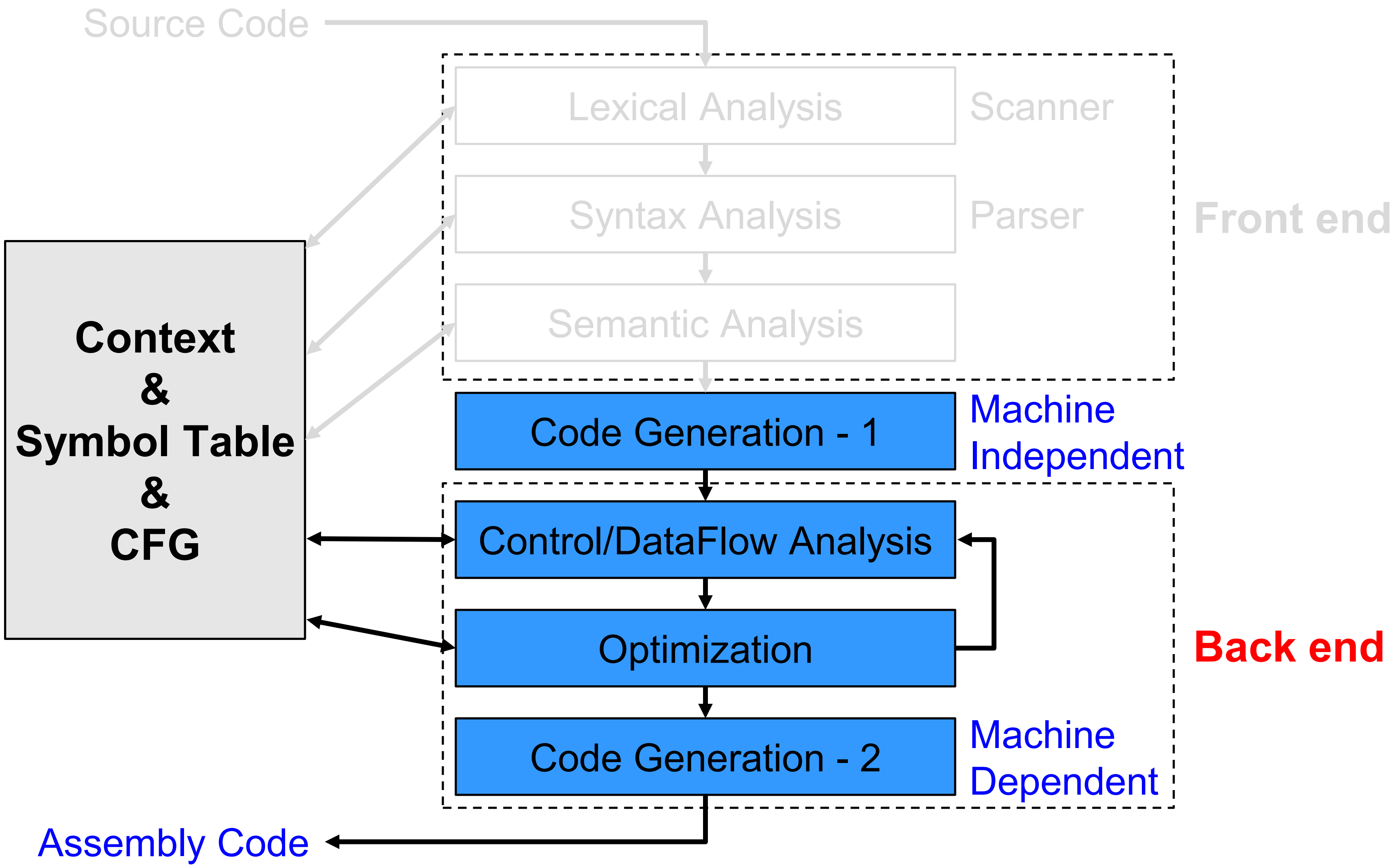
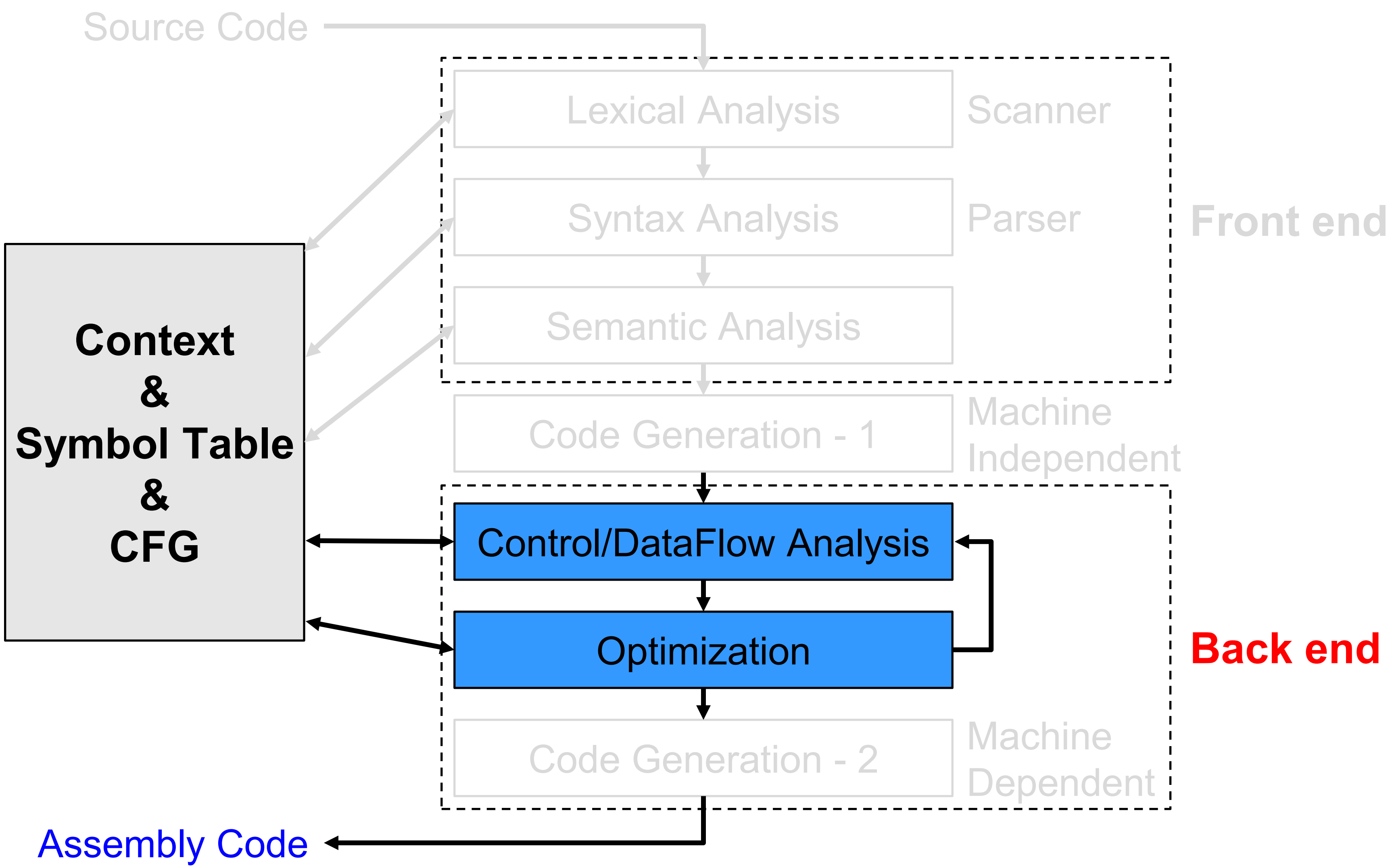
8. Dataflow Analysis

2025 Fall
Hunjun Lee
Hanyang University

What You Will Learn



What You Will Learn



Optimization Overview

- **Optimization seeks to improve a program's resource utilization**
  - Execution time (most often)
  - Code size (consider embedded system)
  - Network messages sent
- **Optimization should not change what the program computes**
  - The answer must be the same

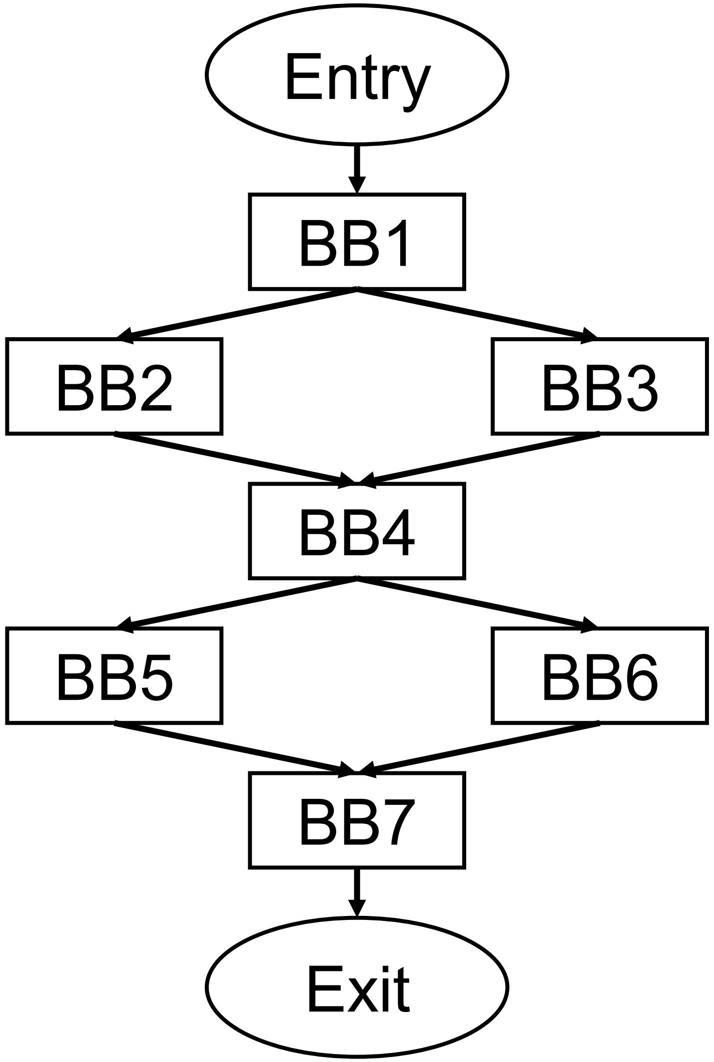
Basic Block (BB)

- **A basic block (BB) is a maximal sequence of instructions with**
  - no labels (except for the first instruction)
  - no jumps (except for the last instruction)

- **All the instructions in a BB has a fixed control flow**
  - A BB is a single-entry, single-exit, straight-line code segment
  - Cannot jump into a BB
  - Cannot jump out of a BB

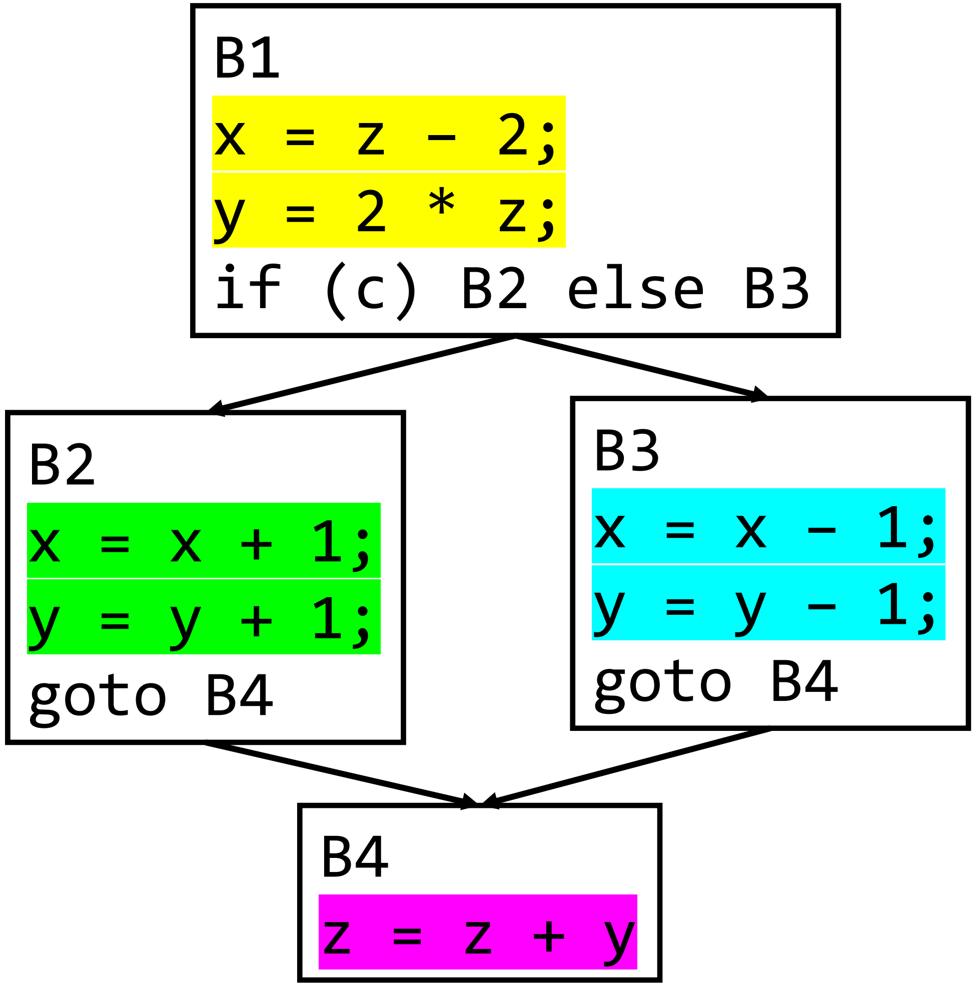
Control Flow Graph (CFG)

- **A control flow graph (CFG) is a directed graph (of a procedure) with**
  - Each basic block is a node in the CFG
  - An edge from a block A to block B indicates that there exists an execution flow from the last instruction of A to the first instruction of B



CFG Example

```
x = z - 2;
y = 2 * z;
if (c) {
    x = x + 1;
    y = y + 1;
}
else {
    x = x - 1;
    y = y - 1;
}
z = x + y
```



Optimization Scope

- **There are three possible granularities of optimizations**
  - Local optimizations
    - Apply to a basic block in isolation
  - Global optimizations (not actually global)
    - Apply to a control-flow graph
  - Inter-procedural optimizations
    - Apply across procedure boundaries

- **The goal of the compiler is to achieve the maximum benefit for minimum cost**

Optimization Types

- **Dataflow optimization**
  - Dataflow is about how a code manipulates the data
  - Can remove redundant computations or simplify computations

- **Control flow optimization**
  - Control flow is about the order of code execution (e.g., branching structure)
  - Can remove unreachable code, change code for reduced computations, …

Optimization Types

- **Dataflow optimization**
  - Dataflow is about how a code manipulates the data
  - Can remove redundant computations or simplify computations

- **Control flow optimization**
  - Control flow is about the order of code execution (e.g., branching structure)
  - Can remove unreachable code, change code for reduced computations, …

Local Optimization: Algebraic Simplification

- **Some instructions that can be deleted**

```
x := x + 0
x := x * 1
```

- **Some instructions can be simplified**

```
// Assume that x is integer
x := x * 0        ➔ x := 0
y := y ** 2       ➔ y := y * y
x := x * 8        ➔ x := x << 3
x := x * 15       ➔ t := x << 4; x := t - x
```

Local Optimization: Constant Folding

- **Constant folding ➔ computes operations on constants at compile time**

```
// Assume that x is integer
x := 2 + 2        ➔ x := 4
if 2 < 0 jump L   ➔ nop
if 2 > 0 jump L   ➔ jump L
```

- **Constant folding can be dangerous on cross-compilation**

Arch x ⊏ **Compiled Code** ⇨ Arch y

```
x := 1.5 + 3.7 ➔ 5.2
```

```
x := 1.5 + 3.7 ➔ 5.19
```

Local Optimization: Unreachable Code

- **Eliminate unreachable basic blocks (code that is unreachable from the initial block)**
  - Make the program smaller, and thus potentially faster

- **Why are there unreachable blocks?**
  - Removing debugging codes in the deployment version (`if (DEBUG) …`)
  - Using only a portion of functions from the libraries
  - Result of other optimizations

Local Optimization: Dead Code Elimination

- **If an assigned register is not used anywhere in the program, the code is dead and can be eliminated**

```
b := z + y
a := b
x := 2 * a
```

⇨

```
b := z + y
a := b
x := 2 * b
```

⇨

```
b := z + y
x := 2 * b
```

Class Exercise

```
a := 5
x := 2 * a     →   x := 10        (constant propagation/folding)
y := x + 6     →   y := 16        (constant propagation/folding)
t := x * y     →   t := x * 16    (constant propagation/folding)
                   t := x << 4    (algebraic simp)
                   t := 160       (constant propagation/folding)
```

Complete Optimization Procedure

- Each optimization itself does little by itself
- Multiple optimizations interact (you see that copy propagation enabled dead code elimination)
- Optimizing is about repeatedly applying the optimization tricks until nothing applies

Class Exercise

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

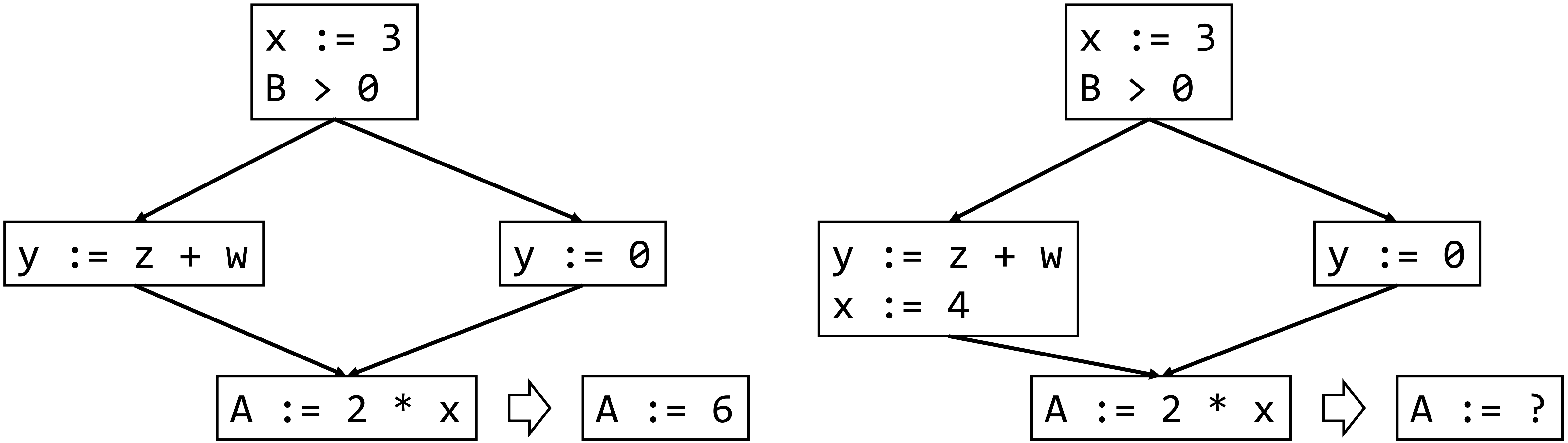
```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

```
a := x * x
f := a + a
g := 6 * f
```

```
a := x * x
g := 12 * a
```

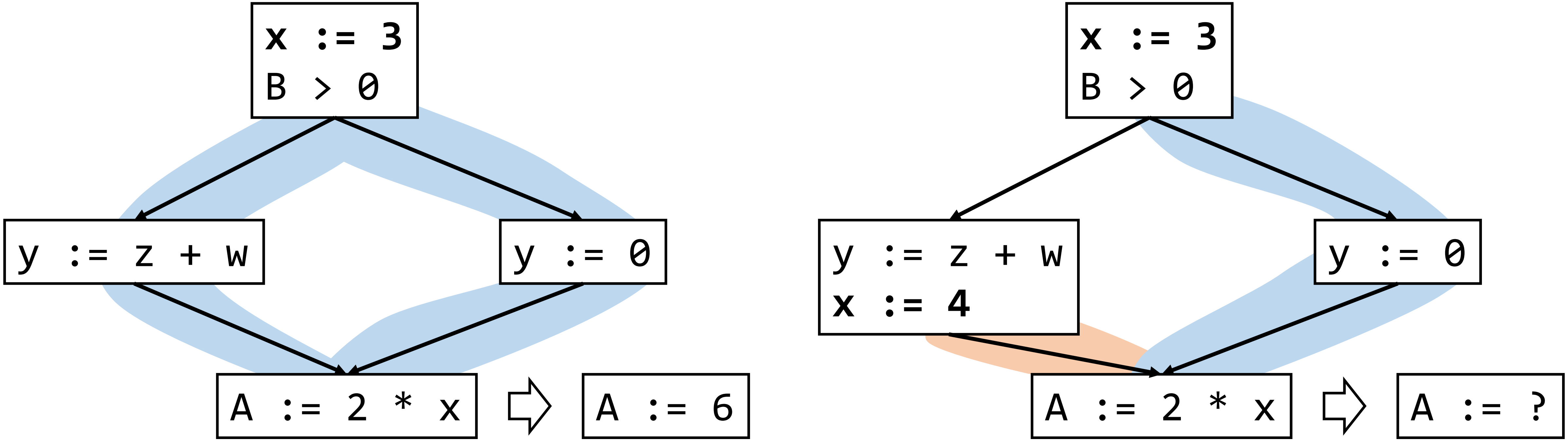
Global Optimization

- **There are cases where the optimization can be performed through the entire control-flow graph**



Global Optimization: Constant Propagation

- To replace x by a constant k, on every path to the use of x, the last assignment is x := k



Features of Global Optimization

- **Global optimization share several features**
  - The optimization depends on knowing a property X at a particular point in the program execution
    - Ex) X ➔ a variable x is a constant
  - Proving X at any point requires knowledge of the entire program (This is extremely expensive)
  - Most compilers rely on conservative methods ➔ If the optimization requires X to be true, then there are various options
    - X is definitely true ➔ it is true
    - Don't know if it is true ➔ it can be deemed as false

Dataflow Analysis

- **Compute the global information on how a given function manipulates its data (at each instruction boundary)**
  - The compiler computes which definitions can reach at each instruction
  - Evaluating the reaching definition should be conservative (incorrect information leads to incorrect optimization)
  - They are used for various global optimizations (e.g., common subexpression elimination, code motion …)

- **There are two hierarchies**
  - Global analysis: compute information at each BB boundary
  - Local analysis: compute information at each instruction within each BB

Local Dataflow Analysis

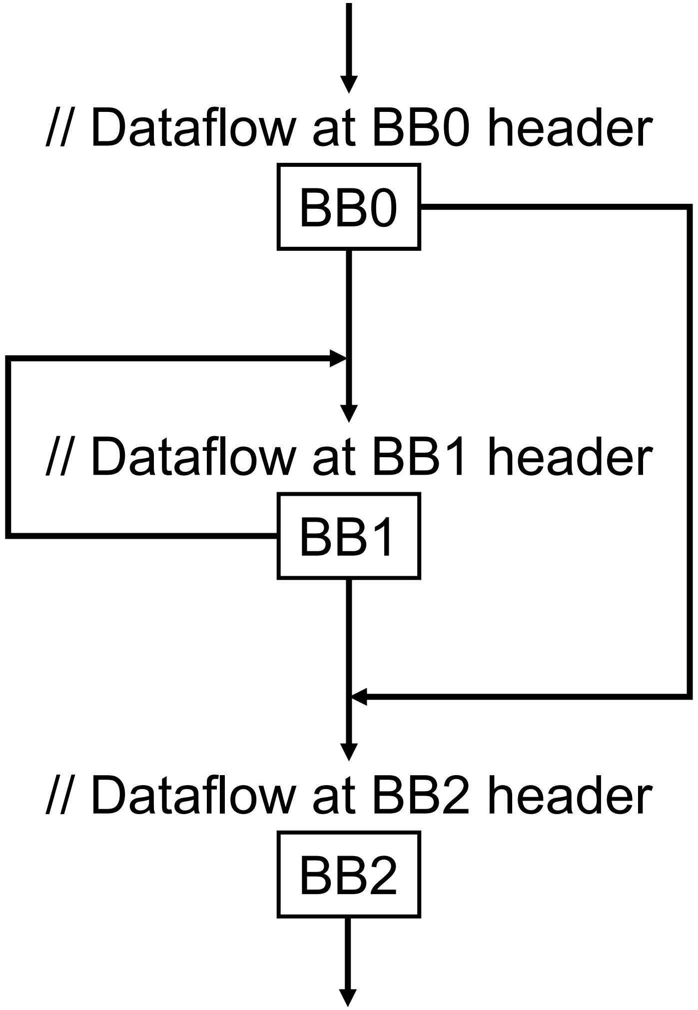
- **Local analysis evaluates information within the BB**
  - Search for redundant instructions within a single BB
  - Analyze the effect of each instruction (e.g., Kill and Generate definitions)
  - Compose the effects of multiple instructions so that we derive information from the beginning (or from the end) of a BB to each instruction

```
// Dataflow info 1
Inst 1
// Dataflow info 2
Inst 2
…
// Dataflow info n
Inst n
```

Global Dataflow Analysis

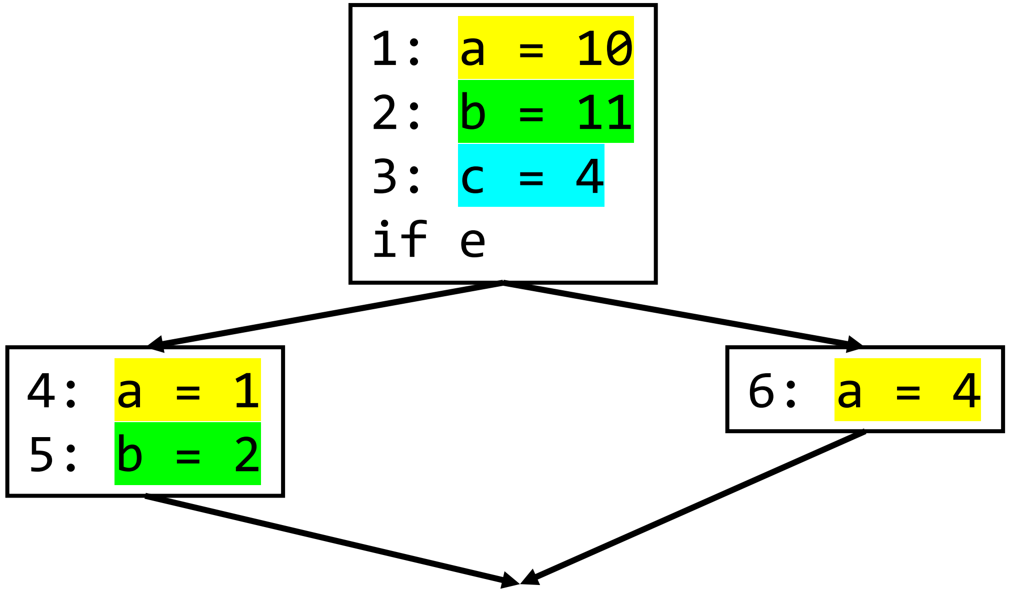
- **How global analysis computes information at each BB boundary?**
  - Summarize the effect of a BB as if it were a single instruction
  - Compose effects of BBs at each BB boundary from the beginning (or from the end), considering control flows on the CFG

- **The dataflow info can be used for evaluating reaching def and live variables**



Example: Reaching Definition

- **A definition of a variable `x` is an instruction that assigns, or may assign (e.g., conditional execution), a value to `x`**
  - A definition `d` reaches a point `p` if there exists a path from the point following `d` to `p` where d is not killed (redefined) along that path



What are reaching definitions here?

Effects of an Instruction (Local Analysis)

- **The same instruction has a different effect depending on the target optimization**

- **For an instruction: `3: a = b + c`**
  - **Kills:** all the old definitions of `a (1, 2)`
  - **Generates:** a new definition `a (3)`

```
1: a = 4
```

```
2: a = 3
3: a = b + c
```

- **We can compute dataflow information thru or across an instruction by applying the effect of the instruction**

Effects of a Basic Block (Global Analysis)

- **Combined effects of instructions at the BB boundary**
  - Calculate which definitions are used, killed, and generated before and after the BB boundary

```
1: a = 4
```

Local analysis (one-by-one)

```
2: a = 3
3: b = 5
4: a = b + c
…
```

generates: {~~2,~~3,4}
kills: {1}

```
1: a = 6
```

```
2: a = y + z
```

generates : {1,2}

```
3: a = 3
4: b = 5
5: a = b + c
…
```

Analysis of a Reaching Definition

- **Problem statement: for each basic block b, determine which definition in a function reaches b**
  - Need information at the beginning of b (IN[b]) and at the end of b (OUT[b])

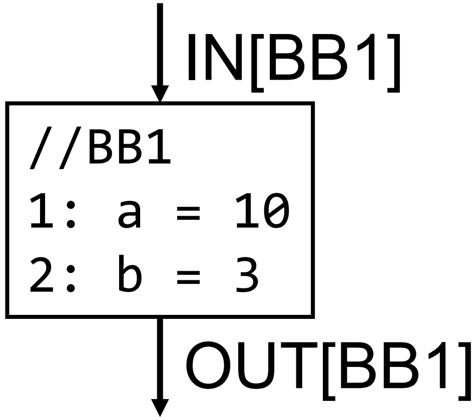
- **Representation**
  - IN[b] and OUT[b]: a set of reaching definitions

- **There are two different types:**
  - Forward: IN[b] is used to compute OUT[b] (e.g., reaching definition)
  - Backward: OUT[b] is used to compute IN[b] (e.g., liveness analysis)

Transfer Function

- **There are necessary functions in reaching definition**
  - GEN[b]: set of locally generated definitions in b
  - KILL[b]: set of definitions in the rest of program, killed by definitions in b

- **A transfer function f_b for a basic block b:**
  - OUT[b] = GEN[b] U (IN[b] – KILL[b])

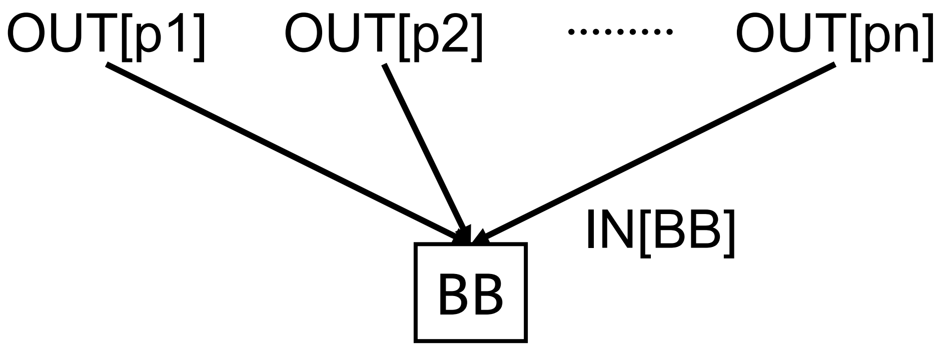


Meet Operator

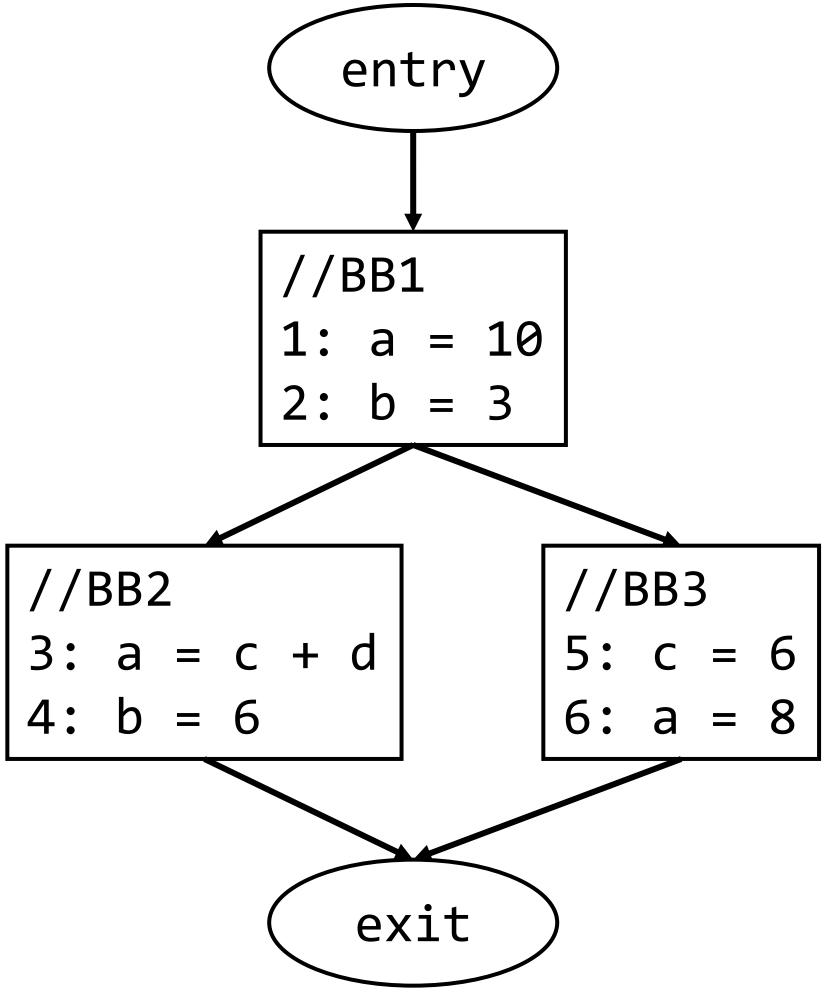
- **We also need a transfer function to compute IN[b] based on the predecessors (incoming edges)**

- **For reaching definition: it is a union of predecessors' OUT**
  - IN[b] = OUT[p1] U OUT[p2] … U OUT[pn] (p1~pn are predecessors of b)

- **To support cyclic graphs, we need repeated computation**



Reaching Definition Example - 1



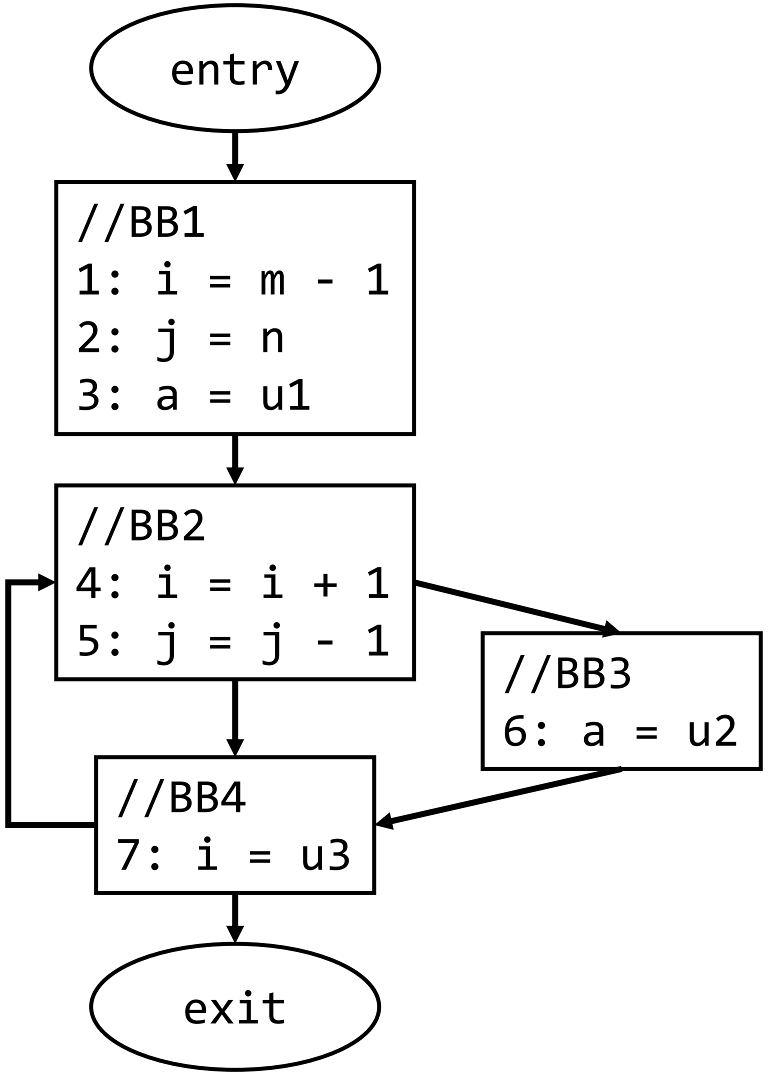
	GEN	KILL
BB1	{1,2}	{3,4,6}
BB2	{3,4}	{1,2,6}
BB3	{5,6}	{1,3}

```
IN[BB1]     = {}
OUT[BB1]    = {1,2}
IN[BB2]     = {1,2}
OUT[BB2]    = {3,4}
IN[BB3]     = {1,2}
OUT[BB3]    = {2,5,6}
IN[exit]    = {2,3,4,5,6}
```

Reaching Definition Algorithm

```
// Input : control flow graph = (N, E, Entry, Exit)
// Initialize
OUT[Entry] = {}
for all nodes i: OUT[i] = {}
worklist = {1 … N}
// Iterate
while worklist != empty {
      pop i from worklist
      IN[i] = U(OUT[p]) // Union of all predecessors
      OUT_Prev = OUT[i]
      OUT[i] = GEN[i] U (IN[i] – KILL[i])
      if (OUT_prev != OUT[i]) { // If change
            for all successors s of i
                  add s to worklist
      }
}
```

Class Exercise

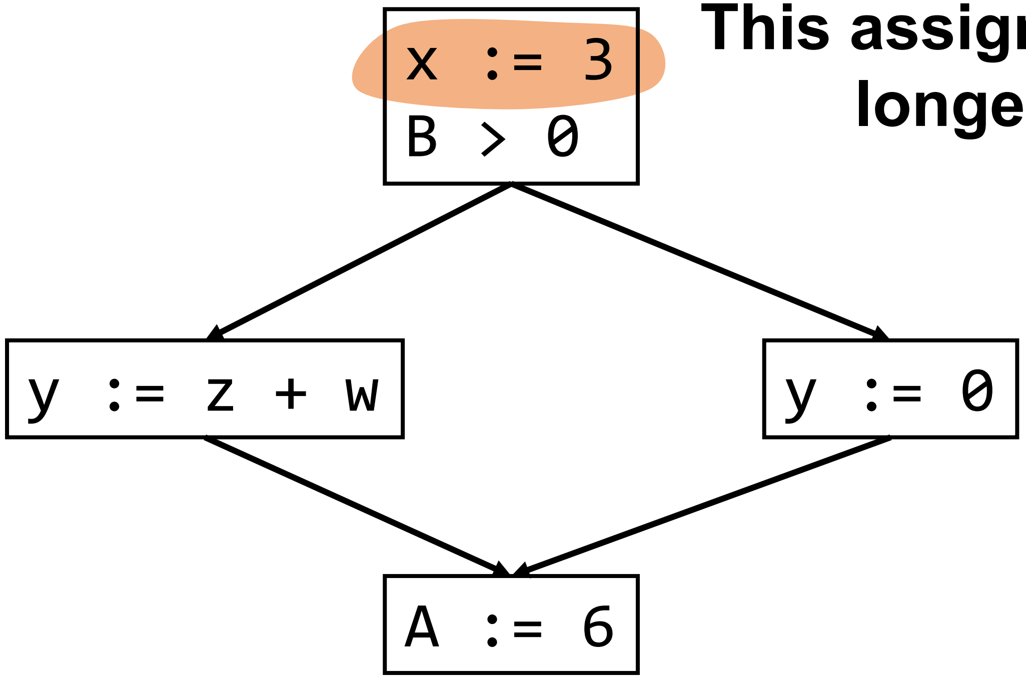


	GEN	KILL
BB1	{1,2,3}	{4,5,6,7}
BB2	{4,5}	{1,2,7}
BB3	{6}	{3}
BB4	{7}	{1,4}

```
IN[BB1]    = {}
OUT[BB1]   = {1,2,3}
IN[BB2]    = {1,2,3}      ➔ {1,2,3,5,6,7}
OUT[BB2]   = {3,4,5}      ➔ {3,4,5,6}
IN[BB3]    = {3,4,5}      ➔ {3,4,5,6}
OUT[BB3]   = {4,5,6}      ➔ {4,5,6}
IN[BB4]    = {3,4,5,6}    ➔ {3,4,5,6}
OUT[BB4]   = {3,5,6,7}    ➔ {3,5,6,7}
```

Liveness Analysis

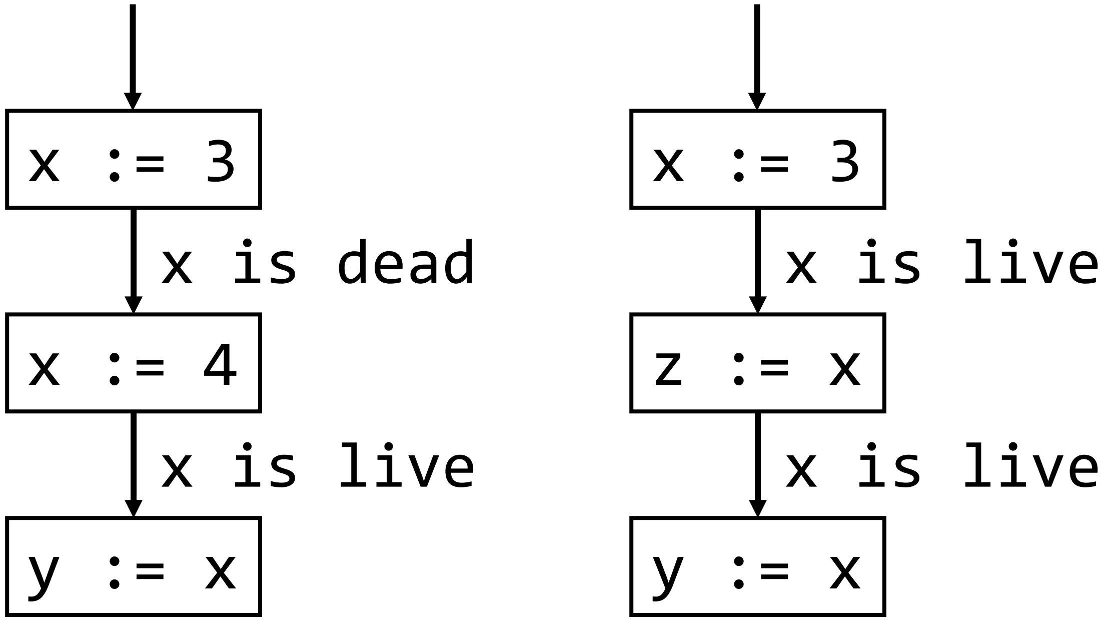
- **Liveness analysis is used to eliminate dead code**
  - liveness indicates that the assigned variable is used in the future
  - An assignment for x is dead if x is dead after the assignment



This assignment is no longer used

Transfer Function

- **There are necessary functions in liveness analysis**
  - USE[b]: set of variables used in b
  - DEF[b]: set of variables defined in b
  - IN[b] / OUT[b]: set of live variables
- **A transfer function f_b for a basic block b:**
  - IN[b] = USE[b] U (OUT[b] – DEF[b])



Meet Operator

- **We also need a transfer function to compute OUT[b] based on the successors (incoming edges)**

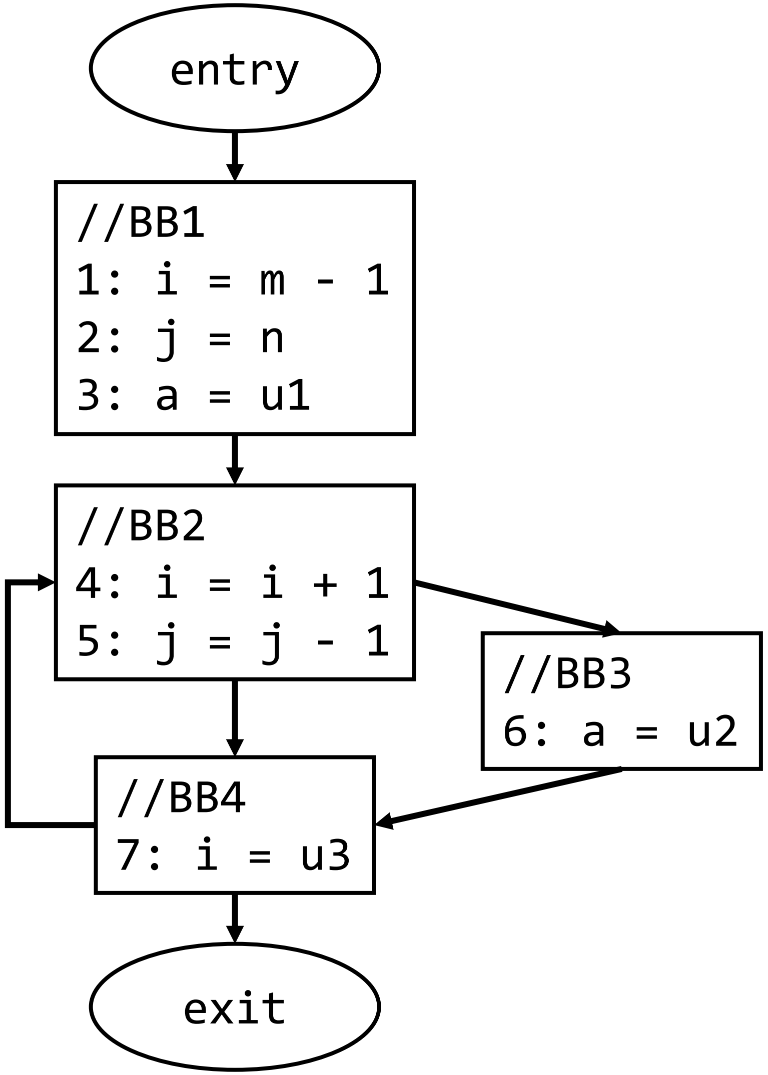
- **For reaching definition: it is a union of successors' IN**
  - OUT[b] = IN[s1] U IN [s2] … U IN[sn] (s1~sn are successors of b)
  - A variable is live if at least a single path as the variable live

- **To support cyclic graphs, we need repeated computation**

Liveness Analysis Algorithm

```
// Input : control flow graph = (N, E, Entry, Exit)
// Initialize
IN[exit] = {}
for all nodes i: IN[i] = {}
worklist = {1 … N}
// Iterate
while worklist != empty {
      pop i from worklist
      OUT[i] = U(IN[s]) // Union of all successors
      IN_Prev = IN[i]
      IN[i] = USE[i] U (OUT[i] – DEF[i])
      if (IN_prev != IN[i]) {
            for all predecessors p of i
                  add p to worklist
      }
}
```

Class Exercise



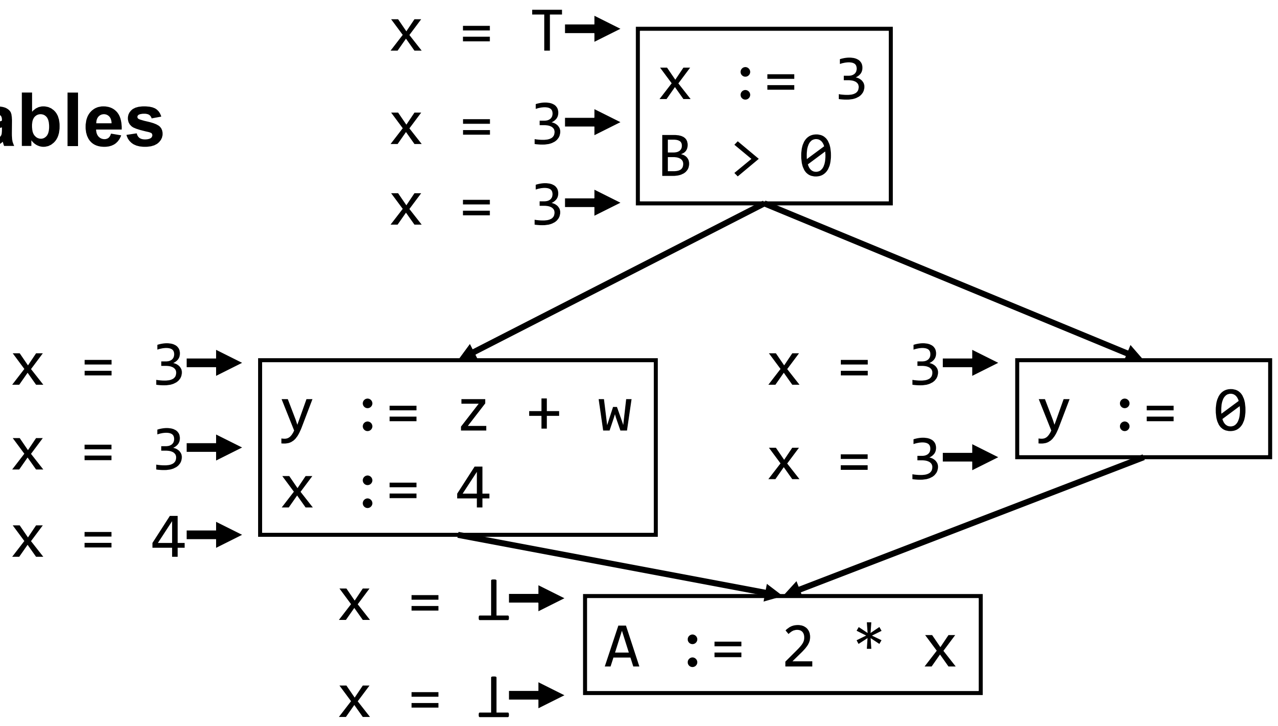
	USE	DEF
BB1	{m,n,u1}	{i,j,a}
BB2	{i,j}	{i,j}
BB3	{u2}	{a}
BB4	{u3}	{i}

```
OUT[BB4]    = {}             → {i,j,u2,u3}
IN[BB4]     = {u3}           → {j,u2,u3}
OUT[BB3]    = {u3}           → {j,u2,u3}
IN[BB3]     = {u2,u3}        → {j,u2,u3}
OUT[BB2]    = {u2,u3}        → {j,u2,u3}
IN[BB2]     = {i,j,u2,u3}    → {i,j,u2,u3}
OUT[BB1]    = {i,j,u2,u3}    → {i,j,u2,u3}
IN[BB1]     = {m,n,u1,u2,u3} → {m,n,u1,u2,u3}
```

Complex Example: Constant Propagation

- **Constant propagation determines whether we can convert a variable to a constant at a point**

- **There are three types of variables**
  - x = T (top, don't know)
  - x = c (constant, x = c)
  - x = ⊥ (bottom, not a constant)



Transfer Function

- **There are necessary functions in constant propagation**
  - IN/OUT[b][x]: the value of the variable x before and after b
  - SET[b]: the list of variables (x) and assigned values (c) in b
    - If x is assigned a non-constant: the value is T
- **We need more complex transfer function**

```
OUT[b] = IN[b]
for SET[b]:
     // SET[b] has the form
     // x = e(w1, w2, …, wn)
     if any wi == ⊥: OUT[b][x] = ⊥
     elif any wi == T:     OUT[b][x] = T
     else:                 OUT[b][x] = eval result (constant)
```

Meet Operator

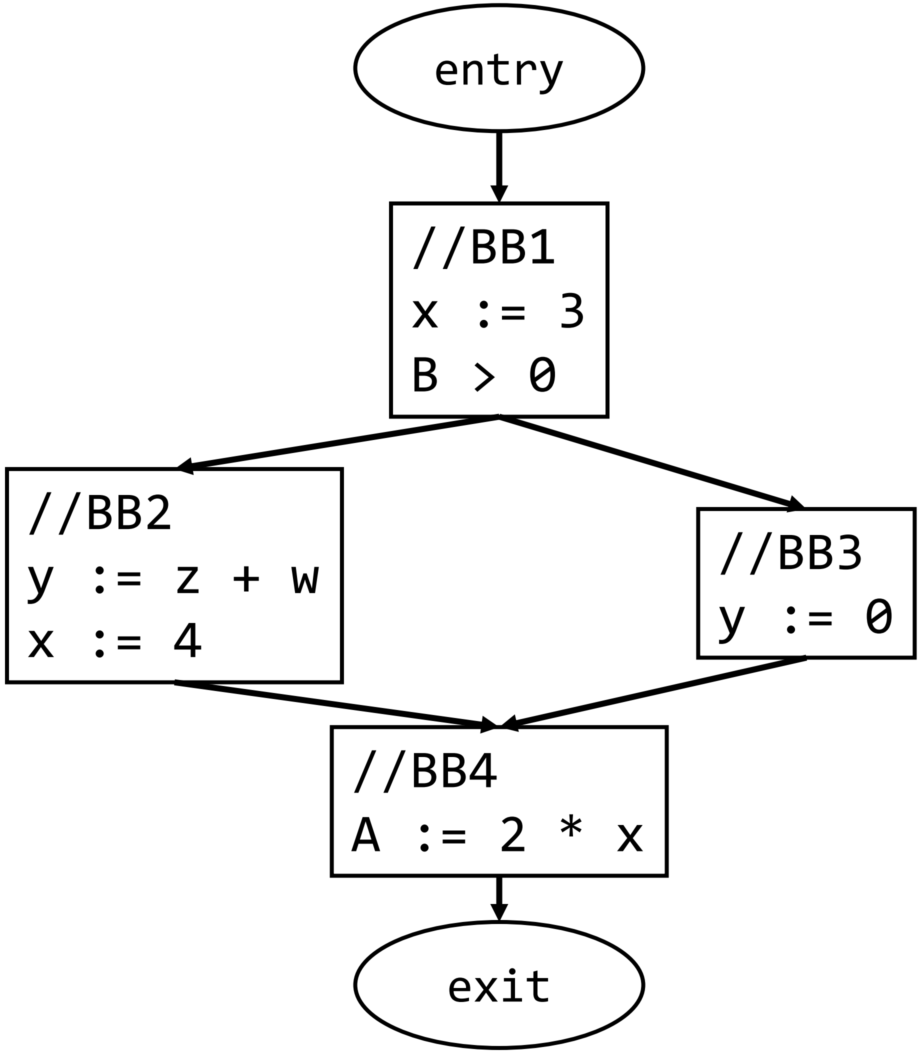
- **We also need a more complex transfer function for the edge**

```
IN[b] = OUT[p1]
for (pk in p2 … pn)
     for (x, v in OUT[pk])
          if v == T:                    continue
          else if v == ⊥:               IN[b][x] = ⊥
          else if v == IN[b][x]:        IN[b][x] = v
          else if v != IN[b][x]:        IN[b][x] = ⊥
```

Constant Propagation Algorithm

```
// Input : control flow graph = (N, E, Entry, Exit)
// Initialize
OUT[Entry] = {x = b for each variable x}
for all nodes i: OUT[i] = {x = b for each variable x}
worklist = {1 … N}
// Iterate
while worklist != empty {
      pop i from worklist
      IN[i] = transfer_edge(OUT[p]) // Union of all predecessors
      OUT_Prev = OUT[i]
      OUT[i] = transfer_block(IN[i])
      if (OUT_prev != OUT[i]) {
            for all successors s of i
                  add s to worklist
      }
}
```

Class Exercise




```
IN[BB1]      = {x = T, y = T, A = T}
OUT[BB1]     = {x = 3, y = T, A = T}
IN[BB2]      = {x = 3, y = T, A = T}
OUT[BB2]     = {x = 4, y = ⊥, A = T}
IN[BB3]      = {x = 3, y = T, A = T}
OUT[BB3]     = {x = 3, y = 0, A = T}
IN[BB4]      = {x = ⊥, y = ⊥, A = T}
OUT[BB4]     = {x = ⊥, y = ⊥, A = ⊥}
```