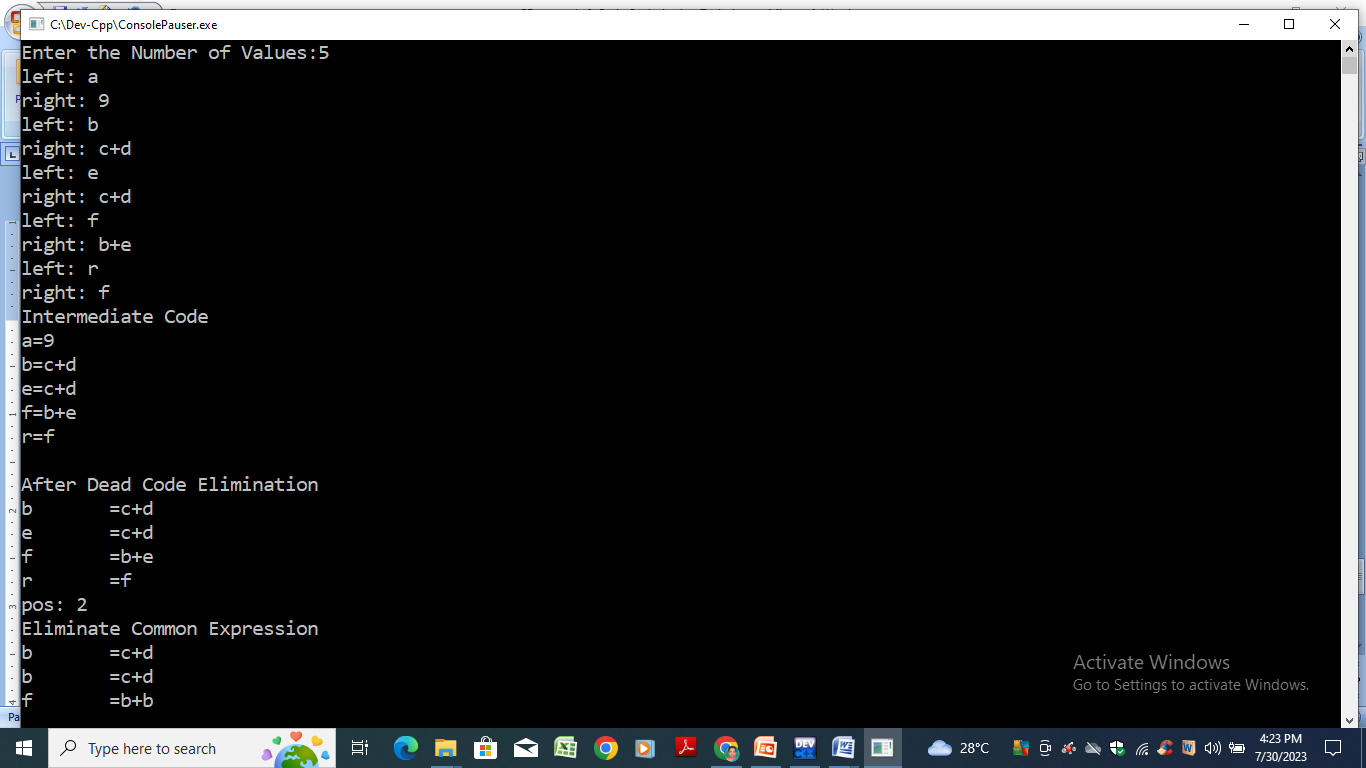
printf("%s\n",pr[i].r);

}

}

}

**Output**

****

**Result**

Thus the program to perform code optimization is compiled and executed successfully and the output is verified.

**Ex: 7**

**Implementation of Back-End of the Compiler**

**Aim**

To Write a C program to implement the Back-end of the compiler.

**Algorithm**

Step1: Start the program.

Step2: Read the given three address codes.

Step3: Generate machine code for each three address code.

Step4: Display a list of machine codes for all the given three address codes.

Step5: Stop the Program.

**Source Code**

#include<stdio.h>

#include<string.h>

#include<conio.h)

main()

{ char icode[10][30],str[20],opr[10]; int i=0;

//clrscr();

printf("\n Enter the set of intermediate code (terminated by exit):\n");

do

{

scanf("%s",icode[i]);

} while(strcmp(icode[i++],"exit")!=0); printf("\n target code generation");

printf("\n\*\*\*\*\*\*\*\*\*\*\*");

i=0;

do

{

strcpy(str,icode[i]);

switch(str[3])

{

case '+': strcpy(opr,"ADD"); break;

case '-': strcpy(opr,"SUB"); break;

case '\*': strcpy(opr,"MUL");break;

case '/':strcpy(opr,"DIV"); break;

}

printf("\n\tMov %c,R%d",str[2],i);

printf("\n\t%s%c,R%d",opr,str[4],i);

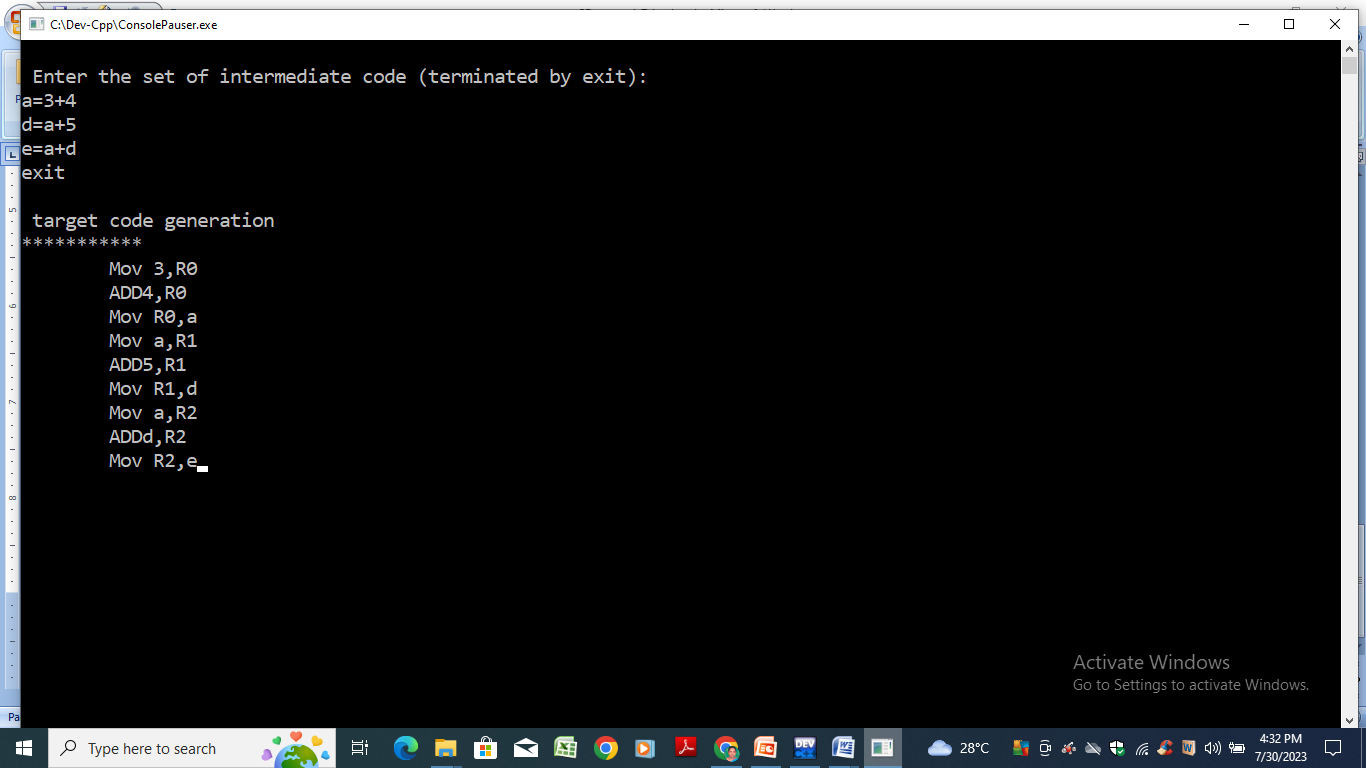
printf("\n\tMov R%d,%c",i,str[0]);

}while(strcmp(icode[++i],"exit")!=0);

//getch();

}

**Output**

****

**Result**

Thus the program to implement the Back-end of the compiler is compiled and executed successfully and the output is verified.