PART A

(PART A : TO BE REFFERED BY STUDENTS)

Experiment No. 9 (A)

Aim : To implement any code optimization techniques

Objective: Develop a program to implement following code optimization techniques:

a. Common sub-expression elimination
b. Reduction in strength

Outcome: Students are able to appreciate role of code optimization and implement various techniques.

Theory:

Compilers should produce target code that is as good as a hand written code. This is quiet difficult in reality. The code generated by compiler can be made to run faster or take less space. This improvement is achieved through some program transformations which are called as optimizations.
Compilers that apply code-improving transformations are called optimizing compilers.

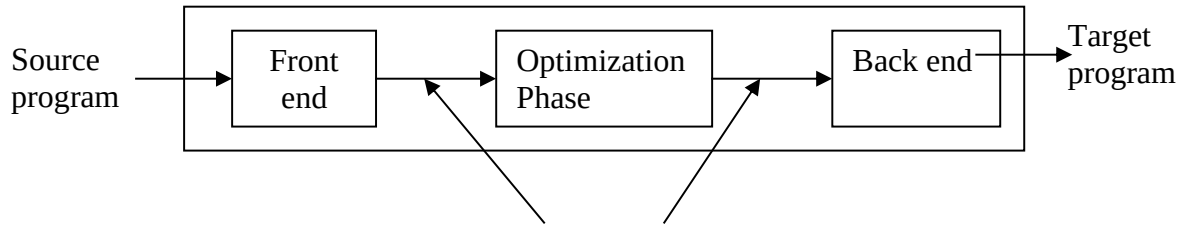


Fig : Schematic of optimizing compiler

A compiler can improve a program without changing the function it computes, using function preserving transformations. The different function-preserving transformations are:

1. Common sub-expressions elimination
2. Copy propagation
3. Dead-code elimination and
4. Constant folding

- **Common sub-expressions**

  If an expression E was previously computed and the values of variables in E have not changed since previous computation, an expression E is called as common sub-expression. We can use previously computed value to avoid re-computation of the expression.
  Consider the following code segment in basic block B1

t6 := 4 * i	**t6 := 4 * i**
x := a[t6]	**x := a[t6]**
t7 := 4 * i	**t8 := 4 * j**
t8 := 4 * j	**t9 :=a[t8]**
t9 :=a[t8]	**a[t6] := t9**
a[t7] := t9	**a[t8] := x**
t10 := 4 * j	**goto B2**
a[t10] := x	
goto B2	

(a) Before (b) **After**

FIGURE Local common sub-expression elimination

- **Reduction in strength**

x := x * 2 à x := x + x

à x := x << 2

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded at the end of the practical)

Roll. No. B48	Name: Aryan Unhale

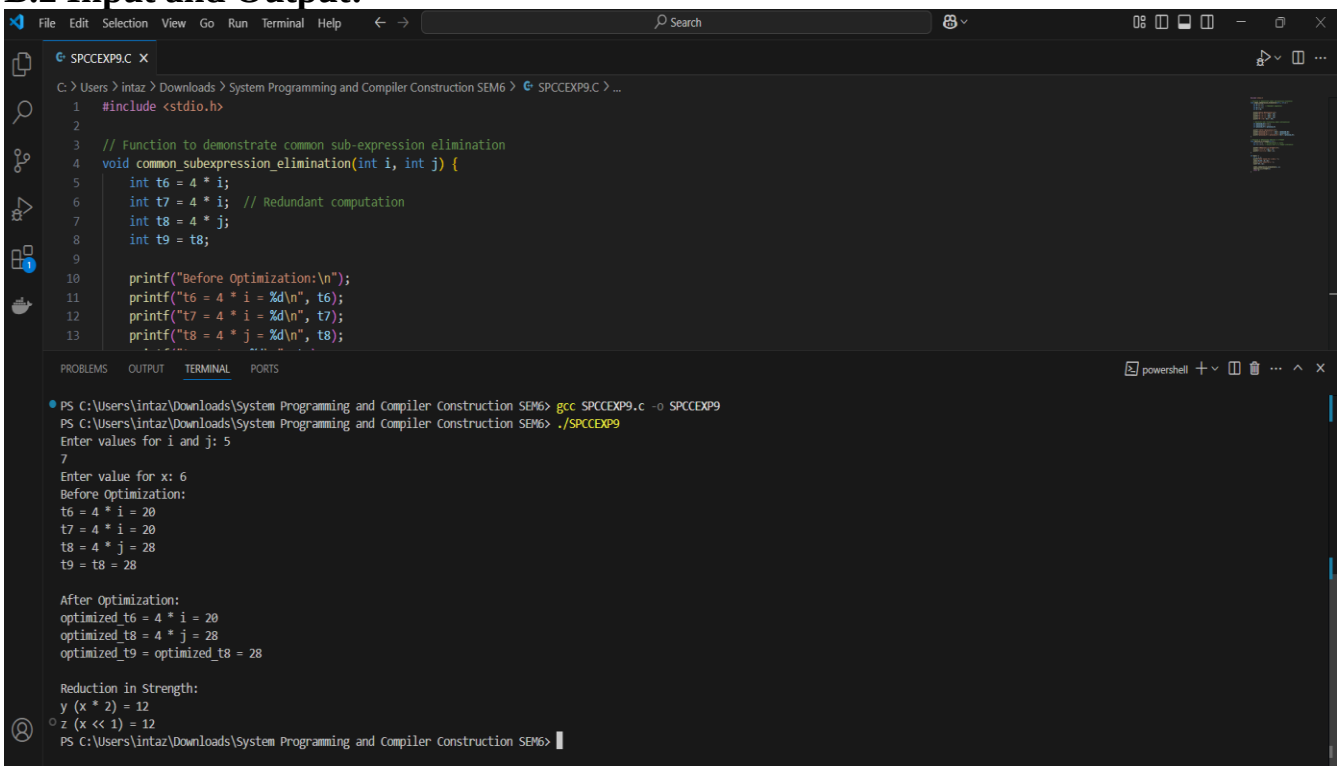
Class: TE-B-COMPS	Batch: B3
Date of Experiment:29/3/25	Date of Submission: 4/4/25
Grade:	

B.1 Software Code written by student:

```
#include <stdio.h>

// Function to demonstrate common sub-expression elimination
void common_subexpression_elimination(int i, int j) {
    int t6 = 4 * i;
    int t7 = 4 * i;  // Redundant computation
    int t8 = 4 * j;
    int t9 = t8;

    printf("Before Optimization:\n");
    printf("t6 = 4 * i = %d\n", t6);
    printf("t7 = 4 * i = %d\n", t7);
    printf("t8 = 4 * j = %d\n", t8);
    printf("t9 = t8 = %d\n", t9);

    // Optimized code: Eliminating common sub-expression
    int optimized_t6 = 4 * i;
    int optimized_t8 = 4 * j;
    int optimized_t9 = optimized_t8;

    printf("\nAfter Optimization:\n");
    printf("optimized_t6 = 4 * i = %d\n", optimized_t6);
    printf("optimized_t8 = 4 * j = %d\n", optimized_t8);
    printf("optimized_t9 = optimized_t8 = %d\n", optimized_t9);
}

// Function to demonstrate reduction in strength
void reduction_in_strength(int x) {
    int y = x * 2;  // Multiplication is costly
    int z = x << 1; // Bitwise shift is a cheaper alternative

    printf("\nReduction in Strength:\n");
    printf("y (x * 2) = %d\n", y);
    printf("z (x << 1) = %d\n", z);
}

int main() {
```

```
    int i, j, x;
    printf("Enter values for i and j: ");
    scanf("%d %d", &i, &j);
    printf("Enter value for x: ");
    scanf("%d", &x);

    common_subexpression_elimination(i, j);
    reduction_in_strength(x);
    return 0;
}
```

B.2 Input and Output:



B.3 Observations and learning:

During this experiment, we implemented code optimization techniques such as common sub-expression elimination and reduction in strength. By applying these optimizations, we observed that redundant computations were eliminated, reducing unnecessary calculations and improving execution efficiency. In the case of common sub-expression elimination, repeated expressions were identified and replaced with precomputed values, reducing redundant operations. Similarly, in reduction in strength, expensive operations like multiplication were replaced with equivalent but computationally cheaper operations, such as addition or bit shifting. These optimizations resulted in reduced execution time and improved resource utilization.

B.4 Conclusion:

The experiment demonstrated the importance of code optimization techniques in improving the efficiency of programs. By applying common sub-expression elimination, we reduced redundant computations, leading to faster execution and lower memory usage. Reduction in strength further optimized the code by replacing computationally expensive operations with simpler alternatives.

B.5 Question of Curiosity

1. Comment on live at that point ?

Live at that point refers to the variables or expressions that are actively used or modified at a specific point within the program's execution. In other words, these are the variables that hold relevant values or contribute to the computation within a particular code segment. Identifying live variables is crucial for optimization techniques such as common sub-expression elimination, as it helps in determining which variables need to be retained or recomputed at various points in the program.

2. What are the the types of block transformation?

There are two types of basic block optimization. These are as follows:

1. Structure preserving transformations:
The primary Structure-Preserving Transformation on basic blocks is as follows:

a. Common subexpression elimination:
In the common sub-expression, you don't need to be computed over and over again. Instead of this you can compute it once and keep it in-store from where it's referenced when encountered again.
a : = b + c
b : = a - d
c : = b + c
d : = a - d
In the above expression, the second and forth expression computed the same expression. So the block can be transformed as follows:
a : = b + c
b : = a - d
5c : = b + c
d : = b

b. Dead-code elimination:
A program may contain a large amount of dead code.
This can be caused when once declared and defined and forget to remove them in this case they serve no purpose.
Suppose the statement x:= y + z appears in a block and x is a dead symbol that means it will never subsequently be used. Then without changing the value of the basic block you can safely remove this statement.

c. Renaming temporary variables:

A statement t:= b + c can be changed to u:= b + c where t is a temporary variable and u is a new temporary variable. All the instances of t can be replaced with u without changing the basic block value.

d. Interchange of the statement:
Suppose a block has the following two adjacent statements:
t1 : = b + c
t2 : = x + y
These two statements can be interchanged without aff ecting the value of the block when the value of t1 does not aff ect the value of t2.

2. Algebraic transformations:
In the algebraic transformation, we can change the set of expressions into an algebraically equivalent set. Thus the expression x:= x + 0 or x:= x *1 can be eliminated from a basic block without changing the set of expressions.
Constant folding is a class of related optimization. Here at compile-time, we evaluate constant expressions and replace the constant expression with their values. Thus the expression 5*2.7 would be replaced by13.5.
Sometimes the unexpected common subexpression is generated by the relational operators like <=, >=, <, >, +, = etc.
6Sometimes the associative expression is applied to expose common subexpression without changing the basic block value. if the source code has the assignments
a:= b + c
e:= c +d +b
The following intermediate code may be generated:
a:= b + c
t:= c +d
e:= t + b

3. State the Normal form of Block.

➔ In compiler construction, a basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. This restricted form makes a basic block highly amenable to analysis. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. Basic blocks form the vertices or nodes in a control flow graph.
➔ The code in a basic block has:
● One entry point, meaning no code within it is the destination of a jump instruction anywhere in the program.
● One exit point, meaning only the last instruction can cause the program to begin executing code in a diff erent basic block.