Code Optimization

Code Optimization

- **Why**
  - Reduce programmers' burden
    - Allow programmers to concentrate on high level concept
    - Without worrying about performance issues
- **Target**
  - Reduce execution time
  - Reduce space
  - Sometimes, these are tradeoffs
- **Types**
  - Intermediate code level
  - Assembly level
    - Instruction selection, register allocation, etc.

Code Optimization

- **Scope**
  - Peephole analysis
    - Within one or a few instructions
  - Local analysis
    - Within a basic block
  - Global analysis
    - Entire procedure or within a certain scope
  - Inter-procedural analysis
    - Beyond a procedure, consider the entire program

Goal is to optimize (minimize)

- Time
  - Runtime
  - Response time
- Space
  - Secondary storage
  - Main memory

Contd..

- Optimization strategies
  - Cost measured in effort, time, risk
  - Tradeoffs often involved
    - Space vs. time
    - Increased complexity
- Optimization difficult because
  - Many strategies, tradeoffs
  - No general algorithm
- Focus--must decide
  - What to optimize--space, time
  - Where to optimize

Platform dependent/ Platform independent optimization

- Code optimization can be also broadly categorized as

- **Platform-dependent** techniques use specific properties of one platform, or rely on parameters depending on the single platform or even on the single processor. Platform dependent techniques involve ***instruction scheduling***, ***instruction-level parallelism, data-level parallelism, cache optimization techniques*** (i.e. parameters that differ among various platforms) and the optimal instruction scheduling might be different even on different processors of the same architecture.
- In **Platform independent** writing or producing different versions of the same code for different processors might be needed. In the case of ***compile-level optimization***, platform-independent techniques are ***generic techniques*** such as ***loop unrolling, reduction in function calls, memory efficient routines, reduction in conditions,*** etc. Compile level optimization impact most CPU architectures in a similar way. Compile level optimization serve to reduce the total Instruction path length required to complete the program and/or reduce total memory usage during the process

Levels of Optimization

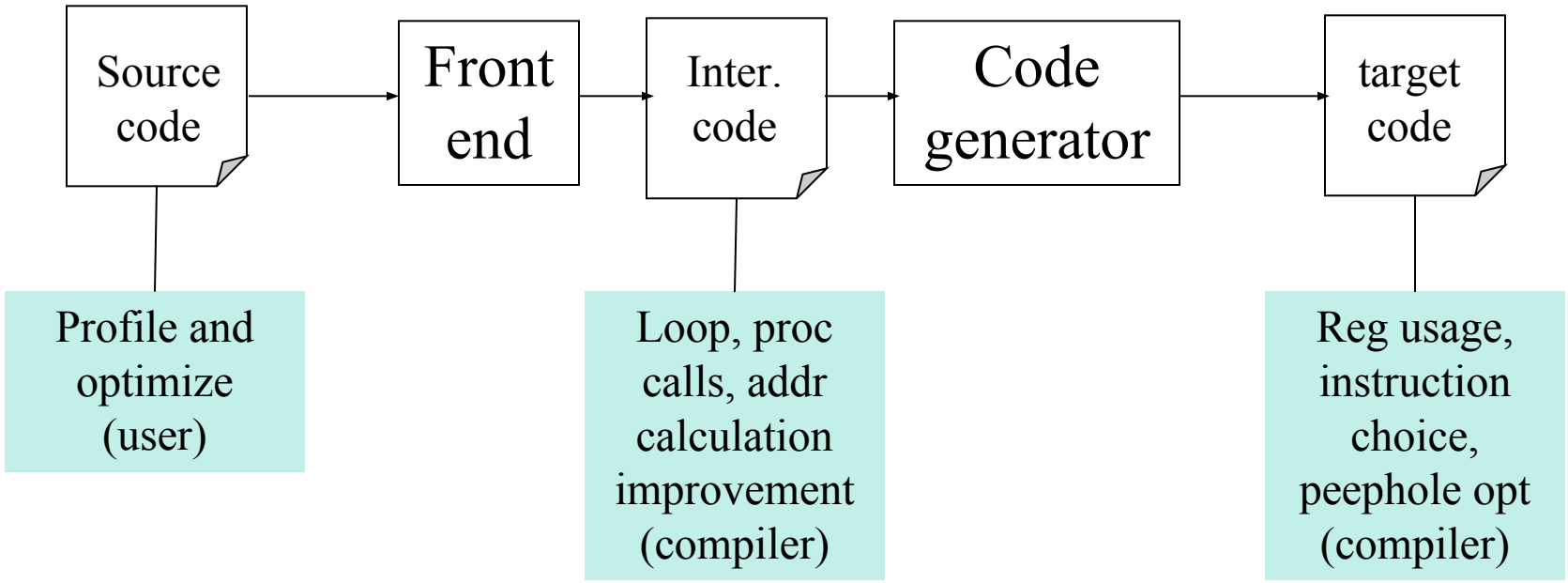
- **Design level:**
  - Architectural design of a system affects its performance. At the highest level, the design may be optimized to make best use of the available resources.
- **Source code level:**
  - avoid poor quality coding can also improve performance.
- **Assembly level**
  - At the lowest level, writing code using an assembly language, designed for a particular hardware platform will normally produce the most efficient code.
- **Run time:** Just in time compilers and Assembler programmers may be able to perform run time optimization exceeding the capability of static compilers by dynamically adjusting parameters according to the actual input or other factors.

Levels of Optimization contd..

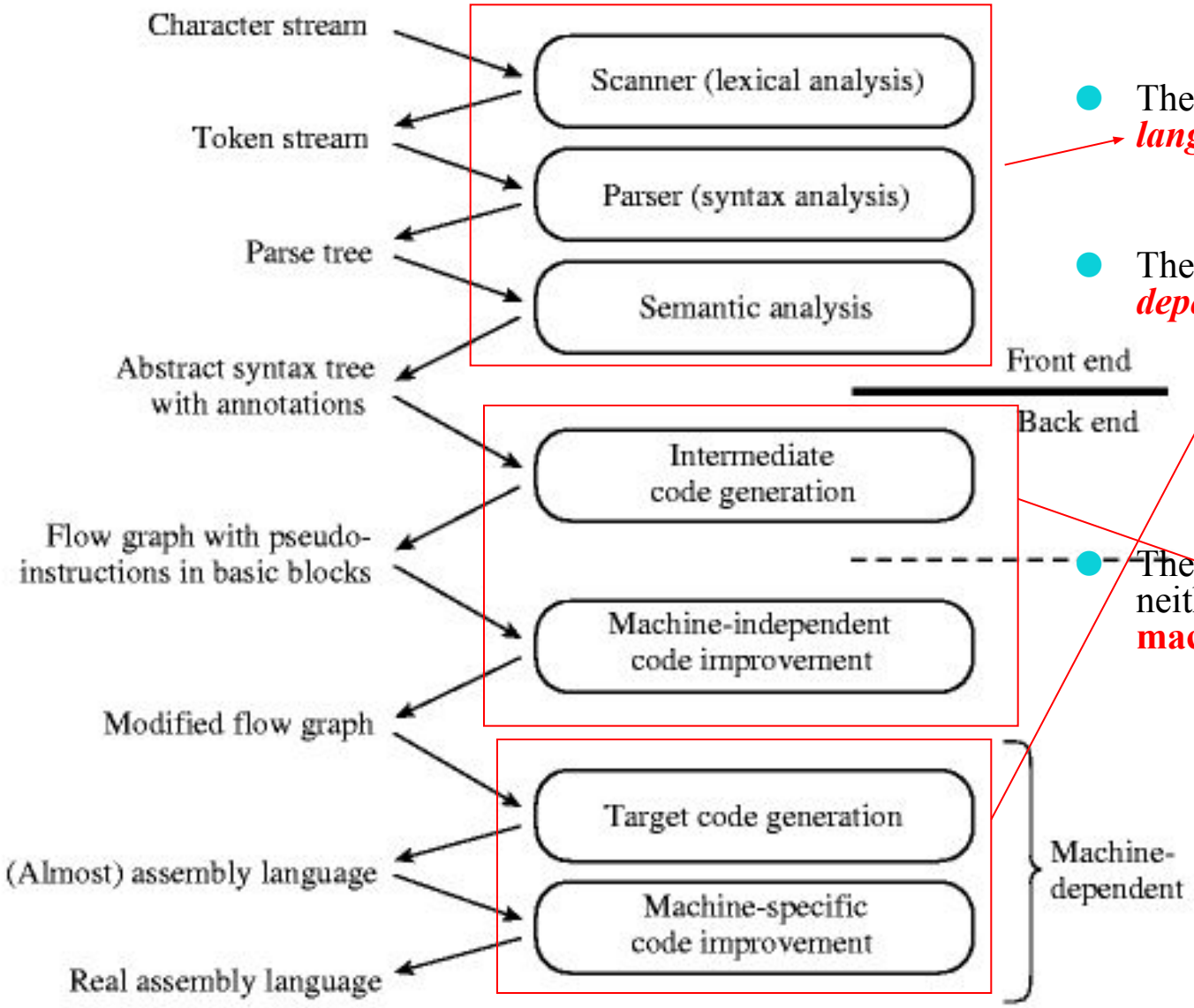
- **Compile level:**
  - Compilers try to generate good code. I.e. Fast
  - Code improvement is challenging
  - Code improvement may slow down the compilation process. In some domains, such as just-in-time compilation, compilation speed is critical.

Levels of Optimization

- Optimization can be done in almost all phases of compilation.

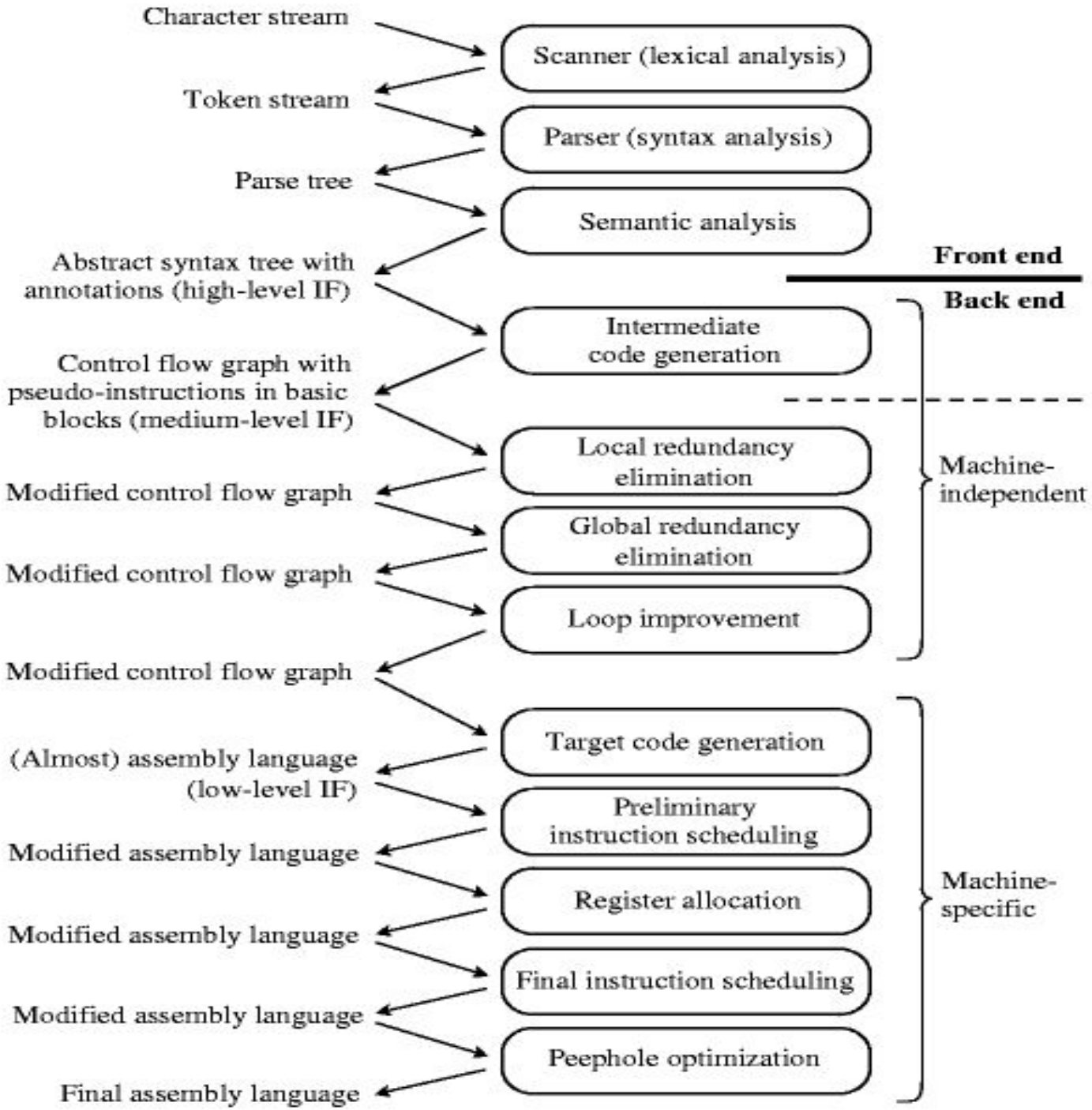


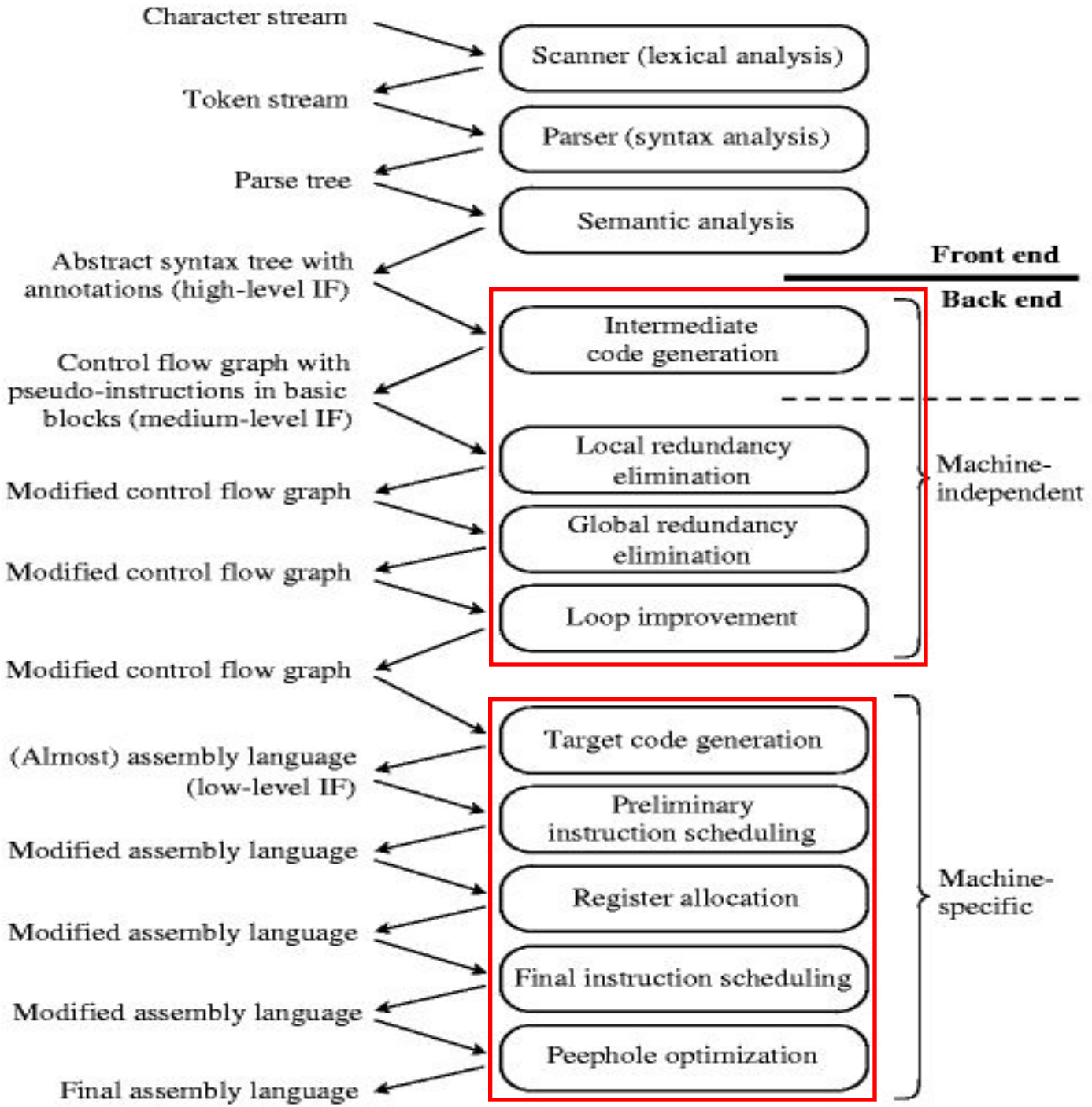
Phases of Compilation



- The first three phases are ***language-dependent***
- The last two are ***machine dependent***
- The middle two dependent on neither the **language nor the machine**

- **Generating highly optimized is a complicated process**
- **We will concentrate on execution speed optimization**




Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Front end
Abstract syntax tree with annotations (high-level IF)
Back end
Intermediate code generation
Control flow graph with pseudo-instructions in basic blocks (medium-level IF)
Local redundancy elimination
Machine-independent
Modified control flow graph
Global redundancy elimination
Modified control flow graph
Loop improvement
Modified control flow graph
Target code generation
(Almost) assembly language (low-level IF)
Preliminary instruction scheduling
Modified assembly language
Register allocation
Machine-specific
Modified assembly language
Final instruction scheduling
Modified assembly language
Peephole optimization
Final assembly language

When to optimize

- When the code doesn't give adequate time/space performance:
  - Early days of computing
  - Early days of desktop computers
  - Specialized computers (military, space)
  - Ubiquitous computing (as tiny computers are embedded in credit cards and other objects)
- When complexity can be reduced
  - If performance is not an issue, aim for clarity, simplicity, maintainability
  - Review your first-cut code to see if it can be transformed into something simpler

Design Levels

Six Design Levels (Reference: Jon Bentley, **Writing Efficient Programs**, Prentice-Hall, 1982)

1. Program design (System structure) -- decomposition into modules
2. Module and routine design (Intramodular structure) -- choice of data structures and algorithms
3. **Code tuning (Writing efficient code) -- source to source transformations**
4. Code compilation (Translation into machine code) -- compiler may outperform human
5. Operating system interaction (System software) -- changing, tuning, bypassing operating system (OS) or database system (DBMS)
6. Hardware -- modify or purchase: microcode, faster CPU, DB machine, array processor, floating point hardware, ASICs (Application Specific Integrated Circuits)

- **Strategy -- optimize where change is possible and payoff is highest**

Methodology

- Design and implement for correctness and clarity
  - Include useful documentation
  - Use modularity for maintenance
  - Identity performance goals, and factor these down to individual modules
- **If** performance unacceptable **then**
  - Monitor program to see where bottlenecks (time, space) are -- instrument code or apply tools
  - Revise data structures and algorithms in critical modules
  - Consider redesign
  - Consider solution by purchase (optimizing compiler, faster DBMS or OS, faster hardware)

Contd..

- **If** performance still not OK **then**
  - Apply source-to-source transformations
  - retain original code as documentation
- **If** still not OK **then**
  - Try lower level designs:
    - Recode critical modules in assembly language
    - Modify/tune/bypass OS or DBMS

Criterion of code optimization

- Must preserve the semantic equivalence of the programs.
- The algorithm should not be modified.
- Transformation, on average should speed up the execution of the program.
- Worth the effort: Intellectual and compilation effort spend on insignificant improvement..
- Transformations are simple enough to have a good effect

Themes behind Optimization Techniques

- **Avoid redundancy:** something already computed need not be computed again
- **Smaller code:** less work for CPU, cache, and memory!
- **Less jumps:** jumps interfere with code pre-fetch
- **Code locality:** codes executed close together in time is generated close together in memory – increase locality of reference
- **Extract more information about code**: More info – better code generation

Redundancy elimination

- Redundancy elimination = determining that two computations are equivalent and eliminating one.
- There are several types of redundancy elimination:
  - Value numbering
    - Associates symbolic values to computations and identifies expressions that have the same value
  - Common subexpression elimination
    - Identifies expressions that have operands with the same name
  - Constant/Copy propagation
    - Identifies variables that have constant/copy values and uses the constants/copies in place of the variables.
  - Partial redundancy elimination
    - Inserts computations in paths to convert partial redundancy to full redundancy.

Peephole Optimization

- Simple compiler do not perform machine-independent code improvement
  - They generates *naïve* code
- It is possible to take the target hole and optimize it
  - Sub-optimal sequences of instructions that match an optimization pattern are transformed into optimal sequences of instructions
  - This technique is known as ***peephole optimization***
  - Peephole optimization usually works by sliding a window of several instructions (a *peephole*)

Optimizing Transformations
(Basic peephole techniques)

- Compile time evaluation
- Common sub-expression elimination
- Code motion
- Strength Reduction
- Dead code elimination
- Copy propagation
- Loop optimization
  - Induction variables and strength reduction

Compile-Time Evaluation

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

Compile-Time Evaluation

- **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`

  - Requirement:
    - After a constant assignment to the variable
    - Until next assignment of the variable
  - Need data flow analysis to exam the propagation

Constant Propagation

- What does it mean?
  - Given an assignment x = c, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.
    - Similar to copy propagation
    - Extra feature: It can analyze constant-value conditionals to determine whether a branch should be executed or not.
- When is it performed?
  - Early in the optimization process.
- What is the result?
  - Smaller code
  - Fewer registers

Common Sub-expression Evaluation

- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
  - The *definition* of the variables involved should not change

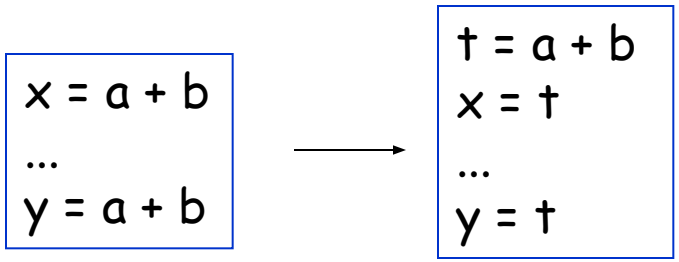
Example:

```
a := b * c          temp := b * c
…                   a := temp
…                   …
x := b * c + 5      x := temp + 5
```

  - Reduce the copying
  - If y is reassigned in between, then this action cannot be performed

Common Subexpression Elimination

- Local common sub expression elimination
  - Performed within basic blocks
  - Algorithm sketch:
    - Traverse BB from top to bottom
    - Maintain table of expressions evaluated so far
      - if any operand of the expression is redefined, remove it from the table
    - Modify applicable instructions as you go
      - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.



```
a := b + c          a := b + c
c := b + c     ⇒    c := a
d := b + c          d := b + c
```

Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to L
```

```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /* commutative */
a = j + a
k = t2
j = t3
a = -b
if t2 go to L
```

the table contains quintuples:
(pos, opd1, opr, opd2, tmp)

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

Code Motion

1. *Code Space reduction*: Similar to common sub-expression elimination but with the objective to reduce code size.

Example: Code hoisting

```
                          temp : = x ** 2
if (a< b) then            if (a< b) then
    z := x ** 2                   z := temp
else                      else  ⟶
    y := x ** 2 + 10                    y := temp + 10
```

"x ** 2" is computed once in both cases, but the code size in the second case reduces.

Code Motion

2 *Execution frequency reduction*: reduce execution frequency of partially available expressions (expressions available atleast in one path)

Example:

```
if (a<b) then          temp = x * 2
    z = x * 2          if (a<b) then
else                       z = temp
y = 10                 else
g = x * 2                  y = 10
                       g = temp;
```

Code Motion

- Move expression out of a loop if the evaluation does not change inside the loop.

Example:

```
while ( i < (max-2) ) ...
```

Equivalent to:

```
t :=  max - 2
while ( i < t ) ...
```

Code Motion

- Safety of Code movement

  Movement of an expression e from a basic block b_i to another block b_j, is safe if it does not introduce any new occurrence of e along any path.

Example: Unsafe code movement

```
                    temp = x * 2
if (a<b) then       if (a<b) then
   z = x * 2           z = temp
else                   else
   y = 10              y = 10
```

Strength Reduction

- Replacement of an operator with a less costly one.

Example:

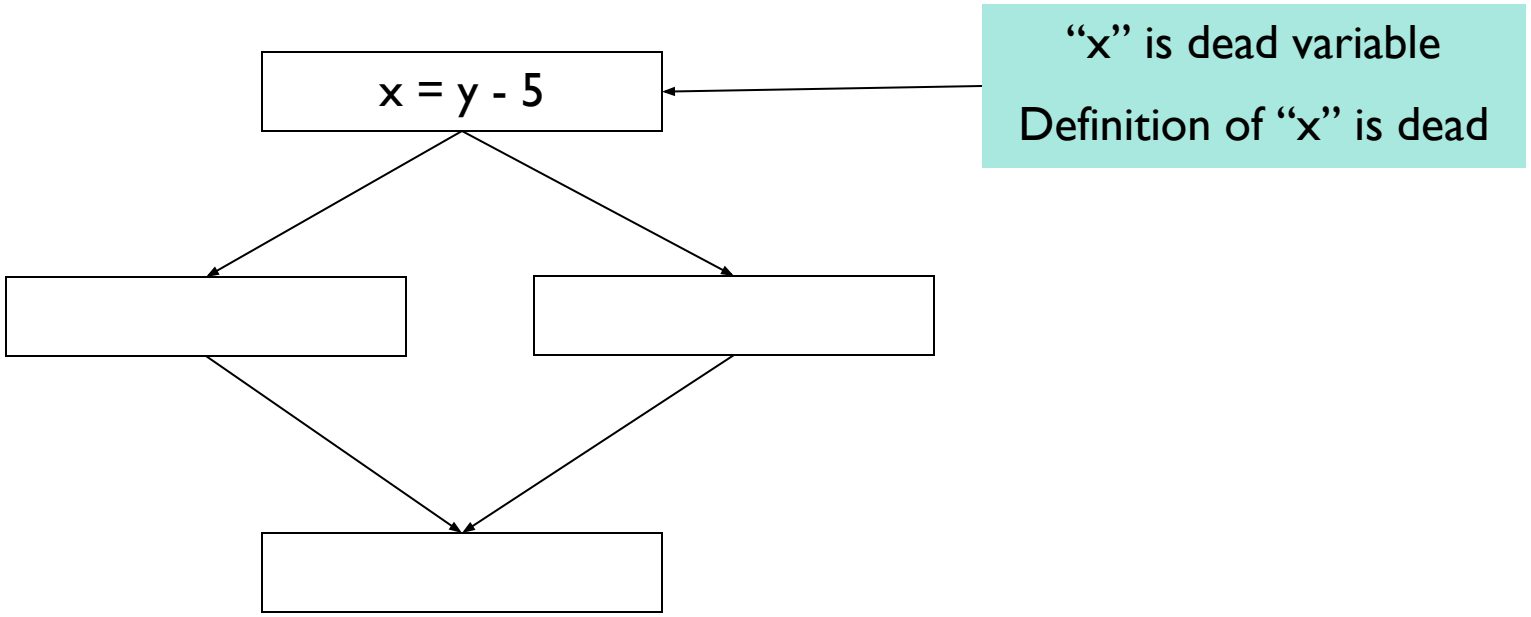
```
                        temp = 5;
for i=1 to 10 do                    for i=1 to 10 do
 …                        …
 x = i * 5                  x = temp
 …                        …
                             temp = temp + 5
 end                       end
```

- Typical cases of strength reduction occurs in address calculation of array references.
- Applies to integer expressions involving induction variables (loop optimization)

Dead Code Elimination

- Dead Code are portion of the program which will not be executed in any path of the program.
  - Can be removed
- Examples:
  - No control flows into a basic block
  - A variable is dead at a point -> its value is not used anywhere in the program
  - An assignment is dead -> assignment assigns a value to a dead variable

Dead Code Elimination



- Beware of side effects in code during dead code elimination

Copy Propagation

- What does it mean?
  - Given an assignment x = y, replace later uses of x with uses of y, provided there are no intervening assignments to x or y.
- When is it performed?
  - At any level, but usually early in the optimization process.
- What is the result?
  - Smaller code

Copy Propagation

- `f := g` are called copy statements or copies
- Use of `g` for `f`, whenever possible after copy statement

Example:

x[i] = a;	x[i] = a;
sum = x[i] + a;	sum = a + a;

- May not appear to be code improvement, but opens up scope for other optimizations.

Loop Optimization

- Decrease the number if instruction in the inner loop
- Even if we increase no of instructions in the outer loop
  - Consumes 90% of the execution time

    $\Rightarrow$ a larger payoff to optimize the code within a loop
- Techniques:
  - Loop invariant detection and code motion
  - Induction variable elimination
  - Strength reduction in loops
  - Loop unrolling
  - Loop peeling
  - Loop fusion

Code Optimization Techniques

- Loop invariant detection and code motion
  - If the result of a statement or expression does not change within a loop, and it has no external side-effect
  - Computation can be moved to the outside of the loop
  - Example

    for (i=0; i<n; i++) a[i] := a[i] + x/y;

    ⇒ for (i=0; i<n; i++) { c := x/y; a[i] := a[i] + c; } // three address code

    ⇒ c := x/y; for (i=0; i<n; i++) a[i] := a[i] + c;
- Induction variable elimination
  - If there are multiple induction variables in a loop, can eliminate the ones which are used only in the test condition
  - Example

    s := 0; for (i=0; i<n; i++) { s := s + 4; }

    ⇒ s := 0; while (s < 4*n) { s := s + 4; }

Code Optimization Techniques

- Strength reduction in loops
  - Example

    s := 0; for (i=0; i<n; i++) { v := 4 * i; s := s + v;)

    ⇒ s := 0; for (i=0; i<n; i++) { v := v + 4; s := s + v;)
- Loop unrolling
  - Execute loop body multiple times at each iteration
  - Get rid of the conditional branches, if possible
  - Allow optimization to cross multiple iterations of the loop
    - Especially for parallel instruction execution
  - Space time tradeoff
    - Increase in code size, reduce some instructions
- Loop peeling
  - Similar to unrolling
  - But unroll the first and/or last iterations

Code Optimization Techniques

- Loop fusion
  - Example

```
for i=1 to N do
    A[i] = B[i] + 1
endfor
for i=1 to N do
    C[i] = A[i] / 2
endfor
for i=1 to N do
    D[i] = 1 / C[i+1]
endfor
```

Before Loop Fusion

```
for i=1 to N do
    A[i] = B[i] + 1
    C[i] = A[i] / 2
    D[i] = 1 / C[i+1]
endfor
```

Is this correct?
Actually, cannot fuse the third loop

Peephole Optimization

- Peephole optimization is very fast
  - Small overhead per instruction since they use a small, fixed-size window

- It is often easier to generate inexperienced code and run peephole optimization than generating good code!

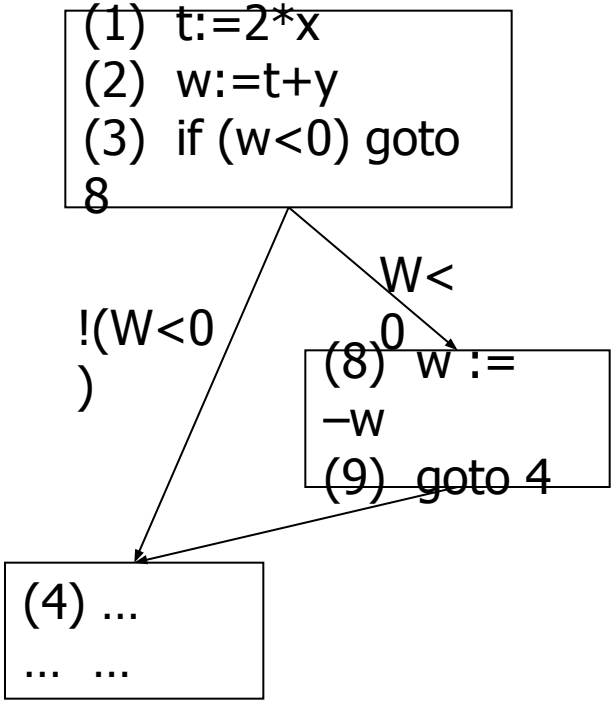
Code Optimization

- Peephole and local analysis generally involve
  - Constant folding
  - Algebraic simplification, strength reduction
- Global analysis generally requires
  - Control flow graph
  - Data flow analysis
  - Loop optimization

Control Flow Graph

- CFG
  - Is a directed graph
  - Nodes are basic blocks
    - A sequence of statements
    - No control flow within a basic block
  - Edges represent execution flow
- Example

(1) t := 2*x;
(2) w := t+y;
(3) if (w<0) goto 8
(4) …
… …
(8) w := –w
(9) goto 4



Construct CFG from Three Address Code

- Identify leader statements
  - First statement of the program
  - Target of any branch statement (conditional or unconditional)
  - Statement that immedia follows a branch or retu statement
- Construct blocks
  - Append the subsequent instructions, up to the statement before the next leader

Easy to know that there is a branch
How about a loop?

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
(13)  ...
```

Finding a Loop in a CFG

- To facilitate loop optimization
- How to find a loop in a CFG?
  - Need to define dominance relation
  - Need to find a back edge of the loop
  - Based on the back edge, find the loop body
- Identify loop at the high level
  - Loop can also be recorded during the intermediate code generation phase
  - But as long as a language allows goto statements, it is necessary to do loop identification

Dominate Relation

- Dominate relation
  - In a single entry CFG, a node a dominates a node b if every path from the start node s to node b goes through a
  - Expressed as $a \leq b$
  - $a < b \Rightarrow a \leq b$ and $a \neq b$
  - Dominate relation is always acyclic
- Dominator set of node b
  - The set of all nodes that dominate b, expressed as dom(b)
  - By definition, $b \in$ dom(b)
    - Each node dominates itself
- Immediate dominator
  - a immediately dominates b if $a < b$ and there is no c such that $a < c < b$
  - Each node, besides the entry node, has a unique immediate dominator

Dominate Relation

- Dominator Sets:

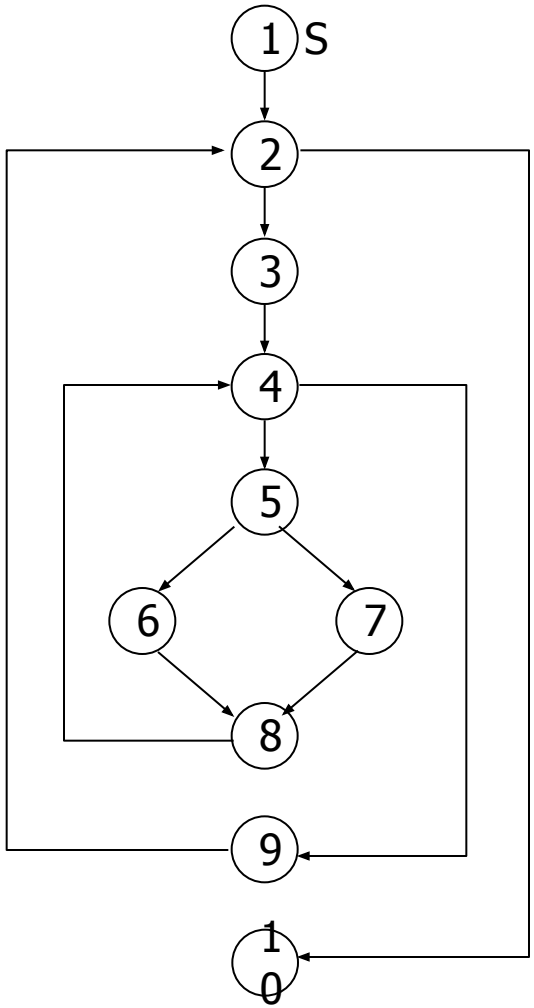
  dom(1) = {1}

  dom(2) = {1, 2}

  dom(3) = {1, 2, 3}

  dom(4) = {1, 2, 3, 4}

- Immediate dominator
  - 1 immediate dominates 2
  - 2 immediate dominates 3, 10

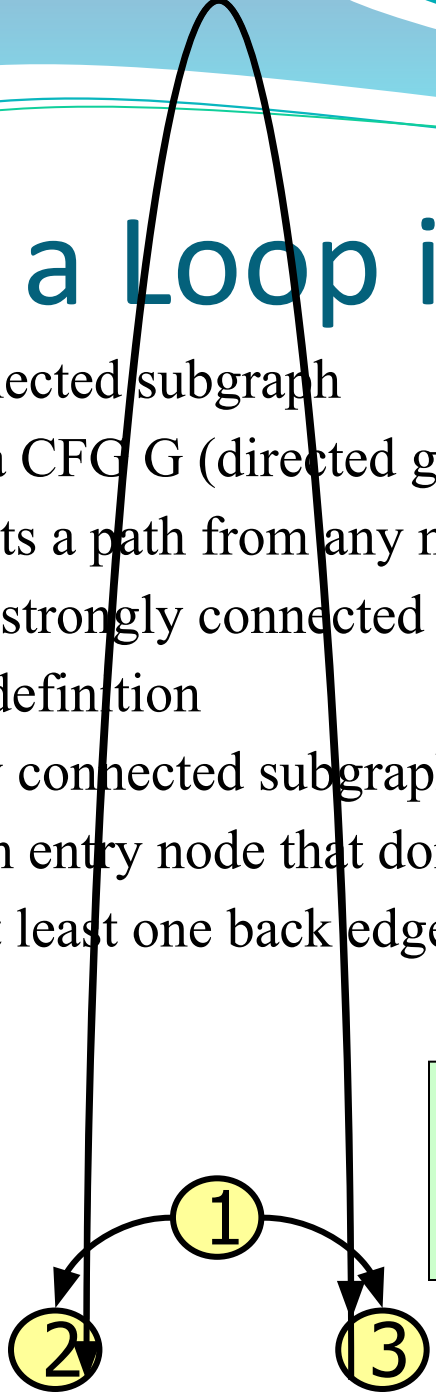


dom(5) = {1, 2, 3, 4, 5}
dom(6) = {1, 2, 3, 4, 5, 6}
dom(7) = {1, 2, 3, 4, 5, 7}
dom(8) = {1, 2, 3, 4, 5, 8}
dom(9) = {1, 2, 3, 4, 9}
dom(10) = {1, 2, 10}

3 imm-dom 4
4 imm-dom 5, 9
5 imm-dom 6, 7, 8

Finding a Loop in a CFG

- Strongly connected subgraph
  - Consider a CFG G (directed graph)
  - There exists a path from any node in G to any other node
  - Then G is strongly connected
- Natural loop definition
  - A strongly connected subgraph in a CFG
  - There is an entry node that dominates all other nodes in the subgraph
  - There is at least one back edge to allow iteration



No, nodes 2 and 3 form a strongly connected subgraph, but there is no entry node that dominates all other nodes.

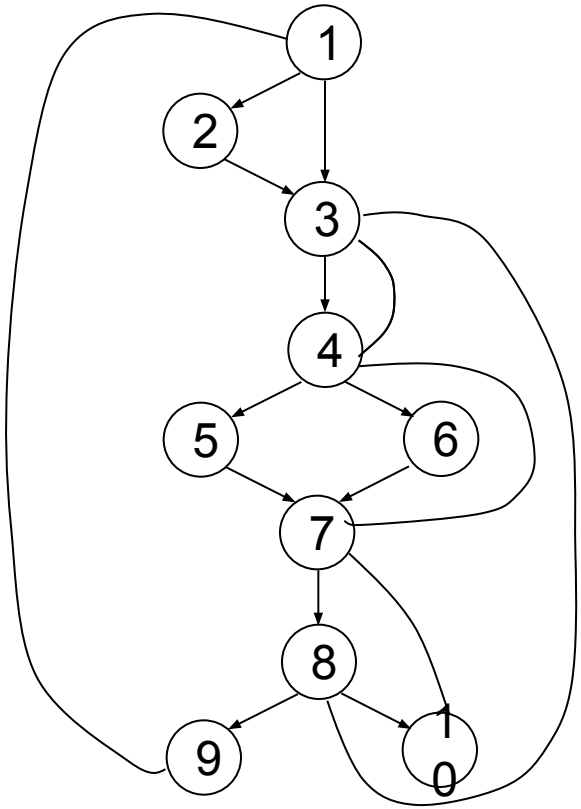
Do nodes 2 and 3 form a loop?

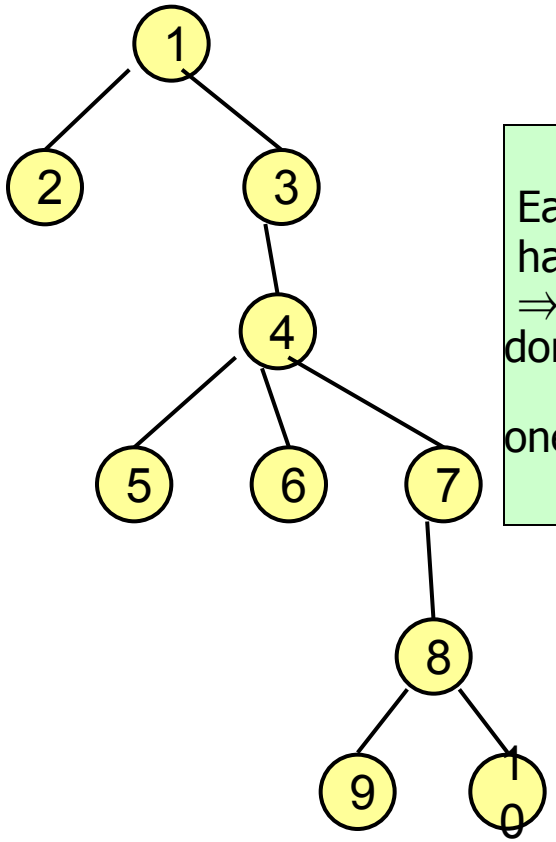
Finding a Loop in a CFG

- How to find natural loops in a CFG?
- Build the dominator tree
  - Starting node is the root
  - A node dominates its descendants
- Find the back edges
  - There exists an edge (b, a) in the CFG
  - a dominates b
  - Then, edge (b, a) is a back edge
- Identify the nodes in a natural loop associated with a back edge (b, a), including all nodes:
  - Dominated by a (a is the entry of the loop)
  - can reach b without going through a

Finding a Loop in a CFG -- Example

- Build the dominator tree
  - Follow the immediate dominate relation

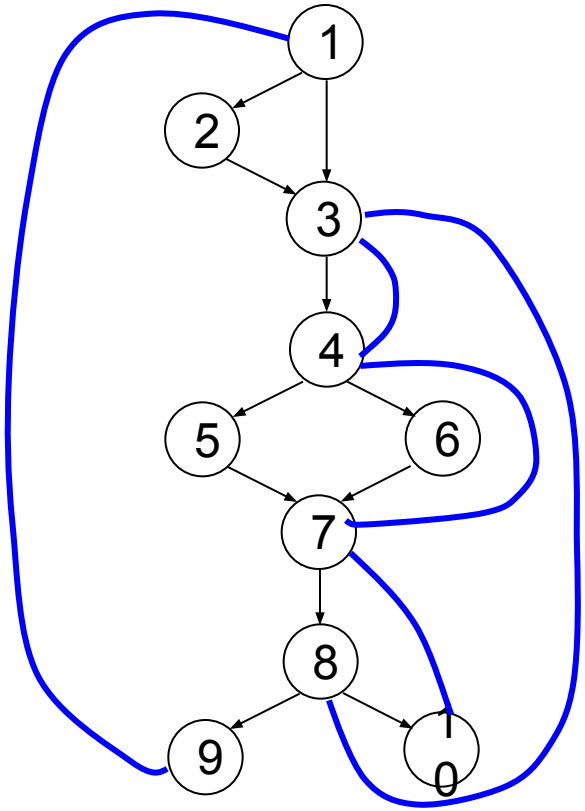


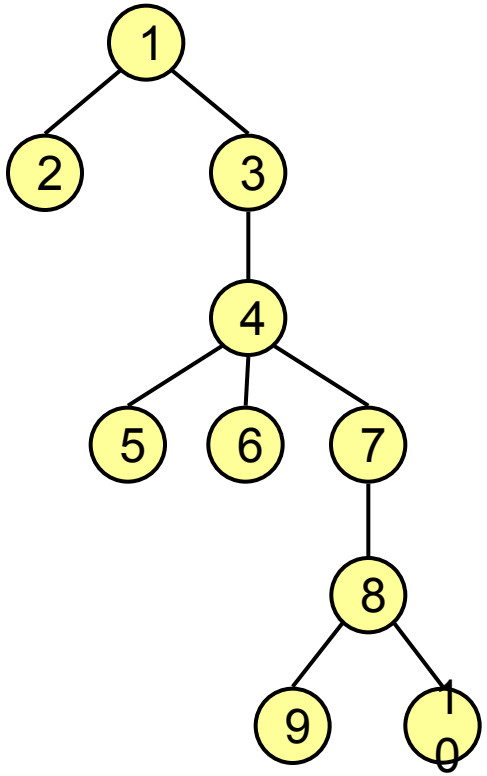


Each node (beside the entry node) has a unique immediate dominator
⇒ always possible to build the dominator tree
(each node has one and only one parent)

Finding a Loop in a CFG -- Example

- Find the back edges
  - From the bottom of the tree, find outgoing edges that go up the tree

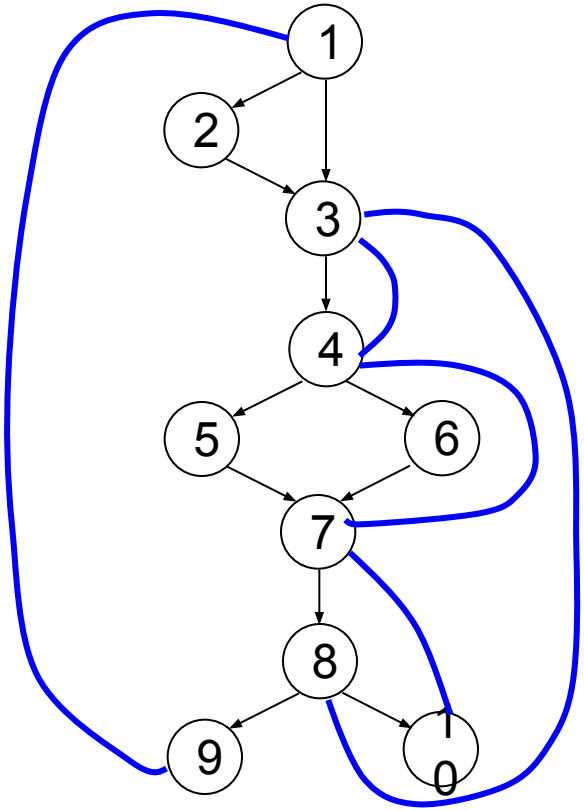


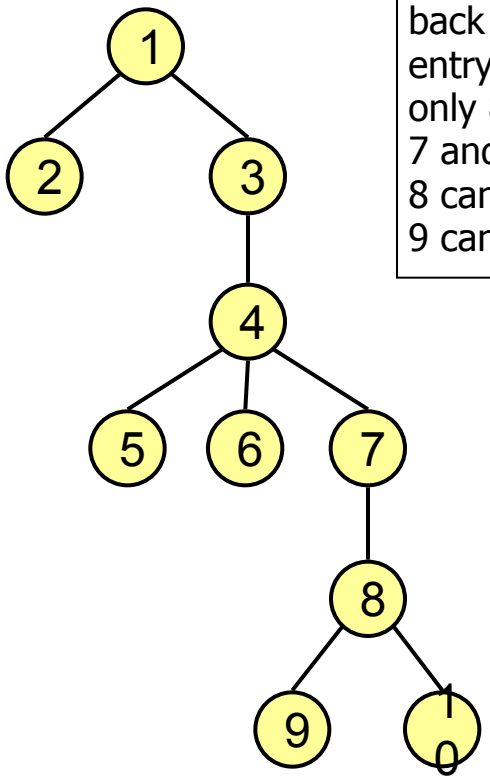


(10, 7)
(9, 1)
(8, 3)
(7, 4)
(4, 3)

Finding a Loop in a CFG -- Example

- Find the natural loop associated with each back edge (b, a)
  - Dominated by *a* (*a* is the entry of the loop)
  - can reach *b* without going through *a*





back edge (10,7)
entry is 7
only 8, 9, 10 are dominated by 7
7 and 10 are in the loop
8 can reach 10 without going through 7
9 cannot reach 10 without going through 7

(10, 7):
entry is 7
{7, 8, 10}
(9, 1)
entry is 1
include all nodes
(8, 3)
{3, 4, 5, 6, 7, 8, 10 }
(7, 4)
{4, 5, 6, 7, 8, 10}
(4, 3)
{3, 4, 5, 6, 7, 8, 10}