



中山大學

SUN YAT-SEN UNIVERSITY

计算机学院（软件学院）

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle

编译原理

第20讲：代码优化(1)

张献伟

xianweiz.github.io

DCS290, 5/23/2024

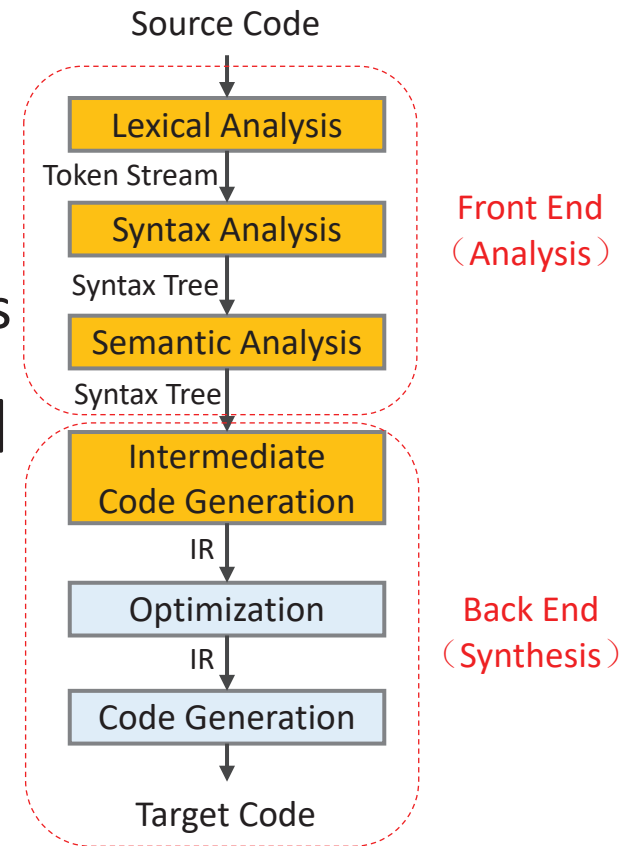


中山大學
SUN YAT-SEN UNIVERSITY




Optimization[代码优化]

- What we have now
 - IR of the source program (+symbol table)
- Goal of optimization[优化目标]
 - Improve the IR generated by the previous step to take better advantage of resources
- A very active area of research[研究热点]
 - Front end phases are **well understood**
 - Unoptimized code generation is **relatively straightforward**
 - Many optimizations are **NP-complete**
 - Thus usually rely on heuristics and approximations



To Optimize: Who, When, Where?

- Manual: source code[人工, 源码]
 - Select appropriate algorithms and data structures
 - Write code that the compiler can effectively optimize
 - Need to understand the capabilities and limitations of compiler opts
- **Compiler**: intermediate representation[编译器, IR] 
 - To generate more efficient TAC instructions
- **Compiler**: final code generation[编译器, 目标代码]
 - E.g., selecting effective instructions to emit, allocating registers in a better way
- Assembler/Linker: after final code gen[汇编/链接, 目标代码]
 - Attempting to re-work the assembly code itself into something more efficient (e.g., link-time optimization)

Example

```
int find_min(const int* array, const int len) {
    int min = a[0];
    for (int i = 1; i < len; i++) {
        if (a[i] < min) { min = a[i]; }
    }
    return min;
}

int find_max(const int* array, const int len) {
    int max = a[0];
    for (int i = 1; i < len; i++) {
        if (a[i] > max) { max = a[i]; }
    }
    return min;
}

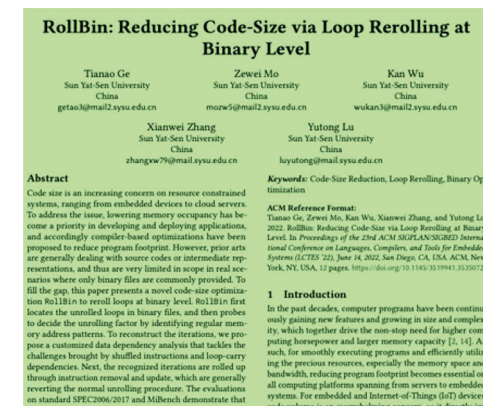
void main() {
    int* array, len, min, max;
    initialize_array(array, &len);
    min = find_min(array, len);
    max = find_max(array, len);
    ...
}
```

Inline
Loop merge

```
void main() {
    int* array, len, min, max;
    initialize_array(array, &len);
    min = a[0]; max = a[0];
    for (int i = 0; i < len; i++) {
        if (a[i] < min) { min = a[i]; }
        if (a[i] > max) { max = a[i]; }
    }
    ...
}
```

Overview of Optimizations

- Goal of optimization is to generate **better** code[更好的代码]
 - Impossible to generate **optimal** code (so, it is improvement, actually)
 - Factors beyond control of compiler (user input, OS/HW design) all affect what is optimal
 - Even discounting above, it's still a NP-complete problem
- Better one or more of the following (in the average case)
 - **Execution time**[运行时间]
 - **Memory usage**[内存使用]
 - **Energy consumption**[能耗]
 - To reduce energy bill in a data center
 - To improve the lifetime of battery powered devices
 - **Binary executable size**[可执行文件大小]
 - If binary needs to be sent over the network
 - If binary must fit inside small device with limited storage
 - Other criteria[其他]
- Should never change program semantics[正确性是前提]



Types of Optimizations[分类]

- Compiler optimization is essentially a transformation[转换]
 - Delete / Add / Move / Modify something
- **Layout-related** transformations[布局相关]
 - Optimizes *where* in memory code and data is placed
 - Goal: maximize **spatial locality**[空间局部性]
 - Spatial locality: on an access, likelihood that nearby locations will also be accessed soon
 - Increases likelihood subsequent accesses will be faster
 - E.g. If access fetches cache line, later access can reuse
 - E.g. If access page faults, later access can reuse page
- **Code-related** transformations[代码相关]
 - Optimizes *what* code is generated
 - Goal: execute least number of most costly instructions



Layout-Related Opt.: Code

- Two ways to layout code for the below example

```
f() {  
  ...  
  h();  
  ...  
}  
g() {  
  ...  
}  
h() {  
  ...  
}
```



code of f()
code of g()
code of h()

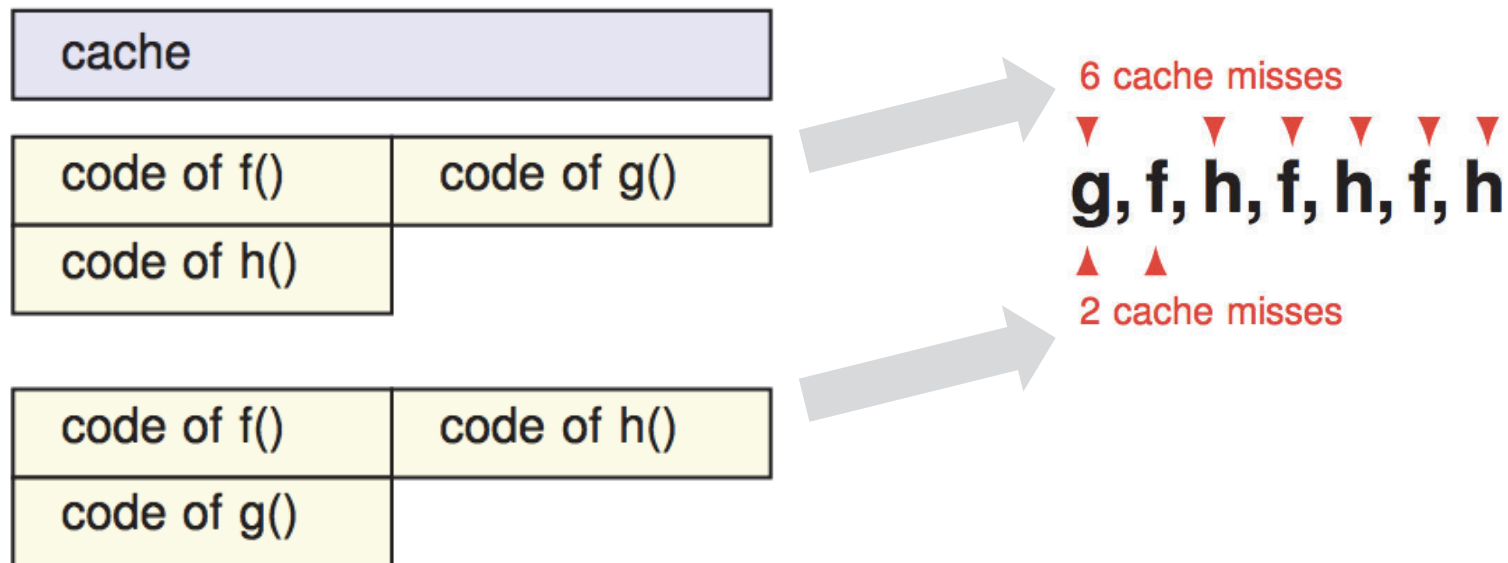
OR



code of f()
code of h()
code of g()

Layout-Related Opt.: Code (cont.)

- Which code layout is better?
- Assume
 - data cache has one N -word line
 - the size of each function is $N/2$ -word long
 - access sequence is “g, f, h, f, h, f, h”



Layout-Related Opt.: Data

- Change the variable declaration order

```
struct S {  
    int x1;  
    int x2[200];  
    int x3;  
} obj[100];  
  
for(...) {  
    ... = obj[i].x1 + obj[i].x3;  
}
```



```
struct S {  
    int x1;  
    int x3;  
    int x2[200];  
} obj[100];  
  
for(...) {  
    ... = obj[i].x1 + obj[i].x3;  
}
```

- Improved spatial locality
 - Now **x1** and **x3** likely reside in same cache line
 - Access to **x3** will always hit in the cache

Layout-Related Opt.: Data (cont.)

- Change AOS (array of structs) to SOA (struct of arrays)

```
struct S {  
    int x;  
    int y;  
} points[100];  
  
for(...) {  
    ... = points[i].x * 2;  
}  
  
for(...) {  
    ... = points[i].y * 2;  
}
```



```
struct S {  
    int x[100];  
    int y[100];  
} points;  
  
for(...) {  
    ... = points.x[i] * 2;  
}  
  
for(...) {  
    ... = points.y[i] * 2;  
}
```

- Improved spatial locality for accesses to 'x's and 'y's

Structure Peeling[结构分离]

```
struct S {  
    int A;  
    int B;  
    int C;  
};
```

A,C – Hot fields
B – Cold field

Peeled structures:

```
struct S.Hot {  
    int A;  
    int C;  
};
```

```
struct S.Cold {  
    int B;  
};
```

<https://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf>

<https://llvm.org/devmtg/2021-02-28/slides/Prashantha-MLIR-LTO.pdf>

Code-Related Optimizations

- Modifying code e.g. **strength reduction**[强度削减]
`A=2*a; ≡ A=a«1;`
- Deleting code e.g. **dead code elimination**
`A=2; A=y; ≡ A=y;`
- Moving code e.g. **code scheduling**
`A=x*y; B=A+1; C=y; ≡ A=x*y; C=y; B=A+1;`
(Now `C=y;` can execute while waiting for `A=x*y;`)
- Inserting code e.g. **data prefetching**[数据预取]
`while (p!=NULL)`
`{ process(p); p=p->next; }`
`≡`
`while (p!=NULL)`
`{ prefetch(p->next); process(p); p=p->next; }`
(Now access to `p->next` is likely to hit in cache)



中山大學
SUN YAT-SEN UNIVERSITY

计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle 编译原理

第21讲：代码优化(2)

张献伟

xianweiz.github.io

DCS290, 5/30/2024

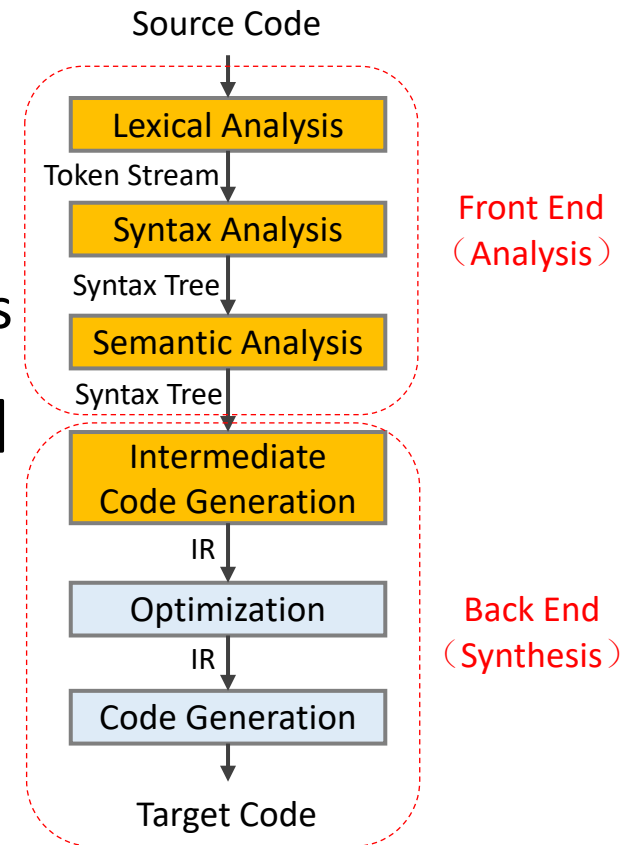


中山大學
SUN YAT-SEN UNIVERSITY



Optimization[代码优化]

- What we have now
 - IR of the source program (+symbol table)
- Goal of optimization[优化目标]
 - Improve the IR generated by the previous step to take better advantage of resources
- A very active area of research[研究热点]
 - Front end phases are **well understood**
 - Unoptimized code generation is **relatively straightforward**
 - Many optimizations are **NP-complete**
 - Thus usually rely on heuristics and approximations



Types of Optimizations[分类]

- Compiler optimization is essentially a transformation[转换]
 - Delete / Add / Move / Modify something
- **Layout-related** transformations[布局相关]
 - Optimizes *where* in memory code and data is placed
 - Goal: maximize **spatial locality**[空间局部性]
 - Spatial locality: on an access, likelihood that nearby locations will also be accessed soon
 - Increases likelihood subsequent accesses will be faster
 - E.g. If access fetches cache line, later access can reuse
 - E.g. If access page faults, later access can reuse page
- **Code-related** transformations[代码相关]
 - Optimizes *what* code is generated
 - Goal: execute least number of most costly instructions



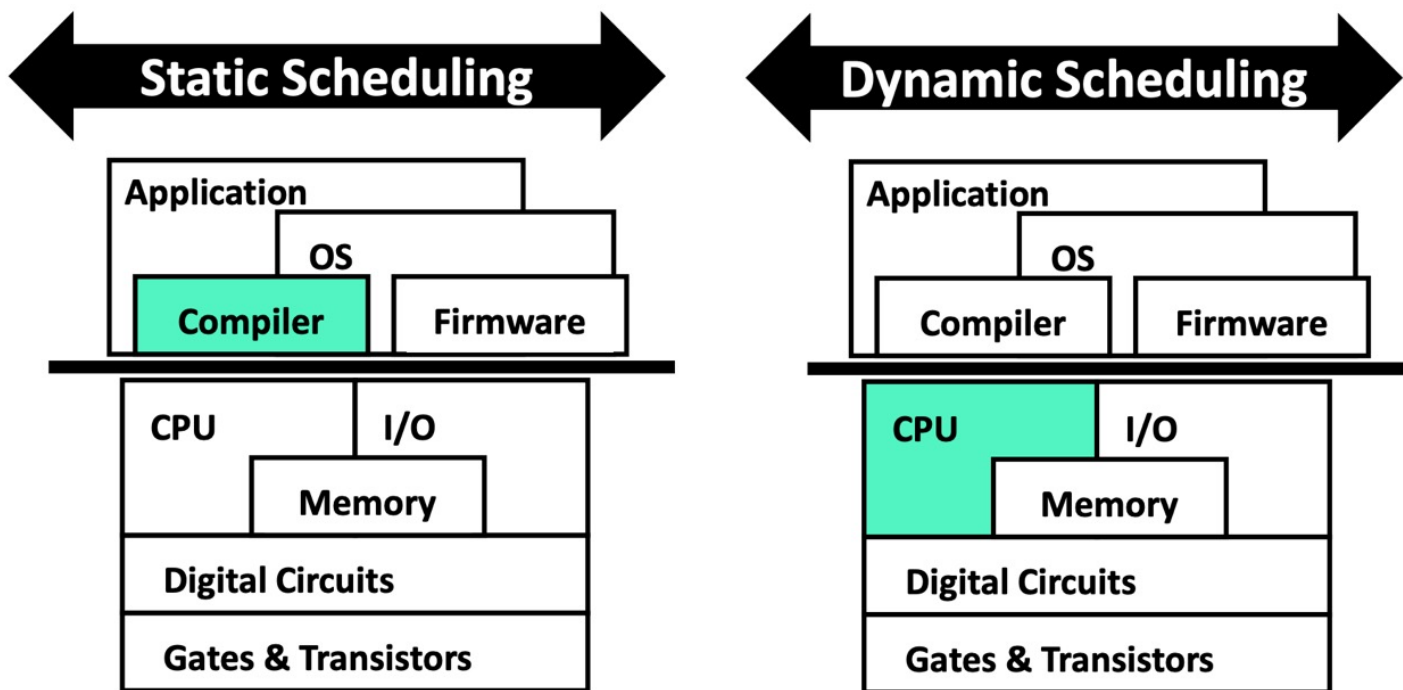
Code-Related Optimizations

- Modifying code e.g. **strength reduction**[强度削减]
`A=2*a; ≡ A=a«1;`
- Deleting code e.g. **dead code elimination**
`A=2; A=y; ≡ A=y;`
- Moving code e.g. **code scheduling**
`A=x*y; B=A+1; C=y; ≡ A=x*y; C=y; B=A+1;`
(Now `C=y;` can execute while waiting for `A=x*y;`)
- Inserting code e.g. **data prefetching**[数据预取]
`while (p!=NULL)`
`{ process(p); p=p->next; }`
`≡`
`while (p!=NULL)`
`{ prefetch(p->next); process(p); p=p->next; }`
(Now access to `p->next` is likely to hit in cache)

↩ Detour: Instruction Scheduling[指令调度]



- Scheduling: act of finding independent instructions
 - Static: done at compile time by the compiler (sw)
 - Dynamic: done at runtime by the processor (hw)
 - Scoreboard, Tomasulo's algorithm, Reorder Buffer (ROB)



↗ Detour: Compiler Tech. to Expose ILP



- Scheduling[调度]
 - To keep a pipeline full, parallelism among insts must be exploited by finding sequences of unrelated insts that can be overlapped in the pipeline[重叠]
 - To avoid a pipeline stall, the execution of a dependent inst must be separated from the source insts by a distance in clock cycles equal to the pipeline latency of that source inst[分隔]
- A compiler's ability to perform the scheduling depends on
 - Amount of ILP in the program[程序特性]
 - Latencies of the functional units in the pipeline[硬件特性]
- Compiler can increase the amount of available of ILP by transforming loops[循环转换]

↗ Detour: Loop Unrolling[循环展开]



- Simply replicates the loop body multiple times, adjusting the loop termination code[复制->调整]
 - Increases the number of insts relative to the branch and overhead insts[增加有效指令数]
 - Eliminates branches, thus allowing insts from different iterations to be scheduled together[消除分支, 共同调度]

```
Loop: fld    f0, 0(x1)
      fadd.d f4, f0, f2
      fsd    f4, 0(x1)
      fld    f6, -8(x1)
      fadd.d f8, f6, f2
      fsd    f8, -8(x1)
      fld    f0, -16(x1)
      fadd.d f12, f0, f2
      fsd    f12, -16(x1)
      fld    f14, -24(x1)
      fadd.d f16, f14, f2
      fsd    f16, -24(x1)
      addi   x1, x1, -32
      bne    x1, x2, loop
```



```
Loop: fld    f0, 0(x1)
      fld    f6, -8(x1)
      fld    f0, -16(x1)
      fld    f14, -24(x1)
      fadd.d f4, f0, f2
      fadd.d f8, f6, f2
      fadd.d f12, f0, f2
      fadd.d f16, f14, f2
      fsd    f4, 0(x1)
      fsd    f8, -8(x1)
      fsd    f12, -16(x1)
      fsd    f16, -24(x1)
      addi   x1, x1, -32
      bne    x1, x2, loop
```

A total of 14 clock cycles
(3.5 cycles per iter)



↶ ↷ Detour: Unrolling Limitations[限制]

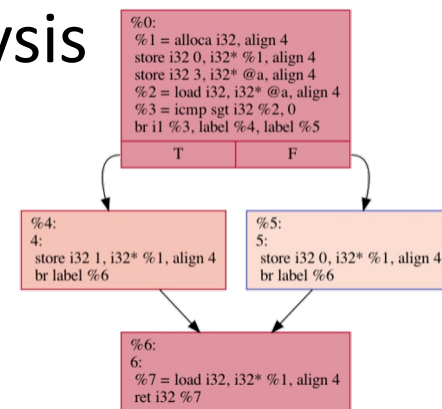


- The gains from loop unrolling are limited by
 - A decrease in the amount of **overhead** amortized with each unroll
 - Unrolled 4 times → 8 times: $\frac{1}{2}$ cycle/iter → $\frac{1}{4}$ cycle/iter
 - Growth in **code size** caused by unrolling
 - May increase in the inst cache miss rate
 - May bring register pressure (more live values)
 - **Compiler** limitations
 - Sophisticated transformations increases the compiler complexity

```
Loop: fld    f0, 0(x1)
      fld    f6, -8(x1)
      fld    f0, -16(x1)
      fld    f14, -24(x1)
      fadd.d  f4, f0, f2
      fadd.d  f8, f6, f2
      fadd.d  f12, f0, f2
      fadd.d  f16, f14, f2
      fsd    f4, 0(x1)
      fsd    f8, -8(x1)
      fsd    f12, -16(x1)
      fsd    f16, -24(x1)
      addi   x1, x1, -32
      bne    x1, x2, loop
```

Control-Flow Analysis[控制流分析]

- The compiling process has done lots of analysis
 - Lexical
 - Syntax
 - Semantic
 - IR
- But, it still doesn't really know how the program does what it does
- **Control-flow analysis** helps compiler to figure out more info about how the program does its work
 - First construct a **control-flow graph** (CFG), which is a graph of the different possible paths program flow could take through a function
 - To build the graph, we first divide the code into basic blocks

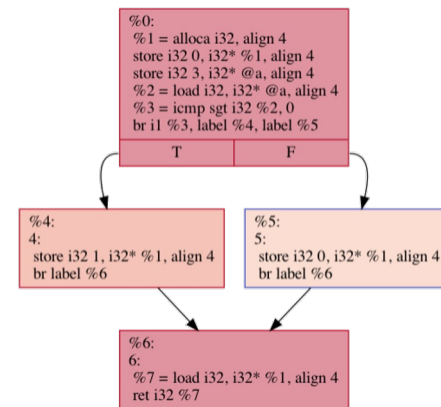


Basic Block[基本块]

- A **basic block** is a maximal sequence of instructions that
 - Except the first instruction, there are no other labels[只第一条入]
 - Except the last instruction, there are no jumps[只末一条出]
- Therefore, [进/出口唯一]
 - Can only jump into the beginning of a block
 - Can only jump out at the end of a block
- Are units of control flow that cannot be divided further
 - All instructions in a basic block execute or none at all[all or nothing]
- Local optimizations are limited to scope of a basic block
- Global optimizations are across basic blocks

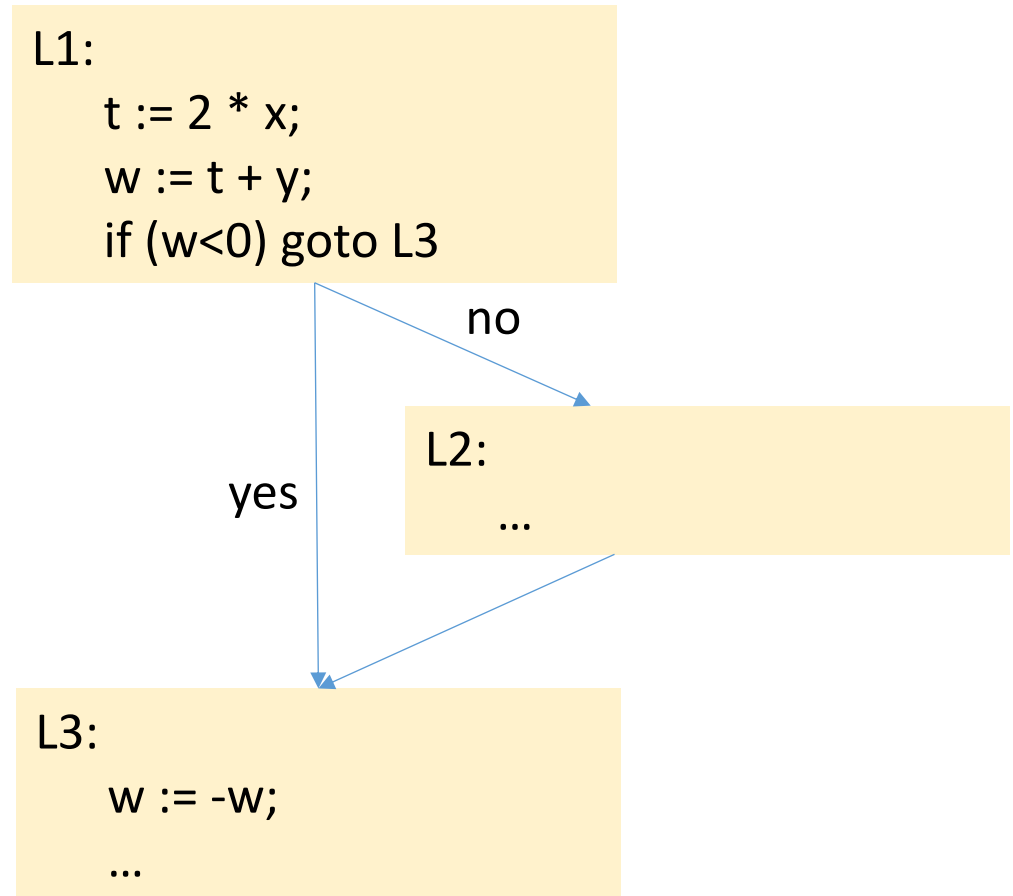
Control Flow Graph[控制流图]

- A **control flow graph** is a directed graph in which
 - **Nodes** are basic blocks
 - **Edges** represent flow of execution between basic blocks
 - Flow from end of one basic block to beginning of another
 - Flow can be result of a control flow divergence
 - Flow can be result of a control flow merge
 - Control statements introduce control flow edges
 - e.g. if-else, for-loop, while-loop, ...
- CFG is widely used to represent a function
- CFG is widely used for program analysis, especially for global analysis/optimization



Example

```
L1:
  t := 2 * x;
  w := t + y;
  if (w < 0) goto L3
L2:
  ...
L3:
  w := -w;
  ...
```



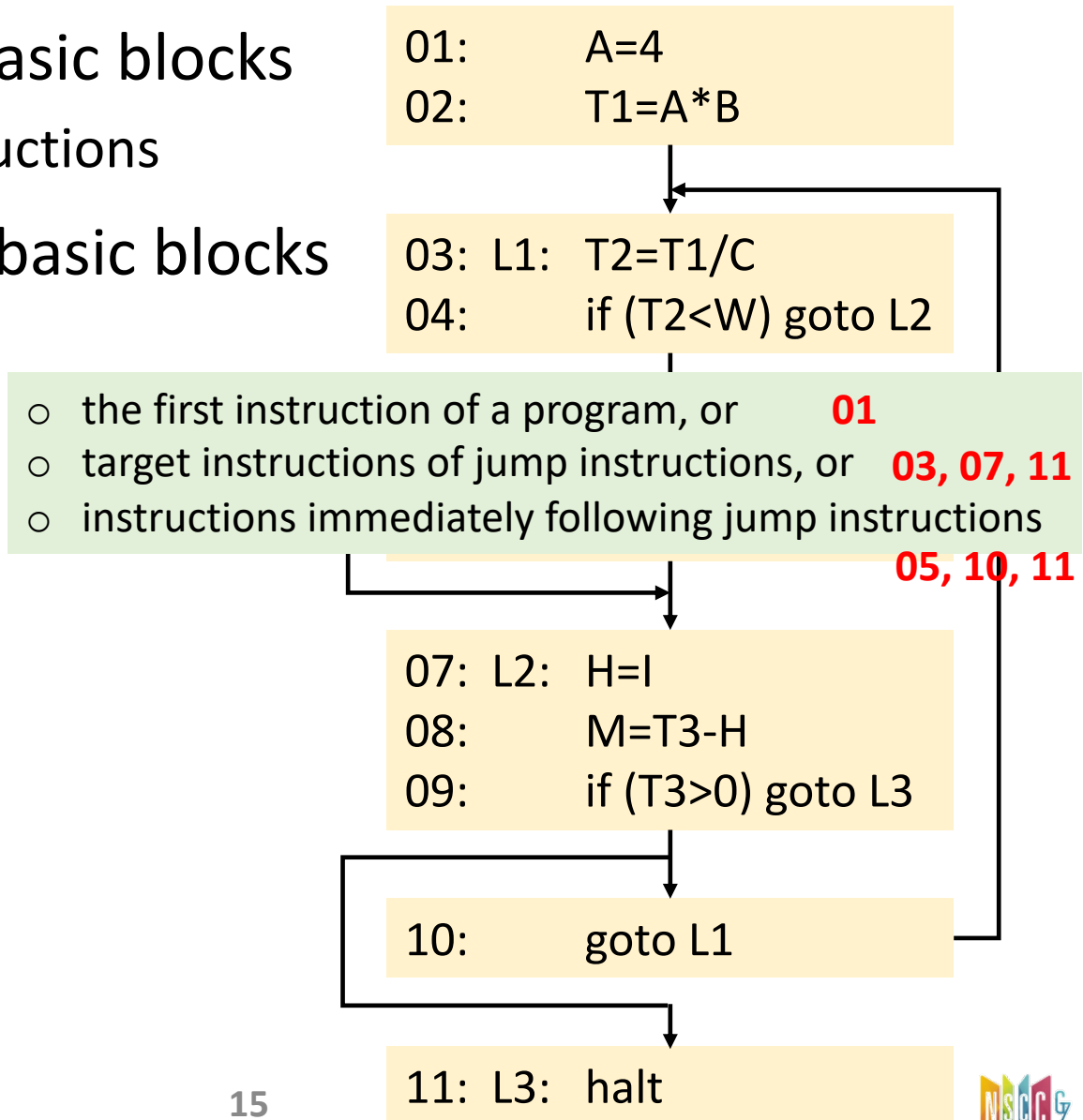
Construct CFG

- Step 1: partition code into basic blocks[分解为基本块]
 - Identify **leader** instructions that are
 - the first instruction of a program, or[首条指令]
 - target instructions of jump instructions, or[跳转目标]
 - instructions immediately following jump instructions[紧跟跳转]
 - A basic block consists of a leader instruction and subsequent instructions before the next leader
- Step 2: add an edge between basic blocks B1 and B2 if[连接基本块]
 - B2 follows B1, and B1 may “fall through” to B2[相邻]
 - B1 ends with a conditional jump to another basic block[若条件假，到达B2]
 - B1 ends with a non-jump instr (B2 is a target of a jump)[无跳转，B1顺序执行到达B2]
 - Note: if B1 ends in an uncond jump, cannot fall through[B1无条件跳转，会绕开B2]
 - B2 doesn't follow B1, but B1 ends with a jump to B2[不相邻，但B2是B1的
跳转目标]

Example

- Partition code into basic blocks
 - Identify leader instructions
- Add edges between basic blocks

```
01:      A=4
02:      T1=A*B
03: L1:  T2=T1/C
04:      if (T2<W) goto L2
05:      M=T1*K
06:      T3=M+1
07: L2:  H=I
08:      M=T3-H
09:      if (T3>0) goto L3
10:      goto L1
11: L3:  halt
```



Local and Global Optimizations

- Local optimizations[局部优化]
 - Optimizations performed exclusively within a basic block
 - Typically the easiest, never consider any control flow info
 - All instructions in scope executed exactly once
 - Examples:
 - constant folding[常量折叠]
 - common subexpression elimination[公共子表达式删除]
- Global optimizations[全局优化]
 - Optimizations performed across basic blocks
 - Scope can contain if / while / for statements
 - Some insts may not execute, or even execute multiple times
 - Note: global here doesn't mean across the entire program
 - We usually optimize one function at a time

Local Optimization: Examples

- Common subexpression elimination[公共子表达式删除]
 - Two operations are common if they produce the same result
 - It is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it[避免重复计算]
- Dead code elimination[死代码删除]
 - If an instruction's result is never used, the instruction is considered “dead” and can be removed from the instruction stream[结果从不使用]

```
y = x + z;  
y = x * x + (x/3)  
z = x * x + y;
```



```
y = x + z;  
t1 = x * x  
t2 = x / 3  
y = t1 + t2  
t3 = x * x  
z = t3 + y;
```



```
y = x + z;  
t1 = x * x  
t2 = x / 3  
y = t1 + t2  
t3 = x * x  
z = t1 + y;
```

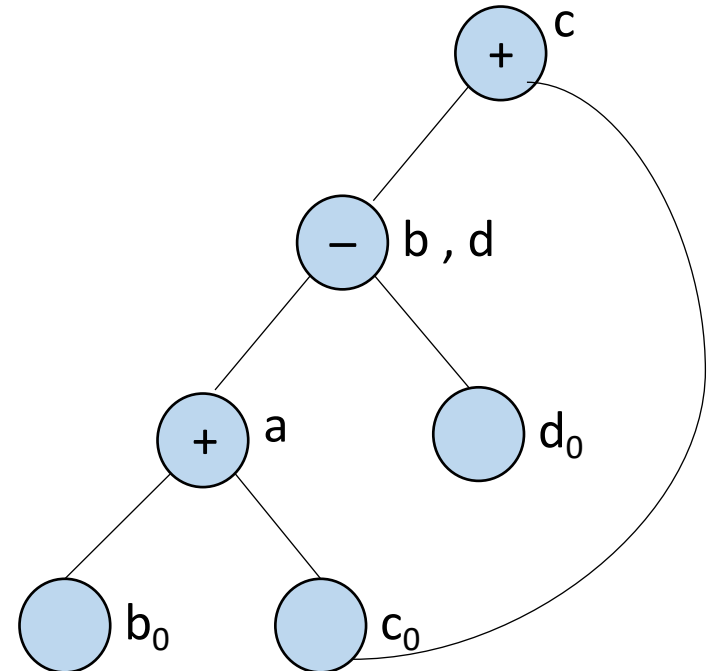
DAG of Basic Blocks

- Many important techniques for local optimization begin by transforming a BB into a DAG (directed acyclic graph)[无环有向图]
- To construct a DAG for a BB as follows
 - Create a node for each of the initial values of the variables appearing in the BB[为变量初始值创建节点，叶子]
 - Create a node N associated with each statement s within the block[为声明语句创建节点，中间]
 - The children of N are those nodes corresponding to statements that are the last definitions, prior to s , of the operands used by s
 - Label N by the operator applied at s [用运算符标注节点]
 - Certain nodes are designated output nodes[某些为输出节点]
 - These are the nodes whose variables are live on exit from the block (i.e., their values may be used later, in another block of the flow graph)

Example: DAG

- (3) $c = b + c$
 - b refers to the node labelled '-'
 - Most recent definition of b
- (4) $d = a - d$
 - Operator and children are the same as the 2nd statement
 - Reuse the node

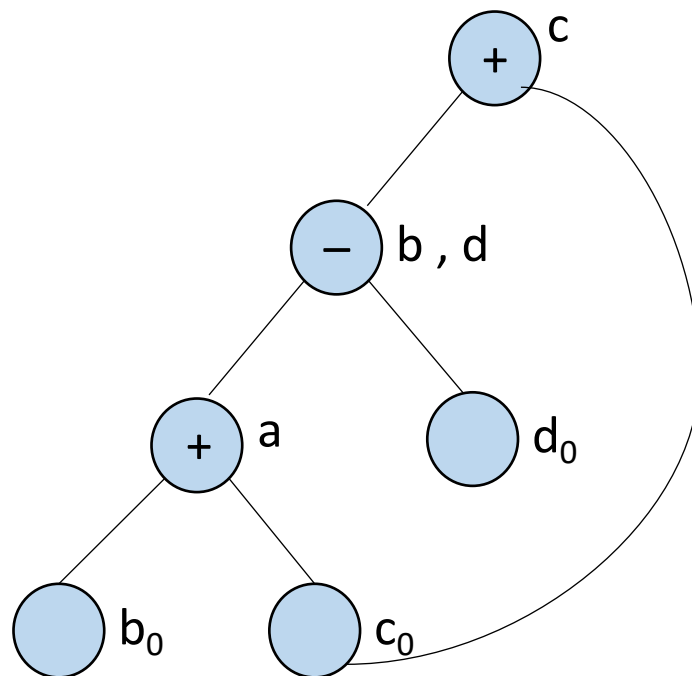
(1) $a = b + c$
(2) $b = a - d$
(3) $c = b + c$
(4) $d = a - d$



Local Opt.: Elimination

- If c is **not live** on exit from the block
 - No need to keep $c = b + c$
- If both b and d are **live**
 - Remove either (2) or (4) : **common subexpr elimination**
 - Add a 4th statement to copy one to the other
- If only a is **live** on exit
 - Then remove nodes from the DAG correspond to dead code
 - $c \rightarrow b, d \rightarrow d_0$
 - This is actually **dead code elimination**

```
(1) a = b + c
(2) b = a - d
(3) c = b + c
(4) d = a - d
```



Local Opt.: Elimination (cont.)

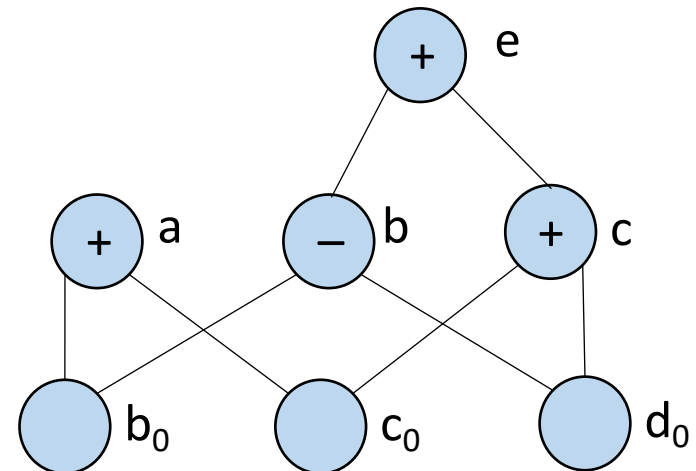
- When finding common subexprs, we really are finding exprs that are guaranteed to compute the same value, no matter how that value is computed[过于严苛]

- Thus miss the fact that (1) and (4) are the same

$$\square b + c = (b - d) + (c + d) = b_0 + c_0$$

- Solution:** algebraic identities[代数恒等式]

(1) $a = b + c$
(2) $b = b - d$
(3) $c = c + d$
(4) $e = b + c$



Local Opt.: Algebraic Identities[代数恒等式]

- Eliminate computations by applying mathematical rules[使用数学规则]
 - Identities: $a * 1 \equiv a$, $a * 0 \equiv 0$, $b \& \text{true} \equiv b$
 - Reassociation and commutativity[重组合、交换]
 - ▣ $(a + b) + c \equiv a + (b + c)$, $a + b \equiv b + a$
- **Strength Reduction**[强度削减]
 - Replacing expensive operations (*multiplication, division*) by less expensive operations (*add, sub, shift*)
 - Some ops can be replaced with cheaper ops
 - Examples
 - ▣ $x = y/8 \rightarrow x = y \gg 3$
 - ▣ $y = y * 8 \rightarrow y = y \ll 3$
 - ▣ $x^2 \rightarrow x * x$
 - ▣ $2 * x \rightarrow x + x$

Local Opt.: Constant Folding[常量折叠]

- **Constant Folding**

- Computing operations on constants at compile time
- Example:

```
#define LEN 100  
x = 2 * LEN;  
if (LEN < 0) print("error");
```

- After constant folding

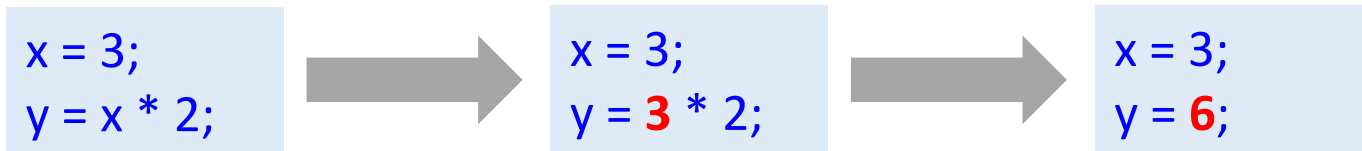
```
x = 200;  
if (false) print("error");
```

- Dead code elimination can further remove the above *if* statement
- Inherently local since scope limited to statement

Local Opt.: Constant Propagation[常量传播]

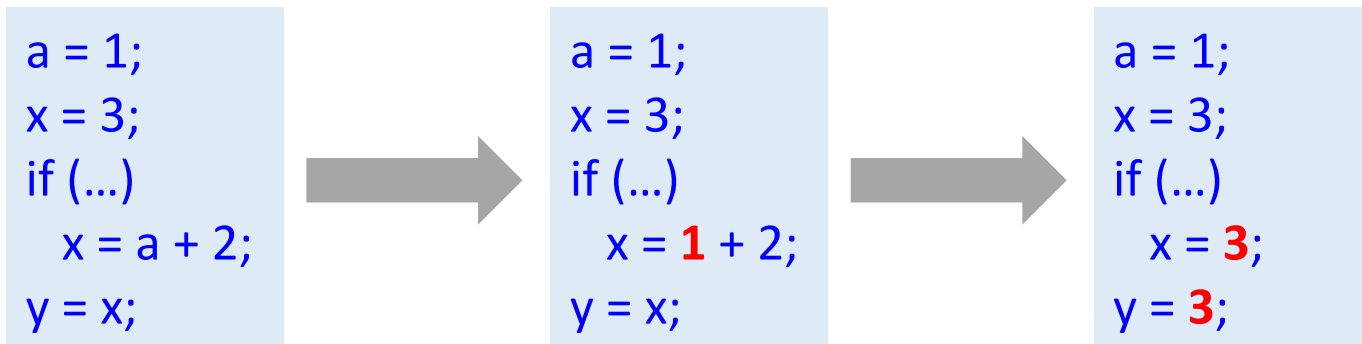
- **Constant Propagation**

- Substituting values of known constants at compile time
- Local Constant Propagation (LCP)



- Some optimizations have both local and global versions

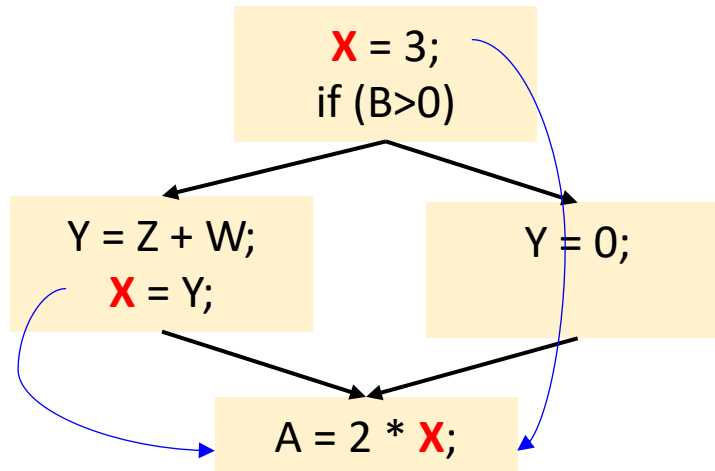
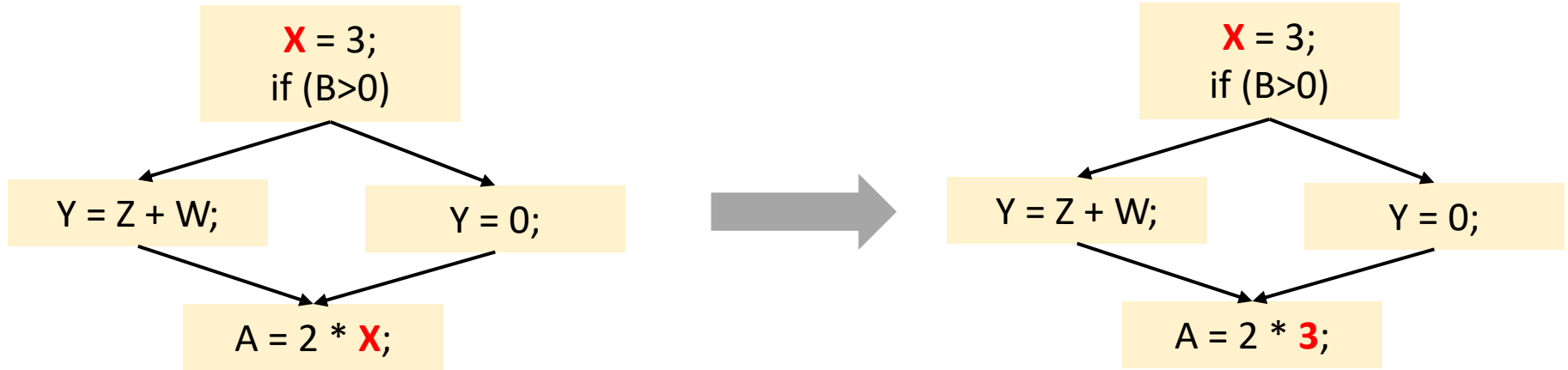
- Global Constant Propagation (GCP)



- GCP more powerful than LCP but also more complicated
 - Must determine `x` is constant across all paths reaching `x`

Global Optimizations

- Extend optimizations to flow of control, i.e. CFG
 - Along **all paths**, the last assignment to X is “X=C”
 - Optimization must be stopped if incorrect in even one path



Global Opt.: Conservative[需保守]

- Compiler must prove some property X at a particular point
 - Need to prove at that point property X holds along all paths
 - Need to be **conservative** to ensure correctness
 - An optimization is enabled only when X is definitely true
 - If not sure if it is true or not, it is safe to say **don't know**
 - If analysis result is **don't know**, no optimization done
 - May lose opt. opportunities but guarantees correctness
- Property X often involves data flow of program
 - E.g. Global Constant Propagation (GCP):
 $X = 7;$
...
 $Y = X + 3;$ // does value of 7 flow into this use of X ?
 - Needs knowledge of **data flow**, as well as control flow
 - Whether data flow is interrupted between points A and B

Global Opt.: Data Flow[数据流]

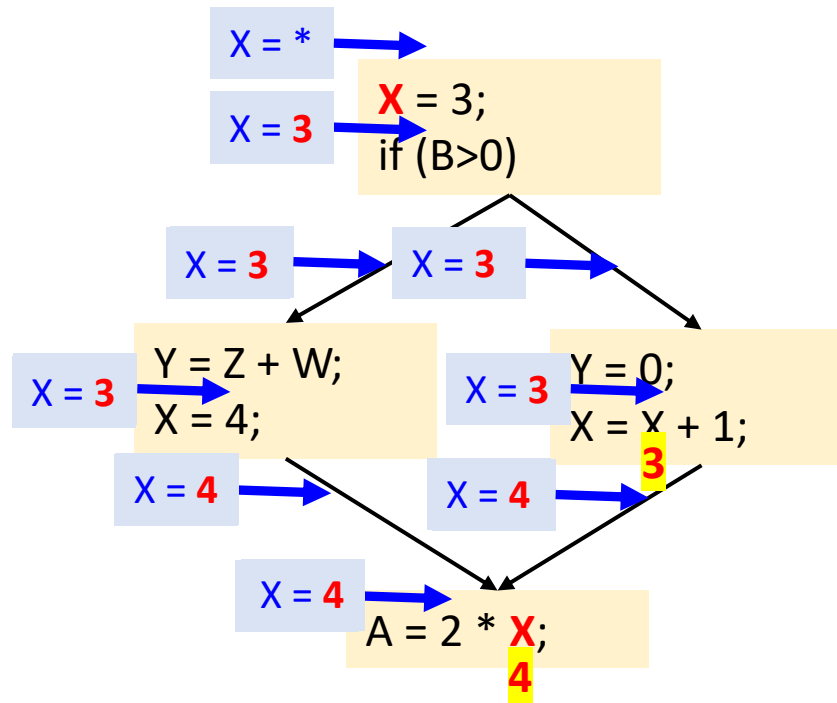
- Most optimizations rely on a property at given point
 - For Global Constant Propagation (GCP):
 $A = B + C;$ // property: $\{A=?, B=10, C=?\}$
 - After optimization:
 $A = 10 + C;$
- For this discussion, let's call these properties *values*
- **Dataflow analysis**: compiler analysis that calculates values for each point in a program
 - Values get propagated from one statement to the next
 - Statements can modify values (for GCP, assigning to variables)
 - Requires CFG since values flow through control flow edges
- **Dataflow analysis framework**: a framework for dataflow analysis that guarantees correctness for **all paths**
 - Does *not* traverse all possible paths (could be infinite)
 - To be feasible, makes **conservative** approximations

Global Constant Propagation (GCP)

- Let's apply dataflow analysis to compute values for GCP
 - Emulates what human does when tracing through code
- Let's use following notation to express the state of a var:
 - $x=*$: not assigned (default)
 - $x=1, x=2, \dots$: assigned to a constant value
 - $x=\#$: assigned to multiple values
- All values start as $x=*$ and are iteratively refined
 - Until they stabilize and reach a fixed point
- Once fixed point is reached, can replace with constants:
 - $x=*$: replace with any constant (typically 0)
 - $x=1, x=2, \dots$: replace with given constant value
 - $x=\#$: cannot do anything

Example

- In this example, constants can be propagated to **$X+1$** , **$2*X$**
- Statements visited in reverse postorder (predecessor first)



`x=*`: not assigned (default)

`x=1, x=2, ...`: assigned to a constant value

`x=#`: assigned to multiple values

Example (cont.)

- Once constants have been globally propagated, we would like to eliminate the dead code

