**37. Write a C Program for code optimization to eliminate common subexpression.**

**AIM:** To write a C program for code optimization to eliminate common subexpression.

**PROGRAM:**

#include <stdio.h>

#include <string.h>

int main() {

// Original code

int a = 5, b = 10, c = 15, d, e;

printf("=== Original Code ===\n");

printf("d = (a + b) \* c;\n");

printf("e = (a + b) \* (c + 2);\n\n");

// Without optimization (recomputes (a+b) twice)

d = (a + b) \* c;

e = (a + b) \* (c + 2);

printf("Without Optimization: d = %d, e = %d\n", d, e);

// With common subexpression elimination

int temp = a + b; // common subexpression

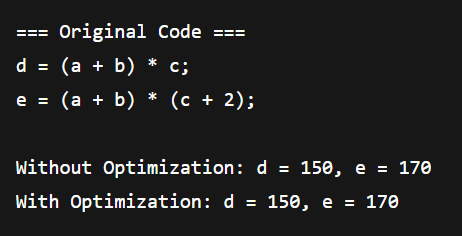
d = temp \* c;

e = temp \* (c + 2);

printf("With Optimization: d = %d, e = %d\n", d, e);

return 0;

}

**OUTPUT:  
**

**38. Write a C program to implement the back end of the compiler.**

**AIM:** To write a C program to implement the back end of compiler.

**PROGRAM:**

#include <stdio.h>

#include <string.h>

int main() {

char dst[20], lhs[20], op[5], rhs[20];

printf("; --- Simple Backend ---\n\n");

while (scanf("%s = %s %s %s", dst, lhs, op, rhs) == 4) {

printf("; %s = %s %s %s\n", dst, lhs, op, rhs);

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

if(strcmp(op,"+")==0) printf("ADD %s\n", rhs);

else if(strcmp(op,"-")==0) printf("SUB %s\n", rhs);

else if(strcmp(op,"\*")==0) printf("MUL %s\n", rhs);

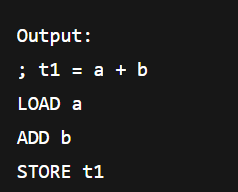
else if(strcmp(op,"/")==0) printf("DIV %s\n", rhs);

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

}

return 0;

}

**OUTPUT:  
**

**39. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Write a LEX specification file to take input C program from a .c file and count the number of characters, number of lines & number of words.  
  Input Source Program: (sample.c)  
 #include <stdio.h>  
 int main()  
 {      
      int number1, number2, sum;  
 printf("Enter two integers: ");  
 scanf("%d %d", &number1, &number2);  
 sum = number1 + number2;        
      printf("%d + %d = %d", number1, number2, sum);  
  return 0;  
 }**

**AIM:** To write a LEX program to count the number of characters, number of lines and number of words.

**PROGRAM:**

%{

#include <stdio.h>

int char\_count = 0, line\_count = 0, word\_count = 0;

%}

ws [ \t\n]+

id [a-zA-Z\_][a-zA-Z0-9\_]\* /\* identifiers \*/

num [0-9]+

str \"([^\\\"]|\\.)\*\"

comment (\/\/.\*)|(\/\\*([^\*]|\\*+[^\*/])\*\\*+\/)

%%

{comment} { /\* ignore comments \*/ }

{ws} {

for(int i=0; yytext[i]; i++)

if(yytext[i]=='\n') line\_count++;

}

{id}|{num}|{str} { word\_count++; char\_count += yyleng; }

. { char\_count++; } /\* any other single char \*/

%%

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

{

if(argc > 1) {

FILE \*fp = fopen(argv[1], "r");

if(!fp) { perror("fopen"); return 1; }

yyin = fp;

}

yylex();

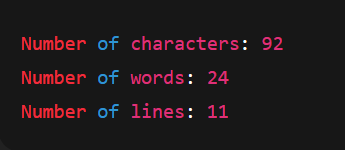
printf("Number of characters: %d\n", char\_count);

printf("Number of words: %d\n", word\_count);

printf("Number of lines: %d\n", line\_count+1); // +1 for last line

return 0;

}

**OUTPUT:  
**

**40. The main function of the Intermediate code generation is producing three address code statements for a given input expression. The three address codes help in determining the sequence in which operations are actioned by the compiler. The key work of Intermediate code generators is to simplify the process of Code Generator. Write a C Program to Generate the Three address code representation for the given input statement.**

**AIM:** To write a C program to generate the three address code representation for the given input statement

**PROGRAM:**

#include <stdio.h>

int t=1;

void tac(char a, char op, char b) {

printf("t%d = %c %c %c\n", t++, a, op, b);

}

int main() {

char expr[50];

printf("Enter expression: ");

scanf("%s", expr);

// handle \* and / first

for(int i=0; expr[i]; i++)

if(expr[i]=='\*'||expr[i]=='/') tac(expr[i-1], expr[i], expr[i+1]), expr[i-1]='t', expr[i]=t-1+'0', expr[i+1]='\0', i=-1;

// then + and -

for(int i=0; expr[i]; i++)

if(expr[i]=='+'||expr[i]=='-') tac(expr[i-1], expr[i], expr[i+1]), expr[i-1]='t', expr[i]=t-1+'0', expr[i+1]='\0', i=-1;

printf("Final result in t%d\n", t-1);

return 0;

}

**OUTPUT:**