Question:
Explain various storage allocation strategies with its merits and demerits. [7M]

Answer:

Storage allocation refers to how memory is assigned to programs and data during execution. There are three main storage allocation strategies:

1. Static Allocation

Definition:
Memory is allocated at compile time before execution begins.

Example:

```
int a = 10;
```

Merits:

- Easy to implement.
- Efficient access to memory using fixed addresses.
- No runtime overhead for allocation.

Demerits:

- Wastes memory if the allocated memory is not used.
- Cannot handle dynamic data structures like linked lists.
- Size must be known at compile time.

2. Stack Allocation

Definition:
Memory is allocated and deallocated in a Last In First Out (LIFO) manner during function calls.

Example:
Local variables inside a function:

```
void func() {
   int x = 5;
}
```

Merits:

- Fast allocation and deallocation.
- Suitable for recursive function calls.
- No fragmentation.

Demerits:

- Limited memory size (stack size is fixed).
- Data is lost when function ends.
- Cannot allocate variable size data at runtime.

3. Heap Allocation (Dynamic Allocation)

Definition:
Memory is allocated at runtime using functions like `malloc()`, `calloc()` in C.

Example:

```
int *p = (int*)malloc(sizeof(int));
```

Merits:

- Flexible, allows variable size memory allocation.
- Can manage complex data structures like trees, graphs, etc.
- Memory can be freed when not required.

Demerits:

- Slower access compared to stack.
- Memory leak issues if not properly deallocated.
- Overhead of managing heap memory.

b) Define activation records. Explain how it is related with runtime storage allocation. (7M)

Answer:

Activation Record:

An **activation record** is a block of memory that stores information about a single execution of a procedure (function or method). It is created when a function is called and removed when the function exits.

It is also called a **stack frame**, and it is stored in the **runtime stack**.

Contents of an Activation Record:

1. **Return Address** – Where to return after function execution.
2. **Parameters** – Values passed to the function.
3. **Local Variables** – Variables declared inside the function.

4. **Temporary Values** – Intermediate values used during evaluation.
5. **Saved Registers** – If any registers are used, their values are saved.

Relation with Runtime Storage Allocation:

- During program execution, memory is divided into:
  - **Code Area** – For compiled code.
  - **Static Area** – For global/static variables.
  - **Heap** – For dynamically allocated memory.
  - **Stack** – For activation records.
- **Each function call** creates a **new activation record** on the **runtime stack**.
- When the function returns, its activation record is **popped off** the stack.

Example:

```
void add(int x, int y) {
    int sum = x + y;
    printf("%d", sum);
}
```

When `add(2, 3)` is called:

- An **activation record** is created with:
  - Parameters: `x=2`, `y=3`
  - Local variable: `sum`
  - Return address

This activation record is pushed to the **runtime stack**.

Q. What is runtime stack? Explain the storage allocation strategies used for recursive procedure calls. [7M]

Answer:

1. Runtime Stack:

- A **runtime stack** (also called **call stack**) is a memory area used to manage **function/procedure calls** during program execution.
- It stores **activation records (stack frames)** of functions, including **local variables, return address, parameters,** and **saved registers**.
- When a function is called, its activation record is **pushed** onto the stack. When the function returns, the record is **popped**.
- It supports **Last-In First-Out (LIFO)** behavior, making it suitable for **nested and recursive calls**.

2. Storage Allocation Strategies for Recursive Calls:

Recursive procedures call themselves. So, **each recursive call needs a separate memory** to store its data (like local variables and return address).
The main strategy used is:

a) Stack-based Allocation:

- Each time a recursive procedure is called, a new **activation record** is pushed onto the runtime stack.
- This ensures that **each call has its own separate copy** of local variables and return address.
- When the call finishes, its activation record is **removed** (popped) from the stack.

 Advantages:

- Supports **dynamic allocation** during runtime.
- Allows **recursive and nested procedure calls** efficiently.

Example:

```
int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n - 1);
}
```

Calling `factorial(3)` will create a stack like this:

```
Top of Stack
--------------
factorial(1)
factorial(2)
factorial(3)
--------------
Bottom of Stack
```

Each call to `factorial(n)` has its **own activation record**, so recursive calls don't overwrite each other's data.

Q. What is a flow graph? Explain how a flow graph can be constructed for a given program.
Also draw the flow graph for the below program:

```
Main()
{
  int sum, n, i;
  sum = 0;
  for i := 1 to n do
    sum := sum + i;
  write(sum);
```

```
}
```

Answer:

Flow Graph:

A **flow graph** is a diagram that represents the **control flow** of a program using **nodes and edges**.

- **Nodes** represent **statements or blocks** of code.
- **Edges (arrows)** represent the **flow of control** (how execution moves).

Flow graphs help in:

- **Understanding program flow**
- **Testing (like path testing or loop testing)**
- **Debugging**

Steps to Construct Flow Graph:

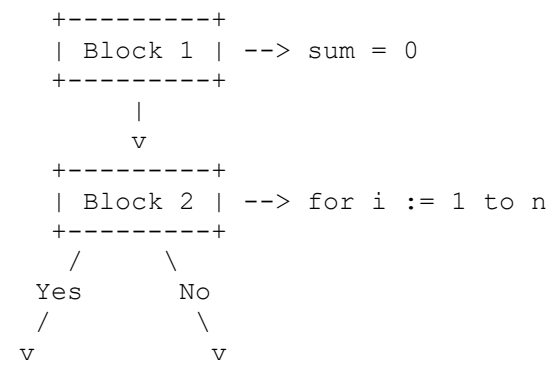
1. Divide the program into **basic blocks** (each block has no jumps except at the end).
2. Number each block.
3. Draw arrows showing the **flow** from one block to another.

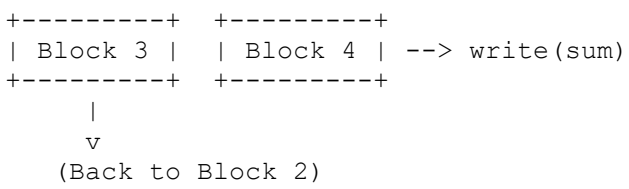
Program Basic Blocks:

Let's break the program into blocks:

- **Block 1**: `sum = 0;`
- **Block 2**: `i := 1 to n` (loop condition)
- **Block 3**: `sum = sum + i;`
- **Block 4**: `write(sum);`

Flow Graph (ASCII Diagram):





Explanation:

- Block 1 initializes `sum`.
- Block 2 checks the loop condition (`i := 1 to n`).
- If true, goes to Block 3 (adds `i` to `sum`) then loops back.
- If false, jumps to Block 4 to print the result.

Q) What are the principles associated with designing calling sequences and the layout of activation records? [7M]

Answer:

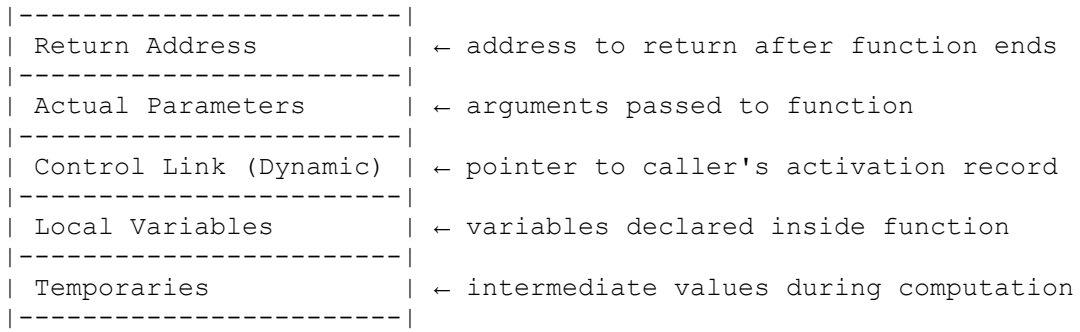
Calling sequence and **activation record layout** are important concepts in the implementation of procedures (functions). Their design affects the efficiency and correctness of function calls and returns.

Principles for Designing Calling Sequences and Activation Records:

1. **Simplicity:**
   - The design should be simple and easy to implement.
   - Should support both recursive and non-recursive procedures.
2. **Efficiency:**
   - Minimize the overhead of procedure calls.
   - Optimize the usage of registers and memory.
3. **Modularity:**
   - Each function should have its own isolated environment.
   - Avoid interference between functions.
4. **Support for Recursion:**
   - Activation records should be created dynamically during recursive calls.
5. **Clear Division of Responsibilities:**
   - Caller and callee should have clearly defined roles in saving and restoring values.
6. **Preservation of Registers:**
   - Decide which registers are saved by the caller and which by the callee (caller-saves vs callee-saves).
7. **Proper Access to Variables:**
   - Ensure access to local, global, and non-local variables.

Activation Record Layout (Example):

An **Activation Record** typically contains the following fields:



Example (Calling Sequence):

Suppose `main()` calls `sum(a, b)`:

1. **Caller (`main`) actions:**
   - Push arguments `a`, `b`.
   - Save return address.
   - Transfer control to `sum`.
2. **Callee (`sum`) actions:**
   - Save control link (dynamic link).
   - Allocate space for local variables.
3. **On Return:**
   - `sum` removes its activation record.
   - Control returns to `main`.

Question:
What is the role of Code Optimizer in a compiler? Is it a mandatory phase? Explain the various sources of optimization with example. [7M]

Answer:

Role of Code Optimizer in Compiler:

- The **Code Optimizer** is a phase in the compiler that improves the intermediate code to make the final program **run faster** and **consume less memory**.
- It **removes unnecessary code**, reduces **redundant operations**, and improves **execution speed**.
- Optimization is done **without changing the output** of the program.

Is Code Optimization a Mandatory Phase?

- **No**, it is **not mandatory**.
- It is an **optional phase**, but very useful for generating **efficient code**.

Sources of Optimization:

1. **Compile-Time Evaluation:**
   - Perform operations during compilation.
   - Example:
   - `int x = 3 * 4; // Compiler changes it to int x = 12;`
2. **Constant Folding:**
   - Combine constant values at compile time.
   - Example:
   - `int y = (2 + 5) * 3; // Optimized to int y = 21;`
3. **Dead Code Elimination:**
   - Removes code that **never affects** the output.
   - Example:
   - `if (false) {`
   - `    x = 10;`
   - `}`
   - `// The above block is removed.`
4. **Common Subexpression Elimination:**
   - Avoids **repeating the same expression**.
   - Example:
   - `a = (b + c) + d;`
   - `e = (b + c) * f;`
   - `// (b + c) computed once and reused`
5. **Loop Optimization:**
   - Improves the **performance of loops**.
   - Example:
     Move invariant code outside the loop.
   - `for (int i = 0; i < 100; i++) {`
   - `    x = y * z;  // If y and z are constant, move outside loop`
   - `}`
6. **Strength Reduction:**
   - Replaces **costly operations** with **cheaper ones**.
   - Example:
   - `x = y * 2;  // Optimized to x = y + y;\`

a) Explain how data flow equations are set up and solved for improving code. [7M]

Answer:

Data flow analysis is used in compilers to collect information about how data values are used and modified in a program. This helps in **code optimization** like dead code elimination, constant propagation, etc.

Steps to set up and solve data flow equations:

1. **Identify basic blocks** in the code.
2. **Construct Control Flow Graph (CFG)**.
3. For each block, define:

    - **GEN[B]**: Variables generated (defined) in B before any use.
    - **KILL[B]**: Variables whose previous definitions are killed in B.
4. Set up **IN[B] and OUT[B] equations**:
    - **IN[B] = ∪ OUT[P]** for all predecessors P of B
    - **OUT[B] = GEN[B] ∪ (IN[B] - KILL[B])**
5. Use **iteration** until IN and OUT values stop changing (reaches a fixed point).

Example:

```
1: a = 5;
2: b = 10;
3: c = a + b;
4: a = 7;
5: d = a + b;
```

Break into basic blocks:

- **B1**: 1, 2
- **B2**: 3
- **B3**: 4, 5

Now:

- **GEN[B1] = {a=5, b=10}**
- **KILL[B3] = {a=5}** (because a is redefined)
- **GEN[B3] = {a=7}**

Apply equations to compute IN and OUT for optimization like **constant propagation**.

b) Discuss basic blocks and flow graphs with an example. [7M]

Answer:

A **Basic Block** is a sequence of instructions with:

- No **branching** in except at the beginning.
- No **branching** out except at the end.

Basic blocks help in identifying areas for optimization.

Flow Graph (Control Flow Graph):

A **Flow Graph** shows flow of control between basic blocks using **nodes** and **edges**.

- **Nodes** represent basic blocks.
- **Edges** show possible flow of control from one block to another.

Steps to identify basic blocks:

1. Start with the **first instruction** as a leader.
2. Any **target of a jump** is a leader.
3. Any instruction **following a jump** is a leader.
4. Each leader starts a new basic block.

Example:

```
1: a = 1;
2: b = 2;
3: if (a < b) goto L1;
4: c = a + b;
5: goto L2;
L1:
6: c = a - b;
L2:
7: print(c);
```

Basic Blocks:

- **B1**: 1, 2, 3
- **B2**: 4, 5
- **B3**: 6
- **B4**: 7

Flow Graph:

- **B1 → B2** (if condition false)
- **B1 → B3** (if condition true)
- **B2 → B4**
- **B3 → B4**

The flow graph helps in optimization like **dead code removal** or **loop analysis**.

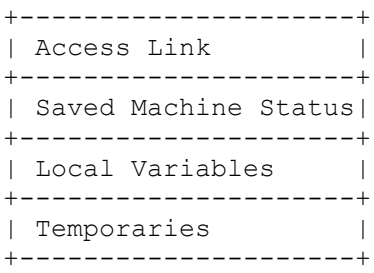
a) Give the general structure of an activation record? Explain the purpose of each component involved in it. [7M]

Answer:

An **activation record** is a block of memory used by the compiler to store information about a function (procedure) call.

General Structure of Activation Record:

```
+---------------------+
| Return Value        |
+---------------------+
| Actual Parameters   |
+---------------------+
| Control Link        |
```



Purpose of Each Component:

1. **Return Value:**
   Stores the result to be returned to the caller.
2. **Actual Parameters:**
   Stores the arguments passed to the function.
3. **Control Link:**
   Points to the activation record of the caller function (helps in returning).
4. **Access Link:**
   Points to non-local variables in the static nesting (used in nested functions).
5. **Saved Machine Status:**
   Stores registers and program counter before the function call.
6. **Local Variables:**
   Stores variables declared inside the function.
7. **Temporaries:**
   Used to hold intermediate results during expression evaluation.

Example:

```
int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

For the above function, activation record will contain:

- `a`, `b` → actual parameters
- `sum` → local variable
- return value → returned sum
- control & access links → for function call management

b) Explain various machine independent code optimization techniques. [7M]

Answer:

Machine Independent Code Optimization means optimizing code without depending on the hardware. It helps improve performance and reduce code size.

Techniques:

1. **Constant Folding:**
   Compute constant expressions at compile time.
   Example:
   `int x = 3 * 4;` ⟶ replaced with `int x = 12;`
2. **Constant Propagation:**
   Replace variables with known constant values.
   Example:
   `int a = 5; int b = a + 2;` ⟶ `int b = 7;`
3. **Dead Code Elimination:**
   Remove code that never affects output.
   Example:
   `if(false) { x = 10; }` ⟶ this block is removed.
4. **Common Subexpression Elimination:**
   Avoid re-evaluation of same expressions.
   Example:
   `a = b + c; d = b + c;` ⟶ compute `b + c` once and reuse it.
5. **Strength Reduction:**
   Replace costly operations with cheaper ones.
   Example:
   `x = x * 2;` ⟶ `x = x + x;`
6. **Copy Propagation:**
   Replace duplicate variable usage.
   Example:
   `a = b; c = a + 1;` ⟶ `c = b + 1;`
7. **Loop Invariant Code Motion:**
   Move code out of loop if it doesn't change.
   Example:
   `for(int i=0; i<n; i++) { x = 5; y[i] = x + i; }` ⟶ move `x = 5` outside loop.

Question:

a) Write a short note on peephole optimization and various operations used in it. [7M]

Answer:

Peephole optimization is a technique used in compiler optimization to improve the performance of a program by making small, localized changes to its intermediate code or machine code. It focuses on optimizing a small window of instructions (a "peephole") by replacing inefficient sequences with more efficient ones.

Key Operations in Peephole Optimization:

1. **Redundant Instruction Removal:**

    - Removes unnecessary instructions that do not contribute to the program's outcome.
    - Example: `MOV R1, R2` followed by `MOV R2, R3` can be replaced with `MOV R1, R3`.
2. **Constant Folding:**
    - Replaces expressions involving constants with their computed values during compilation.
    - Example: `3 + 5` is replaced with `8`.
3. **Common Subexpression Elimination:**
    - Identifies and eliminates repeated expressions by computing the result once and reusing it.
    - Example: If `A + B` appears multiple times, it can be computed once and stored in a register for reuse.
4. **Instruction Combination:**
    - Combines multiple instructions into a single, more efficient one.
    - Example: Two consecutive additions can be combined into one operation using the same operands.
5. **Strength Reduction:**
    - Replaces costly operations like multiplication and division with cheaper operations such as addition or bit shifts.
    - Example: `X * 2` can be replaced by `X << 1`.
6. **Jump Optimization:**
    - Reduces or eliminates unnecessary jumps or branch instructions in the code.
    - Example: A jump to the next instruction can be removed if the flow of control is straightforward.

Example of Peephole Optimization:

Consider the following sequence of instructions:

```
MOV R1, 0
ADD R1, R2
MOV R3, R1
ADD R3, 4
```

After applying peephole optimization, the sequence could be optimized as:

```
MOV R3, R2
ADD R3, 4
```

This optimization removes the redundant `MOV R1, 0` and the unnecessary `MOV R3, R1`.

Peephole optimization is a simple yet effective way to improve the efficiency of code by making small, localized changes that can lead to significant performance improvements.

Question:
Describe loop unrolling and its advantages with your own examples. [7M]

Answer:
Loop unrolling is an optimization technique used to improve the performance of loops in a program by reducing the overhead associated with loop control. This technique involves expanding the loop body so that multiple iterations are performed within a single loop iteration. By doing so, the number of loop control instructions (like incrementing the loop counter and checking the loop condition) is reduced, leading to improved execution speed.

Advantages of Loop Unrolling:

1. **Reduced Loop Overhead:**
   In a typical loop, there are instructions for updating the loop counter and checking the loop condition in every iteration. Loop unrolling minimizes these checks by executing multiple iterations within a single loop iteration.
2. **Improved CPU Pipelining:**
   Modern CPUs can execute multiple instructions simultaneously using pipelining. Unrolling a loop can help in better instruction-level parallelism, as it reduces the frequency of branch instructions, which can hinder the pipeline.
3. **Better Cache Utilization:**
   By accessing data more frequently in a continuous block, loop unrolling improves data locality, which helps in better utilization of CPU caches.

Example of Loop Unrolling:

Consider the following simple loop that adds two arrays element by element:

```
for (int i = 0; i < n; i++) {
    C[i] = A[i] + B[i];
}
```

This loop can be unrolled by processing two elements per iteration:

```
for (int i = 0; i < n; i += 2) {
    C[i] = A[i] + B[i];
    C[i+1] = A[i+1] + B[i+1];
}
```

Explanation:

- In the unrolled loop, two additions are performed in each iteration instead of just one.
- The loop index `i` is incremented by 2, meaning the loop runs fewer times, reducing the loop control overhead.
- This optimization can make the loop execute faster on modern processors by exploiting instruction-level parallelism and better cache behavior.

Question:
Explain static and stack storage allocations with examples. [7M]

Answer:

Static Storage Allocation:
In static storage allocation, the memory for variables is allocated at compile-time, and the memory remains fixed throughout the program's execution. The size and address of the variable are determined before the program runs, and it does not change while the program is running.

- **Example:**
  In a C program, global variables and constants are examples of static storage allocation. For instance, if we declare a global variable like:
- `int x = 10;  // Static allocation`

  The variable `x` will be allocated a fixed memory location before the program starts and will retain its value throughout the program's lifetime.

Stack Storage Allocation:
In stack storage allocation, memory is dynamically allocated at runtime for local variables within functions. When a function is called, its local variables are pushed onto the stack, and when the function ends, the memory is automatically freed. This memory allocation is temporary and managed using a stack structure.

- **Example:**
  In a C program, local variables within a function are allocated using the stack. For example:
- `void function() {`
- `    int a = 5;  // Stack allocation`
- `    int b = 10; // Stack allocation`
- `}`

  Here, the variables `a` and `b` are stored in the stack memory, and their memory is released when the function finishes execution.

Key Differences:

Feature	Static Allocation	Stack Allocation
Memory Allocation	At compile time	At runtime (during function call)
Lifetime of Variables	Throughout the program	Limited to the function's scope
Memory Release	Manually, when program ends	Automatically when function returns
Examples	Global variables, constants	Local variables in functions

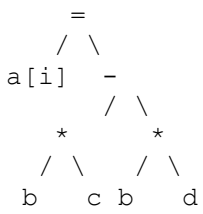
Question:

Translate the arithmetic expression `a[i] = b * c - b * d` into a syntax tree, quadruples, and triples.

Answer:

Syntax Tree:

The syntax tree represents the expression's hierarchical structure. Here's how we can break it down:



Explanation:

- The root of the tree is the assignment operator `=`.
- Left side of `=` is `a[i]`.
- Right side of `=` is the subtraction (`-`) operator, which takes the results of two multiplications (`*`).

Quadruples:

In quadruples, we generate a list of four components: (operator, operand1, operand2, result). Here's how we translate the expression:

1. (`*`, `b`, `c`, `t1`)
   (b * c) is stored in temporary variable `t1`.
2. (`*`, `b`, `d`, `t2`)
   (b * d) is stored in temporary variable `t2`.
3. (`-`, `t1`, `t2`, `t3`)
   Subtract `t1` from `t2` and store the result in `t3`.
4. (`=`, `t3`, , `a[i]`)
   The result `t3` is assigned to `a[i]`.

Triples:

In triples, we only store the operator and its operands, and use indices for results. Here's the translation:

1. (`*`, `b`, `c`) → index 1
2. (`*`, `b`, `d`) → index 2
3. (-, 1, 2) → index 3
4. (`=`, 3, `a[i]`)

Explanation:

- In triples, the result of each operation is identified by an index, and the operands are referenced by their corresponding indices.

Summary:

- **Syntax Tree** represents the hierarchical structure of the expression.
- **Quadruples** use four fields (operator, operand1, operand2, result).
- **Triples** use only three fields (operator, operand1, operand2), referencing previous results by indices.

Question:

Write the pseudocode for finding the sum of 'n' numbers. Identify the basic blocks and construct the flow graph for it. Explain the rules used for this. (7M)

Answer:

Pseudocode to Find the Sum of 'n' Numbers:

```
START
    Set sum = 0
    Read n
    FOR i = 1 TO n
        Read number
        sum = sum + number
    END FOR
    Print sum
END
```

Basic Blocks:

1. **Block 1: Initialization** – The sum is initialized to 0 and the value of `n` is read.
    - Statement: `sum = 0` and `Read n`
2. **Block 2: Loop Setup** – The loop is set up to iterate from 1 to `n`.
    - Statement: `FOR i = 1 TO n`
3. **Block 3: Input and Sum Calculation** – Inside the loop, the number is read, and the sum is updated.
    - Statement: `Read number` and `sum = sum + number`
4. **Block 4: Output** – The sum is printed after the loop ends.
    - Statement: `Print sum`

Flow Graph Construction:

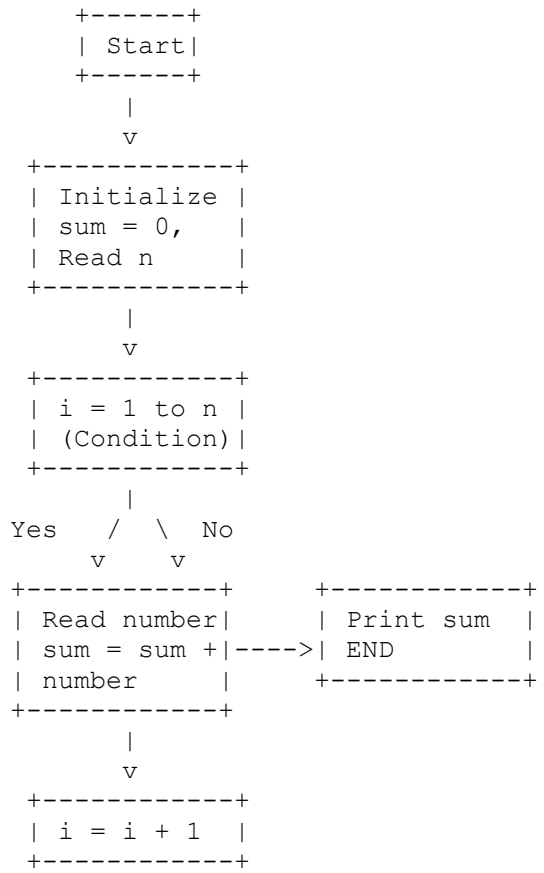
The flow graph is a representation of the flow of control in the program. Each basic block is represented as a node, and edges represent the flow between them.

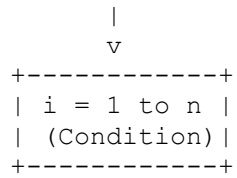
1. **Node 1:** Start of the program. Initializes `sum` to 0 and reads `n`.
2. **Node 2:** The loop condition `FOR i = 1 TO n` is checked.
    - If true, it moves to Node 3.
    - If false, it moves to Node 4 to print the sum and terminate the program.
3. **Node 3:** The loop body where a number is read, and the sum is updated.
4. **Node 2:** After the loop body, control returns to check the loop condition again.
5. **Node 4:** Print the final sum and exit.

Rules for Flow Graph Construction:

1. **Basic Block Representation:** Each block represents a sequence of statements without any branches, i.e., a block that executes sequentially without any condition or decision.
2. **Control Flow Edges:** An edge is drawn from one basic block to another to indicate the flow of control. This could be based on decisions (such as loops or conditions) or sequential execution.
3. **Loop Handling:** For loops, we create two nodes: one for the loop condition and one for the loop body. After executing the loop body, the control returns to the condition check.
4. **End Conditions:** Once the loop condition fails, the flow graph moves to the block where the result is printed, marking the end of the program.

Flow Graph for the Pseudocode:





Explanation of Flow Graph Rules:

1. **Start Node:** Represents the beginning of the program.
2. **Condition Nodes:** Used for loops or conditional checks. For example, the loop condition `FOR i = 1 TO n` is represented as a decision node.
3. **Action Nodes:** Represent the actions being performed, such as reading input, updating variables, or printing output.
4. **End Nodes:** Mark the end of the program or control flow, like the final step where the sum is printed.

This flow graph shows how the control flows through different parts of the program, based on loop conditions and actions performed inside the loop.

Question: Explain the following peephole optimization techniques with examples: i) Elimination of Redundant Code
ii) Elimination of Unreachable Code
[7M]

Answer:

i) Elimination of Redundant Code:

Redundant code refers to instructions or operations that produce the same result and do not contribute to the final output. This code can be removed to improve the efficiency of the program without affecting its correctness.

Example:
Consider the following code:

```
MOV R1, #5
MOV R2, #5
ADD R3, R1, R2
```

In the above code, both `MOV R1, #5` and `MOV R2, #5` are loading the same value into different registers. Since both registers hold the same value and are used in the `ADD` instruction, we can remove the second `MOV` instruction as it is redundant.

Optimized Code:

```
MOV R1, #5
ADD R3, R1, R1
```

Here, we removed the redundant `MOV R2, #5` and used `ADD R3, R1, R1` instead, achieving the same result with fewer instructions.

ii) Elimination of Unreachable Code:

Unreachable code refers to parts of the program that cannot be executed due to the flow of control. These code segments are typically placed after a `return`, `break`, or `goto` statement that ensures the program will never reach them. Removing such code helps in reducing the program size and improving performance.

Example: Consider the following code:

```
int example() {
    int a = 5;
    return a;
    int b = 10;  // Unreachable code
}
```

In the above example, the line `int b = 10;` is unreachable because the function returns before it. This line will never be executed, and therefore, it is considered redundant.

Optimized Code:

```
int example() {
    int a = 5;
    return a;
}
```

Here, we have removed the unreachable code `int b = 10;`, as it will never be executed. This makes the code more efficient and easier to maintain.

Question: Explain briefly about storage organization. [7M]

Answer:

In compiler design, **storage organization** refers to the method used by a compiler to manage memory for storing various data such as variables, constants, and intermediate code during the execution of a program. It involves efficient allocation, tracking, and deallocation of memory.

There are mainly two types of storage organizations:

1. **Static Storage Allocation:**
    - In this method, memory is allocated at compile-time, and the size and location of variables are determined before execution. The variables remain in memory throughout the program's lifetime.

- **Example:** Global variables in a program are statically allocated. For instance, in a C program, a global variable declared outside any function has static storage.

2. **Dynamic Storage Allocation:**
   - Memory is allocated at runtime, and variables are created and destroyed dynamically. This allocation is handled through mechanisms like **heap memory** and **stack memory**.
   - **Example:** Local variables inside a function are dynamically allocated on the stack during the function's execution and are deallocated once the function exits.

Example in Compiler Design: During compilation, the compiler needs to manage both **local variables** (which are allocated dynamically on the stack) and **global variables** (which are statically allocated). For instance, in a function, a local variable `int x` will be allocated memory on the stack, whereas a global variable `int y` will be allocated memory statically.

Question: Discuss briefly about Structure Preserving Transformations. [7M]

Answer:

In compiler design, **Structure Preserving Transformations (SPTs)** refer to the techniques used to modify the structure of a program while preserving its original semantics (meaning or behavior). These transformations are crucial during intermediate code optimization phases, where the goal is to improve the efficiency of the program without changing the output. SPTs maintain the program's logical structure and correctness while optimizing its performance.

Key points about Structure Preserving Transformations:

1. **Preservation of Semantics**: The most important feature of an SPT is that it preserves the program's behavior, meaning that after the transformation, the program should still produce the same output for the same input.
2. **Optimization Techniques**: These transformations are primarily applied to optimize the intermediate code generated by the compiler. Optimizations can include removing redundant code, simplifying expressions, and improving control flow.
3. **Examples of Structure Preserving Transformations**:
   - **Constant Folding**: This optimization involves evaluating constant expressions at compile time rather than runtime. For example:
   - `int x = 3 + 4;`

     This can be transformed to:

     ```
     int x = 7;
     ```

     The structure of the code remains intact, but the computation is simplified.

- **Constant Propagation**: If a variable is assigned a constant value, that value can be propagated throughout the code wherever the variable is used. For example:
- `int a = 5;`
- `int b = a + 10;`

  After constant propagation, this becomes:

  ```
  int b = 15;
  ```

- **Dead Code Elimination**: This involves removing parts of the code that do not affect the program's output. For example, if a variable is assigned a value but is never used, it is removed.

4. **Advantages**:
   - **Increased Efficiency**: By applying structure-preserving transformations, compilers generate code that is more efficient in terms of time and space complexity.
   - **Reduced Resource Usage**: Unnecessary computations and redundant code are eliminated, reducing the overall size and execution time.

Example:

Consider the following code snippet:

```
int x = 10;
int y = x * 2;
int z = x + 5;
```

By applying **constant folding** and **constant propagation**, the optimized version can be:

```
int x = 10;
int y = 20;   // x * 2 is evaluated at compile time
int z = 15;   // x + 5 is evaluated at compile time
```

Q: Describe in detail about Peephole Optimization. What is a flow graph? Explain with a suitable example. [14M]

Answer:

Peephole Optimization

Peephole optimization is a type of code optimization technique in which a small window or "peephole" of a few instructions is examined, and the unnecessary or redundant instructions are replaced with more efficient ones. This technique is often applied during the code generation phase of compilers, and it focuses on small-scale improvements that can make a big impact on performance. The optimization is done by looking at a small set of instructions (usually 3 to 5) and transforming them into a more optimized form.

Key objectives of Peephole Optimization:

- Remove redundant code.
- Replace inefficient code with more efficient alternatives.
- Reduce the size of the code without affecting the logic.

Common Peephole Optimizations include:

1. **Constant Folding**: If an expression involves constants (e.g., `5 + 3`), it can be simplified to the result (`8`).
2. **Dead Code Elimination**: Removing code that doesn't affect the program (e.g., setting a variable and never using it).
3. **Sub-expression Elimination**: Replacing expressions that appear multiple times with their computed value.
4. **Strength Reduction**: Replacing expensive operations like multiplication or division with cheaper operations like addition or subtraction.

Example of Peephole Optimization:

Before Peephole Optimization:

```
x = 5 + 3;  // Constant Expression
y = x * 2;  // Multiplication
```

After Peephole Optimization:

```
x = 8;      // Constant Folding
y = x + x;  // Strength Reduction (multiplication by 2 replaced with
addition)
```

Flow Graph

A **Flow Graph** is a directed graph that represents the control flow of a program. It consists of nodes, where each node represents a basic block of instructions, and edges, which represent the flow of control from one block to another.

In a flow graph:

- **Nodes** represent basic blocks (a sequence of instructions with no branches).
- **Edges** represent possible execution paths, showing how control can move from one block to another.

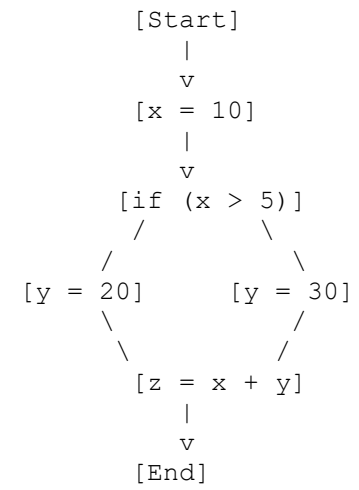
Flow graphs are used in compilers for optimization and in program analysis to detect loops, dead code, and unreachable statements.

Example of Flow Graph:

Consider the following simple code segment:

```
1.  x = 10;
2.  if (x > 5) {
3.      y = 20;
4.  } else {
5.      y = 30;
6.  }
7.  z = x + y;
```

The flow graph for the above code would look like this:



Explanation:

- The graph starts at the instruction `x = 10` and follows through to the `if` statement.
- Depending on the condition `x > 5`, the control flow splits between the two branches (`y = 20` or `y = 30`).
- Finally, it reaches `z = x + y` and ends the program.

Applications of Flow Graph:

1. **Control Flow Analysis:** Understanding how control flows through a program.
2. **Loop Detection:** Identifying loops in the program for optimization purposes.
3. **Reachability Analysis:** Finding unreachable code that can be eliminated.

Question:

Explain the contents of an activation record with an example.

What is an induction variable and identify the induction variables in the following statements in C:

```
for( i=0,j=1,k=2,l=4 ; i< n ; i++){
    j = j + 2;
```

```
    k = k + 3;
    l = l + 4;
    sum = sum + i * j * k;
}
```

Answer:

1. Contents of an Activation Record:

An **activation record** (also known as a **stack frame**) is a data structure that holds information about a function call. It is created when a function is invoked and destroyed once the function execution completes. The activation record contains the following elements:

- **Return Address**: The address to return to after the function completes.
- **Parameters**: The values or references passed to the function.
- **Local Variables**: Variables declared inside the function.
- **Saved Registers**: Registers that need to be preserved between function calls.
- **Control Link**: A pointer to the activation record of the calling function (used for stack unwinding).
- **Data Link**: A pointer to the activation record of the currently running function's stack (if needed).

Example: When a function is called, a new activation record is pushed onto the call stack. For instance, in the function `int sum(int a, int b)`, the activation record will store `a`, `b`, return address, and any local variables used within the function.

2. Induction Variable:

An **induction variable** is a loop variable that changes in a regular, predictable pattern, typically incrementing or decrementing by a constant value in each iteration of a loop. Induction variables help in analyzing the loop behavior and are important in optimization techniques like loop unrolling.

Induction Variables in the Given C Code:

In the given loop:

```
for( i = 0, j = 1, k = 2, l = 4 ; i < n ; i++) {
    j = j + 2;
    k = k + 3;
    l = l + 4;
    sum = sum + i * j * k;
}
```

- **i** is the **induction variable** because it is being incremented by 1 in each iteration of the loop (`i++`).

- **j**, **k**, and **l** are not induction variables because they are updated by a fixed amount within the loop body but do not change directly with respect to the loop index in a regular, linear fashion. They are modified based on constant values (e.g., `j = j + 2`).

Thus, **i** is the primary induction variable in this loop.

Question:

Consider the following intermediate code statements numbered from 1 to 12:

```
1  i = 0
2  if i < n goto 4
3  goto 11
4  t1 = 4 * i
5  t2 = a[t1]
6  t3 = t2 + 1
7  t4 = 4 * i
8  b[t4] = t3
9  i = i + 1
10 goto 2
11 t5 = 4 * i
12 b[t5] = 0
```

(a) Construct a control flow graph for the given code and explain which of the 4 * i computations in statement 7 and statement 11 are redundant.

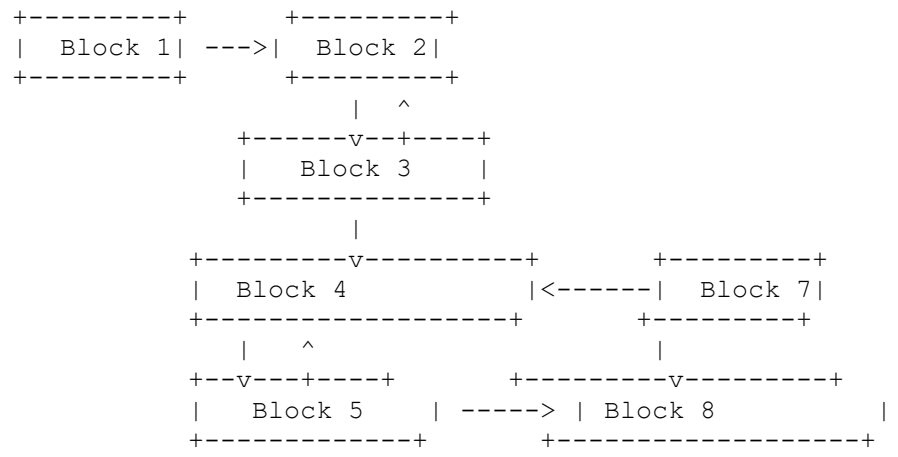
Answer:

Control Flow Graph Construction:

1. **Basic Blocks:**
   - **Block 1**: Statement 1.
   - **Block 2**: Statement 2 (condition checking).
   - **Block 3**: Statement 3 (goto 11).
   - **Block 4**: Statements 4 to 8 (computation and assignments).
   - **Block 5**: Statement 9 (increment).
   - **Block 6**: Statement 10 (goto 2).
   - **Block 7**: Statement 11.
   - **Block 8**: Statement 12.
2. **Control Flow Edges:**
   - Block 1 → Block 2
   - Block 2 → Block 3 (if `i >= n`)
   - Block 2 → Block 4 (if `i < n`)
   - Block 4 → Block 5
   - Block 5 → Block 6
   - Block 6 → Block 2 (loop continues)

- Block 3 → Block 7 (exit condition)
- Block 7 → Block 8

Control Flow Graph:



Redundant Computations of `4 * i`:

The computation `4 * i` is performed in **Statement 4** and **Statement 7**. Let's analyze:

1. **Statement 4**: The expression `t1 = 4 * i` computes the value of `4 * i` for the current value of `i` and stores it in `t1`. This value is then used in statement 5 (`t2 = a[t1]`).
2. **Statement 7**: The expression `t4 = 4 * i` computes `4 * i` again for the same value of `i`, but this time the result is used to update the array `b` at position `b[t4] = t3`.
3. **Statement 11**: The expression `t5 = 4 * i` computes `4 * i` after the loop ends, and this value is used to assign `b[t5] = 0`.

Redundancy:

- The computation of `4 * i` in **Statement 7** is redundant because the same value (`4 * i`) is already computed in **Statement 4** during the loop. Instead of recalculating it in Statement 7, we could reuse the value stored in `t1` from Statement 4.
- The computation in **Statement 11** of `4 * i` is also redundant, as it is simply the value of `i` at the end of the loop, which can be reused from previous calculations (i.e., the loop counter itself).

Question:

Explain with an example why the static allocation strategy is not appropriate for languages like C. [7M]

Answer:

In static memory allocation, memory is allocated at compile time, and the size and number of variables are fixed during the program's execution. This approach is not ideal for languages like C for the following reasons:

1. **Lack of Flexibility**: In C, the size of arrays and variables cannot be changed during program execution. This is problematic when the exact size of data is not known at compile time. For example, consider a program that processes an unknown number of student records. Using static memory allocation would require the programmer to decide on a fixed array size at compile time, leading to either wasted memory or insufficient space.

   Example:

   ```
   #include <stdio.h>
   #define MAX_STUDENTS 100

   int main() {
       int student_scores[MAX_STUDENTS];
       // But if there are only 10 students, the rest of the array is
   wasted
       return 0;
   }
   ```

   In this example, if there are fewer than 100 students, the remaining array memory goes unused. On the other hand, if there are more than 100 students, the program will not have enough space, leading to errors.

2. **Wasted Memory**: In static allocation, memory for the largest possible case must be reserved. This often leads to wasted memory when the program doesn't require all the space.
3. **Inability to Handle Dynamic Data**: If a program needs to handle dynamic or user-defined data (like reading an unknown number of user inputs), static allocation fails because it cannot adjust memory based on runtime conditions.

Question:

Show the contents of the activation record for the function call `f(6)` where the definition of `f` is given as follows, and `f` is called from the main function:

```
int f(int n){
  if (n == 0)
    return 1;
  else
    return n * f(n-1);
}
```

Answer:

When the function `f(6)` is called, an activation record is created for each recursive call. Each activation record stores the following details:

1. **Return address** – where the control should go after the function call is completed.
2. **Parameters** – the arguments passed to the function (in this case, `n`).
3. **Local variables** – any local variables used inside the function (if any).
4. **Saved registers** – storing the values of registers that will be used in the current function call.

Let's now break down the recursive calls and show how the activation records look.

Function Call Stack for `f(6)`:

1. **Call to f(6):**
   - **Return Address**: Main function (where control should return after `f(6)` finishes).
   - **Parameter `n`**: 6
   - **Local Variable**: None.
2. **Call to f(5):**
   - **Return Address**: To the point where `n * f(n-1)` is evaluated in `f(6)`.
   - **Parameter `n`**: 5
   - **Local Variable**: None.
3. **Call to f(4):**
   - **Return Address**: To the point where `n * f(n-1)` is evaluated in `f(5)`.
   - **Parameter `n`**: 4
   - **Local Variable**: None.
4. **Call to f(3):**
   - **Return Address**: To the point where `n * f(n-1)` is evaluated in `f(4)`.
   - **Parameter `n`**: 3
   - **Local Variable**: None.
5. **Call to f(2):**
   - **Return Address**: To the point where `n * f(n-1)` is evaluated in `f(3)`.
   - **Parameter `n`**: 2
   - **Local Variable**: None.
6. **Call to f(1):**
   - **Return Address**: To the point where `n * f(n-1)` is evaluated in `f(2)`.
   - **Parameter `n`**: 1
   - **Local Variable**: None.
7. **Call to f(0):**
   - **Return Address**: To the point where `n * f(n-1)` is evaluated in `f(1)`.
   - **Parameter `n`**: 0
   - **Local Variable**: None.
   - **Base Case Reached**: Since `n == 0`, `f(0)` returns 1.

Unwinding the Stack:

After `f(0)` returns 1, the previous function calls return one by one, performing the multiplication at each level:

- `f(1)` returns `1 * f(0) = 1 * 1 = 1`
- `f(2)` returns `2 * f(1) = 2 * 1 = 2`
- `f(3)` returns `3 * f(2) = 3 * 2 = 6`
- `f(4)` returns `4 * f(3) = 4 * 6 = 24`
- `f(5)` returns `5 * f(4) = 5 * 24 = 120`
- `f(6)` returns `6 * f(5) = 6 * 120 = 720`

Final Activation Record for `f(6)`:

The activation record for the final call `f(6)` looks like this:

- **Return Address**: Main function (to return the result of `f(6)`).
- **Parameter `n`**: 6
- **Local Variable**: None (no local variables).
- **Return Value**: 720

Question:

Explain in detail common sub-expression elimination, copy propagation, and dead code elimination optimizations with examples. (7 Marks)

Answer:

Optimizations in compilers aim to improve the efficiency of the generated code. Some of the commonly used optimizations are:

1. Common Sub-expression Elimination (CSE)

Definition:
Common Sub-expression Elimination (CSE) is an optimization technique that identifies and eliminates expressions that are computed multiple times. It involves finding expressions that appear more than once and replacing them with a temporary variable.

Example:
Consider the following code:

```
a = b + c;
d = b + c;
```

Here, the expression `b + c` is computed twice. With CSE, we can eliminate the redundancy:

```
t = b + c;
a = t;
```

```
d = t;
```

By storing the result of `b + c` in a temporary variable `t`, we avoid recalculating it.

2. Copy Propagation

Definition:
Copy Propagation is an optimization where assignments like `a = b;` are replaced with `a` wherever `b` is used. This helps in eliminating unnecessary assignments and simplifies the code.

Example:
Consider the following code:

```
a = b;
c = a + 5;
```

Here, `a` is just a copy of `b`, so we can propagate the value of `b` directly:

```
c = b + 5;
```

By replacing `a` with `b`, we simplify the code and remove the redundant assignment.

3. Dead Code Elimination

Definition:
Dead Code Elimination is an optimization technique that removes code that does not affect the program's output. This usually involves eliminating unreachable code or instructions that are never used (e.g., calculations whose results are not used).

Example:
Consider the following code:

```
a = 5;
b = a + 2;
c = b + 3;
```

Now, if `c` is never used in the program, the code related to `c` is considered dead code:

```
a = 5;
b = a + 2;
```

In this case, `c = b + 3;` is eliminated because `c` is not used, and it has no effect on the program's output.

Summary:

- **Common Sub-expression Elimination (CSE)** reduces repeated calculations by reusing the results.
- **Copy Propagation** eliminates redundant assignments by replacing variables with their equivalent values.
- **Dead Code Elimination** removes code that doesn't contribute to the final result.

1) Purpose of Live Variable Data Flow Analysis (7 Marks)

Live Variable Analysis is a technique used in compilers to determine which variables in a program hold values that are needed for future computations. A **live variable** is a variable that holds a value that may be used in the future (either directly or indirectly). This analysis is essential in optimizing programs by eliminating unnecessary computations, such as dead code elimination, which reduces execution time and memory usage.

Purpose of Live Variable Analysis:

- **Dead Code Elimination**: Identifies and removes code that does not contribute to the final output, improving performance.
- **Register Allocation**: Helps in efficient register usage by identifying variables that are no longer needed.
- **Optimization**: Helps in optimizing the program by focusing on the variables that are live and eliminating those that are not.

Example:

Consider the following code:

```
int a = 10, b = 20, c;
c = a + b;
b = c * 2;
```

In the above code:

- After the statement `c = a + b;`, `a` and `b` are **live** because their values will be used in the next computation (`b = c * 2;`).
- However, the variable `a` is no longer used after `c = a + b;`, making it **dead** in subsequent computations. This can be safely eliminated in an optimization step.

2) Structure Preserving Transformations (7 Marks)

Structure Preserving Transformations are compiler transformations that change the internal representation of the program without altering the program's semantic meaning. These transformations aim to optimize the program while maintaining its original structure and functionality.

Key Points:

- **Semantic Integrity**: The program's output should remain the same after the transformation.
- **Efficiency**: The transformation aims to improve efficiency, such as reducing the execution time or memory usage, but the logic of the program should not be affected.
- **Examples of Structure-Preserving Transformations**:
    - **Loop Unrolling**: This transformation reduces the overhead of looping by replicating the loop body multiple times.
        - Example:
        - ```
          for (int i = 0; i < 4; i++) {
              a[i] = b[i] + 10;
          }
          ```

          After unrolling:

          ```
          a[0] = b[0] + 10;
          a[1] = b[1] + 10;
          a[2] = b[2] + 10;
          a[3] = b[3] + 10;
          ```

    - **Strength Reduction**: Replaces expensive operations with cheaper ones.
        - Example: Replacing multiplication by a constant with addition.
        - ```
          for (int i = 0; i < 10; i++) {
              a[i] = i * 2;
          }
          ```

          Transformed to:

          ```
          for (int i = 0; i < 10; i++) {
              a[i] = i + i;
          }
          ```

(a) Stack Allocation Strategy:

Stack Allocation refers to the method of allocating memory in a program during execution. This strategy uses the stack data structure to allocate memory for local variables and function calls. The memory for a function's local variables is automatically allocated when the function is called and deallocated when the function returns.

In a stack, memory is allocated in a Last In First Out (LIFO) manner, which means that the most recently called function will have its local variables stored at the top of the stack.

Example:

Consider the following simple program:

```
int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

```
int main() {
    int x = 10, y = 20;
    int result = add(x, y);
    return 0;
}
```

In this program:

- When `add(x, y)` is called, memory for the parameters `a`, `b`, and the local variable `sum` is allocated on the stack.
- Once the function `add` completes, the memory used by these variables is deallocated.

Working of Stack Allocation:

1. The call to `main` pushes a new stack frame for `main` with local variables `x`, `y`, and `result`.
2. The call to `add(x, y)` pushes a new stack frame containing `a`, `b`, and `sum`.
3. After `add` returns, its stack frame is removed, and control returns to `main`.

The stack grows and shrinks as functions are called and return, ensuring automatic memory management for local variables.

(b) Basic Block and Algorithm for Constructing Control Flow Graph:

Basic Block: A **Basic Block** is a sequence of consecutive statements in a program with a single entry point and a single exit point. It has no branches (except at the entry and exit points). It is a fundamental unit of control flow in a program.

For example, the following code represents a basic block:

```
int a = 5;
a = a + 1;
printf("%d", a);
```

This is a basic block because there are no jumps or branches within this sequence of instructions.

Control Flow Graph (CFG): A **Control Flow Graph (CFG)** is a representation of the flow of control within a program. The nodes represent basic blocks, and the edges represent the flow of control between these blocks.

Algorithm for Constructing Control Flow Graph (CFG):

1. **Identify Basic Blocks:**
    - Traverse the intermediate code or program, and identify sequences of instructions with a single entry and exit point.
    - Each identified sequence forms a basic block.
2. **Create Nodes for Basic Blocks:**
    - For each basic block, create a node in the CFG.

3. **Connect the Nodes:**
    - If there is a jump (like `goto`, `if`, `while`, etc.) from one basic block to another, draw an edge from the current block to the target block.
    - For every function call, an edge should be created from the current block to the called function block.
4. **Handle Entry and Exit:**
    - The entry point of the program has no incoming edges, and the exit point has no outgoing edges.

Example:

Consider the following intermediate code (three-address code):

```
1.  a = 5
2.  b = 10
3.  if a > b goto L1
4.  c = a + b
5.  goto L2
L1: d = a - b
L2: e = c + d
```

- Basic blocks can be identified as follows:
    - **Block 1**: `a = 5; b = 10`
    - **Block 2**: `if a > b goto L1`
    - **Block 3**: `c = a + b`
    - **Block 4**: `goto L2`
    - **Block 5**: `d = a - b`
    - **Block 6**: `e = c + d`
- The control flow graph will be:
    - Block 1 → Block 2
    - Block 2 → Block 3 (if the condition is false)
    - Block 2 → Block 5 (if the condition is true)
    - Block 3 → Block 4
    - Block 4 → Block 6
    - Block 5 → Block 6

The constructed CFG will look like this:

```
(Entry) → Block 1 → Block 2 → Block 3 → Block 4 → (Exit)
                        → Block 5 →
```

This graph represents how the program's control flows, showing the different paths it can take based on conditions and jumps.

Question:

Perform available expression analysis on the following intermediate code statements numbered from 1 to 9.

i) Argue whether the expression `a*b` is available at statement 6 and statement 8.

ii) Argue whether the expression `b*c` is available at statement 5 and statement 9.

Intermediate Code:

1. `x = a*b`
2. `y = b*c`
3. `if a>20 goto 6`
4. `z = a*b`
5. `w = b*c`
6. `p = a*b`
7. `a = a+20`
8. `q = a*b`
9. `r = b*c`

Answer:

i) Availability of `a*b` at statement 6 and statement 8:

- **Statement 6 (`p = a*b`)**: The expression `a*b` is available at this point. This is because it is computed at statement 1 (`x = a*b`), and no modification to `a` or `b` has occurred before statement 6. Therefore, the expression `a*b` is still available in the previous statements (1 and 4), making it available at statement 6.
- **Statement 8 (`q = a*b`)**: The expression `a*b` is not available here. This is because at statement 7, `a` is updated with `a = a + 20`, which modifies the value of `a`. As a result, the expression `a*b` from earlier (statement 1, 4, or 6) is no longer valid at statement 8 because `a` has changed.

ii) Availability of `b*c` at statement 5 and statement 9:

- **Statement 5 (`w = b*c`)**: The expression `b*c` is available at statement 5. This is because it is computed in statement 2 (`y = b*c`), and no modification to `b` or `c` has occurred before statement 5. Therefore, the expression `b*c` from statement 2 is still valid here.
- **Statement 9 (`r = b*c`)**: The expression `b*c` is available at statement 9. This is because no modifications have been made to `b` or `c` between statement 5 and statement 9. Thus, the expression `b*c` from statement 2 and 5 remains valid at statement 9.

In summary:

- **`a*b` is available at statement 6 but not at statement 8** due to the update of `a` in statement 7.
- **`b*c` is available at both statement 5 and statement 9** because there were no changes to `b` or `c` between these statements.