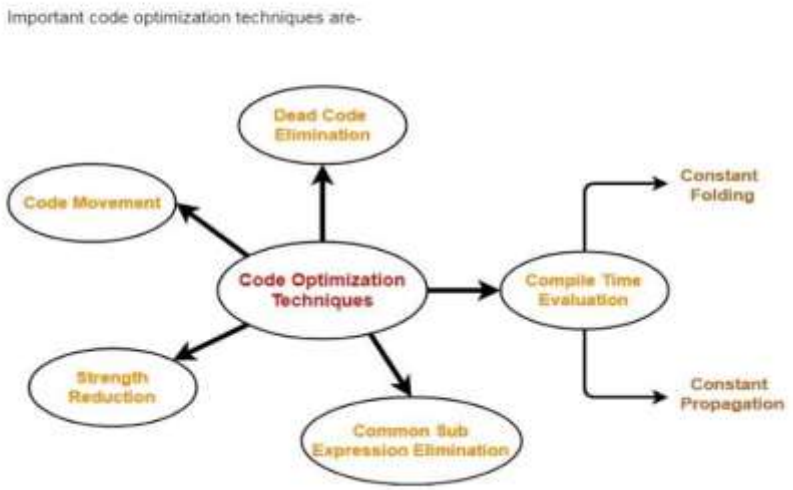
Code Optimization

What are sources of optimization?

Optimization in coding involves enhancing code performance by eliminating unnecessary lines and rearranging statements. The sources of optimization include various techniques and principles aimed at improving code efficiency and execution speed. Here are the key sources of :

Principle sources of optimization/ Code Optimization Techniques-



1. **Compile Time Evaluation**:
    - **Constant Folding**: Involves evaluating expressions with constant operands at compile time and replacing them with their results.
    - **Constant Propagation**: Substitutes variables with constant values during compilation if the value remains constant throughout the program.
2. **Common Sub-Expression Elimination**:
    - Identifies and eliminates redundant expressions that have been computed before to avoid recomputation.
3. **Code Movement**:

- Involves moving code outside loops if its presence inside or outside the loop does not affect the program's outcome, thus avoiding unnecessary execution.

4. **Dead Code Elimination**:
    - Eliminates code statements that are unreachable, never executed, or whose output is not used in the program.
5. **Strength Reduction**:
    - Focuses on replacing complex and costly operators with simpler and more efficient ones to optimize code performance.
6. **Peephole Optimization**:
    - A local optimization technique that evaluates small code segments and optimizes them based on predefined rules to improve execution speed, reduce code size, eliminate dead code, and simplify code.

These optimization techniques play a crucial role in enhancing code efficiency, reducing memory usage, and improving overall program performance. By applying these principles effectively, developers can create optimized code that runs faster and utilizes resources more efficiently.

What is peephole optimization?

Peephole optimization is a local optimization technique used in compilers to enhance the efficiency of generated code. It focuses on evaluating a small segment of code, typically a few instructions, and optimizing them based on predefined rules. This technique works by identifying and eliminating redundant or unnecessary instructions within this small window of code. Peephole optimization aims to improve code performance by reducing execution time, minimizing code size, eliminating dead code, and simplifying the code structure without altering the program's output.

Objectives of Peephole Optimization in Compiler Design:

- **Increasing Code Speed**: Enhancing the execution speed of the generated code by removing redundant or unnecessary instructions.
- **Reducing Code Size**: Minimizing the size of the generated code by replacing lengthy sequences of instructions with shorter, more efficient ones.
- **Eliminating Dead Code**: Getting rid of unreachable or redundant code segments that do not impact the program's output.
- **Simplifying Code**: Making the generated code more readable and manageable by removing unnecessary complexities.

Peephole Optimization Techniques:

1. **Redundant Instruction Elimination**: Removing redundant loads, stores, or other instructions that do not affect the program's outcome.
2. **Unreachable Code**: Eliminating code segments that are never accessed or executed due to programming constructs.
3. **Flow of Control Optimization**: Streamlining the flow of control by removing unnecessary jumps or loops that do not contribute to the program's logic.
4. **Algebraic Simplification**:
    - There are occasions where algebraic expressions can be made simple.

- For example, the expression a = a + 0 can be replaced by a itself and the expression a = a + 1 can simply be replaced by INC a.

5. **Reduction in strength**:
    - There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.
    - for example, **x * 2 can be replaced x+x.**

6. **Machine idioms**
    - So The target instructions have equivalent machine instructions for performing some have operations.
    - Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
    - Example: Some machines have auto-increment or auto decrement addressing modes. These modes can use in a code for a statement like i=i+1

Peephole optimization plays a vital role in improving code quality and performance by fine-tuning small sections of code, resulting in faster execution, reduced memory usage, and more streamlined programs.

What are basic blocks?

Basic blocks play a fundamental role in code optimization, especially in the context of compiler design and program analysis.

Definition of Basic Blocks:

- **Basic Blocks**: A basic block is a sequence of intermediate code with a single entry point and a single exit point. It is a fundamental unit of code that does not contain any jumps in the middle, making it a linear sequence of instructions.

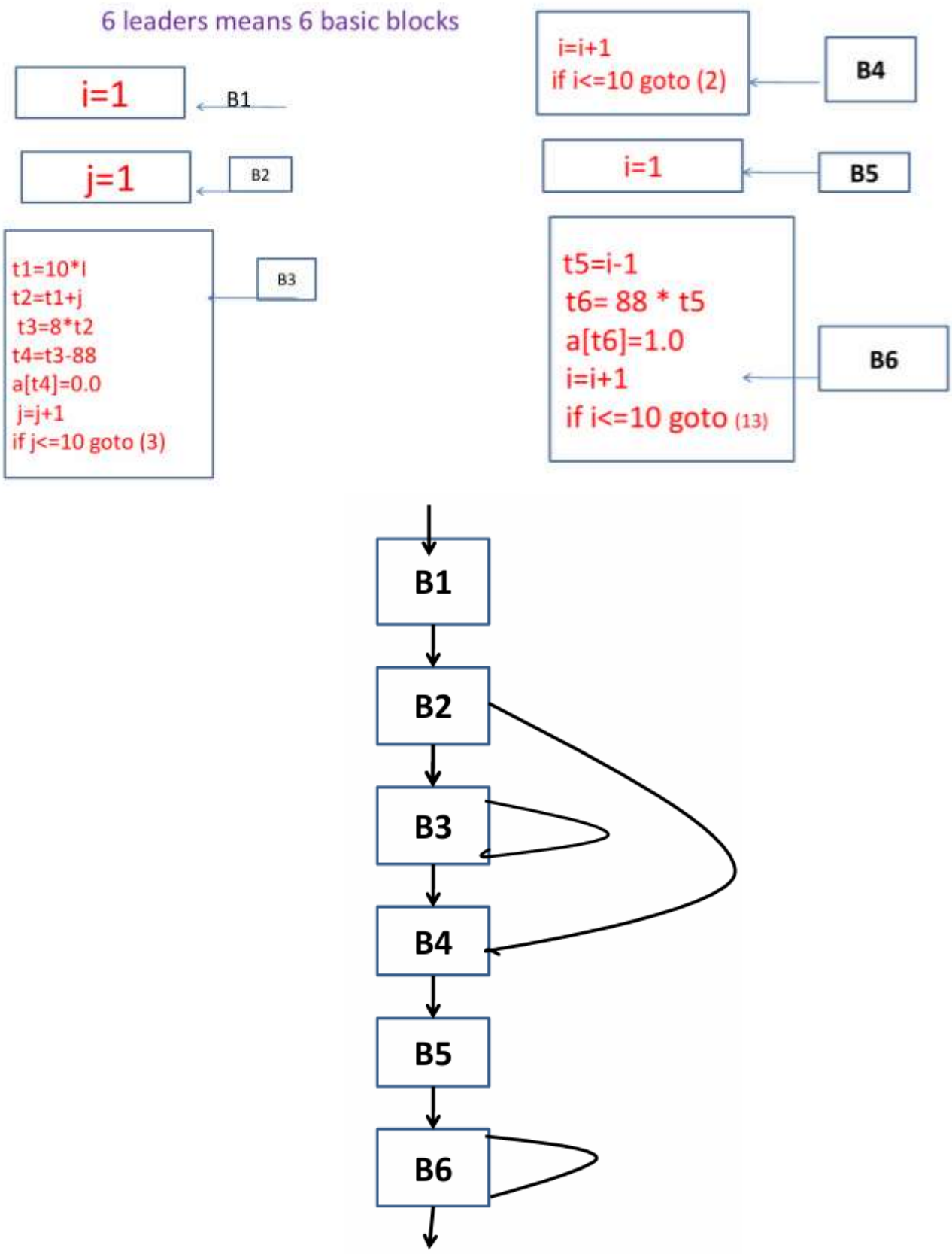
Algorithm for Constructing Basic Blocks:

1. **Identifying Leaders**: The first line of code and addresses of conditional and unconditional goto statements are considered leaders.
2. **Creating Basic Blocks**: Basic blocks are formed from one leader to the line before the next leader, ensuring that each block has a single entry and exit point.

Example of Basic Blocks:

- Consider the following code snippet:

```
1) i=1
2) j=1
3) t1=10*I
4) t2=t1+j
5) t3=8*t2
6) t4=t3-88
7) a[t4]=0.0
8) j=j+1
9) if j<=10 goto (3)
10) i=i+1
11) if i<=10 goto (2)
12) i=1
13) t5=i-1
14) t6= 88 * t5
15) a[t6]=1.0
16) i=i+1
17) if i<=10 goto (13)
```



- This code can be divided into six basic blocks: B1, B2, B3, B4, B5, and B6, each with a distinct entry and exit point.

Optimization of Basic Blocks using Directed Acyclic Graphs (DAGs):

- **DAGs**: Directed Acyclic Graphs are a special type of Abstract Syntax Tree used for optimizing basic blocks.
- **Construction**: DAGs are constructed from Three Address Code to identify common sub-expressions, optimize computations, and simplify code.
- **Applications**: DAGs help in determining redundant computations, identifying dependencies, and optimizing code by eliminating unnecessary instructions.

Understanding basic blocks is crucial in code optimization as it allows for targeted optimizations within specific segments of code, leading to improved performance, reduced memory usage, and streamlined execution.

What are loops in flow graphs?

Flow Graph Representation:

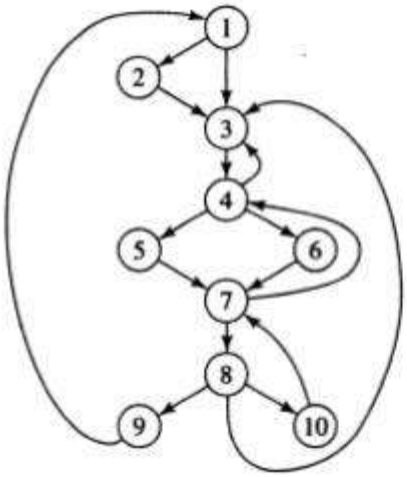
Flow graphs represent three-address statements where nodes denote computations and edges signify the flow of control.

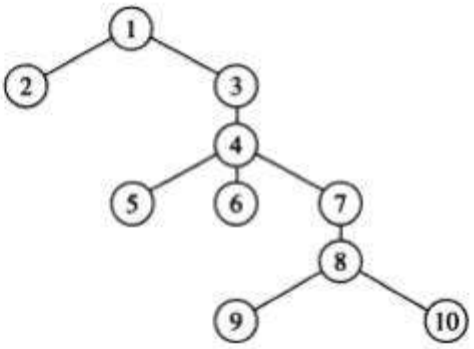
1. **Dominators**:

   In a flow graph, a node dd dominates node nn if every path from the initial node of the flow graph to nn passes through dd. This dominance relationship is denoted as d dom nd dom n. The concept of dominance is fundamental in identifying loops and optimizing code.

   Properties:

   - The initial node dominates all nodes.
   - Every node dominates itself.
   - Example:
     - Node 1 dominates every node.
     - Node 2 dominates itself.
     - Node 3 dominates all but 1 and 2.
     - Node 4 dominates all but 1, 2, and 3.
     - Nodes 5 and 6 dominate only themselves.
     - Node 7 dominates 7, 8, 9, and 10.
     - Node 8 dominates 8, 9, and 10.
     - Nodes 9 and 10 dominate only themselves.





2. **Natural Loops**:
   Natural loops are identified based on the presence of back edges in a flow graph. A loop must have a single entry point, called the header, and there must be at least one way to iterate the loop. Back edges in a flow graph indicate loops, and the natural loop of a back edge consists of the destination node of the back edge (dd) and all nodes that can reach the source node (nn) of the back edge without going through dd.
   - A loop must have a single entry point called the header and must have at least one path back to the header.
   - Loops can be identified by searching for back edges in the flow graph, where the head dominates the tail.
   - Example:
     - Back edges:
       $7 \rightarrow 4$
       $10 \rightarrow 7$
       $4 \rightarrow 3$
       $8 \rightarrow 3$
       $9 \rightarrow 1$
   - The natural loop of a back edge consists of the tail plus the set of nodes that can reach the tail without going through the head.

   Algorithm for Constructing Natural Loops: Given a flow graph GG and a back edge $n \rightarrow dn \rightarrow d$, the algorithm constructs the natural loop consisting of all nodes in the loop. It starts with the source node nn and iteratively adds nodes to the loop by considering their predecessors until all nodes in the loop are identified.

3. **Inner Loops**:
   Inner loops are loops that contain no other loops. When two natural loops have the same header but are not nested within each other, they are combined and treated as a single loop. Inner loops are important for understanding the structure of code and optimizing transformations.
   - Loops that contain no other loops are termed as inner loops.
   - Two loops with the same header but not nested within each other are combined and treated as a single loop.

4. **Pre-Headers**:
   Pre-headers are introduced before loop headers to facilitate certain code transformations that require moving statements before the loop. Pre-headers ensure that all edges entering the loop header from outside the loop are redirected to the pre-header.
   - Pre-Headers are created to move statements before the loop header.
   - Pre-Headers only have the header as a successor, and all edges entering the header from outside the loop enter the pre-header. it.

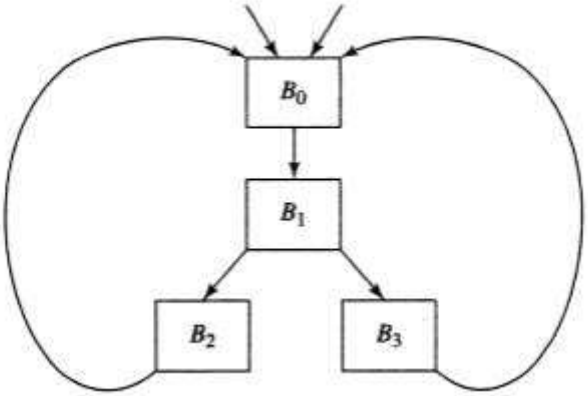


Fig. 5.4 Two loops with the same header

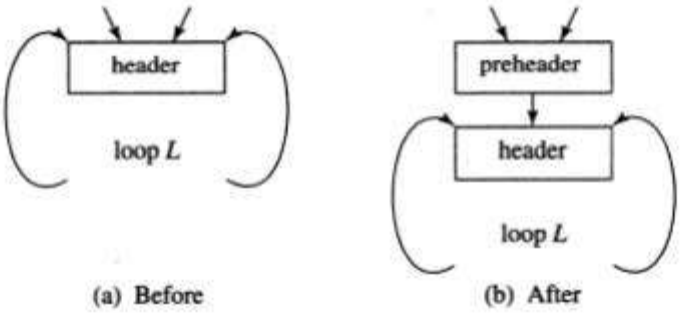


Fig. 5.5 Introduction of the preheader

5. **Reducible Flow Graphs**:
   Reducible flow graphs are special flow graphs where code optimization transformations are easier to perform. In reducible flow graphs, loops are well-defined, dominators can be efficiently calculated, and data flow analysis problems can be solved effectively. Structured flow-of-control statements lead to reducible flow graphs.

- Special flow graphs where loops are unambiguously defined, dominators can be easily calculated, and data flow analysis problems can be efficiently solved.
- Properties:

1. No jumps into the middle of loops from outside.
2. The only entry to a loop is through its header.

- A flow graph is reducible if edges can be partitioned into forward and back edges with specific properties.