Review of Compiler Optimizations

1. Redundant Expression Elimination (Common Subexpression Elimination)

   Use an address or value that has been previously computed. Consider control and data dependencies.

2. Partially Redundant Expression (PRE) Elimination

   A variant of Redundant Expression Elimination. If a value or address is redundant along *some* execution paths, add computations to other paths to create a fully redundant expression (which is then removed).

   Example:

```
if (i > j)
   a[i] = a[j];
a[i] = a[i] * 2;
```

3. Constant Propagation

If a variable is known to contain a particular constant value at a particular point in the program, replace references to the variable at that point with that constant value.

4. Copy Propagation

After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable (until one or the other of the variables is reassigned).

(This may also "set up" dead code elimination. Why?)

5. Constant Folding

An expression involving constant (literal) values may be evaluated and simplified to a constant result value. Useful when constant propagation is performed.

6. Dead Code Elimination

Expressions or statements whose values or effects are unused may be eliminated.

7. Loop Invariant Code Motion

An expression that is *invariant* in a loop may be moved to the loop's header, evaluated once, and reused within the loop.

Safety and *profitability* issues may be involved.

8. Scalarization (Scalar Replacement)

A field of a structure or an element of an array that is repeatedly read or written may be copied to a local variable, accessed using the local, and later (if necessary) copied back.

This optimization allows the local variable (and in effect the field or array component) to be allocated to a register.

9. Local Register Allocation

Within a *basic block* (a straight line sequence of code) track register contents and reuse variables and constants from registers.

10. Global Register Allocation

Within a subprogram, frequently accessed variables and constants are allocated to registers. Usually there are *many more* register candidates than available registers.

11. Interprocedural Register Allocation

Variables and constants accessed by more than one subprogram are allocated to registers. This can *greatly* reduce call/return overhead.