1. Basic Block

Definition

A **basic block** is a **sequence of consecutive statements** in a program such that:

- Control **enters only at the first statement**
- Control **leaves only at the last statement**
- There are **no jumps or branches** inside the block except at the end

In simple words, once execution starts in a basic block, **all statements execute in order without interruption**.

Characteristics of a Basic Block

1. Single entry point
2. Single exit point
3. No branching inside the block
4. All statements execute sequentially

Example

Consider the following code:

1: a = b + c;

2: d = a * e;

3: if (d > 10) goto L1;

4: f = d - 1;

5: L1: g = f + 2;

Basic Blocks:

- **Block B1**
- a = b + c;
- d = a * e;
- if (d > 10) goto L1;
- **Block B2**

- f = d - 1;
- **Block B3**
- g = f + 2;

Why Basic Blocks are Important

- Simplify program analysis
- Used in optimization
- Used to construct CFG
- Makes code generation easier

📌 **Exam Tip:**
Write **definition + properties + example**.

---

2. Control Flow Graph (CFG)

Definition

A **Control Flow Graph (CFG)** is a **graphical representation of program control flow**, where:

- **Nodes** represent **basic blocks**
- **Edges** represent **possible flow of control** between blocks

CFG shows **how execution moves from one basic block to another**.

Why CFG is Needed

- Helps in program analysis
- Used in optimization techniques
- Identifies loops and branches
- Used in data flow analysis

Construction of Control Flow Graph (CFG)

CFG construction is done in **three main steps**.

Step 1: Identify Leaders

A **leader** is the first statement of a basic block.

A statement is a leader if:

1. It is the **first statement** of the program
2. It is the **target of a jump**
3. It **immediately follows a jump statement**

Step 2: Form Basic Blocks

- Each leader starts a new basic block
- Block ends just before the next leader

Step 3: Construct Edges

Add directed edges:

- From one block to the next (sequential flow)
- From a block with a jump to the target block
- From conditional blocks to both true and false targets

Example CFG Construction

Code:

1: a = 10;

2: if (a > 5) goto L1;

3: b = a + 1;

4: goto L2;

5: L1: b = a - 1;

6: L2: c = b * 2;

Basic Blocks:

- B1: a = 10; if (a > 5) goto L1;
- B2: b = a + 1; goto L2;
- B3: b = a - 1;
- B4: c = b * 2;

CFG Edges:

- B1 → B2 (false)
- B1 → B3 (true)
- B2 → B4
- B3 → B4

📌 **Exam Tip:**
Draw a **CFG diagram** with arrows.

Introduction

Code optimization is an important phase of the compiler in which the compiler tries to **improve the performance of the program** without changing its meaning. Optimization helps in:

- Reducing execution time
- Saving memory
- Reducing number of instructions

Among many optimization techniques, the most important ones are:

1. Loop Optimization
2. Loop-Invariant Computation
3. Peephole Optimization
4. Global Data-Flow Analysis

---

1. Loop Optimization

Definition

Loop optimization refers to a set of techniques used to **improve the efficiency of loops**, since loops execute many times and consume a large portion of execution time.

Even a small improvement inside a loop can give **large performance gain**.

Common Loop Optimization Techniques

a) Loop Unrolling

- Repeats the loop body multiple times
- Reduces loop control overhead

📌 Example:

Before optimization

```
for(i = 0; i < 4; i++)
  a[i] = b[i];
```

After unrolling

```
a[0] = b[0];
```

```
a[1] = b[1];
a[2] = b[2];
a[3] = b[3];
```

✓ Reduces number of loop checks and jumps

---

b) Loop Fusion

- Combines two loops with same bounds into one

📌 Example:

```
for(i=0;i<n;i++) a[i]=b[i]+1;
for(i=0;i<n;i++) c[i]=a[i]*2;
```

After fusion:

```
for(i=0;i<n;i++) {
  a[i]=b[i]+1;
  c[i]=a[i]*2;
}
```

✓ Reduces loop overhead

---

c) Loop Fission

- Splits one loop into multiple loops
- Improves cache performance

Advantages of Loop Optimization

- Faster execution
- Better CPU utilization
- Reduced overhead

---

2. Loop-Invariant Computation (Loop-Invariant Code Motion)

Definition

A **loop-invariant computation** is an expression inside a loop whose **value does not change across loop iterations**.

Such computations should be moved **outside the loop**.

Why It Is Needed

If an expression gives the same result every time:

- No need to recompute it
- Saves time and instructions

Example

Before optimization

```
for(i = 0; i < n; i++) {
  x = a * b;
  y[i] = x + i;
}
```

Here, a * b does not depend on i.

After optimization

```
x = a * b;
for(i = 0; i < n; i++) {
  y[i] = x + i;
}
```

✓ Computation done only once

Conditions for Loop-Invariant Code

- Operands are constants or not modified inside loop
- Expression result remains same

Advantages

- Reduces redundant computations
- Improves loop performance
- Very effective optimization

---

3. Peephole Optimization

Definition

Peephole optimization is a **local optimization technique** applied to a **small set of consecutive instructions** (like looking through a small “peephole”).

It is usually applied **after code generation**.

How It Works

- Compiler looks at a small window of instructions
- Replaces inefficient code with efficient code

Common Peephole Optimizations

a) Redundant Instruction Elimination

MOV R1, R1

→ Removed

---

b) Algebraic Simplification

x = x + 0;

→ Removed

x = x * 1;

→ Removed

---

c) Strength Reduction

Replace costly operations with cheaper ones.

x = y * 2;

→

x = y << 1;

Advantages

- Simple and fast
- Improves machine-level code
- Reduces instruction count

Limitations

- Works on small code fragments only
- Cannot perform global optimization

---

4. Global Data-Flow Analysis

Definition

Global data-flow analysis is an optimization technique that analyzes **how data values flow across the entire program**, not just a single block.

It works on the **Control Flow Graph (CFG)**.

Why It Is Needed

Some optimizations require information from:

- Multiple basic blocks
- Different paths of execution

Key Data-Flow Information

- **Reaching Definitions**

- **Live Variables**
- **Available Expressions**
- **Constant Propagation**

Example: Live Variable Analysis

A variable is **live** if its value is used later.

x = 10;

y = 20;

print(x);

Here:

- x is live
- y is dead → can be removed

Optimizations Enabled by Data-Flow Analysis

- Dead code elimination
- Constant propagation
- Register allocation
- Code motion

Advantages

- Powerful optimization
- Improves overall program performance
- Enables advanced optimizations

Limitations

- Complex
- Requires CFG
- Time-consuming

Comparison of Optimization Techniques

Technique	Scope	Type
Loop Optimization	Loop	Local
Loop-Invariant Motion	Loop	Local
Peephole Optimization	Small instruction window	Local
Data-Flow Analysis	Whole program	Global