Logo, company name

Description automatically generated

**COMSATS University Islamabad (CUI)**

**Lab Terminal**

**Submitted to: Sir Bilal Bukhari**

**Submitted By: Ziyad Amjad**

**Reg No: FA21-BCS-057**

**Course: Topics in Computer Science**

**Date: 3 jan, 2025**

**Question 3:**

## 1. Constant Folding

**What it is:**  
Constant folding involves evaluating constant expressions at compile-time rather than at run-time. For example, if your code has something like x = 2 + 3;, it can be directly replaced with x = 5; during compilation.

**How to implement:**

1. When parsing or constructing your Abstract Syntax Tree (AST), check if an operation’s operands are all constant (literals).
2. If so, compute the result immediately and replace the expression subtree with a single node holding the computed value.
3. Propagate that constant value through subsequent intermediate code generation and final code emission.

**Benefits:**

* Reduces run-time computation.
* Simplifies subsequent optimization passes because fewer complex expressions remain.

## 2. Dead Code Elimination

**What it is:**  
Dead code refers to instructions or blocks that do not affect the program’s final output. Identifying and removing them leads to more concise and efficient generated code.

**How to implement:**

1. Conduct a liveness analysis on variables and expressions. Track where variables are read and written.
2. Identify code paths (assignments, method calls, etc.) that do not contribute to any live variable or program state.
3. Remove those instructions or entire blocks if they are not reachable or do not affect externally visible behavior.

**Benefits:**

* Decreases code size.
* Eliminates unnecessary memory writes or computations.

## 3. Peephole Optimization

**What it is:**  
Peephole optimization scans small windows (“peepholes”) of generated instructions (often in the intermediate representation or final assembly) and optimizes them locally. It looks for patterns like unnecessary loads and stores, jump-to-jump sequences, or redundant moves that can be replaced with simpler sequences.

**How to implement:**

1. After generating a sequence of intermediate or final code, scan through consecutive instructions in pairs or short tuples.
2. Look for patterns such as:
   * load r1, x; store x, r1; which can often be removed or simplified.
   * goto L1; L1: goto L2; which can collapse into a single jump.
   * Redundant comparisons (e.g., comparing a variable that was just compared without any modifications in between).
3. Substitute or remove instructions based on these patterns.

**Benefits:**

* Very straightforward to implement incrementally.
* Results in tighter, faster machine code.

## 4. Common Subexpression Elimination (CSE)

**What it is:**  
Common Subexpression Elimination identifies expressions that are repeated within a given scope. If the expression’s operands have not changed, the computation can be done once and reused.

**How to implement:**

1. During or after AST creation, annotate expressions with a representation of their operation and operands (e.g., (+, x, y) for x + y).
2. Maintain a table (often called an expression table or value numbering table) to see if the same operation+operand pair has appeared before.
3. If it has, replace the repeated expression with a reference to the previously computed value (a temporary variable or register).

**Benefits:**

* Minimizes redundant calculations.
* Reduces the number of instructions in the final program.

## 5. Strength Reduction

**What it is:**  
Strength reduction replaces expensive operations (like multiplication or division) with cheaper ones (like addition or bitwise shifts) when possible. For instance, x \* 2 can often become x << 1 in many lower-level representations (though in Java bytecode, the JIT might handle this, it’s still a worthwhile step in an IR).

**How to implement:**

1. Identify operations such as multiplication by a power of two.
2. Replace them with left shifts (e.g., x \* 8 becomes x << 3).
3. Similarly, division by a power of two can be replaced by right shifts, provided you handle edge cases (like signed vs. unsigned shifts if you’re targeting a lower-level language).

**Benefits:**

* Reduces execution time, especially on hardware where multiplication/division is more expensive than shifts.
* Helps the compiler produce code that can run faster, especially in tight loops.