**CS3501COMPILER DESIGN LABORATORY**

**COURSE OBJECTIVES:**

* To learn the various phases of compiler.
* 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:**

**Upon completion of this course, student will be able to**

CO1:Understand the techniques in different phases of a compiler.

CO2:Design a lexical analyzer 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.No:1 IMPLEMENTATION OF LEXICAL ANALYZER**

**Date:**

**AIM:**

To develop a lexical analyzer to identify identifiers, constants, comments, operators etc using the C program.

**ALGORITHM:**

**Step 1:** Start the program.  
**Step 2:** Declare all the variables and file pointers.  
 **Step 3:** Display the input program.  
**Step 4:** Separate the keyword in the program and display it.  
**Step 5:** Display the header files of the input program  
**Step 6:** Separate the operators of the input program and display it.  
 **Step 7:** Print the punctuation marks.  
**Step 8:**Print the constant that is present in the input program.  
**Step 9:** Print the identifiers of the input program.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

void main()

{

inti,j,k,flag=0,p=0,count;

char a[20][20]={"",""},temp[20];

char\*key[]={"if","float","int","char","void","main","while","else","do",

"printf","scanf","switch","for","while","double"};

clrscr();

printf("Enter the c program - Enter zz at the end.\n");

while(1)

{

scanf("%s",a[p]);

if(!strcmp(a[p],"zz"))

break;

p=p+1;

}

count=0;

while(count<p)

{

for(i=0;i<strlen(a[count]);i++)

{

if((a[count][i]=='/')&&(a[count][i+1]=='/'))

{

printf("%s is comment line\n",a[count]);

i=strlen(a[count]);

}

if((a[count][i]=='/')&&(a[count][i+1]=='\*'))

{

i=i+2;

while((a[count][i]!='\*')&&(a[count][i+1]!='/'))

{

printf("%c",a[count][i]);

i=i+1;

if(i>=strlen(a[count]))

{

count=count+1;

i=0;

}

}

i=i+2;

printf(" is comment line\n");

}

if((a[count][i]=='#'))

{

printf("%s is a header file\n",a[count]);

i=strlen(a[count]);

}

if(isalpha(a[count][i]))

{

j=0;

while(isalnum(a[count][i]))

{

temp[j]=a[count][i];

i++;

j++;

}

i=i-1;

temp[j]='\0';

for(k=0;k<15;k++)

{

if(!strcmp(key[k],temp))

{

flag=1;

break;

}

else

flag=0;

}

if(flag==1)

printf("%s is a keyword\n",temp);

else

printf("%s is an identifier\n",temp);

}

else if(isdigit(a[count][i]))

{

j=0;

while(isdigit(a[count][i]))

{

temp[j]=a[count][i];

i++;

j++;

}

temp[j]='\0';

printf("%s is number\n",temp);

}

else if((a[count][i]=='=')&&(a[count][i+1]=='='))

{

i=i+1;

printf("%c%c is an equivalent operator\n",a[count][i- 1],a[count][i]);

}

else if((a[count][i]=='='))

printf("%c this is an assignment operator\n",a[count][i]);

else if((a[count][i]=='!')&&(a[count][i+1]=='='))

{

printf("%c%c this is an not equal to operator\n",a[count][i],a[count][i+1]);

i=i+1;

}

else if((a[count][i]=='?')&&(a[count][i+1]==':'))

{

printf("%c%c this is an ternary

operator\n",a[count][i],a[count][i+1]);

i=i+1;

}

else if((a[count][i]=='>')&&(a[count][i+1]=='='))

{

printf("%c%c this is an greater than or equal to opertor\n",a[count][i],a[count][i+1]);

i=i+1;

}

else if((a[count][i]=='<'))

printf("%c is less than operator\n",a[count][i]);

else if((a[count][i]=='>'))

printf("%c is greater than operator\n",a[count][i]);

else if((a[count][i]=='<')&&(a[count][i+1]=='='))

{

printf("%c%c this is an less than or equal to operator\n",a[count][i],a[count][i+1]);

i=i+1;

}

else if((a[count][i]=='+')&&(a[count][i+1]=='+'))

{

printf("%c%c this is an increment operator\n",a[count][i],a[count][i+1]);

i=i+1;

}

else if((a[count][i]=='-')&&(a[count][i+1]=='-'))

{

printf("%c%c this is an decrement operator\n",a[count][i],a[count][i+1]);

i=i+1;

}

else if((a[count][i]=='(')||(a[count][i]==')')||(a[count][i]=='{')||(a[count][i]=='}'))

printf("%c is paranthesis\n",a[count][i]);

else

if((a[count][i]=='+')||(a[count][i]=='-')||(a[count][i]=='\*')||(a[count][i]=='/')||(a[count][i]=='%'))

printf("%c is an arithmetic operator\n",a[count][i]);

}

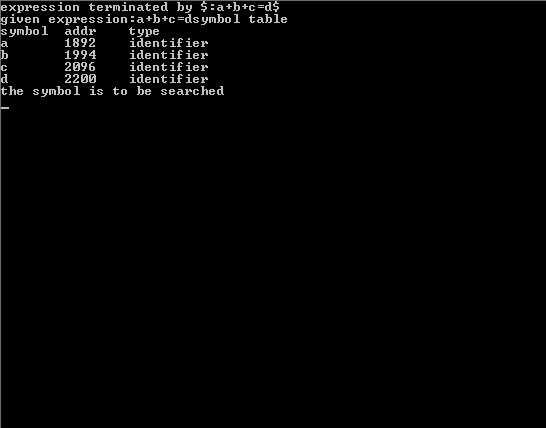
count=count+1;

}

getch();

}

**OUTPUT:**



**4.** List the phases that constitute the front end of a

compiler

**4.** List the phases that constitute the front end of a

compiler

**4.** List the phases that constitute the front end o

**RESULT:**

Thus the program for the exercise on lexical analysis using lex has been successfully executed

and output is verified

**Ex.No:2 IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL**

**Date:**

**AIM:**

To write a program for implementing a Lexical analyzer using the LEX tool.

**ALGORITHM:**

**Step 1:**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

**Step 2:**In the definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is the alphabet and the remaining letters are alphanumeric.

**Step 3:**In the rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, newline characters, numbers, strings, identifiers, beginning and end of blocks, comment statements, preprocessor directive statements etc.

**Step 4:**Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

**Step 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.

**Step 6:**In the user subroutine section, the main routine calls yylex(). yywrap() is used to get more input.

**Step7:**The LEX command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc 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.

**PROGRAM:**

%{

int com=0;

%}

delim [\t\n]

ws {delim}+

key "int"|"float"|"char"|"double"|"do"|"if"|"else"|"for"|"switch"|"void"|"main"

letter [A-Z a-z]

digit [0-9]

id {letter}+({letter}|{digti})\*

number {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

op [+-/\*<>=]

pun ["|%|.|:|,|:]

par [“(“|”)”|”{“|”}”]

%%

{ws} {}

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

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

{number} {printf("%s is a number\n",yytext);}

{op} {printf("%s is a operator\n",yytext);}

"/\*".\* {com=1;printf("comment line begins %s",yytext);}

.\*"\*/" {com=0;printf("comment line ends %s",yytext);}

"//" {printf("%s is a comment line",yytext);}

{id} {if(!com) printf("%s is an identifier\n",yytext);}

else {printf("%s",yytext);}

{pun} {printf("%s is a punctuator",yytext);}

{par} {printf("%s is paranthesis",yytext);}

%%

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

{

if(argc>1)

{

FILE \*f1;

f1=fopen(argv[1],"r");

if(!f1)

exit(0);

yyin=f1;

}

yylex();

}

intyywrap()

{

return 1;

}

**INPUT FILE**:

#include<stdio.h>

void main()

{

int a=0;

if(a>13)

printf("Hello");

/\*hai\*/

//welcome//

}

**OUTPUT:**



**RESULT:**

Thus the program for the exercise on lexical analysis using lex has been successfully executed and output is verified.

**Ex.No:3**

**Date: RECOGNIZING A VALID ARITHMETIC EXPRESSION**

**AIM:**

To write a YACC program to recognize a valid arithmetic expression.

**ALGORITHM:**

Step1: Start the program.

Step2: Reading an expression.

Step3: Checking the validating 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

**PROGRAM:**

**//Recognizing a Valid Expression using YACC**

%{

#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

%}

%token num,let

%left '+''-''\*''/'

%%

stmt:expr"\n" {printf("VALID EXPRESSION.\n"); exit(0);}|

error{yyerror(" NOT VALID "); exit(0); }

expr: expr'+'expr|expr'-'expr|expr'\*'expr|expr'/'expr|'('expr')'|num1|let1;

num1:num1 num|num;

let1:let1 let|let;

%%

main()

{

printf("Enter an Expression :");

yyparse();

}

yylex()

{

char ch;

ch=getchar();

if(isdigit(ch))

return num;

if(isalpha(ch))

return let;

returnch;

}

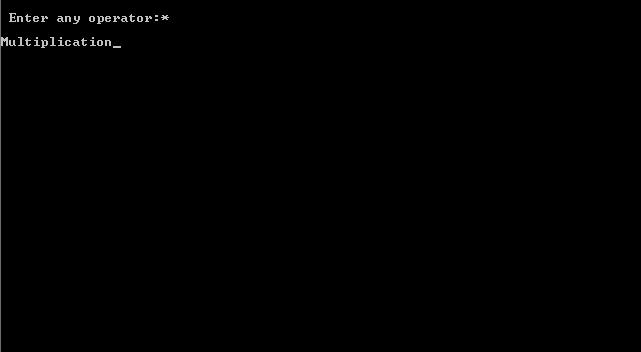
yyerror(char \*s)

{

printf("%s",s);

}

**OUTPUT:**



**RESULT:**

Thus the program for the exercise on the syntax using YACC has been executed successfully and Output is verified.

Mention the various notational shorthands for

representing regular expression

**Ex.No:4**

**Date:**

**RECOGNIZING A VALID VARIABLE**

**AIM :**

To write a YACC 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: Reading an identifier

Step3: Checking the validating of the given identifieraccording to the rule using YACC.

Step4: Using identifier rule print the result of the given values

Step5: Stop the program

**PROGRAM:**

**//Recognizing a Valid Identifier using YACC**

%{

#include<stdio.h>

#include<ctype.h>

%}

%token let dig

%%

TERM:XTERM '\n'{printf("\nACCEPTED\n");

exit(0);

}

|error{yyerror("REJECTED\n");

exit(0);

}

;

XTERM:XTERMlet|XTERMdig|let

;

%%

yylex()

{

char ch;

ch=getchar();

if(isalpha(ch))

return let;

if(isdigit(ch))

return dig;

returnch;

}

main()

{

printf("Enter a variable:");

yyparse();

}

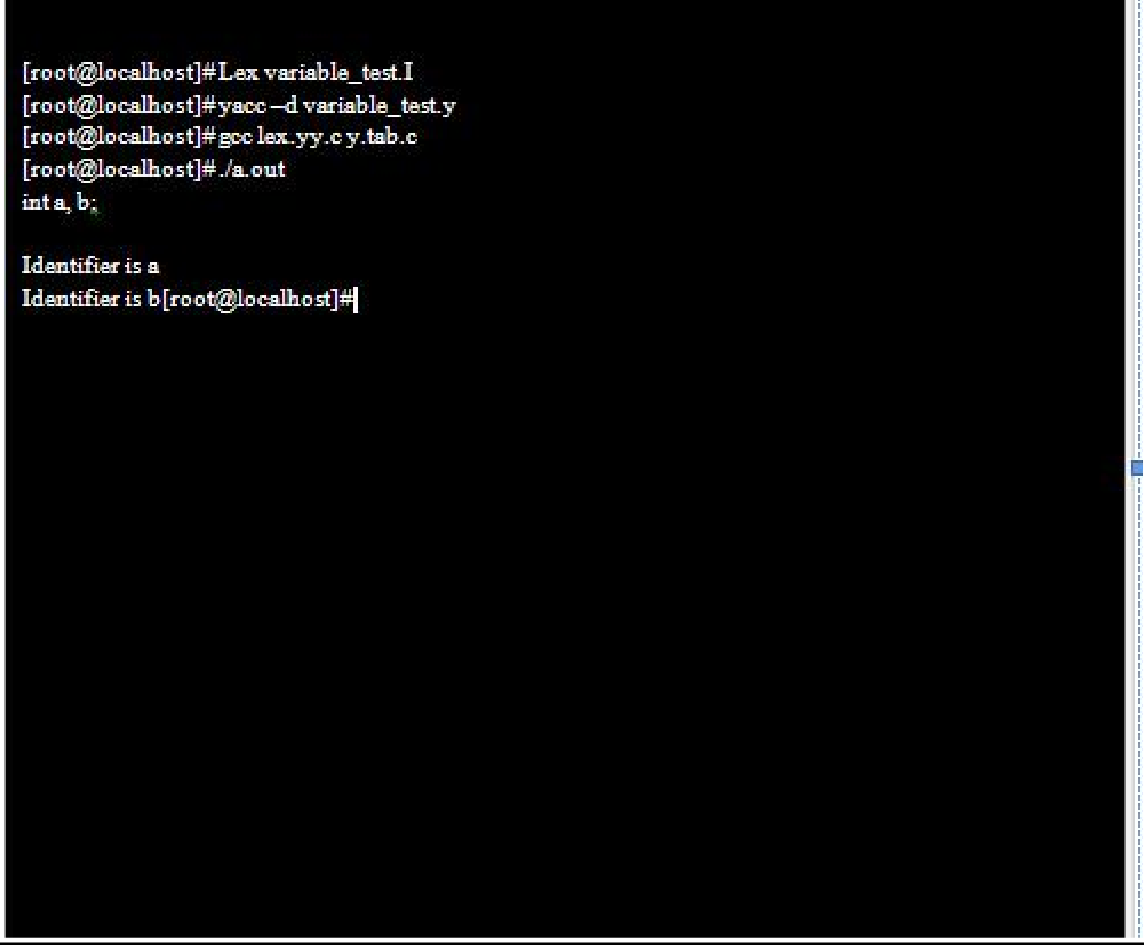
yyerror(char \*s)

{

printf("%s\n",s);

}

**OUTPUT:**



**RESULT :**

Thus the program for the exercise on the syntax using YACC has been executed successfully and Output is verified.

**Ex.No:5**

**Date: RECOGNIZING A VALID CONTROL STRUCTURES SYNTAX OF C LANGUAGE**

**AIM:**

To write a YACC program to recognize a valid control structures syntax of C language

**AGORITHM**:

Step1: Start the program.

Step2: Reading a control structure.

Step3: Checking the validating of the given control structure according to the rule using yacc.

Step4: Using control structure rule print the result

Step5: Stop the program

**PROGRAM**:

%{

#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:

CODE:(nif.l)

%{

#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 for the exercise on YACC program to recognize valid control structures syntax of C language has been executed successfully and Output is verified.

**Ex.No:6**

**Date:**

**IMPLEMENTATION OF ARITHMETIC CALCULATOR USING LEX AND YACC**

**AIM:**

To write a program for implementing a calculator for computing the given expression using semantic rules of the YACC tool and LEX.

**ALGORITHM:**

**Step 1:**A YACC source program has three parts as follows:

       Declarations %% translation rules %% supporting C routines

**Step 2:**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.

**Step 3:**Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

**Step 4:**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 processing this file.

**Step 5:**Main- The required main program that calls the yyparse subroutine to start the program.

**Step 6:**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.lex 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.

**Step 8:**calc.lex contains the rules to generate these tokens from the input stream.

**PROGRAM:**

**Calc1.l**

%{

#include<stdio.h>

#include<math.h>

#include "y.tab.h"

%}

%option noyywrap

%%

[0-9]+ {yylval.dval=atoi(yytext);

return num;

}

\n|. {returnyytext[0];}

%%

void yyerror(char \*str)

{

printf("%s",str);

exit(1);

}

int main()

{

printf("Enter Expression");

yyparse();

return(0);

}

**Calciii.y**

%{

#include<stdio.h>

int yylex(void);

#include<ctype.h>

%}

%union

{

floatdval;

}

%token<dval>num

%left '+' '-'

%left '\*' '/'

%right '^'

%nonassoc UMINUS

%type<dval>e

%%

s:e '\n' {printf("%f",$1);exit(0);}|error '\n'{yyerror("reenter the input");exit(0);}

e:num|e'+'e {$$=$1+$3;}

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

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

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

|e'^'e {inti,j=$1;

for(i=1;i<$3;i++)

{

j=j\*$1;

$$=j;

}

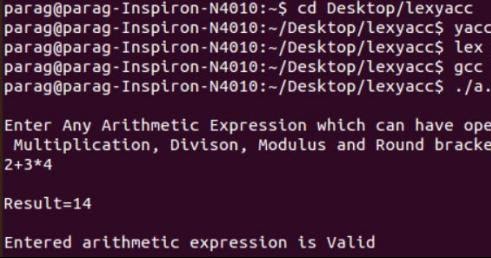
}

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

|'-'e%prec UMINUS {$$=-$2;};

%%

**OUTPUT:**



**RESULT:**

Thus the program for implementing a calculator for computing the given expression using semantic rules of the YACC tool and LEX.

**Ex.No:7**

**Date:**

**GENERATION OF THREE-ADDRESS CODE USING LEX AND YACC**

**AIM:**

To write a program for generating three-address code using semantic rules of the YACC tool and LEX.

**ALGORITHM:**

**Step 1:**A YACC source program has three parts as follows:

       Declarations %% translation rules %%  supporting C routines

**Step 2:**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.

**Step 3:**Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

**Step 4:**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 processing this file.

**Step 5:**Main- The required main program that calls the yyparse subroutine to start the program.

**Step 6:**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.lex 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.

**Step 8:**yacc.y contains the rules to generate three address code.

**PROGRAM:**

**lex.l**

%{

#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.y**

%{

#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");

}

}

int yywrap(){

return 1;

}

int main(){

printf("Enter the expression: ");

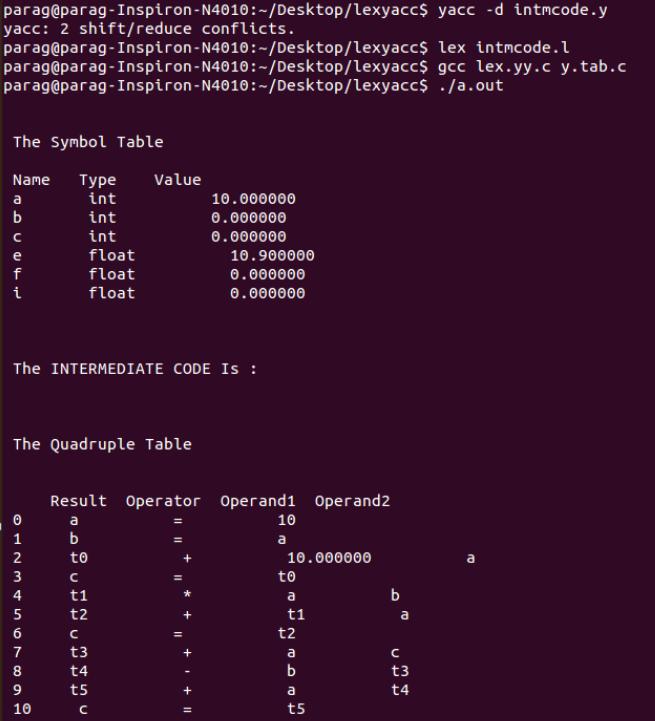
yyparse();

threeAdd();

return 0;

}

**OUTPUT:**



**RESULT:**

Thus the program has been executed successfully and Output is verified.

**Ex.No:8**

**Date:**

**IMPLEMENTATION OF TYPE CHECKING**

**AIM:**

 To write a C program to implement type checking.

**ALGORITHM:**

**Step 1:** Start the program.  
**Step 2:** Track the global scope type information (e.g. classes and their members)  
 **Step 3:** Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.

**Step 4:** If type found correct, do the operation.

**Step 5:** Type mismatches, semantic error will be notified.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

struct symbol{

char name[10],type[10];

}s[10];

void main(){

int i=0,k=0,l=0,m=0,flag=0;

char temp[50],temp1[50],t1[50],b[10],c[10],str[50],buf[50];

char \*a[10]={"int","float","char","double"};

char \*op[10]={"+","-","\*","/","<",">"};

FILE \*fp;

clrscr();

fp=fopen("sample.c","r");

while(!feof(fp))

{

fscanf(fp,"%s",str);

for(i=0;i<4;i++)

{

if(strcmp(str,a[i])==0)

{

strcpy(s[k].type,str);

fscanf(fp,"%s",buf);

strcpy(s[k].name,buf);

k++;

}

}

}

printf("symbol table\n");

for(k=0;k<10;k++)

{

printf("%s\t%s\t\n",s[k].type,s[k].name);

}

fclose(fp);

fp=fopen("sample.c","r");

while(!feof(fp))

{

fscanf(fp,"%s",temp);

strcpy(t1,temp);

fscanf(fp,"%s",temp);

for(l=0;l<10;l++)

{

if(strcmp(temp,op[l])==0)

{

fscanf(fp,"%s",temp1);

for(m=0;m<10;m++)

{

if(strcmp(t1,s[m].name)==0)

strcpy(b,s[m].type);

if(strcmp(temp1,s[m].name)==0)

strcpy(c,s[m].type);

flag=1;

}

}

if(flag==1)

{

if(strcmp(b,c)==0)

printf("valid");

else

printf("invalid");

flag=0;

}

}

}

getch();

}

**INPUT FILE:**

**Sample.c**

main()

{

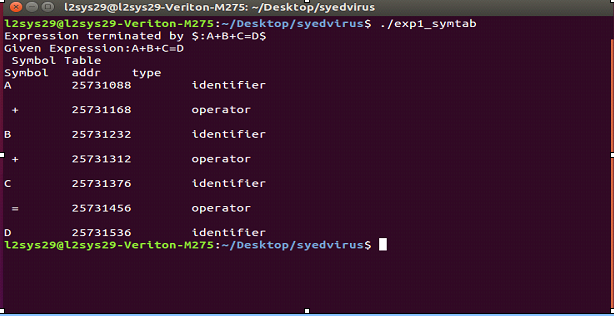
int a;

float b;

if(a>b);

}

**OUTPUT:**



**RESULT:**

Thus the program has been executed successfully and Output is verified.

**Ex.No:9**

**Date:**

**IMPLEMENTATION OF CODE OPTIMIZATION TECHNIQUES**

**AIM:**

To write a program for the implementation of the Code Optimization Technique.

**ALGORITHM:**

**Step 1:** Generate the program for the factorial program using for and do-while loop to specify optimization technique.

**Step 2:** In-for-loop variable initialization is activated first and the condition is checked next. If the condition is true the corresponding statements are executed and a specified increment/decrement operation is performed.

**Step 3:**  The for-loop operation is activated till the condition fails.

**Step 4:**  In the do-while loop the variable is initialized and the statements are executed then the condition checking and increment/decrement operation are performed.

**Step 5:**  When comparing both for and do-while loops for optimization do-while is best because first the statement execution is done then only the condition is checked. So, during the statement execution itself, we can find the inconvenience of the result and no need to wait for the specified condition result.

**Step 6:**  Finally when considering Code Optimization in a loop do-while is best with respect to performance.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

struct op

{

char l;

char r[20];

}

op[10],pr[10];

void main()

{

inta,i,k,j,n,z=0,m,q;

char \*p,\*l;

chartemp,t;

char \*tem;

clrscr();

printf("Enter the number of values:\n");

scanf("%d",&n);

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

{

printf("left:");

op[i].l=getche();

printf("\t right:");

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

}

printf("\n 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("After 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",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';

strcpy(pr[i].r,'\0');

}

}

}

printf("Optimized code\n");

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

{

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

{

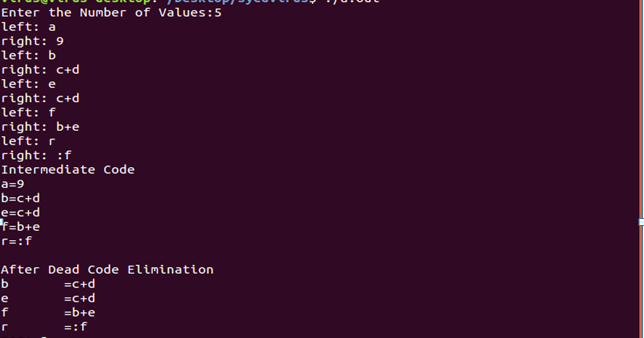
printf("%c=",pr[i].l);

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

}}

getch();}

**OUTPUT:**



**RESULT:**

Thus the program has been executed successfully and Output is verified.

**Ex.No:10**

**Date:**

**IMPLEMENTATION OF THE BACK END OF THE COMPILER**

**AIM:**

To implement the back end of the compiler which takes the three address codes and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be a simple move, add, sub, jump. Also, simple addressing modes are used.

**ALGORITHM:  
 Step 1:** Start the program.  
**Step 2:** Open the source file and store the contents as quadruples.

**Step 3:** Check for operators, in quadruples, if it is an arithmetic operator generator it or if the assignment operator generates it, else perform unary minus on register C.  
**Step 4:** Write the generated code into the output definition of the file

**Step 5:** Print the output

**Step 6:** Stop the program.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

int label[20];

int no=0;

int main()

{

FILE \*fp1,\*fp2;

int check\_label(int n);

char fname[10],op[10],ch;

char operand1[8],operand2[8],result[8];

int i=0,j=0;

clrscr();

printf("enter file name of the intermediate code");

scanf("%s",&fname);

fp1=fopen(fname,"r");

fp2=fopen("target.txt","w");

if(fp1==NULL||fp2==NULL)

{

printf("\nerror opening file");

getch();

exit(0);

}

while(!feof(fp1))

{

fprintf(fp2,"\n");

fscanf(fp1,"%s",op);

i++;

if(check\_label(i))

{

fprintf(fp2,"\nlabel#%d:",i);

}

if(strcmp(op,"print")==0){

fscanf(fp1,"%s",result);

fprintf(fp2,"\n\t OUT %s",result);

}

if(strcmp(op,"goto")==0){

fscanf(fp1,"%s",operand2);

fprintf(fp2,"\n\tJMP label#%s",operand2);

label[no++]=atoi(operand2);

}

if(strcmp(op,"[]=")==0){

fscanf(fp1,"%s%s%s",operand1,operand2,result);

fprintf(fp2,"\n\t STORE %s[%s],%s",operand1,operand2,result);

}

if(strcmp(op,"uminus")==0){

fscanf(fp1,"%s%s",operand1,result);

fprintf(fp2,"\n\t MOV R1,-%s",operand1);

fprintf(fp2,"\n\t MOV %s,R1",result);

}

switch(op[0]){

case '\*':fscanf(fp1,"%s%s%s",operand1,operand2,result);

fprintf(fp2,"\n\t MOV R0,%s",operand1);

fprintf(fp2,"\n\t MOV R1,%s",operand2);

fprintf(fp2,"\n\t MUL R0,R1");

fprintf(fp2,"\n\t MOV %s,R0",result);

break;

case '+':fscanf(fp1,"%s%s%s",operand1,operand2,result);

fprintf(fp2,"\n\t MOV R0,%s",operand1);

fprintf(fp2,"\n\t MOV R1,%s",operand2);

fprintf(fp2,"\n\t ADD R0,R1");

fprintf(fp2,"\n\t MOV %s,R0",result);

break;

case '-':fscanf(fp1,"%s%s%s",operand1,operand2,result);

fprintf(fp2,"\n\t MOV R0,%s",operand1);

fprintf(fp2,"\n\t MOV R1,%s",operand2);

fprintf(fp2,"\n\t SUB R0,R1");

fprintf(fp2,"\n\t MOV %s,R0",result);

break;

case '/':fscanf(fp1,"%s%s%s",operand1,operand2,result);

fprintf(fp2,"\n\t MOV R0,%s",operand1);

fprintf(fp2,"\n\t MOV R1,%s",operand2);

fprintf(fp2,"\n\t DIV R0,R1");

fprintf(fp2,"\n\t MOV %s,R0",result);

break;

case '%':fscanf(fp1,"%s%s%s",operand1,operand2,result);

fprintf(fp2,"\n\t MOV R0,%s",operand1);

fprintf(fp2,"\n\t MOV R1,%s",operand2);

fprintf(fp2,"\n\t DIV R0,R1");

fprintf(fp2,"\n\t MOV %s,R0",result);

break;

case '=':fscanf(fp1,"%s%s",operand1,result);

fprintf(fp2,"\n\t MOV %s,%s",result,operand1);

break;

case '>':j++;

fscanf(fp1,"%s%s%s",operand1,operand2,result);

fprintf(fp2,"\n\t JGT %s,%s label#%s",operand1,operand2,result);

label[no++]=atoi(result);

break;

case '<':fscanf(fp1,"%s%s%s",operand1,operand2,result);

fprintf(fp2,"\n\t JLT %s,%s label#%s",operand1,operand2,result);

label[no++]=atoi(result);

break;

}}

fclose(fp2);

fclose(fp1);

fp2=fopen("target.txt","r");

if(fp2==NULL)

{

printf("\nError Opening in file");

getch();

exit(0);

}

do

{

ch=fgetc(fp2);

printf("%c",ch);

}while(ch!=EOF);

fclose(fp2);

getch();

return 0;

}

intcheck\_label(int k)

{

int i;

for(i=0;i<no;i++)

{

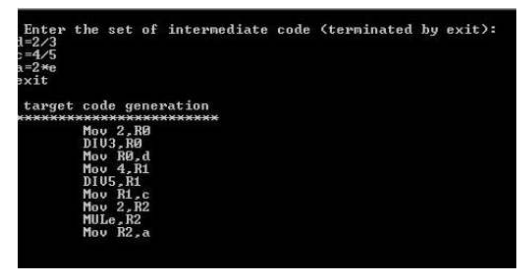
if(k==label[i])

return 1;

}

return 0;}

**OUTPUT:**



**RESULT:**

Thus the program has been executed successfully and Output is verified.