**Q3) Give example of Code Optimizations used in mini compiler.**

**1. Dead Code Elimination**

This optimization removes parts of the code that will **never be executed** or do not contribute to the program's final output. Dead code unnecessarily bloats the program, wastes memory, and slightly slows down execution. Removing it makes the program cleaner and more efficient.

**Why It's Useful:**

1. Reduces the size of the program.
2. Removes clutter, making debugging and maintaining code easier.
3. Minimizes unnecessary computations during runtime.

**Example 1:**

**Before Optimization:**

int x = 10;

int y = 20;

x = y + 5; // `x` is reassigned before it is used

return y; // The value of `x` is not used

**After Optimization:**

int y = 20;

return y;

Here, x = 10 and x = y + 5 are completely removed because x is never used in the output.

**Example 2:**

**Before Optimization:**

void compute() {

int a = 5;

int b = 10;

if (false) {

b = a + 20; // This block will never execute

}

printf("%d", b);

}

**After Optimization:**

void compute() {

int b = 10;

printf("%d", b);

}

The if (false) block is eliminated because it is never executed.

**2. Constant Folding**

Constant folding evaluates constant expressions **at compile time** instead of runtime, replacing the expression with its computed value. This reduces runtime computations and speeds up execution.

**Why It's Useful:**

1. Saves computation time at runtime.
2. Simplifies expressions, leading to shorter and cleaner intermediate code.
3. Eliminates unnecessary steps in code execution.

**Example 1:**

**Before Optimization:**

int result = (10 \* 5) + (20 / 4) - 3;

**After Optimization:**

int result = 55;

Here, the compiler calculates the result of (10 \* 5) + (20 / 4) - 3 at compile time, replacing the entire expression with 55.

**Example 2:**

**Before Optimization:**

float area = 3.14 \* 5 \* 5;

**After Optimization:**

float area = 78.5;

The value of 3.14 \* 5 \* 5 is computed at compile time and replaced with 78.5.

**3. Constant Propagation**

Constant propagation substitutes a variable that holds a constant value with its value throughout the code. This improves readability and enables further optimizations like constant folding.

**Why It's Useful:**

1. Eliminates intermediate variables, reducing memory usage.
2. Improves opportunities for other optimizations, like dead code elimination.
3. Simplifies code for better understanding.

**Example 1:**

**Before Optimization:**

int a = 10;

int b = a + 5;

int c = b \* 2;

**After Optimization:**

int b = 15;

int c = 30;

Here, a is a constant, so it is propagated into the calculation of b. Then, b is propagated into c.

**Example 2:**

**Before Optimization:**

const int multiplier = 5;

int result = multiplier \* 10;

**After Optimization:**

int result = 50;

The constant multiplier is propagated directly into the expression.

**4. Loop Optimization**

Loops are a major source of computational expense in programs. Optimizing them can dramatically improve performance. Techniques include **loop unrolling**, **invariant code motion**, and **strength reduction**.

**Why It's Useful:**

1. Reduces the number of instructions executed inside loops.
2. Minimizes redundant computations, especially for large datasets.
3. Exploits modern CPU architectures for faster execution.

**Example 1: Loop Invariant Code Motion**

Code that doesn’t change during the loop is moved outside the loop to prevent redundant calculations.

**Before Optimization:**

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

int constant = x \* y; // Does not depend on `i`

array[i] = constant + i;

}

**After Optimization:**

int constant = x \* y;

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

array[i] = constant + i;

}

Here, x \* y is computed once outside the loop instead of during each iteration.

**Example 2: Loop Unrolling**

In loop unrolling, multiple iterations are combined into a single iteration to reduce overhead.

**Before Optimization:**

for (int i = 0; i < 4; i++) {

array[i] = i \* 2;

}

**After Optimization:**

array[0] = 0 \* 2;

array[1] = 1 \* 2;

array[2] = 2 \* 2;

array[3] = 3 \* 2;

This eliminates the loop altogether, which can be more efficient for small, fixed loops.

**5. Inline Function Expansion**

Replacing function calls with the actual body of the function reduces the overhead of calling and returning from a function. This is particularly beneficial for small functions called repeatedly in performance-critical code.

**Why It's Useful:**

1. Reduces function call overhead (e.g., stack operations).
2. Allows further optimizations (e.g., constant folding or propagation within the inlined code).
3. Improves performance for small and frequently used functions.

**Example 1:**

**Before Optimization:**

int square(int x) {

return x \* x;

}

int main() {

int result = square(5);

}

**After Optimization:**

int main() {

int result = 5 \* 5;

}

The function call is replaced with the actual computation, avoiding the overhead.

**Example 2:**

**Before Optimization:**

int add(int a, int b) {

return a + b;

}

int main() {

int sum = add(3, 4);

}

**After Optimization:**

int main() {

int sum = 3 + 4;

}

Here, the function add is inlined directly into the code.