# Ex No: 1

# Date:

# IMPLEMENT CODE TO RECOGNIZE TOKENS IN C

**AIM:**

To implement the program to identify C keywords, identifiers, operators, end statements like [], {} using C tool.

**ALGORITHM:**

● We identify the basic tokens in c such as keywords, numbers, variables, etc.

● Declare the required header files.

● Get the input from the user as a string and it is passed to a function for processing.

● The functions are written separately for each token and the result is returned in the

form of bool either true or false to the main computation function.

● Functions are issymbol() for checking basic symbols such as () etc , isoperator() to

check for operators like +, -, \*, / , isidentifier() to check for variables like a,b,

iskeyword() to check the 32 keywords like while etc., isInteger() to check for numbers

in combinations of 0-9, isnumber() to check for digits and substring().

● Declare a function detecttokens() that is used for string manipulation and iteration

then the result is returned from the functions to the main. If it’s an invalid identifier

error must be printed.

● Declare main function get the input from the user and pass to detecttokens() function.

# PROGRAM:

# #include<stdio.h>

# int main(){

# int count=0,k=0,i=0;

# char a[25];

# printf("Enter expression : ");

# fgets(a,25,stdin);

# while(a[i]!='\0'){

# if(isalpha(a[i])){

# printf("%c - identifier\n",a[i]);

# }

# else if(a[i]=='+' || a[i]=='-'||a[i]=='\*'||a[i]=='/'){

# printf("%c - arithmetic operator\n",a[i]);

# }

# else if(a[i]=='='){

# printf("%c - assignment operator\n",a[i]);

# }

# else if(isdigit(a[i])){

# char b[k];

# while(isdigit(a[i])){

# b[k++]=a[i];

# i++;

# }

# printf("%s - digit\n",b);

# k=0;

# i=i-1;

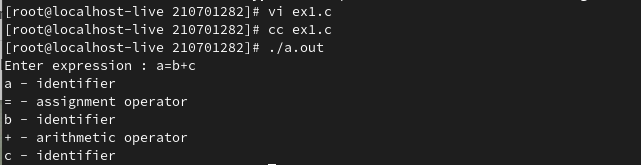
# }

# i++;

# }

# }

**OUTPUT:**



# RESULT:

# Ex No: 2

# Date:

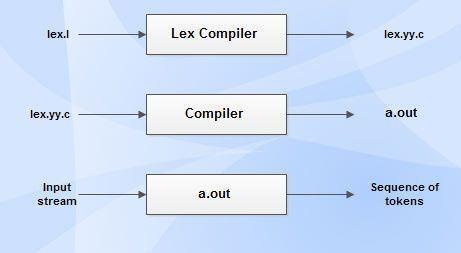
# IMPLEMENT A LEXICAL ANALYZER TO COUNT THE NUMBER OF WORDS USING LEX TOOL

**AIM:**

To implement the program to count the number of words in a string using LEX tool.

**STUDY:**

Lex is a tool in lexical analysis phase to recognize tokens using regular expression. Lex tool itself is a lex compiler.



* lex.l is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
* lex.yy.c is compiled by the C compiler to a file called a.out.
* The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
* yylval is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.
* The attribute value can be numeric code, pointer to symbol table or nothing.
* Another tool for lexical analyzer generation is Flex.

# STRUCTURE OF LEX PROGRAMS:

Lex program will be in following form declarations

%%

translation rules

%%

auxiliary functions

**ALGORITHM:**

* Declare necessary header files and variables in the beginning.
* Define rules in the form of regular expressions to identify words and newline characters.
* Increment a counter each time a word is matched.
* Reset the counter when encountering a newline character and print the count.
* Implement the main function to initiate lexical analysis and return 0.

**PROGRAM:**

%{

#include<stdio.h>

#include<string.h>

int i = 0;

%}

/\* Rules Section\*/

%%

([a-zA-Z0-9])\* {i++;} /\* Rule for counting

number of words\*/

"\n" {printf("%d\n", i); i = 0;}

%%

int yywrap(void){}

int main()

{

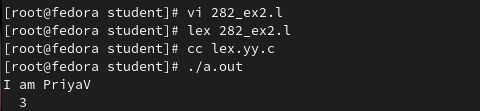
// The function that starts the analysis

yylex();

return 0;

}

**OUTPUT:**



**RESULT:**

# Ex No: 3

# Date:

**DEVELOP A LEXICAL ANALYZER TO RECOGNIZE TOKENS** **USING LEX TOOL**

**AIM:**

To implement the program to identify C keywords, identifiers, operators, end statements like [], {} using LEX tool.

**ALGORITHM:**

* Define patterns for C keywords, identifiers, operators, and end statements using regular expressions. Use %option noyywrap to disable the default behavior of yywrap.
* Utilize regular expressions to match patterns for C keywords, identifiers, operators, and end statements. Associate each pattern with an action to be executed when matched.
* Define actions to print corresponding token categories for matched patterns. Handle special cases like function declarations, numeric literals, and processor directives separately.
* Open the input file (sample.c in this case) for reading. Start lexical analysis using yylex() to scan the input and apply defined rules.
* Increment a counter (n) each time a newline character is encountered. Print the total number of lines at the end of the program execution.

**PROGRAM:**

%option noyywrap

letter [a-zA-Z]

digit [0-9]

id [\_|a-zA-Z]

AO [+|-|/|%|\*]

RO  [<|>|<=|>=|==]

pp [#]

%{

int n=0;

%}

%%

"void" printf("%s return type\n",yytext);

{letter}\*[(][)] printf("%s  Function\n",yytext);

"int"|"float"|"if"|"else" printf("%s keywords\n",yytext);

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

{id}({id}|{digit})\* printf("%s Identifier\n",yytext);

{digit}{digit}\* printf("%d Numbers\n",yytext);

{AO} printf("%s Arithmetic Operators\n",yytext);

{RO} printf("%s Relational Operators\n",yytext);

{pp}{letter}\*[<]{letter}\*[.]{letter}[>] printf("%s processor

Directive\n",yytext);

[\n] n++;

"."|","|"}"|"{"|";" printf("%s others\n",yytext);

%%

int main()

{

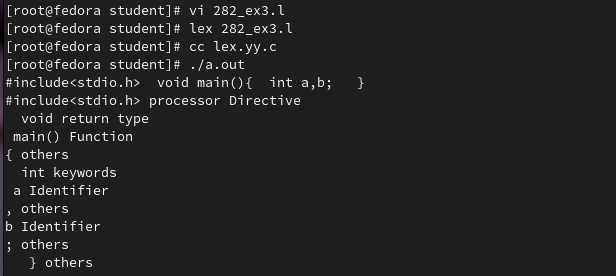
yyin=fopen("sample.c","r");

yylex();

printf("No of Lines %d\n",n);

}

**OUTPUT:**

****

3

3

**RESULT:**

**Ex No: 4 Date:**

# DESIGN A DESK CALCULATOR USING LEX TOOL

# AIM:

To create a calculator that performs addition, subtraction, multiplication and division using lex tool.

# ALGORITHM:

# ● In the headers section declare the variables that is used in the program including

# header files if necessary.

# ● In the definitions section assign symbols to the function/computations we use along

# with REGEX expressions.

# ● In the rules section assign dig() function to the dig variable declared.

# ● In the definition section increment the values accordingly to the arithmetic functions

# respectively.

# ● In the user defined section convert the string into a number using atof() function.

# ● Define switch case for different computations.

# ● Define the main () and yywrap() function.

# PROGRAM:

# %{

# int op = 0,i;

# float a, b;

# %}

# dig [0-9]+|([0-9]\*)"."([0-9]+)

# add "+"

# sub "-"

# mul "\*"

# div "/"

# pow "^"

# ln \n

# %%

# {dig} {digi();}

# {add} {op=1;}

# {sub} {op=2;}

# {mul} {op=3;}

# {div} {op=4;}

# {pow} {op=5;}

# {ln} {printf("\n The Answer :%f\n\n",a);}

# %%

# digi(){

# if(op==0)

# a=atof(yytext);

# else{

# b=atof(yytext);

# switch(op){

# case 1:a=a+b;

# break;

# case 2:a=a-b;

# break;

# case 3:a=a\*b;

# break;

# case 4:a=a/b;

# break;

# case 5:for(i=a;b>1;b--)

# a=a\*i;

# break;

# }

# op=0; } }

# main(int argv,char \*argc[])

# {

# yylex();}

# yywrap()

# {

# return 1;

# }

# OUTPUT:

**RESULT:**

**Ex No: 5 Date:**

**RECOGNIZE AN ARITHMETIC EXPRESSION USING LEX AND YACC**

**AIM:**

To check whether the arithmetic expression using lex and yacc tool.

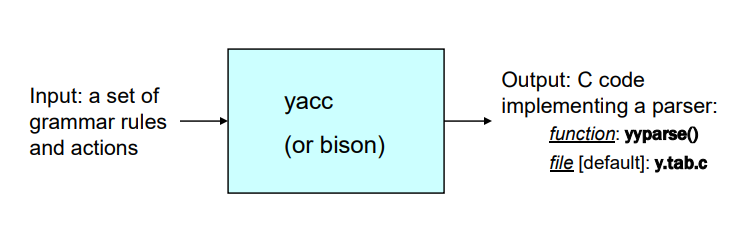
**ALGORITHM:**

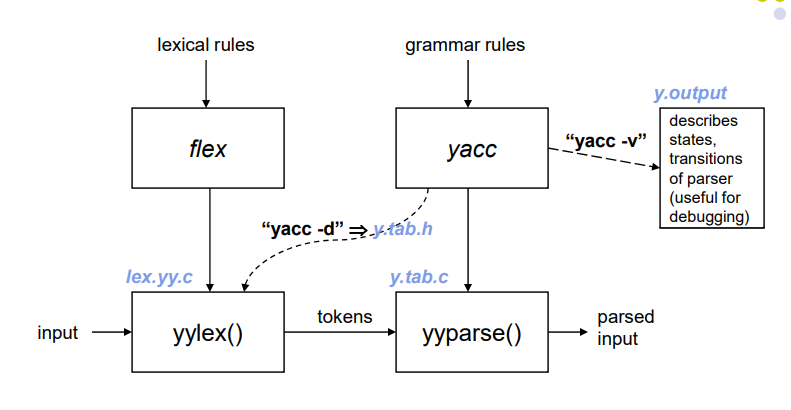
* Using the flex tool, create lex and yacc files.
* In the C include section define the header files required.
* In the rules section define the REGEX expressions along with proper definitions.
* In the user defined section define yywrap() function.
* Declare the yacc file inside it in the C definitions section declare the header files required along with an integer variable valid with value assigned as 1.
* In the Yacc declarations declare the format token num id op.
* In the grammar rules section if the starting string is followed by assigning operator or identifier or number or operator followed by a number or open parenthesis followed by an identifier. The x could be an operator followed by an identifier or operator or no operator then declare that as valid expressions by making the valid stay in 1 itself.
* In the user definition section if the valid is 0 print as Invalid expression in yyerror() and define the main function.

**LEX AND YACC WORKING :**

Parser generator:

* Takes a specification for a context-free grammar.
* Produces code for a parser.





**PROGRAM:**

**validexp.l:**

%{

#include<stdio.h>

#include "y.tab.h"

%}

%%

[a-zA-Z]+ return VARIABLE;

[0-9]+ return NUMBER;

[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

**validexp.y:**

%{

    #include<stdio.h>

%}

%token NUMBER

%token VARIABLE

%left '+' '-'

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

%left '(' ')'

%%

S: VARIABLE'='E {

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

       return 0;

     }

E:E'+'E

 |E'-'E

 |E'\*'E

 |E'/'E

 |E'%'E

 |'('E')'

 | NUMBER

 | VARIABLE

;

%%

void main()

{

   printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");

   yyparse();

}

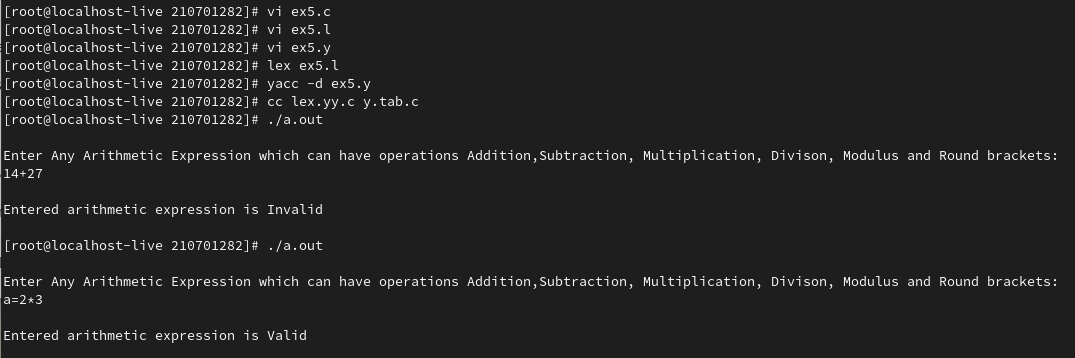
void yyerror()

{

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

}

**OUTPUT:**



**RESULT:**

**Ex No: 6**

**Date:**

**RECOGNIZE A VALID VARIABLE WITH LETTERS AND DIGITS USING LEX AND YACC**

**AIM:**

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

**ALGORITHM:**

* Define lexical rules in variable.l with regex to match valid variables: start with a letter, followed by letters or digits. Tokenize input, distinguishing letters and digits.
* Use lexer (variable.l) to tokenize input into meaningful units like letters and digits.
* Implement grammar rules in parser (variable.y) for recognizing valid variable names using context-free grammar. Incorporate lexer tokens into parsing.
* In parser, implement error handling to detect invalid variable names. Set a flag (e.g., valid) to mark invalid identifiers.
* Check validity post-parsing; if flag remains true, indicate valid identifier. Otherwise, display message for invalid input.

**PROGRAM:**

**variable.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;

}

**variable.y:**

%{

    #include<stdio.h>

    int valid=1;

%}

%token digit letter

%%

start : letter s

s :     letter s

      | digit s |;

%%

int yyerror()

{

    printf("\nIts not a identifier!\n");

    valid=0;

    return 0;

}

int main() {

    printf("\nEnter a name to test for an identifier: ");

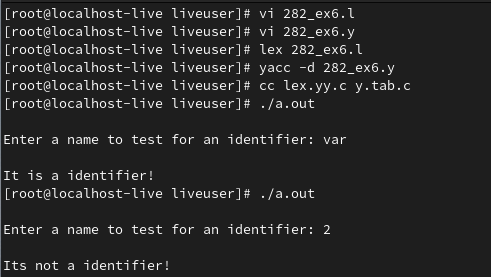
    yyparse();

    if(valid) {

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

    } }

**OUTPUT:**

****

**RESULT:**

**Ex No: 7**

**Date:**

**EVALUATE EXPRESSION THAT TAKES DIGITS, \*, + USING LEX AND YACC**

**AIM:**

To perform arithmetic operations that takes digits,\*, + using lex and yacc.

**ALGORITHM:**

* Define rules in evaluate.l to recognize digits and ignore whitespace, returning tokens for numbers. Utilize yylval to pass token values to parser.
* Break down input into tokens (numbers) in evaluate.l, associating each with its respective value.
* Use parser (evaluate.y) to implement grammar rules for arithmetic expressions, considering precedence and associativity of operators. Generate a result for each expression.
* Implement error handling in evaluate.y to detect invalid expressions. Set a flag if errors occur during parsing.
* After parsing, check if the flag remains unset. If so, indicate that the arithmetic expression is valid; otherwise, display an error message.

**PROGRAM:**

**evaluate.l:**

%{

#include<stdio.h>

#include "y.tab.h"

extern int yylval;

%}

%%

[0-9]+ {

       yylval=atoi(yytext);

       return NUMBER;

    }

[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

**evaluate.y:**

%{

#include<stdio.h>

int flag=0;

%}

%token NUMBER

%left '+' '-'

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

%left '(' ')'

%%

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;}

;

%%

void main()

{

   printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");

   yyparse();

  if(flag==0)

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

}

void yyerror()

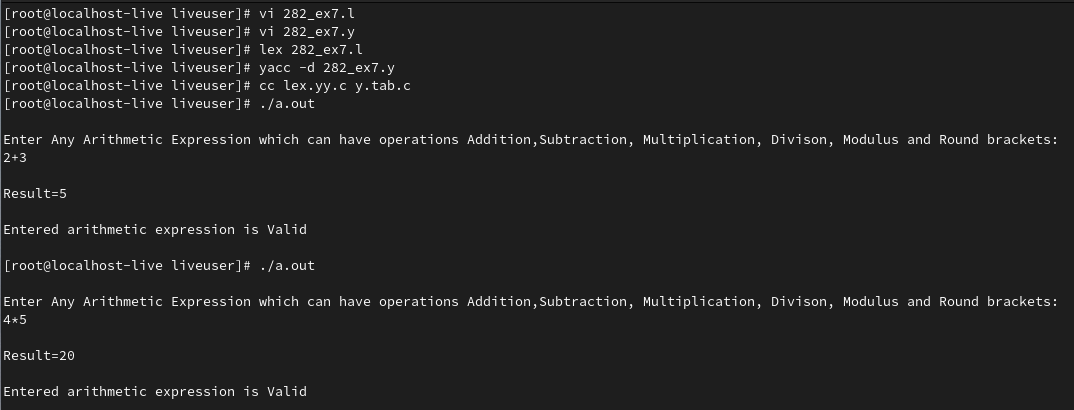
{

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

   flag=1;

}

**OUTPUT:**

****

**RESULT:**

**Ex No: 8**

**Date:**

**GENERATE THREE ADDRESS CODES**

**AIM:**

To generate three address code using C program.

**ALGORITHM:**

* Get address code sequence.
* Determine current location of 3 using address (for 1st operand).
* If the current location does not already exist, generate move (B, O).
* Update address of A (for 2nd operand).
* If the current value of B and () is null, exist.
* If they generate operator () A, 3 ADPR.
* Store the move instruction in memory.

**PROGRAM:**

#include<stdio.h>

#include<string.h>

#include<ctype.h>

typedef struct

{

char var[10]; int alive;

}

regist;

regist preg[10];

void substring(char exp[],int st,int end)

{

int i,j=0;

char dup[10]="";

for(i=st;i<end;i++)

dup[j++]=exp[i];

dup[j]='0';

strcpy(exp,dup);

}

int getregister(char var[])

{

int i; for(i=0;i<10;i++)

{

if(preg[i].alive==0)

{

strcpy(preg[i].var,var);

break;

}

}

return(i);

}

void getvar(char exp[],char v[])

{

int i,j=0;

char var[10]="";

for(i=0;exp[i]!='\0';i++)

if(isalpha(exp[i]))

var[j++]=exp[i];

else

break;

strcpy(v,var);

}

void main()

{

char basic[10][10],var[10][10],fstr[10],op;

int i,j,k,reg,vc,flag=0;

printf("\nEnter the Three Address Code:\n");

for(i=0;;i++)

{

gets(basic[i]);

if(strcmp(basic[i],"exit")==0)

break;

}

printf("\nThe Equivalent Assembly Code is:\n");

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

{

getvar(basic[j],var[vc++]);

strcpy(fstr,var[vc-1]);

substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));

getvar(basic[j],var[vc++]);

reg=getregister(var[vc-1]);

if(preg[reg].alive==0)

{

printf("\nMov R%d,%s",reg,var[vc-1]);

preg[reg].alive=1;

}

op=basic[j][strlen(var[vc-1])];

substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));

getvar(basic[j],var[vc++]);

switch(op)

{

case '+':

printf("\nAdd");

break; case '-':

printf("\nSub");

break;

case '\*':

printf("\nMul");

break;

case '/':

printf("\nDiv");

break;

}

flag=1;

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

{

if(strcmp(preg[k].var,var[vc-1])==0)

{

printf("R%d, R%d",k,reg);

preg[k].alive=0;

flag=0;

break;

}

}

if(flag)

{

printf(" %s,R%d",var[vc-1],reg);

printf("\nMov %s,R%d",fstr,reg);

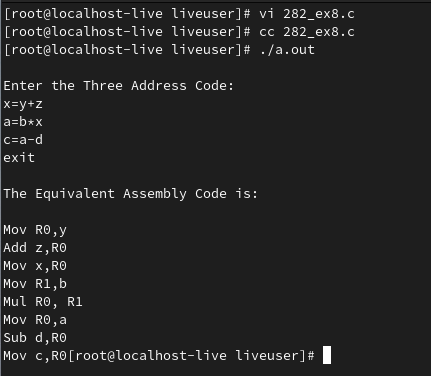
}

strcpy(preg[reg].var,var[vc-3]);

}

}

**OUTPUT:**

****

**RESULT:**

**Ex No: 9**

**Date:**

**IMPLEMENT CODE OPTIMIZATION TECHNIQUES CONSTANT FOLDING**

**AIM:**

To write a C program to implement Constant Folding (Code optimization Technique).

**ALGORITHM:**

* The desired header files are declared.
* The two file pointers are initialized one for reading the C program from the file and one for writing the converted program with constant folding.
* The file is read and checked if there are any digits or operands present.
* If there is, then the evaluations are to be computed in switch case and stored.
* Copy the stored data to another file.
* Print the copied data file.

## PROGRAM:

#include<stdio.h>

#include<string.h>

void main() {

char s[20];

char flag[20]="//Constant";

char result,equal,operator;

double op1,op2,interrslt;

int a,flag2=0;

FILE \*fp1,\*fp2;

fp1 = fopen("input.txt","r");

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

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

while(!feof(fp1)) {

if(strcmp(s,flag)==0) {

flag2 = 1;

}

if(flag2==1) {

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

result=s[0];

equal=s[1];

if(isdigit(s[2])&& isdigit(s[4])) {

if(s[3]=='+'||'-'||'\*'||'/') {

operator=s[3];

switch(operator) {

case '+':

interrslt=(s[2]-48)+(s[4]-48);

break;

case '-':

interrslt=(s[2]-48)-(s[4]-48);

break;

case '\*':

interrslt=(s[2]-48)\*(s[4]-48);

break;

case '/':

interrslt=(s[2]-48)/(s[4]-48);

break;

default:

interrslt = 0;

break; }

fprintf(fp2,"/\*Constant Folding\*/\n");

fprintf(fp2,"%c = %lf\n",result,interrslt);

flag2 = 0;

}

} else {

fprintf(fp2,"Not Optimized\n");

fprintf(fp2,"%s\n",s);

}

} else {

fprintf(fp2,"%s\n",s);

}

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

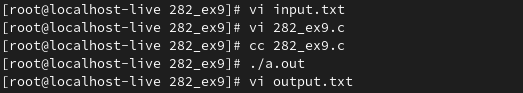
}

fclose(fp1);

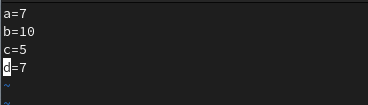
fclose(fp2);

}

**OUTPUT:**

****

**//output.txt**

****

**RESULT:**

**Ex No: 10**

**Date:**

**IMPLEMENT CODE OPTIMIZATION TECHNIQUES**

**DEAD CODE AND COMMON SUB EXPRESSION ELIMINATION**

**AIM:**

To write a C program to implement the dead code elimination and common sub expression elimination (code optimization) techniques.

**ALGORITHM:**

* Start
* Create the input file which contains three address code.
* Open the file in read mode.
* If the file pointer returns NULL, exit the program else go to 5.
* Scan the input symbol from left to right.
* Store the first expression in a string.
* Compare the string with the other expressions in the file.
* If there is a match, remove the expression from the input file.
* Perform these steps 5-8 for all the input symbols in the file.
* Scan the input symbol from the file from left to right.
* Get the operand before the operator from the three address code.
* Check whether the operand is used in any other expression in the three address code.
* If the operand is not used, then eliminate the complete expression from the three-address code else go to 14.
* Perform steps 11 to 13 for all the operands in the three address code till end of the file is reached.
* Stop.

**PROGRAM:**

#include<stdio.h>

#include<conio.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;

  clrscr();

  printf("enter no of values");

  scanf("%d", & n);

  for (i = 0; i < n; i++)

{

    printf("\tleft\t");

    op[i].l = getche();

    printf("\tright:\t");

    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);

  }

  //sub expression elimination

  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);

  }

  // duplicate production elimination

  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");

  for (i = 0; i < z; i++)

{

    if (pr[i].l != '\0') {

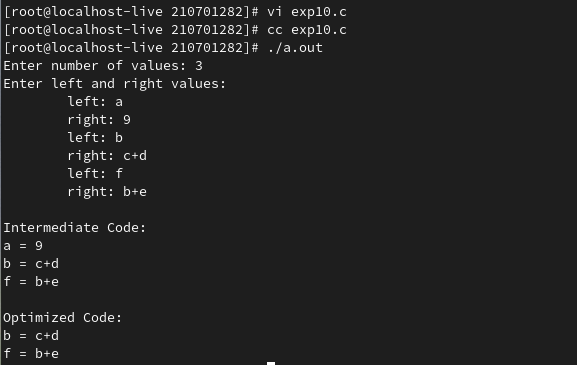
      printf("%c=", pr[i].l);

      printf("%s\n", pr[i].r);

    }    }   getch();

}

**OUTPUT:**



**RESULT:**

**For Index**

# IMPLEMENT CODE TO RECOGNIZE TOKENS IN C

# IMPLEMENT A LEXICAL ANALYZER TO COUNT THE NUMBER OF WORDS USING LEX TOOL

# DEVELOP A LEXICAL ANALYZER TO RECOGNIZE TOKENS USING LEX TOOL

# DESIGN A DESK CALCULATOR USING LEX TOOL

# RECOGNIZE AN ARITHMETIC EXPRESSION USING LEX AND YACC

# RECOGNIZE A VALID VARIABLE WITH LETTERS AND DIGITS USING LEX AND YACC

# EVALUATE EXPRESSION THAT TAKES DIGITS, \*, + USING LEX AND YACC

# GENERATE THREE ADDRESS CODES

# IMPLEMENT CODE OPTIMIZATION TECHNIQUES CONSTANT FOLDING

# IMPLEMENT CODE OPTIMIZATION TECHNIQUES DEAD CODE AND COMMON SUB EXPRESSION ELIMINATION