**MODULE BANK-2**

**Subject: CD Section: 15**

**PART-A**

**1.** a) Analyze the process of generating **Three-Address Code (TAC)** for the expression:

x = (a + b) \* (c - d) + (e \* f - g);

b) Compare TAC and syntax tree representation for the same expression.  
 c) Discuss applications of TAC in compiler optimization with reference to **nested arithmetic   
 expressions**.

**2.** a) For the expression ((a + b) \* (c + d)) - (e / f), draw the **syntax tree** and analyze   
 its structure.  
 b) Construct a DAG for the same expression and identify repeated subexpressions.  
 c) Compare syntax tree and DAG for handling **common subexpression elimination**.

**3.** a) Using the code snippet below, show how **Directed Acyclic Graph (DAG)** eliminates   
 redundancy:

t1 = (a + b) \* c;

t2 = (a + b) \* d;

t3 = (a + b) \* c;

b) Analyze how DAG reduces **storage usage** compared to syntax tree.  
 c) Discuss DAG’s application in **compiler back-end optimization** for large expressions.

**4.** a) Explain **storage organization** in runtime environment with this program:

int a = 5;

float b[20];

int factorial(int n) {

return (n <= 1) ? 1 : n \* factorial(n-1);

}

b) Analyze stack and heap allocation for arrays and recursive calls in this example.  
 c) Discuss how improper storage management may lead to **stack overflow or memory leaks**.

**5.** a) Draw the **activation tree** for the recursive Fibonacci program:

int fib(int n) {

if (n <= 1) return n;

else return fib(n-1) + fib(n-2);

}

b) Construct the **activation record** for fib(4).  
 c) Analyze how stack allocation supports **multiple recursive calls** in Fibonacci.

**6.** a) Identify the **sources of optimization** in the following loop:

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

sum = sum + (i \* 2);

prod = prod \* (8 \* 4);

}

b) Compare **algebraic simplification** and **strength reduction** using this loop.  
c) Discuss elimination of **loop-invariant code** in the above program.

**7.** a) Construct **basic blocks** for the code:

t1 = a + b;

if (t1 > c) {

if (d > e) t2 = d - e;

else t2 = d + e;

} else {

t2 = t1 \* c;

}

x = t2 + f;

b) Draw the corresponding **flow graph**.  
 c) Analyze how **control flow complexity** affects optimization.

**8.** a) Analyze **local optimization** and **global optimization** in the following program:

x = (a + b) \* 2;

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

if (y > 100)

z = (a + b) \* 2 - d;

else

z = y + d;

b) Show how **global optimization** reduces redundancy across basic blocks.  
 c) Discuss applications of global optimization in **compiler loop optimizations**.

**9.** a) Compare **loop unrolling**, **loop fusion**, and **loop invariant code motion** for:

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

for (int j = 0; j < m; j++)

C[i][j] = A[i][j] + B[i][j];

b) Analyze how **loop fusion** improves cache performance in matrix multiplication:

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

for (int j = 0; j < m; j++)

X[i][j] = A[i][j] + 1;

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

for (int j = 0; j < m; j++)

Y[i][j] = X[i][j] \* 2;

c) Explain applications of loop optimization in **scientific computing (FFT, matrix factorization)**.

**10.** a) Analyze **instruction selection** and **register allocation** for:

t1 = (a + b) \* (c - d);

t2 = t1 + (e \* f);

t3 = t2 / g;

b) Compare register allocation with register assignment in code-generation algorithms.  
 c) Analyze issues in code generation such as instruction selection, register allocation, and   
 memory management.