Contents

Code Generation
Code Optimization

Prepared By:
Prashant Gautam
2021

Code Generator

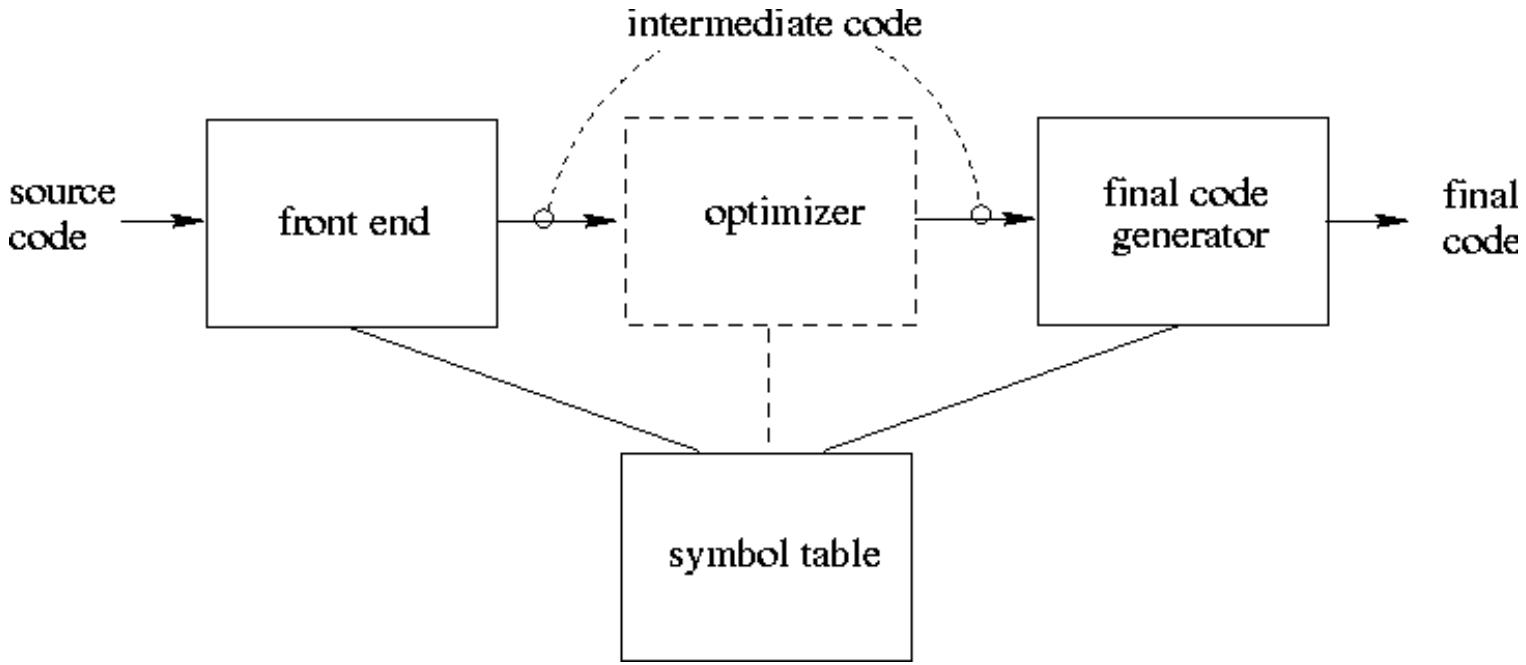
- Code generator is the final phase of compiler which takes as input intermediate representation produced by front end along with the relevant symbol table information and produces as output a symantically equivalent target code.
- The output of intermediate code generator may be given directly to code generation or may pass through code optimization before generating code.
- Code produced by compiler must be correct and be of high quality.
- Source-to-target program transformation should be *semantics preserving* and effective use of target machine resources.
- Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is undecidable.

This phase generates the target code consisting of assembly code.

1. Memory locations are selected for each variable;

2. Instructions are translated into a sequence of assembly instructions;

3. Variables and intermediate results are assigned to memory registers.



Issues in Design of Code generation:

Target code mainly depends on available instruction set and efficient usage of registers. The main issues in design of code generation are:

- Input to the Code Generator
- The Target Program
- Instruction Selection
- Register Allocation
- Evaluation Order

Input to the Code Generator

- The input to the code generator is
  - The intermediate representation of the source program produced by the front-end along with the symbol table.
- Choices for the IR
  - Three-address representations such as 3AC: quadruples, triples, indirect triples
  - Virtual machine representations: bytecode
  - Linear representations: postfix notation
  - Graphical representation: syntax tree, DAG's
- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

The Target Program

- The output of the code generator is the target program. The output may be :
  a. Absolute machine language - It can be placed in a fixed memory location and can be executed immediately.
  b. Relocatable machine language - It allows subprograms to be compiled separately.
  c. Assembly language - Code generation is made easier.
- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.
- The most common target-machine architecture are RISC, CISC, and stack based.
  - A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
  - A CISC machine typically has few registers, two-address instructions, and variety of addressing modes, several register classes, variable-length instructions, and instruction with side effects.
  - In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack.

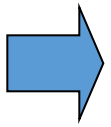
Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by a target machine.
- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size. The former statement can be translated into the latter statement as shown below:
  Instruction selection is important to obtain efficient code
- It depens upon the nature of instruction set architecture.
- For each type of three address code, we can design a code skeleton that defines the target code to be generated. For example we translate three-address code *x*:=*y*+*z*

to:

MOV *y*, **R0**
ADD *z*, **R0**
MOV R0, *x*

`a:=a+1`



```
MOV a,R0
ADD #1,R0
MOV R0,a
```

Cost = 6

Better

`ADD #1,a`

Cost = 3

Better

`INC a`

Cost = 2

Register Allocation

- Accessing values in registers is much faster than accessing main memory.
- A key problem in code generation is deciding what values to hold in what registers.
- Efficient utilization is particularly important.
- The use of registers is often subdivided into two sub problems:

  1. **Register Allocation**, during which we select the set of variables that will reside in registers at each point in the program.

  2. **Register assignment**, during which we pick the specific register that a variable will reside in.

- Finding an optimal assignment of registers to variables is difficult, even with single-register machine.

- Mathematically, the problem is NP-complete.

Register Allocation

Example

```
t:=a*b
t:=t+a
t:=t/d
```

{ R1=t }

```
MOV a,R1
MUL b,R1
ADD a,R1
DIV d,R1
MOV R1,t
```

```
t:=a*b
t:=t+a
t:=t/d
```

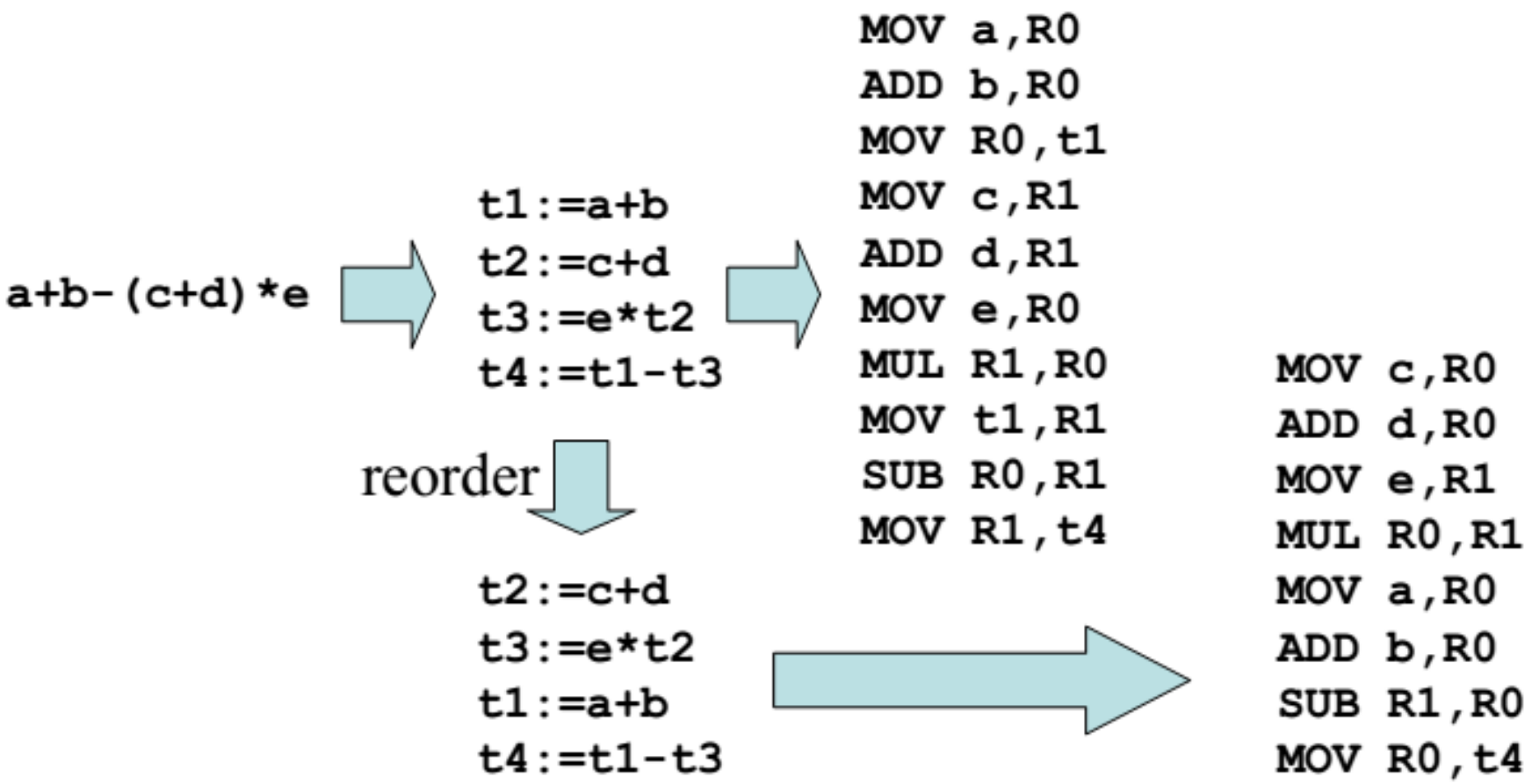
{ R0=a, R1=t }

```
MOV a,R0
MOV R0,R1
MUL b,R1
ADD R0,R1
DIV d,R1
MOV R1,t
```

Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- When the instructions are independent, their evaluation order can be changed.
- However, picking a best order in the general case is a difficult NP-complete problem.

Evaluation Order



The Target Language

- Familarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- In this chapter, target language is assembly code for a simple computer that is representative of many register macnines i.e. A Simple Target Machine Model

A Simple Target Machine Model

- Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.
- The underlying computer is a byte-addressable machine with *n* general-purpose registers.
- Assume the following kinds of instructions are available:
  - **Load operations : LD r, x** loads value in location x in register r.
  - **Store operations: ST x, r** stores value in register r in location x.
  - **Computation operations: SUB r1, r2, r3** computes r1 = r2 – r3
  - **Unconditional jumps: BR L** causes control to branch to machine instruction with label L.
  - **Conditional jumps: BLTZ r, L** causes to jump to label L if the value in register r is less than zero.

Contd…

- Assume a variety of addressing modes:
  - A variable name *x* referring o the memory location that is reserved for *x* , i.e. the l-value of x.
  - Indexed address, *a* (*r*), where *a* is a variable and *r* is a register.

    For example: LD r1, a (r2) has the effect of setting r1 = contents(a + contents(r2)).
  - A memory can be an integer indexed by a register, for example, **LD R1, 100(R2)** has the effect of setting r1 = contents (100 + contents(R2)).
  - Two indirect addressing modes: **r* means the memory location found in the location represented by the contents of register r. and *100(*r*) means the location found in the location obtained by adding 100 to the contents of r. eg. LD r1, *100(r2)
  - Immediate constant addressing mode: LD r1, #100.

A Simple Target Machine Model

X = Y - Z

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2     // R1 = R1 - R2
ST  x, R1          // x = R1
```

b = a[i]
(8-byte elements)

```
LD  R1, i          // R1 = i
MUL R1, R1, 8      // R1 = R1 * 8
LD  R2, a(R1)      // R2 = contents(a + contents(R1))
ST  b, R2          // b = R2
```

x = *p

```
LD  R1, p          // R1 = p
LD  R2, 0(R1)      // R2 = contents(0 + contents(R1))
ST  x, R2          // x = R2
```

```
a[j] = c ⇒ LD  R1, c
           LD  R2, j
           MUL R2, R2, 8
           ST  a(R2), R1
```

```
if x < y goto L  ⇒  LD   R1, x
                    LD   R2, y
                    SUB  R1, R1, R2
                    BLTZ R1, L
```

Basic Block

- *Definition*: A basic block *B* is a sequence of consecutive instructions such that:
  1. control enters *B* only at its beginning;
  2. control leaves *B* at its end (under normal execution); and
  3. control cannot halt or branch out of *B* except at its end.
- This implies that if any instruction in a basic block B is executed, then *all* instructions in B are executed.
  - ⇒ for program analysis purposes, we can treat a basic block as a single entity.

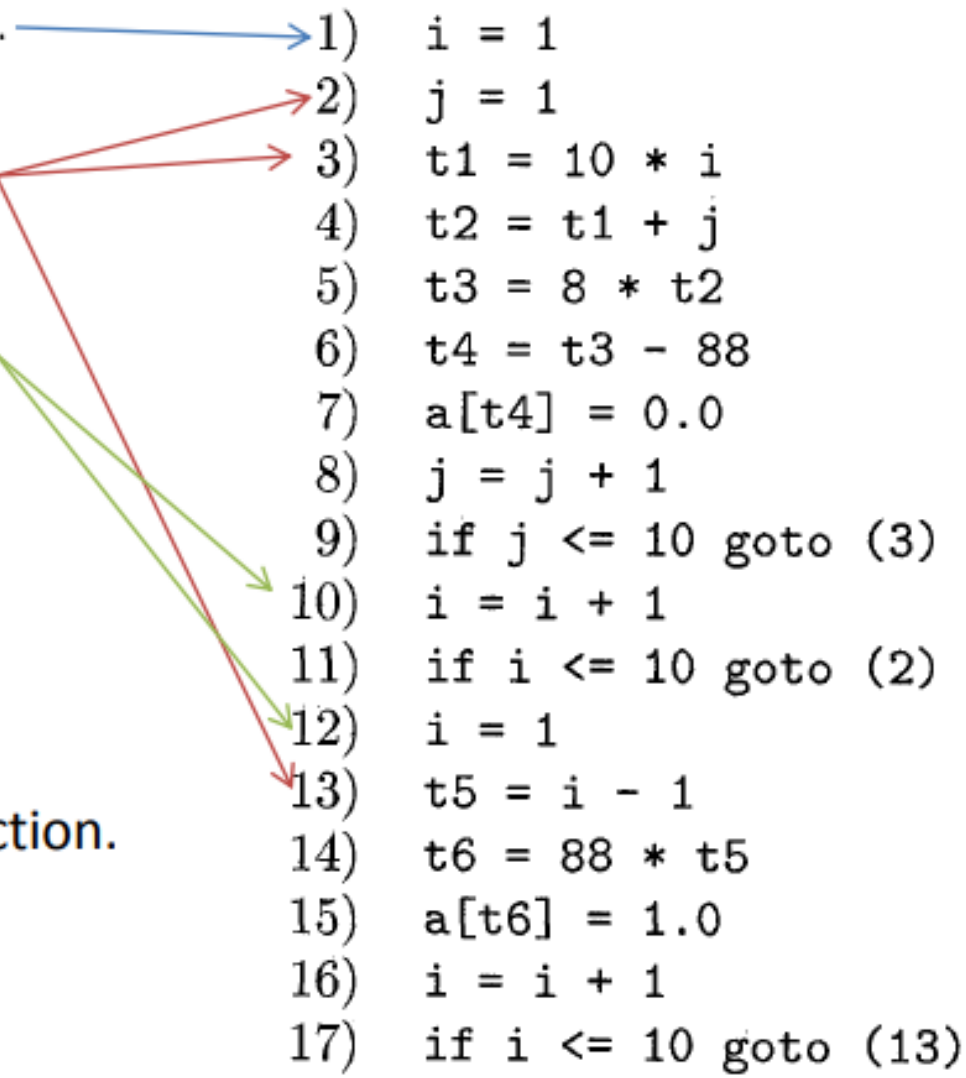
Example: Basic Block

- The following sequence of three-address statements forms a basic block:

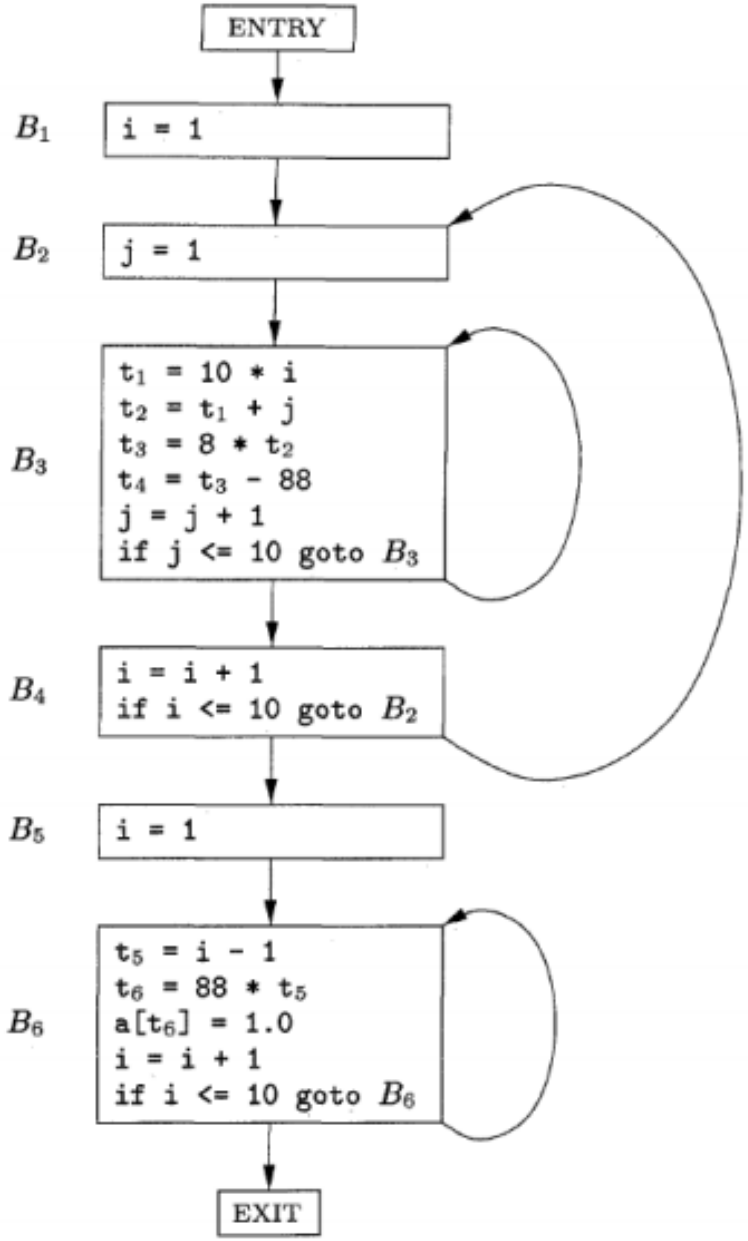
```
t1 : = a * a
t2 : = a * b
t3 : = 2 * t2
t4 : = t1 + t3
t5 : = b * b
t6 : = t4 + t5
```

Fist we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.



Basic block starts with a leader instruction and stops before the following leader instruction.


ENTRY
B_1
i = 1
B_2
j = 1
B_3
t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6
EXIT

Basic Block Construction

1. Determine the set of *leaders*, i.e., the first instruction of each basic block:
   - the entry point of the function is a leader;
   - any instruction that is the target of a branch is a leader;
   - any instruction following a (conditional or unconditional) branch is a leader.
2. For each leader, its basic block consists of:
   - the leader itself;
   - all subsequent instructions up to, but not including, the next leader.

Example: Construct Basic Block

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```

```
     MOV 1,R0
     MOV n,R1
     JMP L2
```

```
L1:  MUL 2,R0
     SUB 1,R1
```

```
L2:  JMPNZ R1,L1
```

Control Flow Graph

- *Definition*: A control flow graph for a function is a directed graph $G = (V, E)$ such that:
  - each $v \in V$ is a basic block; and
  - there is an edge $a \rightarrow b \in E$ iff control can go directly from *a* to *b*.
- *Construction*:
  1. identify the basic blocks of the function;
  2. there is an edge from block *a* to block *b* if:
     i. there is a (conditional or unconditional) branch from the last instruction of *a* to the first instruction of *b*; or
     ii. b immediately follows a in the textual order of the program, and a does not end in an unconditional branch.

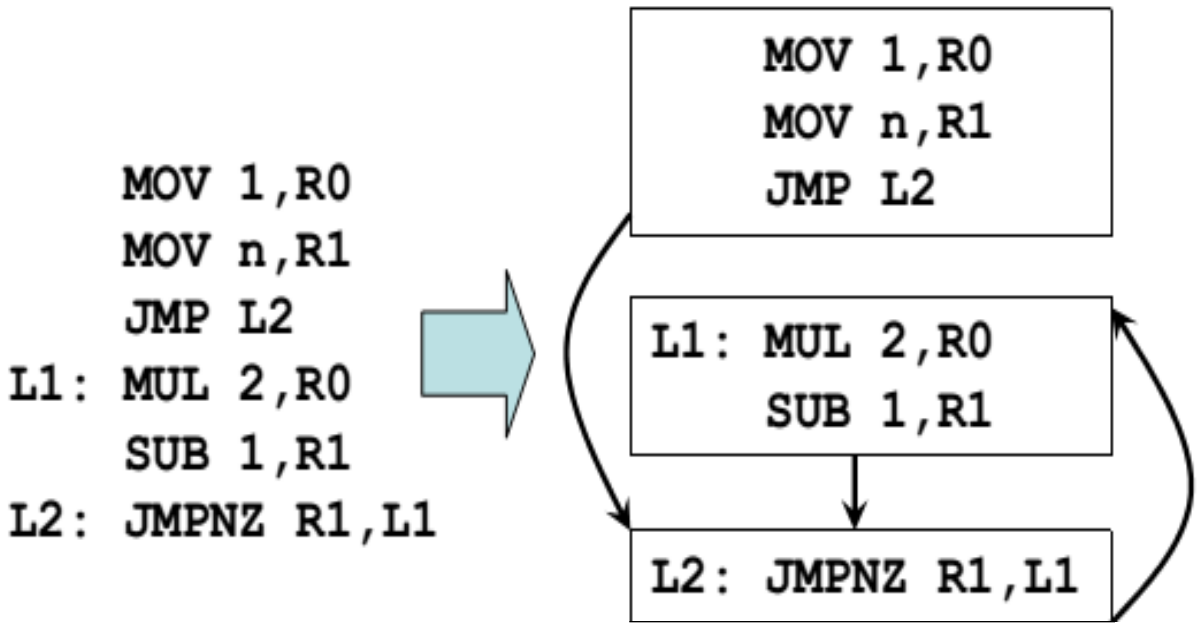


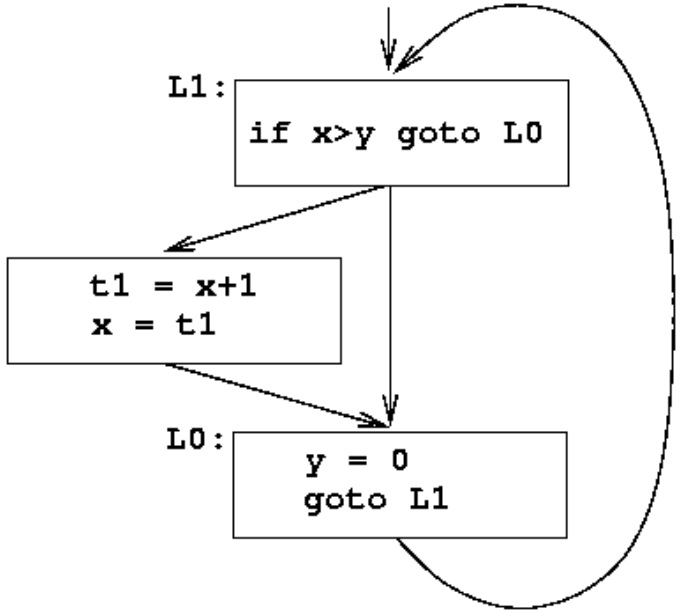
Fig. Flow graph

Basic Blocks and Flow Graphs

- For program analysis and optimization, we need to know the program's control flow behavior.
- For this, we:
  - group three-address instructions into *basic blocks*;
  - represent control flow behavior using *control flow graphs*.

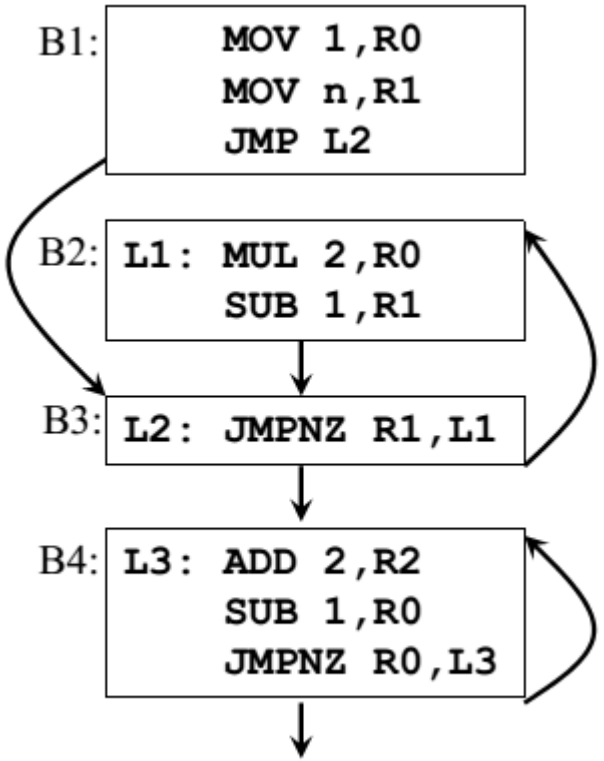
Example:

```
L1: if x > y goto L0
    t1 = x+1
    x = t1
L0: y = 0
    goto L1
```



Loops

- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry
- Virtually every program spends most of its time in executing its loops, it is especially imoportant to generate good code for the loop
- Many code transformations depend upon the identification of the loops in a flow graph.
- Strongly connected components: { B2, B3}, {B4 } , There is a path of length one or more from one node to the another to make a cylcle.
- Entry are: B3 and B4 for the loops.
- A loop that consists of no other loop is called inner loop.



Code Optimization

- Code Optimization phase is mainly used to optimize the code for better utilization of memory and reduce the time taken for execution.
- Code optimization takes input from intermediate code generator and performs machine independent optimization.
- Code optimizer may also take input from code generator and perform machine dependent code optimization.
- Compilers that use code optimization transformations are called as optimizing compilers.
- Code optimization does not consider target machine properties for optimization (like register allocation and memory management) if input is from intermediate code generator.

Optimizations

- Local optimizations
  - Apply to basic Blocks in isolations
- Global Optimizations
  - Apply across basic Blocks
- Peephole Optimizations
  - Apply across boundaries

Optimizing of Basic Block

- We can obtain a substantial improvement in the running time of code merely by performing local optimization within each basic block by itself.
- More thorough global optimization, which looks at how information flows among the basic blocks of a program.
- Some of local optimizations are:
  - Compile time evaluation
  - Common sub-expression elimination
  - Code motion
  - Strength Reduction
  - Dead code elimination
  - Algebraic Transformations

Compile-Time Evaluation

- Expressions whose values can be pre-computed at the compilation time
- Two ways:
  - Constant folding
  - Constant propagation

- **Constant folding**: Evaluation of an expression with constant operands to replace the expression with single value.
  Example:

```
area := (22.0/7.0) * r ^ 2
```

```
area := 3.14286 * r ^ 2
```

Compile-Time Evaluation

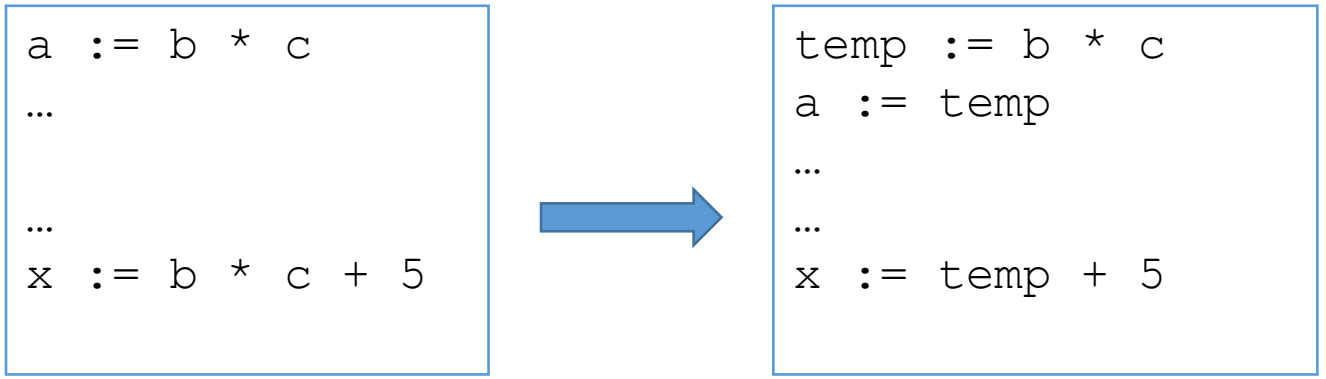
- **Constant Propagation**: Replace a variable with constant which has been assigned to it earlier.
- Example:

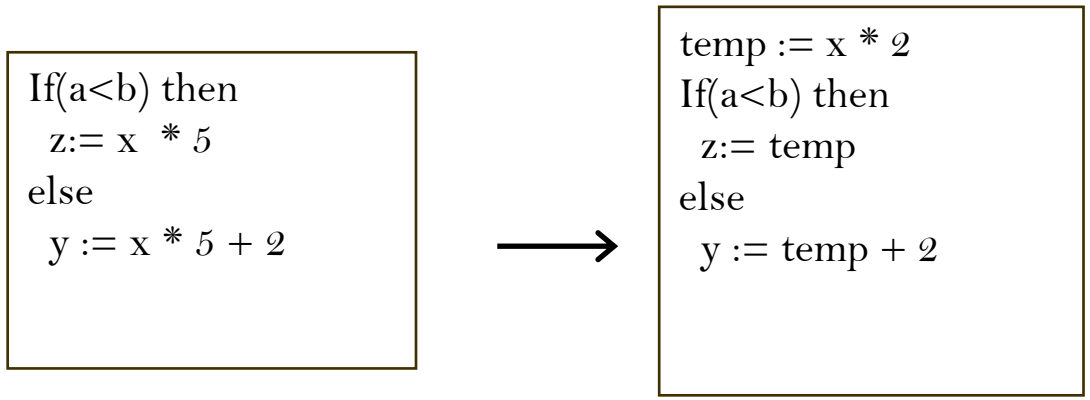
```
pi := 3.14286
area = pi * r ^ 2
```

```
area = 3.14286 * r ^ 2
```

Common Sub-expression Elimination

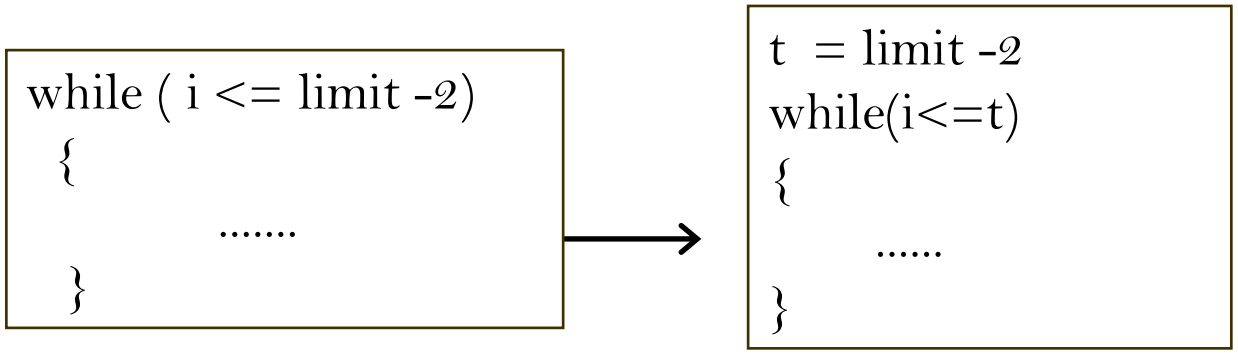
- Local common sub-expression elimination is performed within basic blocks.
- Compute the expression only once and assign it to some temporary





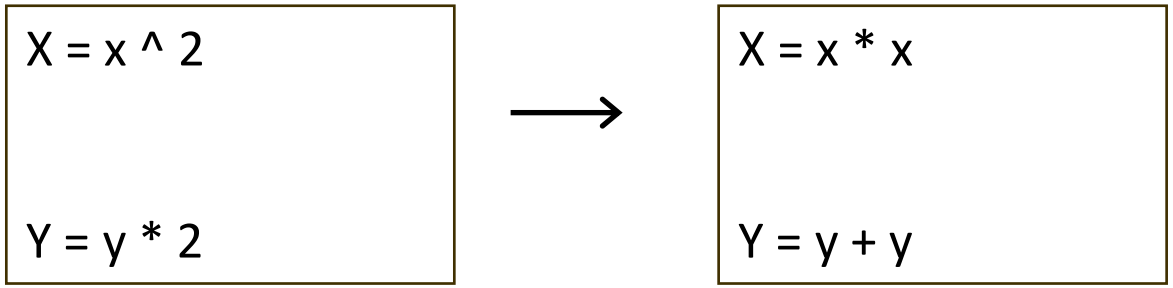
Code Motion

- Moving code from one part of the program to other without modifying the algorithm
  - Reduce size of the program
  - Reduce execution frequency of the code subjected to movement
- This transformation takes an expression that yields the same result independent of the number of times a loop is executed (i.e. loop invariant computation) and evaluates the expression before the loop.
- Similar to common sub-expression elimination but with the objective to reduce code size.



Strength Reduction

- Replacement of an operator with a less costly one.



Dead Code Elimination

- Dead Code are portion of the program which will not be executed in Basic block.

```
If(a==b)
{
      b=c ;
       …..
       return b ;
       c = 0 ;
}
```

```
If(a==b)
{
b=c ;
…..
return b ;
}
```

```
debug = FALSE;
...
if (debug)
   print.....
```

```
debug = FALSE
.................
.................
.............
```

Algebraic Simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1

- Some statements can be simplified

  x := x * 0 ⇒ x := 0

  y := y ** 2 ⇒ y := y * y

Contd…

- It implies that amount of time taken for optimization should be very less when compared to the reduction of overall execution time. Generally, a fast non optimizing compilers are preferred for debugging programs.
- Local Optimization: Consider each basic block by itself. (All compilers.)
- Global Optimization: Consider each procedure by itself. (Mostcompilers.)
- Inter-Procedural Optimization: Consider the control flow between procedures. (A few compilers do this.)

Peephole Optimization

- A simple but effective technique for locally improving the target code is peephole optimization, which examines a short sequence of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence whenever possible.
- Peephole optimization can also be applied directly after intermediate code generation to improve the IR.

contd…

- The peephole is a small, sliding window on a program.
- That is, the “peephole” is a short sequence of (usually contiguous) instructions
  – The optimizer replaces the sequence with another equivalent one (but faster)

  Characteristics of Peephole Optimization:
  – Redundant instruction elimination
  – Flow-of-control optimizations
  – Algebraic simplifications
  – Use of machine idioms

Eliminating Redundant Loads and Stores

- Consider

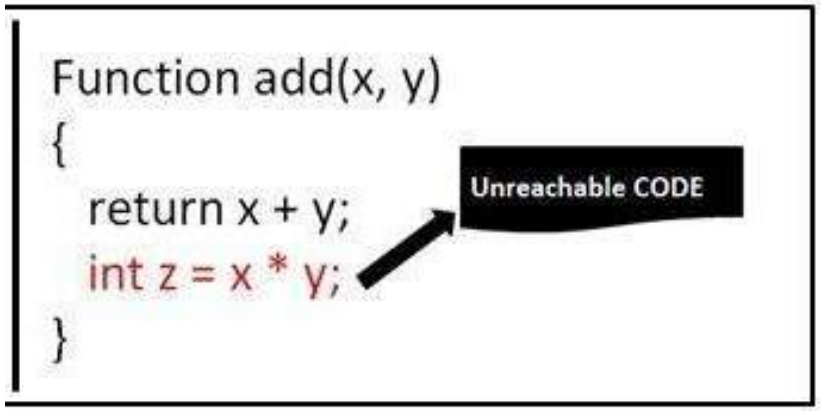
  MOV R0,a

  MOV a,R0

- The second instruction can be deleted because first ensures value of a in R0, but only if it is not labeled with a target label
- Peephole represents sequence of instructions with at most one entry point
- The first instruction can also be deleted if *live*(**a**) = false

Eliminating unreachable code

– Code that is unreachable in the control-flow graph
– Basic blocks that are not the target of any jump or "fall through" from a conditional
– Such basic blocks can be eliminated



Using Machine Idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.
- For example, some machines have auto-increment and auto-decrement addressing modes.
- Using these modes can greatly improve the quality of the code when pushing or popping a stack.
- These modes can also be used for implementing statements like $a = a + 1$.
- Eg. INC a

Algebraic Simplifications

If statements like:

```
a = a + 0
a  =a * 1
```

are generated in the code, they can be eliminated, because zero is an additive identity, and one is a multiplicative identity.

Code hoisting

- Moving computations outside loops
- Saves computing time
- In the following example (2.0 * PI) is an invariant expression there is no reason to recompute it 100 times.

```
DO I = 1, 100
        ARRAY(I) = 2.0 * PI * I
ENDDO
```

- By introducing a temporary variable 't' it can be transformed to:

```
t = 2.0 * PI
DO I = 1, 100
        ARRAY(I) = t * I
END DO
```

Dead store elimination

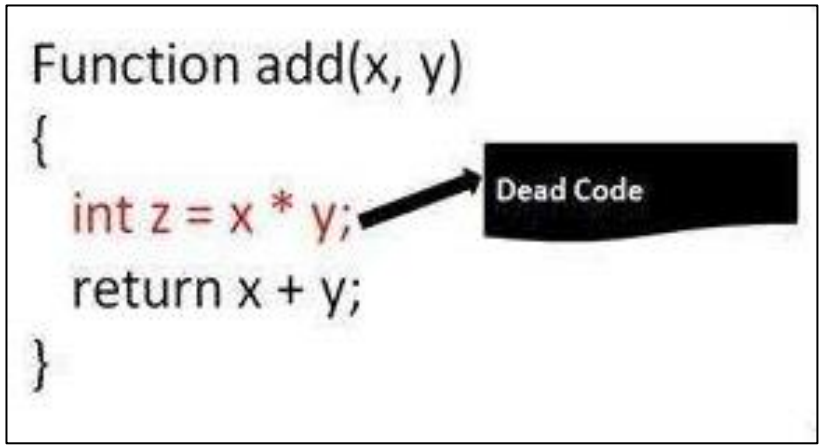
- If the compiler detects variables that are never used, it may safely ignore many of the operations that compute their values.

- Dead code is code that is never executed or that does nothing useful. May appear from copy propagation:

```
T1 := k
...

x := x + T1
y := x - T1
...
```

```
...
 x := x + k
y := x - k
...
```



Eliminating common sub-expressions

- Optimization compilers are able to perform quite well:

```
X = A * LOG(Y) + (LOG(Y) ** 2)
```

- Introduce an explicit temporary variable t:

```
t = LOG(Y)
X = A * t + (t ** 2)
```

- Saves one 'heavy' function call, by an elimination of the common sub-expression LOG(Y), the exponentiation now is:

```
X = (A + t) * t
```

Flow-of-control optimizations

Explain Flow-of-control optimizations.

```
goto L1
...
Ll: goto L2
```

Can be replaced by:

```
goto L2
...
Ll: goto L2
```

```
if a<b goto L1
...
Ll: goto L2
```

Can be replaced by:

```
if a<b goto L2
...
Ll: goto L2
```

Code Motion

- Any code inside a loop that always computes the same value can be moved before the loop.
- Example:

```
while (i <= limit-2)
do {loop code}
```

where the loop code doesn't change the limit variable. The subtraction, limit-2, will be inside the loop. Code motion would substitute:

```
t = limit-2;
while (i <= t)
do {loop code}
```

Thank you

Some Important Questions

Explain the principle sources of code optimization with example.

- Common sub expression elimination
- Dead code elimination
- Loop optimization
- Copy propagation
- Constant folding

Eliminating common sub-expressions

- Optimization compilers are able to perform quite well:

  X = A * LOG(Y) + (LOG(Y) ** 2)

- Introduce an explicit temporary variable t:

  t = LOG(Y)

  X = A * t + (t ** 2)

- Saves one 'heavy' function call, by an elimination of the common sub-expression LOG(Y), the exponentiation now is:

  X = (A + t) * t

Dead store elimination

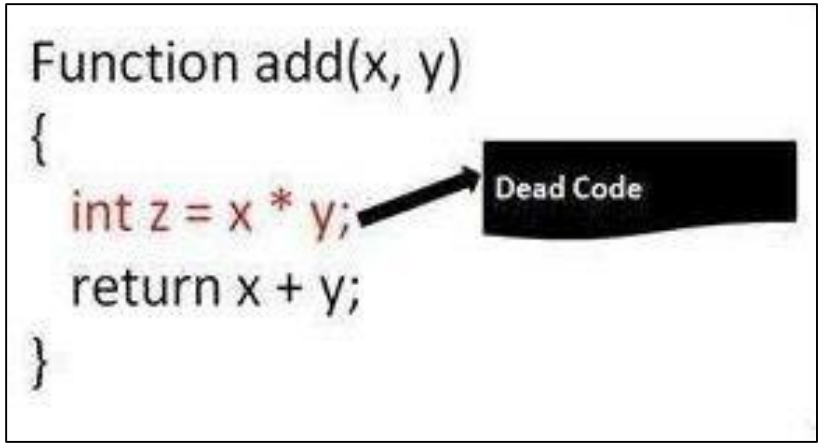
- If the compiler detects variables that are never used, it may safely ignore many of the operations that compute their values.

- Dead code is code that is never executed or that does nothing useful. May appear from copy propagation:

```
T1 := k
...

x := x + T1
y := x - T1
...
```

```
...
 x := x + k
y := x - k
...
```



Copy propagation

Before

```
x = y
z = 1 + x
```

Has data dependency

Compile

After

```
x = y
z = 1 + y
```

No data dependency

Constant Folding

Before

```
add = 100
aug = 200
sum = add + aug
```

After

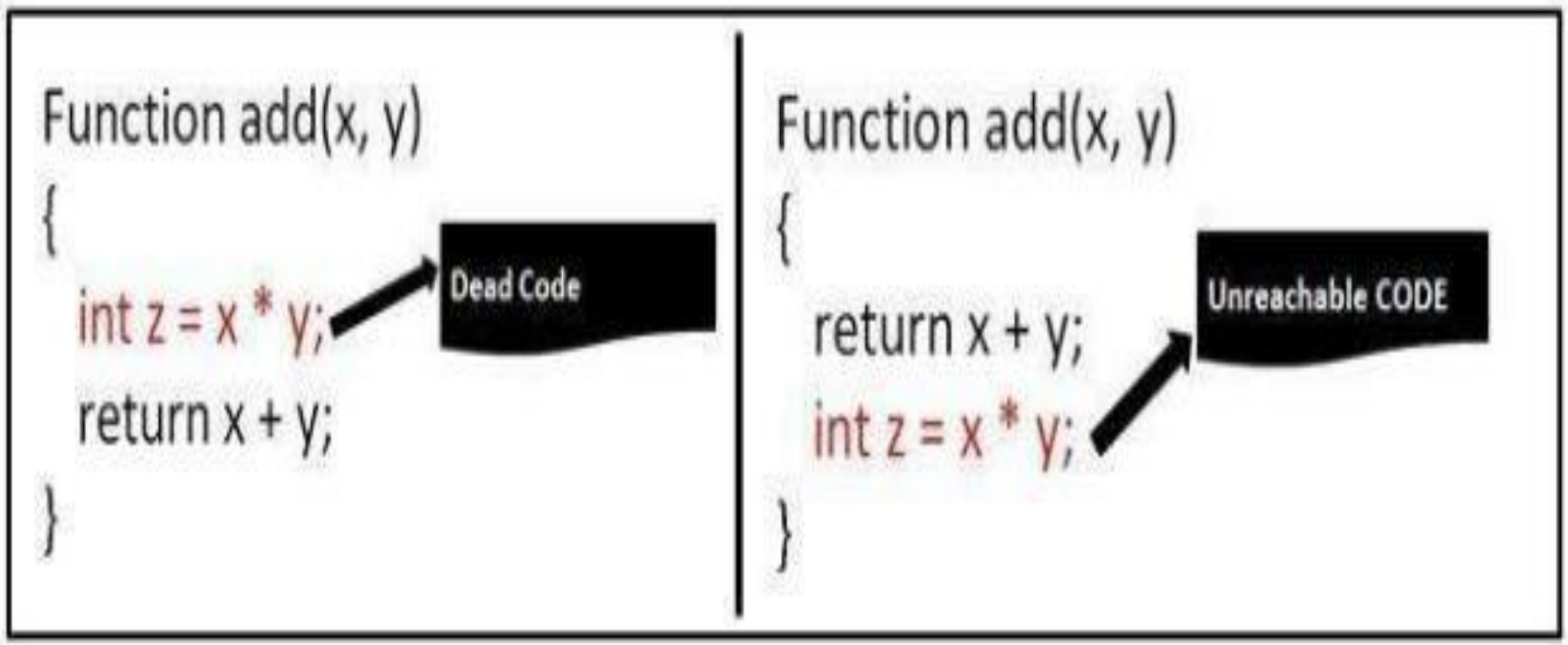
```
sum = 300
```

Notice that **sum** is actually the sum of two constants, so the compiler can precalculate it, eliminating the addition that otherwise would be performed at runtime.

Explain the unreachable code optimization.

- Dead code elimination
  - Unreachable code elimination
  - Redundant statement

- Unreachable code, a part of the source code that will never be executed due to inappropriate exit points/control flow.
- The other kind of unreachable code is referred as dead code, although dead code might get executed but has no effect on the functionality of the system.


Function add(x, y)
{
int z = x * y;
return x + y;
}
Dead Code
Function add(x, y)
{
return x + y;
int z = x * y;
}
Unreachable CODE

Characteristics of Peephole Optimization:

– Redundant instruction elimination
– Flow-of-control optimizations
– Algebraic simplifications
– Use of machine idioms