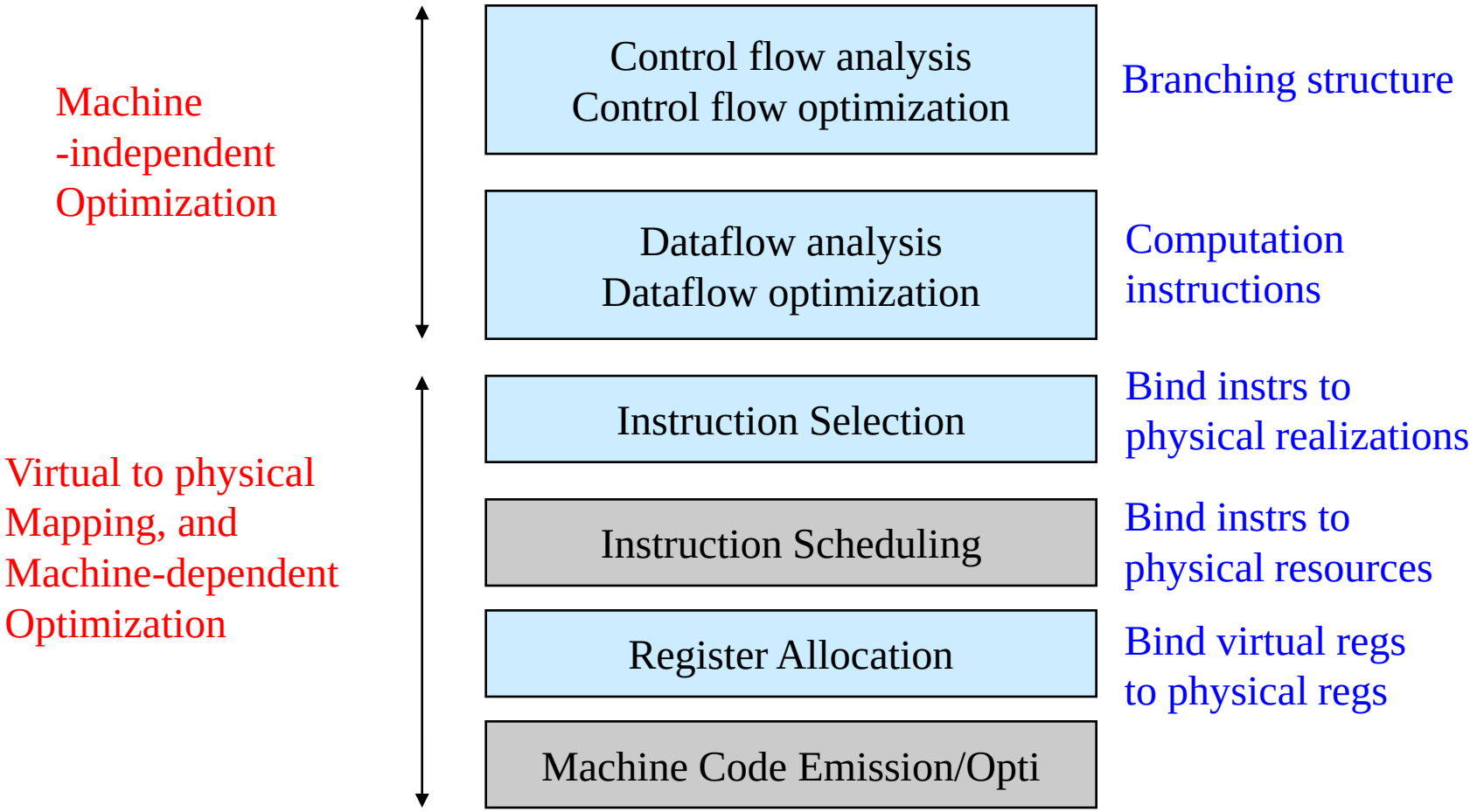
# Compiler (컴파일러)– Optimization and Analysis

2015년 2학기

충남대학교 컴퓨터공학과

조은선

# 컴파일러 후반부 (Backend)




**후반부 = 빠른 코드로 바꾸기 + 실제 머신에서 도는 코드로 바꾸기**
**(+ 코드 크기 줄이기, ...)**

# Optimization

- "최적화"
  - 주어진 입력 프로그램과 의미적으로 동등하면서 좀더 효율적인 코드로 바꾸는 것
    - 효율적 : 실행시간이 짧고, 기억장소 요구량이 최소화됨
- 방법
  - 가급적 계산의 횟수를 줄이고,
  - 보다 빠른 명령을 사용
  - **분석**을 이용하고..
- 종류 – 여러 가지 분류 방법
  - Control flow analysis vs. data flow analysis
  - Inner basic block (local) opt. vs. inter basic block (global) opt.
  - cyclic (loop) code opt. vs. acyclic opt.

# Control Flow Analysis

# Control Flow

- Control flow
  - 프로그램의 가능한 수행순서
  - 관심 있는 것은 pc=pc+1 외에 pc의 변동 상황

  ➔ 즉, branch (분기)!
- Execution ➔ dynamic control flow
  - branch (분기) 에서 어느 쪽으로 가는지에 대한 것
  - 동적인 예측 : “profiling”
- Compiler ➔static control flow
  - input 을 모르는 상황, 정확한 예측은 불가하고,
  - worst case를 생각해서 예측

# Control Flow Analysis

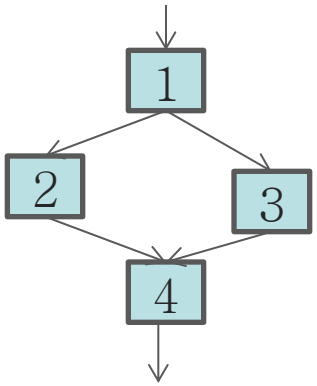
- 정적 성질 (static property)
  - 프로그램 수행 없이, 즉 수행 중 실제 분기 방향에 관계없이 도출되는 성질
- CFA (Control Flow Analysis)
  - 코드의 분기 구조를 수행 전에 (컴파일 단계에서) 분석 하여 CFG를 만들고
  - 정적 성질을 도출하여
  - 코드를 최적화 하는 것이 목적

# Basic Blocks (BB)

- 의미 : 동일한 execution condition을 적용 받는 instruction 묶음
- 정의 : 다음 조건을 만족하는 일련의 instructions
  - 제어 흐름이 시작 instruction 으로 부터만 시작되고 끝 instruction 에서만 밖으로 나감
  - 끝 instruction 외에는 분기가 없음
- 성격 :
  - 프로그램상에서 일렬로 나타나는 instruction 들만 한 BB 내에 있음
  - BB 내의 한 instruction이 수행된다는 것은 모두 수행을 의미

# 예) gcc

```
#include <stdio.h>
int main(){
        int i;
        printf("> ");
        scanf("%d", &i);
        printf("₩n ");

        if (i > 0)
                printf("Hello World.₩n");
        else
                printf("Hi World.₩n");
        return 1;
}
~
"tt.c" 13L, 174C
```




**tt.c**

**tt.c.13.cfg**
after **gcc –da tt.c**

```
4 basic blocks, 6 edges.

Basic block 1 prev -1, next 2, loop_depth 0, count 0, freq 0.
Predecessors:  ENTRY (fallthru)
Successors:  2 (fallthru) 3

Basic block 2 prev 1, next 3, loop_depth 0, count 0, freq 0.
Predecessors:  1 (fallthru)
Successors:  4

Basic block 3 prev 2, next 4, loop_depth 0, count 0, freq 0.
Predecessors:  1
Successors:  4 (fallthru)

Basic block 4 prev 3, next -2, loop_depth 0, count 0, freq 0.
Predecessors:  2 3 (fallthru)
Successors:  EXIT [100.0%]  (fallthru)

(note 2 0 4 NOTE_INSN_DELETED)
                                                    14,1          8%
```

# Basic Blocks (Cont' )

- Basic block 구하기
  - BB의 첫instruction (leader) 을 구하고
    - 프로그램의 시작 instruction
    - Branch의 target instruction
    - Branch 직후의 instruction
  - 다음 leader 이전까지의 모든 코드를 구한다
- 예) Leader: L1, L4,L10, L2,L7, (L10), L8

  BB : <L1> <L2,L3> <L4,L5,L6>

  <L7> <L8, L9> <L10, L11>

참고) "beq x y L"

: x가 y와 동일하면 L로 점프

```
L1: r7 = [r8]
L2: r1 = r2 + r3
L3: beq r1, 0, L10
L4: r4 = r5 * r6
L5: r1 = r1 + 1
L6: beq r1 100 L2
L7: beq r2 100 L10
L8: r5 = r9 + 1
L9: r7 = r7 - 3
L10: r9 = [r3]
L11: [r9] = r1
```

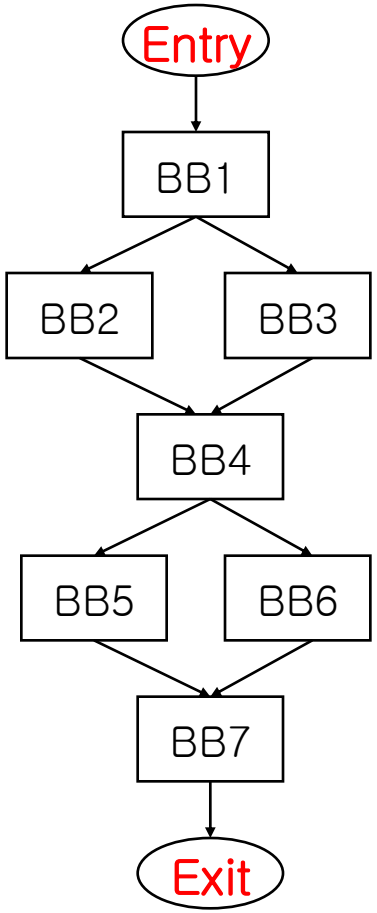
# Class Problem

- 다음 코드에서 basic block 들을 구하시오.(hint: for문을 if, goto 등을 써서 바꾸고 구하시오.)

```
for (i=0; i< N; i++)
      stmt1;
stmt2;
```

# Control Flow Graph (CFG)

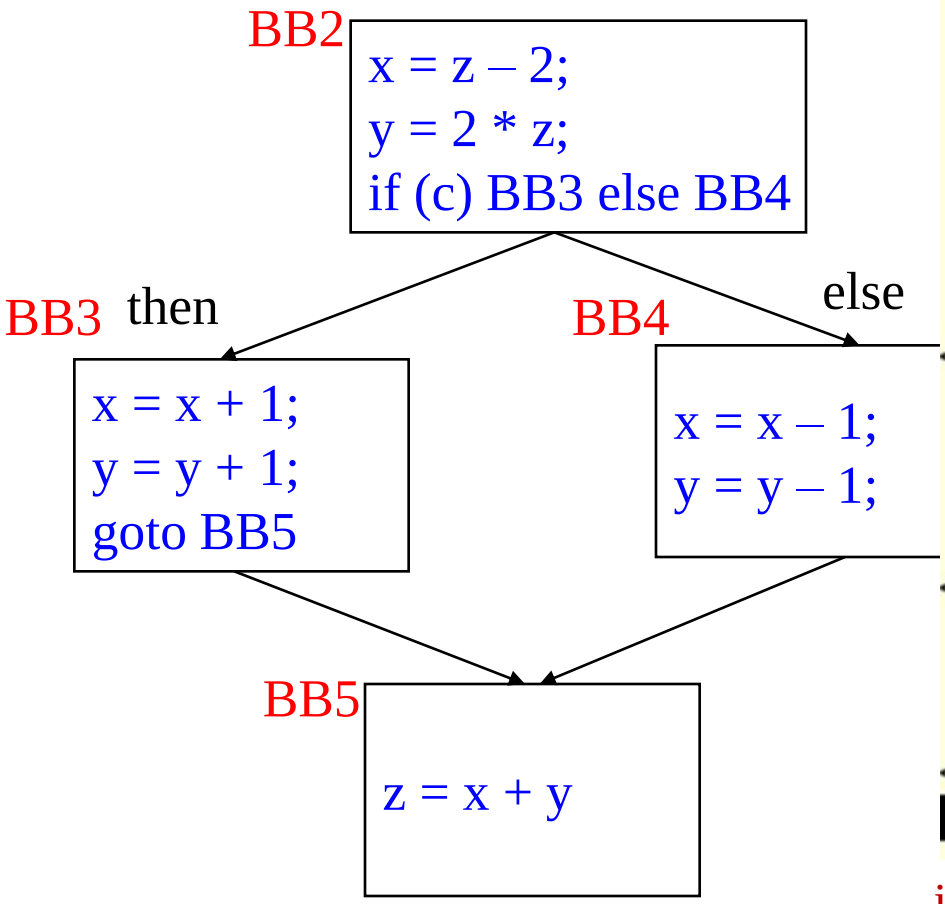
- Control Flow Graph (CFG)
  - "Node가 BB이고 edge가 이들의 수행 순서를 나타내는 그래프"
  - 분기 등으로 연결된 두 BB 간에는 edge 존재
- 사실, CFG는 컴파일러 마다 형태가 다양함.
- 그러나 대부분 2개의 추가 (virtual) 노드 도입
  - Entry node
  - Exit node



# 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
```

in <x.c >



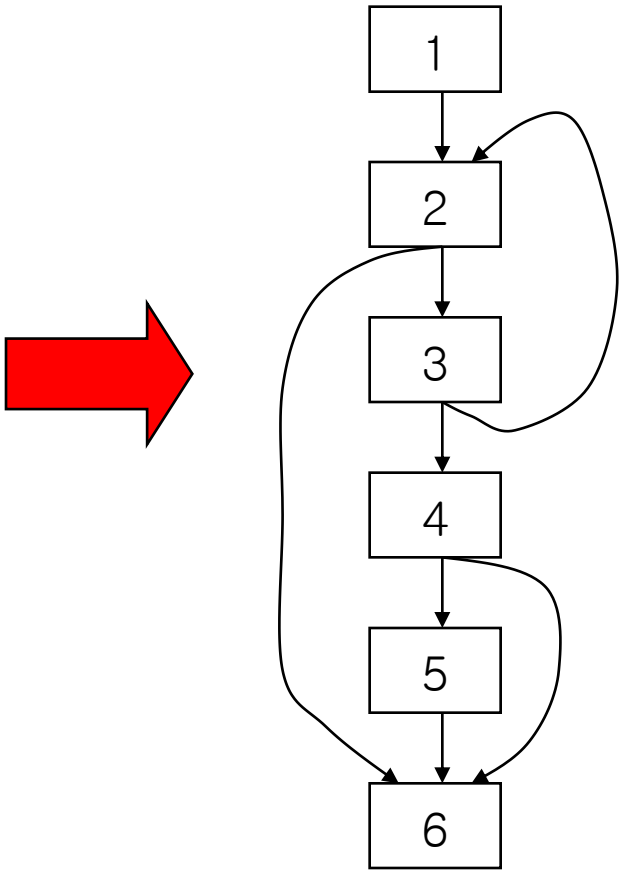

File Edit View

```
<bb 2>:
  c = 0;
  z = c;
  y = z;
  x = y;
  x = z + -2;
  y = z * 2;
  if (c != 0)
    goto <bb 3>;
  else
    goto <bb 4>;

<bb 3>:
  x = x + 1;
  y = y + 1;
  goto <bb 5>;

<bb 4>:
  x = x + -1;
  y = y + -1;

<bb 5>:
  z = x + y;
```

in <x.c.013t.cfg>

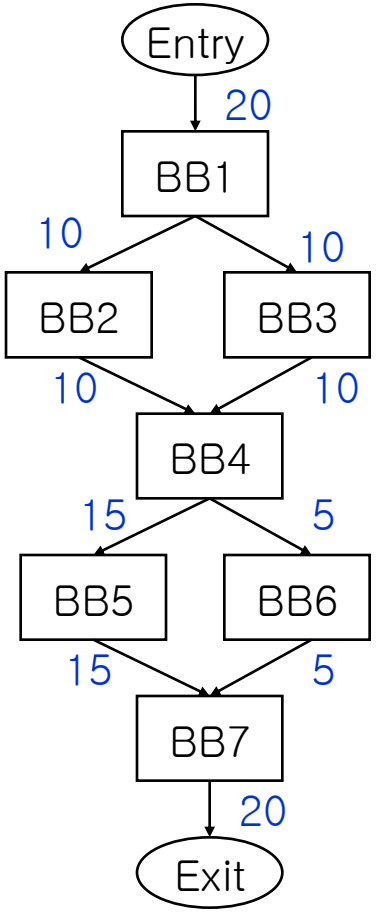
# CFG Example (more)

| | |
|---|---|
| 1 | L1: r7 = [r8] |
| 2 | L2: r1 = r2 + r3<br>L3: beq r1, 0, L10 |
| 3 | L4: r4 = r5 * r6<br>L5: r1 = r1 + 1<br>L6: beq r1 100 L2 |
| 4 | L7: beq r2 100 L10 |
| 5 | L8: r5 = r9 + 1<br>L9: r7 = r7 - 3 |
| 6 | L10: r9 = [r3]<br>L11: [r9] = r1 |



# Weighted CFG

- Profiling (프로파일링) – 몇 번 돌려보고 빈도등 결과를 얻음
- Control flow profiling
  - edge profile
  - block profile
  - path profiling
- Weighted CFG – CFG에 프로파일 결과를 annotation으로 붙인 것
- 자주 일어나는 상황에 대해 효과적인 optimization이 가능



# Acyclic Code Optimization

# Acyclic Code Optimization

- Acyclic Code
  - Loop 가 없는 코드
  - 분석 및 최적화가 상대적으로 쉬움

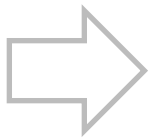
- 종류
  1. Inner basic block opt. = Intra opt. = Local opt.
     - 부분적인 관점 (단일 block 내) 에서 비효율적인 코드를 구분해 내고 좀더 효율적인 코드로 개선
  2. Inter-basic block opt. = Global opt.
     - basic block 간 관계를 분석하고 이를 고려하여 optimize

# Inner Basic Block Optimization

## 1. Common subexpression elimination

- 공통된 부분이 반복해서 나타나는 경우, 한 번만 계산

```
x = b+ c;
y = b+ c+ d;
z = (b+ c) * e;
```



```
x = b + c;
y = x + d;
z = x * e;
```

## 2. Algebraic simplication

- 수학적인 대수 법칙을 이용하여 식을 간소화
- x = y + 0; ➔ x = y; 또는 x =1*y; ➔ x = y;
- t1 = 4*j+ 1; t7 = t1-4*j; ➔ t1= 4*j+ 1; t7 =1;

## 3. Strength reduction

- 같은 의미를 가지면서 좀더 연산자의 비용이 적은 연산자로 바꾸는 것

a = b ** 2 ➔ a = b*b

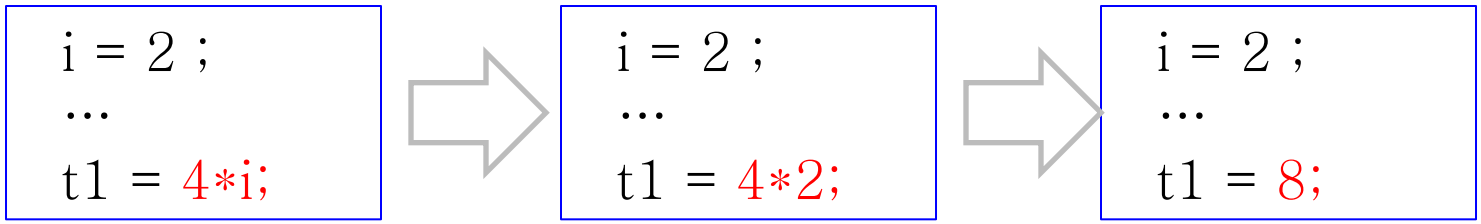
x = 3*a ➔ x = a+ a+ a

y = a/5 ➔ y = a*0.2

y = a/4 ➔ y = a shift-right 2

## 4. Constant folding / propagation

- folding : 컴파일 시간에 상수식을 직접 계산하여 그 결과를 필요한 곳에 사용
- propagation : 고정된 값을 가지는 변수를 상수로 대체

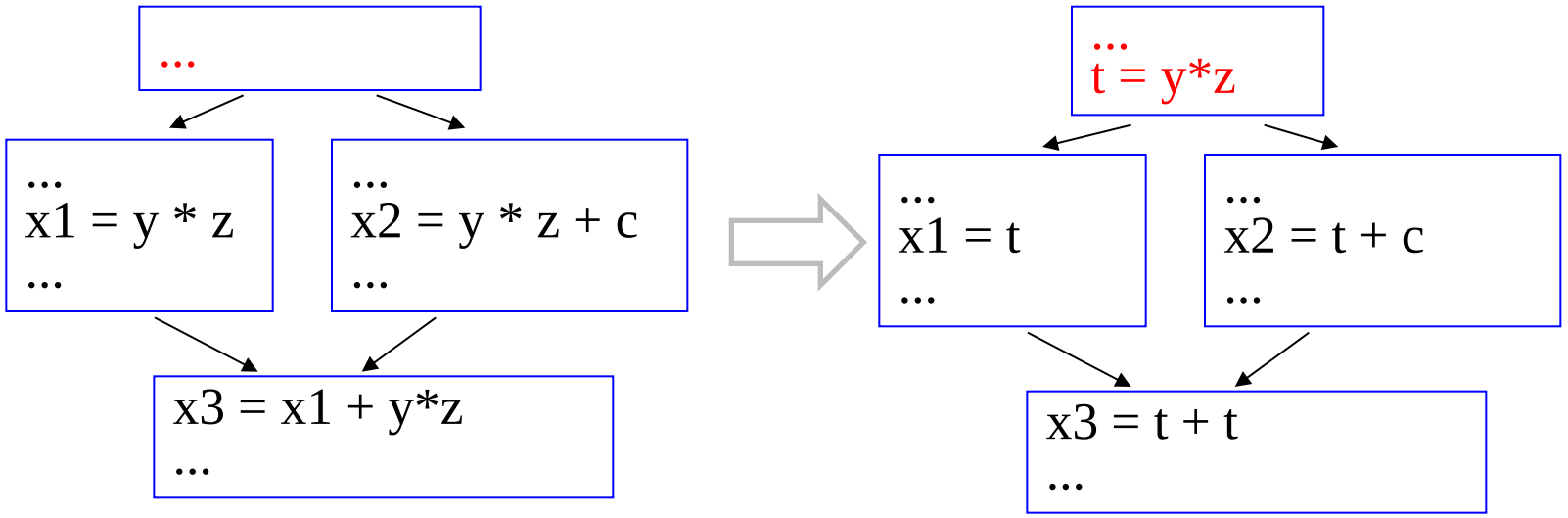


# Inter-Basic Block Optimization

1. Global application of inner-basic block optimization

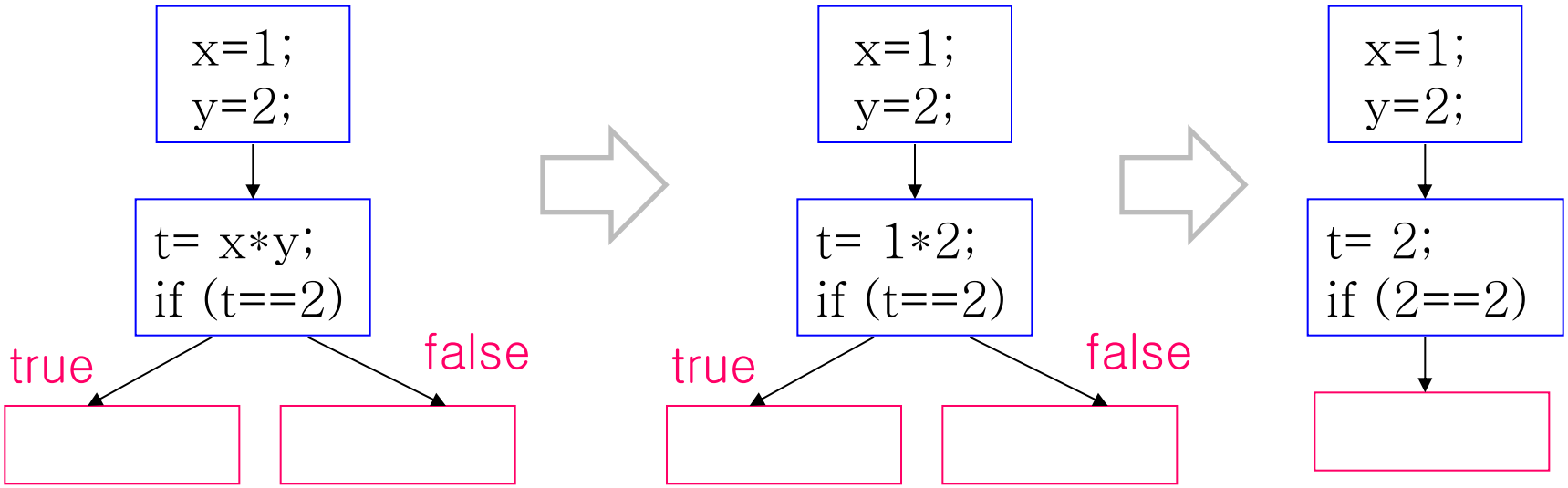
1-1. Global common subexpression elimination

- basic block 간에 나타나는 공통 부분식에 대해서도 한번만 계산



## 1-2. Global constant folding / propagation

- 변수의 정의와 사용이 서로 다른 block에 걸쳐 있는 경우에 대해서도 상수를 인식하여, basic block 간 상수 폴딩과 전파가 가능

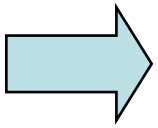


## 2. Other transformations

- Branch 숫자 줄이고, basic block 더 크게 만들기, 코드 길이 줄이기..

### 2-1. Branch to unconditional branch

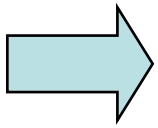
```
L1: if (a < b) goto L2
. . .
L2: goto L3
```



```
L1: if (a < b) goto L3
. . .
L2: goto L3  → may be deleted
```

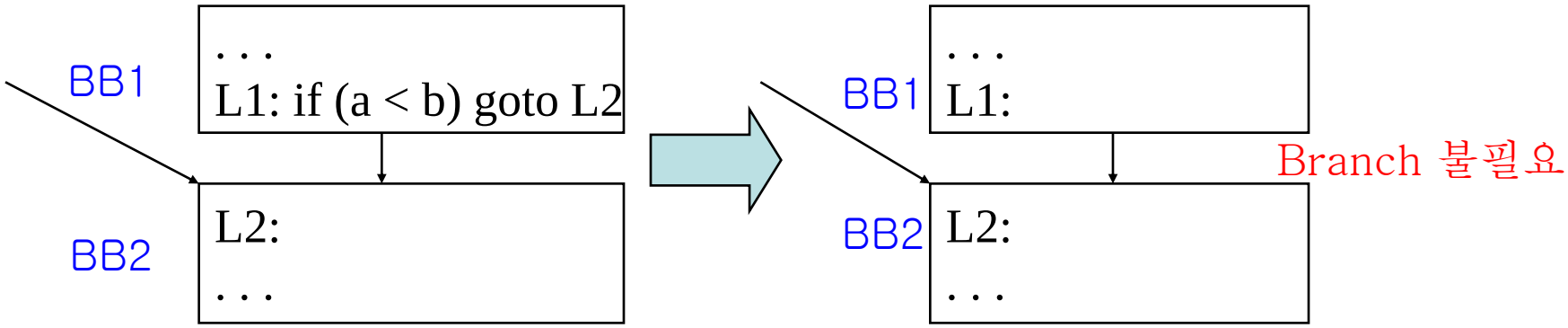
### 2-2. Unconditional branch to branch

```
L1: goto L2
. . .
L2: if (a < b) goto L3
L4:
```

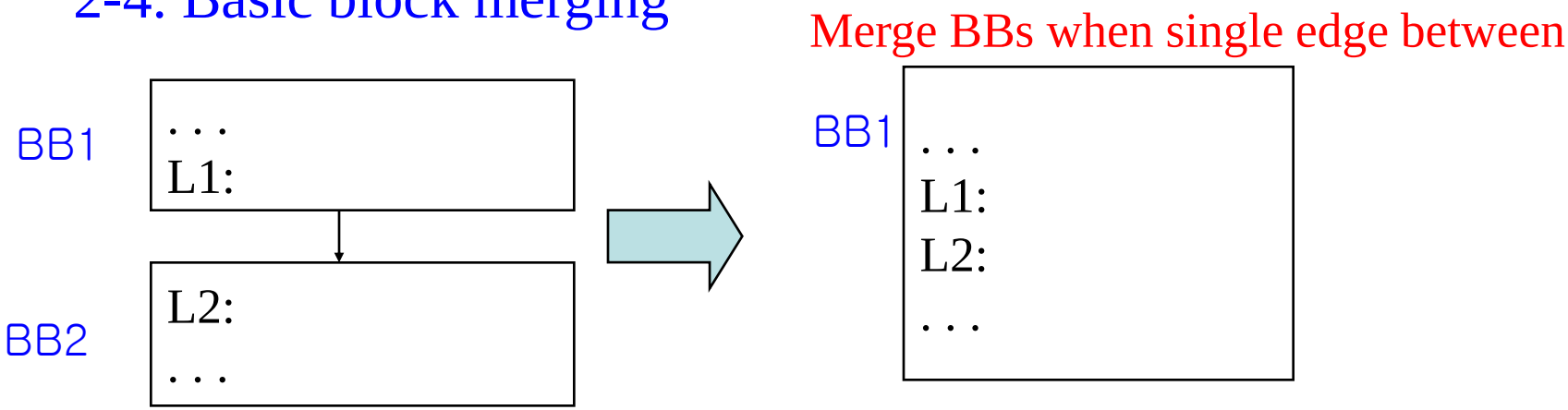


```
L1: if (a < b) goto L3
goto L4:
. . .
L2: if (a < b) goto L3  → may be deleted
L4:
```

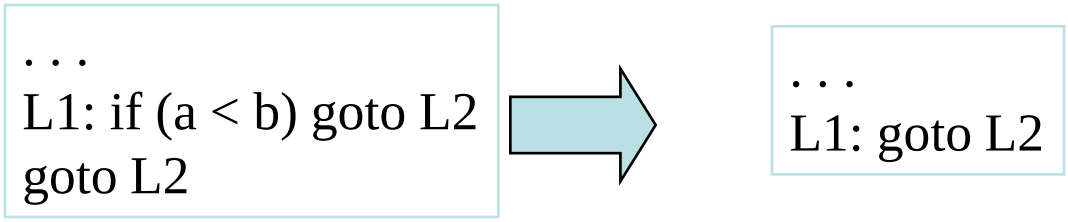
## 2-3. Branch to next basic block (next instr)



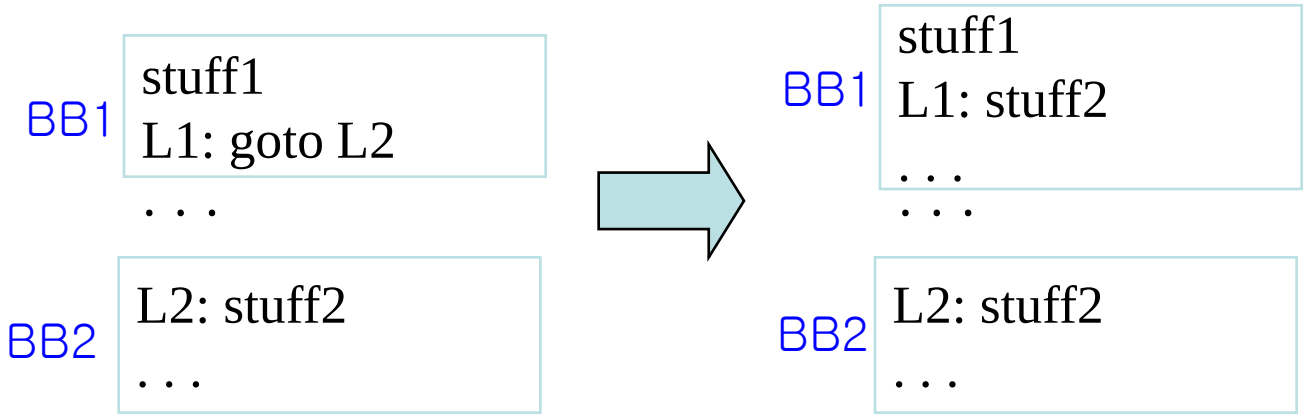

## 2-4. Basic block merging



## 2-5. Branch to same target



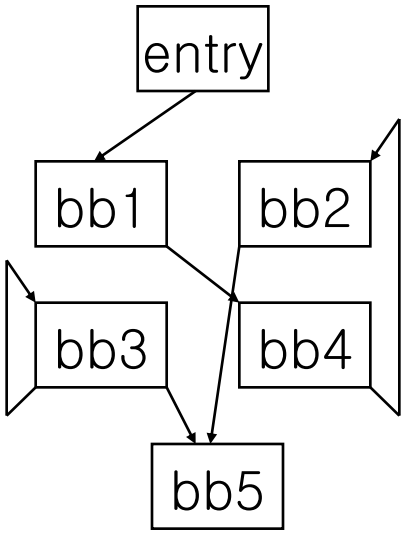

## 2-6. Branch target expansion




*Conditional Branch 의 경우도 동일*

# 2-7. Unreachable Code Elimination

- Entry 에서 도달할 수 없으면 'unreachable' block
- Algorithm
  - mark (e) {
    e.visited = true
    for all sucessors e' of e
    mark(e')
    }
  - for all blocks e
    e.visited = false
  - Call mark(entry)
  - Find unmarked blocks and eliminate them




*Which BB(s) can be deleted?*

# Class Problem

주어진 프로그램을 최대한 optimize 해보시오.

| |
|---|
| L1: if (a < b) goto L11 |
| L2: goto L7 |
| L3: goto L4 |
| L4: stuff4<br>L5: if (c < d) goto L15 |
| L6: goto L2 |
| L7: if (c < d) goto L13 |
| L8: goto L12 |
| L9: stuff 9<br>L10: if (a < c) goto L3 |
| L11:goto L9 |
| L12: goto L2 |
| L13: stuff 13<br>L14: if (e < f) goto L11 |
| L15: stuff 15<br>L16: rts |

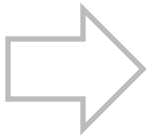
# Loop Detection and Analysis

# Loop Optimization

- Loop 는 한번 optimize 해두면 효과가 큼
  - 각종 optimization 을 loop basic block 들에 하면 효과가 크다.
  - common subexpr elim., algebraic simp., strength red. ...
- 그외 Loop 고유의 opt. 기법

  1. Loop unrolling
     - loop body를 펼쳐서 반복 회수 branch 등을 줄임, 다른 opt. 기회도 높임
     - loop body가 적을 때 특히 유용

```
i=0;
while (i<1000) {
   z[i]= 0;
   i++;
}
```

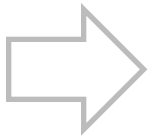


```
i=0;
while (i<1000) {
    z[i]= 0;
    z[i+1] = 0;
    i+=2;
}
```

## 2. Loop invariant

- 매번 동일한 값을 내는 문장은 loop밖으로 이동

```
for (i=0 ; i < 1000 ; i++)
        z[i]= cos(x) + i*sin(x);
```

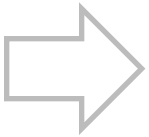


```
c =cos(x); s=sin(x);
for (i=0; i < 1000 ; i++)
        z[i]=c+i*s;
```

## 3. Count up to zero

- loop의 종료 조건을, 동일한 의미를 가지면서 0과 비교하는 식으로 바꾸기

```
for (i=1; i <= n ; i++)
        ...
```



```
for (i:= n ; i > 0; i--)
        ...
```

# Loop Detection

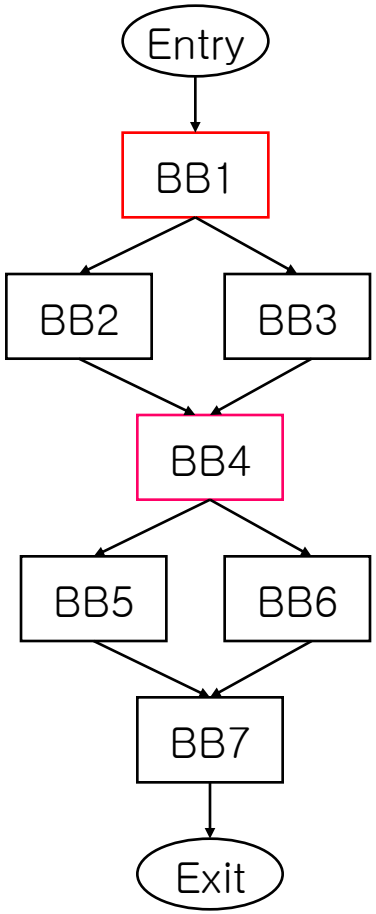
- 'Natural Loop' : 성질
  - 하나의 entry point block인 'header'가 하나 있고
    - Header는 loop 내의 다른 block 들을 dominate
  - 되돌아가는 edge인 'backedge' 가 적어도 하나는 존재
    - Loop를 이루려고 하다 보니
- Backedge detection
  - Edge n→d 가 d (target)가 n(source)를 dominate 할 때 backedge 임.

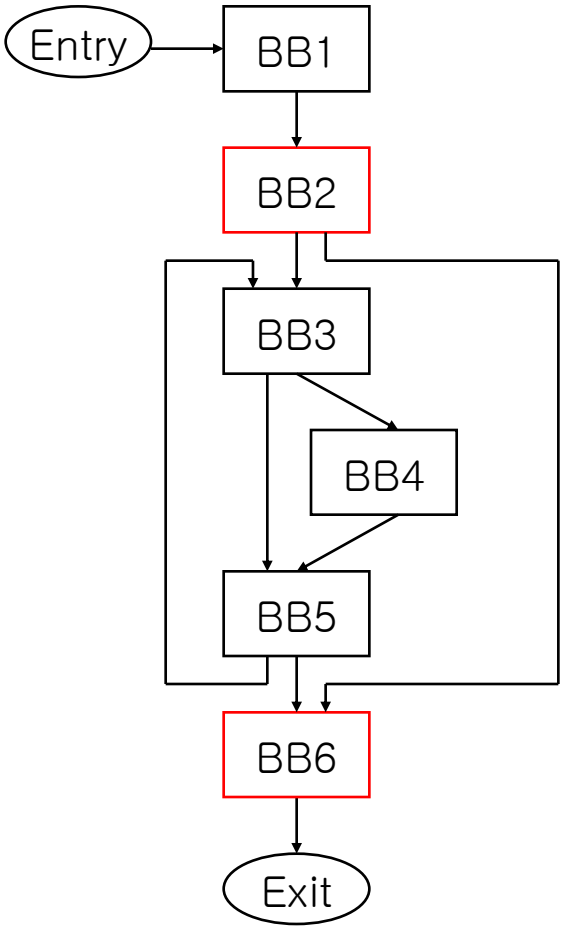
# Dominator

- Dominator
  - “CFG 에서 Entry 에서 node y까지 가는 모든 path에서 node x를 거치게 된다면 x가 y의 dominator 라고 함”
- “x dominates y” 의 의미

  ➔ “y 수행 전에는 반드시 x 수행된다.”
- 성질
  - 자기자신은 항상 dominate함
  - x dominate y이고 y dominate z 면 x가 z를 dominate.
  - x, y 둘다 z를 dominate 한다면, x dominate y 던지 y dominate x 임

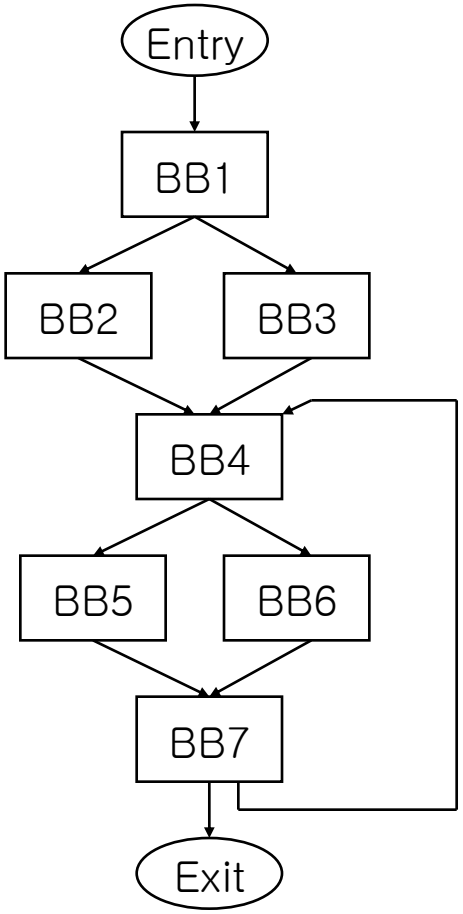
# Dominator Examples



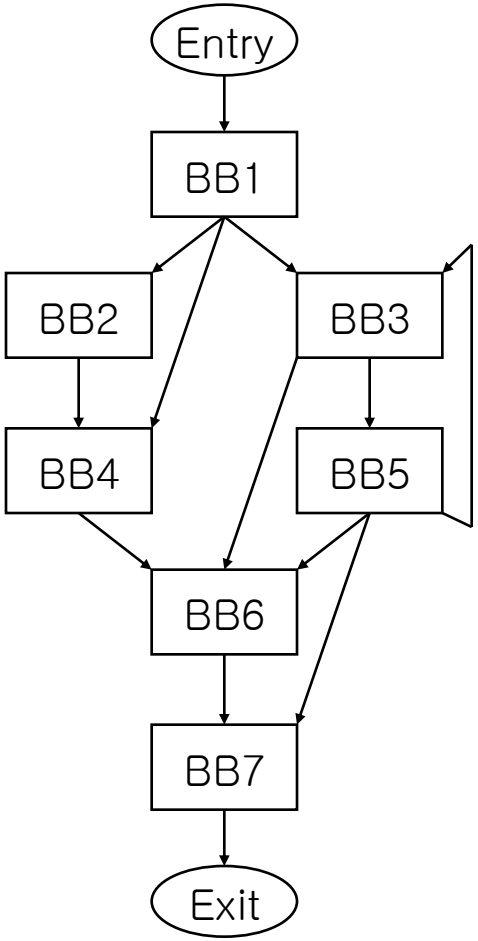
# Dominator Analysis

- dom(BBi) = set of BBs that dominate BBi
- 초기화
  - dom(ENTRY) = ENTRY
  - dom(everything else) = all nodes
  - change = true
- Iterative computation
  - while change, do
    - change = false
    - for each BB (except ENTRY)
      - tmp(BB) = BB + {모든predecessor의 dom의 교집합}
      - if (tmp(BB) != dom(BB))
        - » dom(BB) = tmp(BB)
        - » change = true



# Class Problem

각 BB 마다 dom을 구하라.



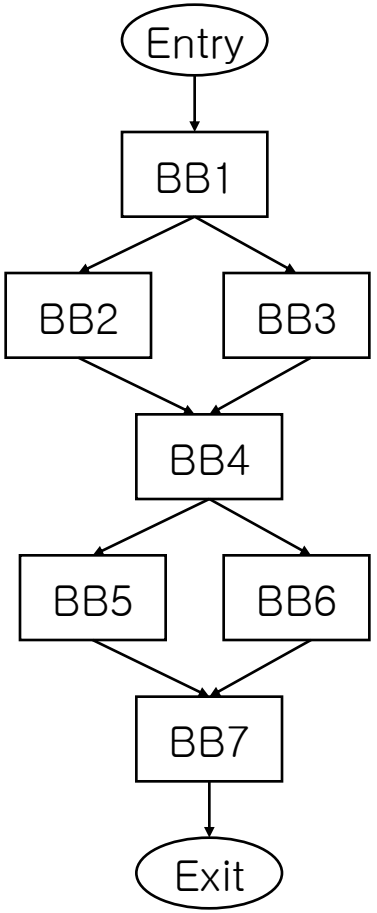
# Dominators과 optimization

- Optimization 범위
  - Global optimization : inter (inner) BB
  - Local optimization : intra BB
- Dominator의 용도
  - 대부분의 global optimization에 이용

예)

❖ Loop detection !

❖ Redundant computation : dominator 에서 계산된 것이 그대로 수행되면 (그리고 그사이 안 변한다면) redundant 하므로 제거



# Loop Detection (복습)

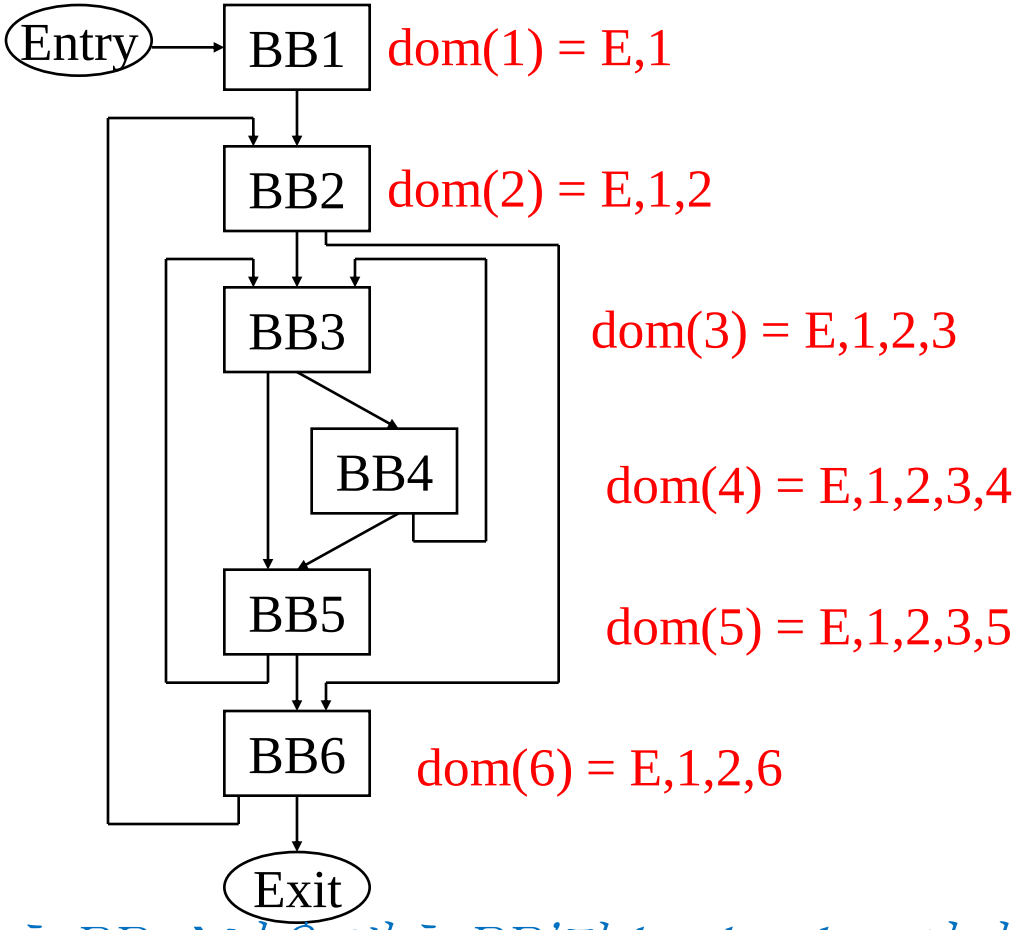
- 'Natural Loop' : 성질
  - 하나의 entry point block인 'header'가 하나 있고
    - Header는 loop 내의 다른 block 들을 dominate
  - 되돌아가는 edge인 'backedge' 가 적어도 하나는 존재
    - Loop를 이루려고 하다 보니

- Backedge detection
  - Edge n→d 가 d (target)가 n(source)를 dominate 할 때 backedge 임.

# Backedge Example

*Target dominates Source?*

E → 1 : No
1 → 2 : No
2 → 3 : No
2 → 6 : No
3 → 4 : No
3 → 5 : No
4 → 3 : Yes
4 → 5 : No
5 → 3 : Yes
5 → 6 : No
6 → 2 : Yes
6 → X : No




*예에서는 단순히 '높은 번호 BB → 낮은 번호 BB'면 back edge 이나, 항상 이렇게 쉽진 않다.*

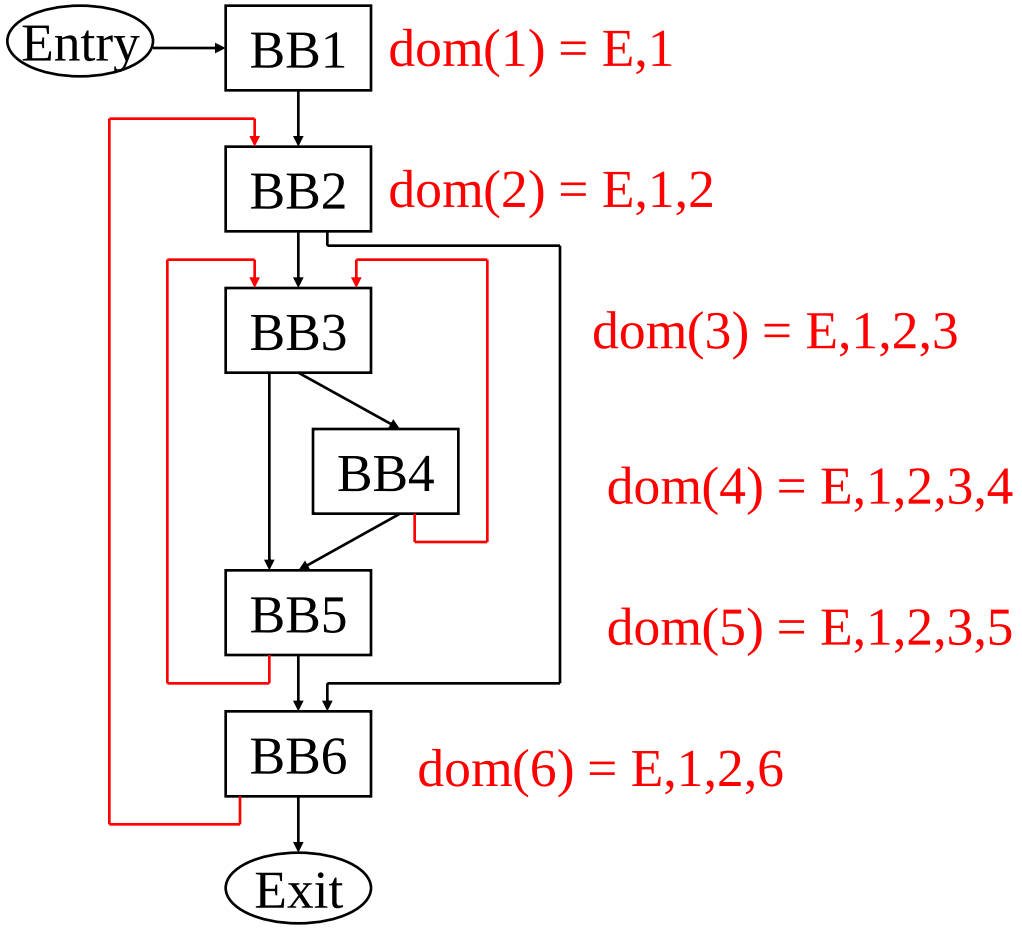
# Loop Detection Steps

- Natural Loop를 인식하는 단계
  1. Dominate 관계를 구하고,
  2. Backedge (n→d) 를 찾음
     - Loop Header는 d
  3. Loop Basic Block 찾음
     - Loop를 이루는 basic block : n, d 는 포함
     - d + {x | d 가 dominate x 하고, and
       d를 안 거치고 x 로부터 n로 가는 path가 존재}
  4. 공통 header loop는 하나로 취급
     - Loop Backedge = LoopBackedge1 + LoopBackedge2
     - LoopBB = LoopBB1 + LoopBB2
     - Header는 (여전히) 모든 LoopBB 을 dominate

# Loop Detection Example

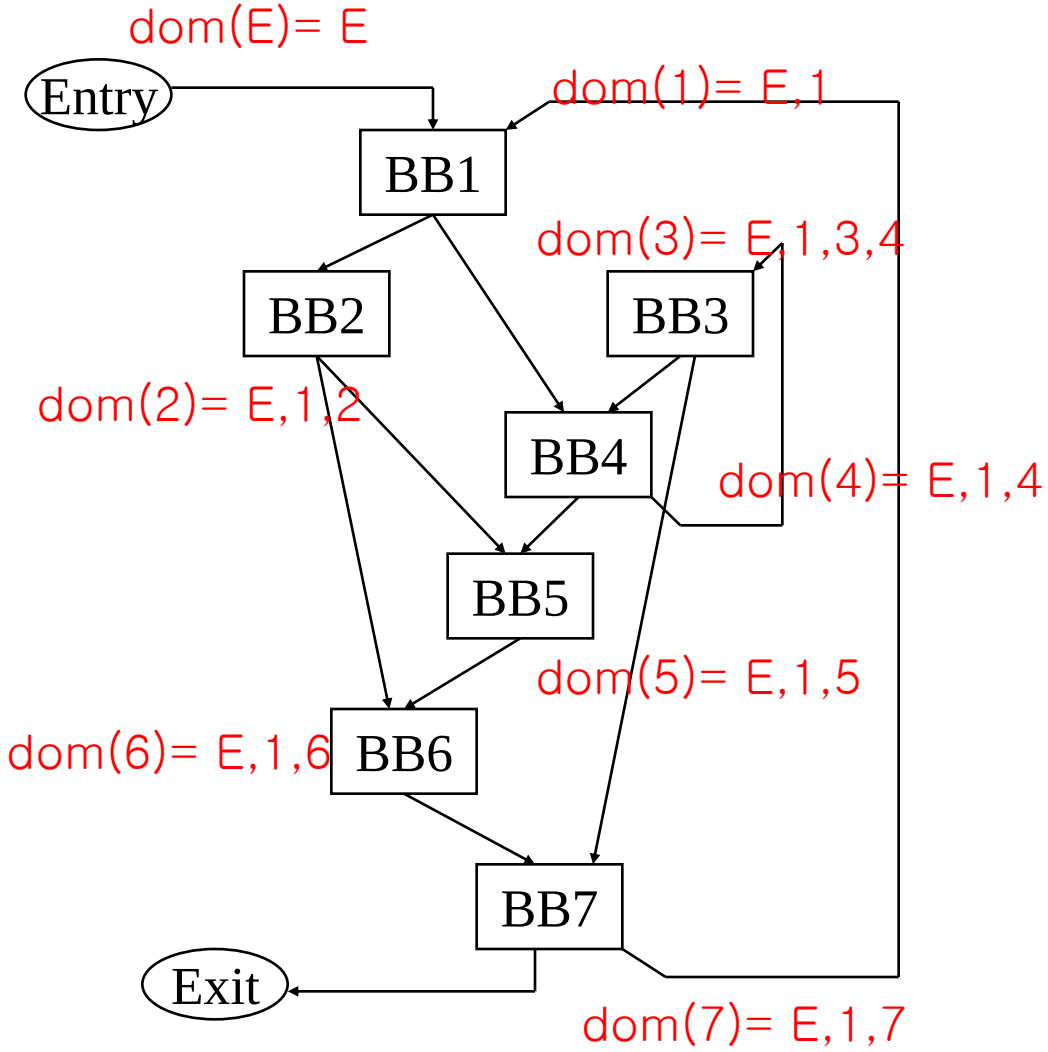
Loop1: defined by 6 → 2
  LoopBB = 2,3,4,5,6
Loop2: defined by 4 → 3
  LoopBB = 3,4
Loop3: defined by 5 → 3
  LoopBB = 3,4,5

Merge loops 2,3
  LoopBB = 3,4,5
  Backedges = 4→3, 5→3



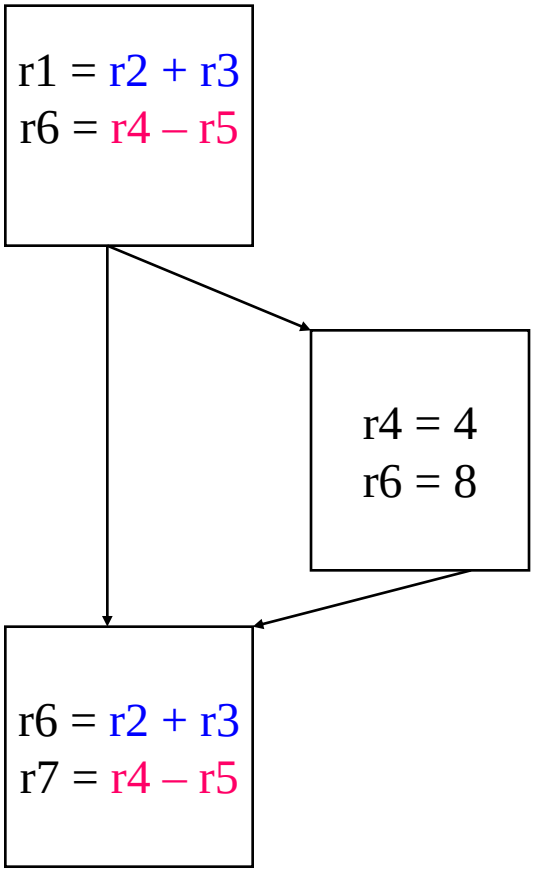
# Class Problem

Natural Loop 를 찾으시오.
(backedge 와 header, loop basic block)



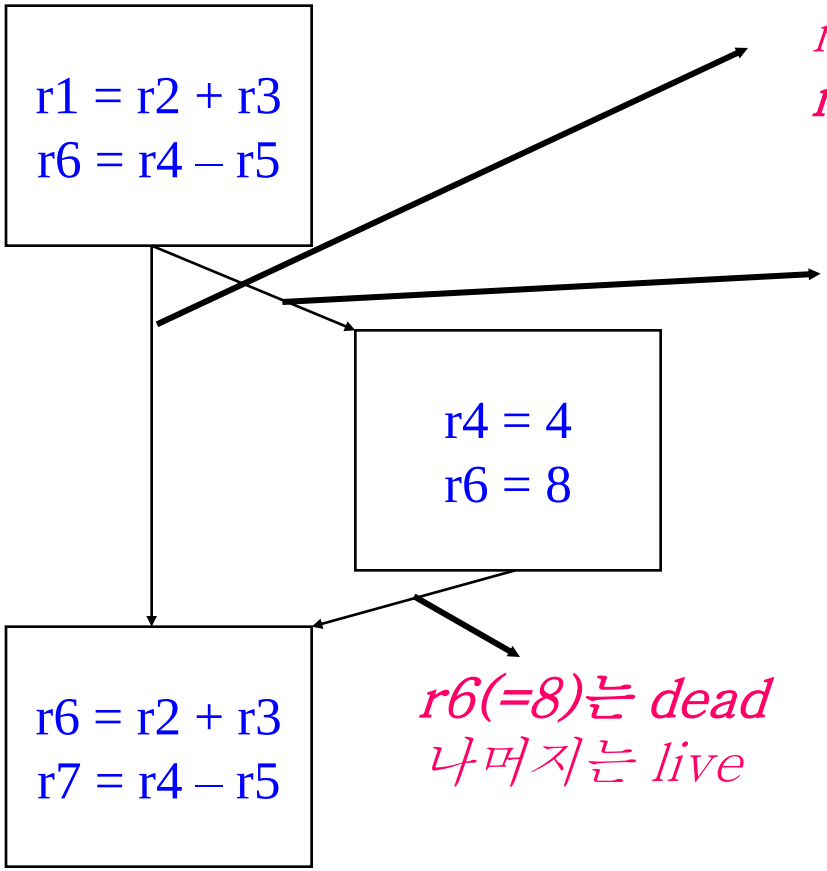
# Dataflow Analysis

# Dataflow Analysis + Optimization




- Classical optimizations 예
  - Common subexpression elimination (공통부분식 제거)
    - r2 + r3?  r4 - r5?
  - Register allocation
    - Virtual register 인 r1과 r7을 같은 actual register 에 저장해도 될까?
- Control flow analysis
  - Basic block 을  black box로 보고, branch 만 생각
- Data flow analysis
  - 프로그램 내에 각 data 값들이 생성/소멸되는 정보를 모으는 것
  - Basic block 내의 operation 을 봄

# Live Variable (Liveness) Analysis




*r1,r2,r3,r4,r5는 live*
*r6 (=r4-r5) 은 dead*

*r4, r6(=r4-r5)는 dead*
*나머지는 live*

*r6(=8)는 dead*
*나머지는 live*

- Live variable analysis : 현재 (특정 프로그램 point에서) 변수 값이 미래에 사용될 것인지를 알아보는 것

*결론 : "r6 = r4 – r5 는 필요 없다!"*

# Live Variable Analysis – Algorithms

다음과 같은 데이터를 모으는 것

- USE/DEF 한 block 내
  - USE = set of external *variables* consumed in the BB
  - DEF = set of variables defined in the BB
- IN/OUT block들 간
  - IN = set of variables that are live at the entry point of a BB
  - OUT = set of variables that are live at the exit point of a BB
- *note)*
  - *IN/OUT 은 backward 로 계산하면 좋음*
  - *IN/OUT 은 USE/DEF 를 써서 정의되며*
  - *BB 진입시 IN 외의 register는 free 시키고, BB 나갈 때는 OUT 외에 register 는 free 시키면 좋음.*

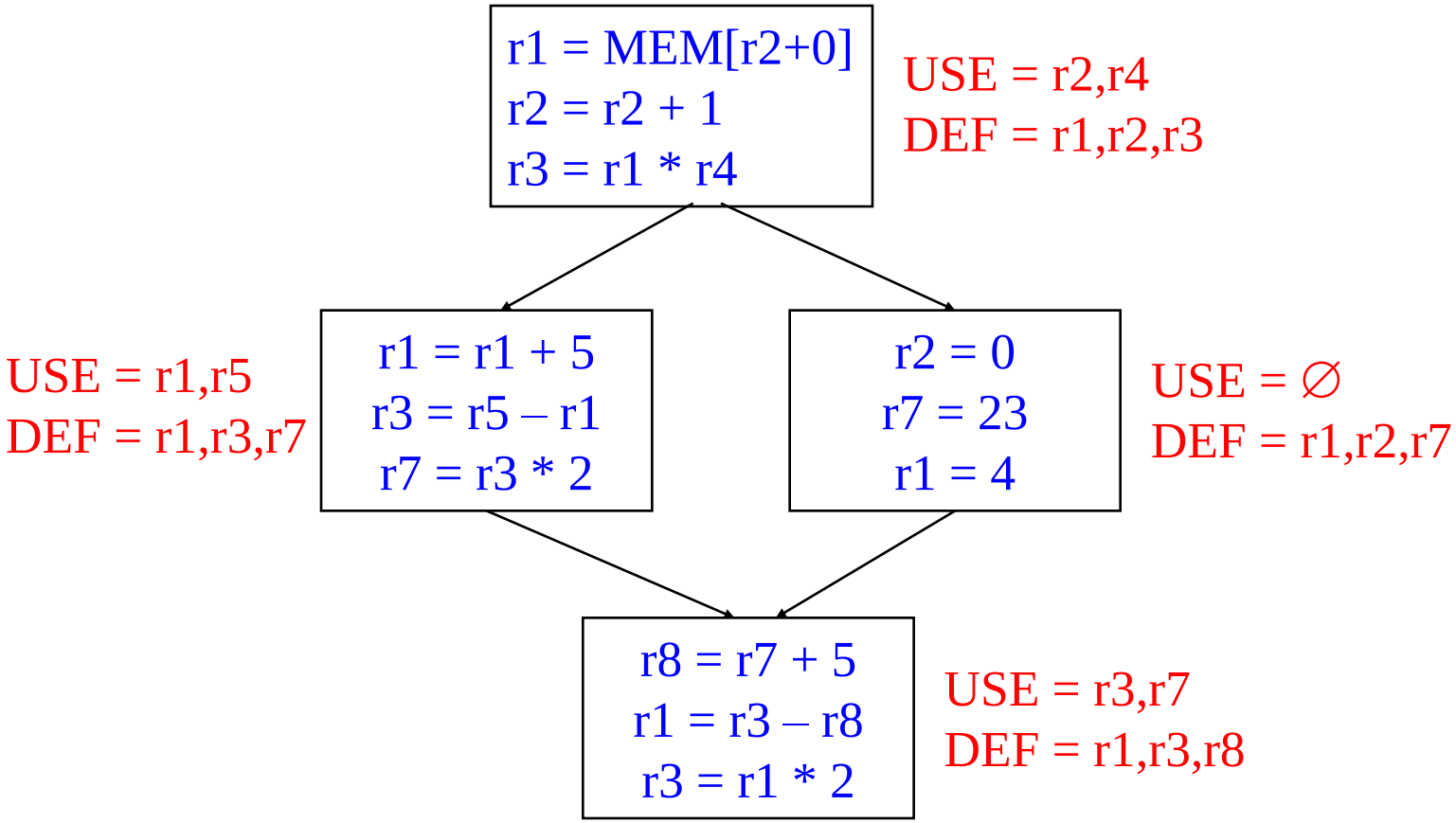
# Compute USE/DEF Sets For *Each* BB

```
for each basic block X, do
    DEF(X) = ∅
    USE(X) = ∅
    for each operation in sequential order in X, op, do
        for each source operand of op, src, do
            if (src not in DEF(X)) then
                USE(X) += src
            endif
        endfor
        for each destination operand of op, dest, do
            DEF(X) += dest
        endfor
    endfor
endfor
```

*dest : = $src_1$+$src_2$*

- DEF : block내 모든 LHS들
- USE : 바깥에서 들어온 값이 *block내에서 새로 정의되기 전에* 사용된 것들

# Example DEF/USE Calculation

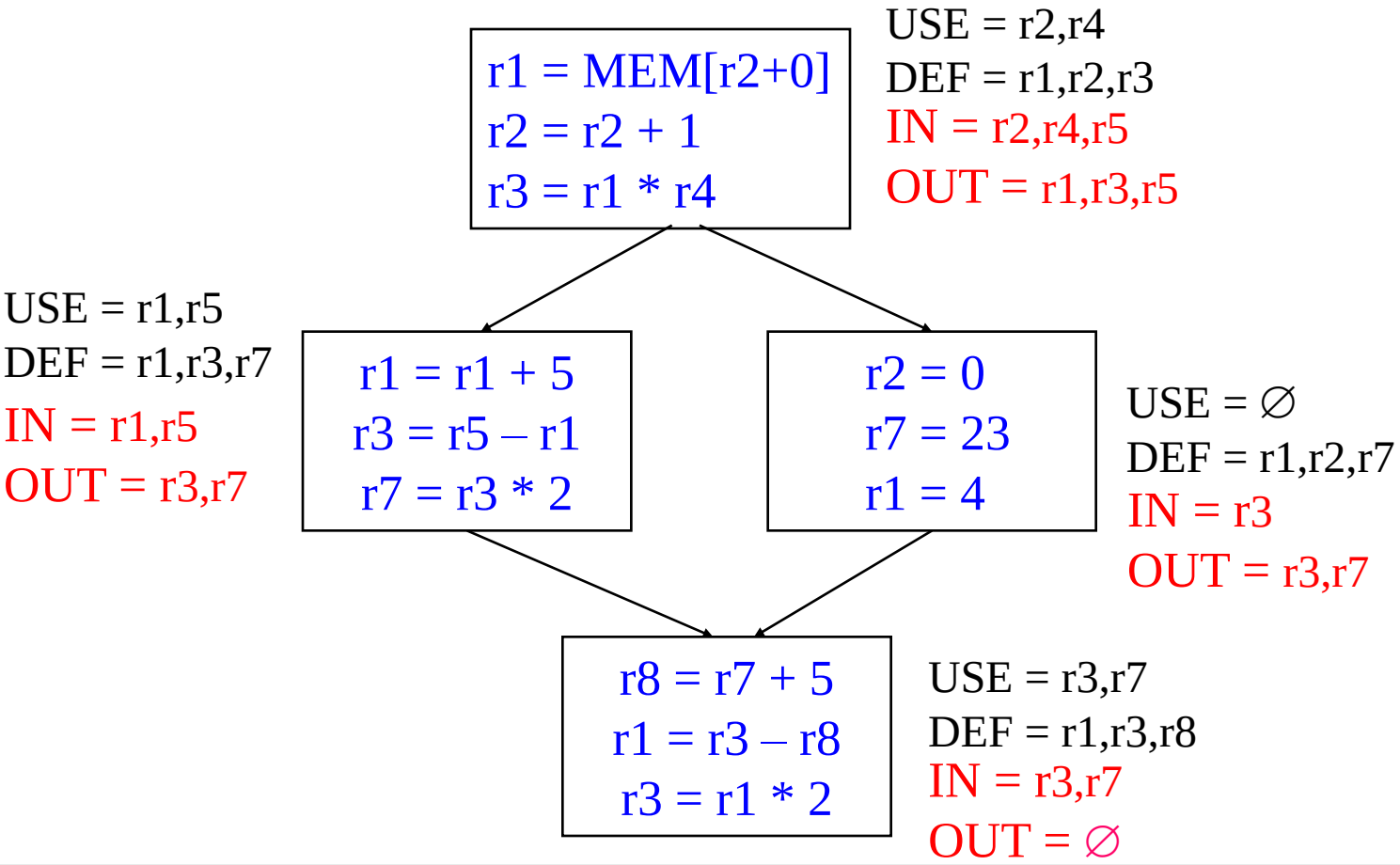


# Compute IN/OUT Sets For *All* BBs

initialize IN($X$) to ∅ for all basic blocks $X$
*change* = 1
while (*change*) do
    *change* = 0
    for each basic block, $X$, do
        old_IN = IN($X$)
        OUT($X$) = $\cup_{Y \in \text{successor}(X)}$ IN($Y$)
        IN($X$) = USE($X$) + (OUT($X$) – DEF($X$))
        if (old_IN != IN($X$)) then
            *change* = 1
        endif
    endfor
endfor

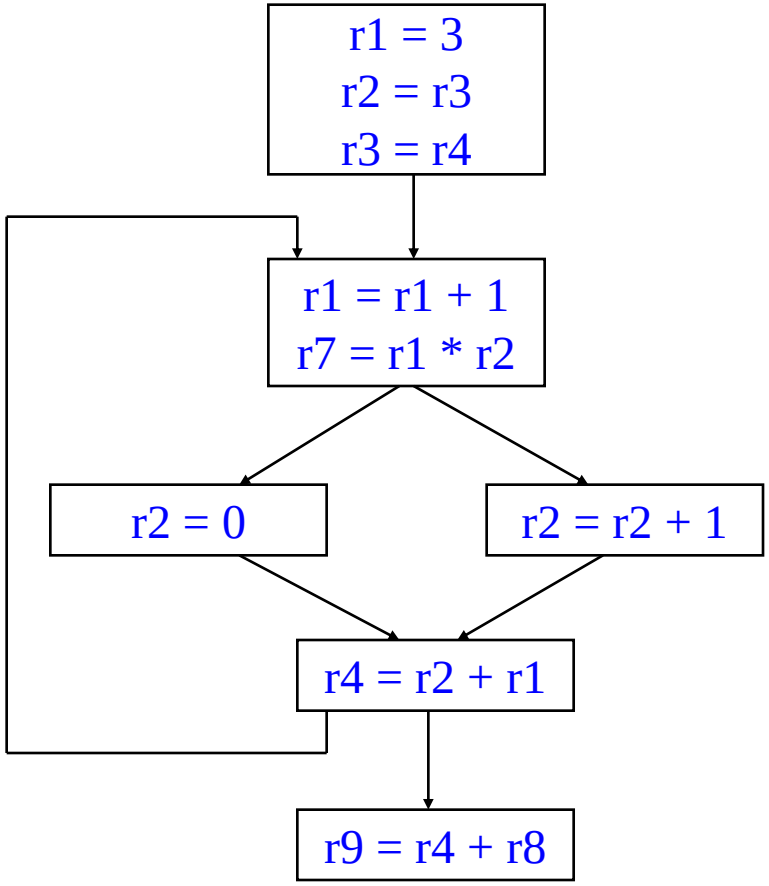
- IN : Block에 들어오는 시점에서 계산한 live변수
  진입시점 변수 값이 내 block 안에서 사용되는 것들 *USE(X)*
  + 내 block 안에서는 그 값이 안 바뀐채로
  다음 block 에서 사용될 것들 *OUT(X) – DEF(X)*
- OUT : 뒤따라오는 다음 block (successor) 들의
  변수 사용 정보 *IN(Y)* 를 모은 것

# Example IN/OUT Calculation



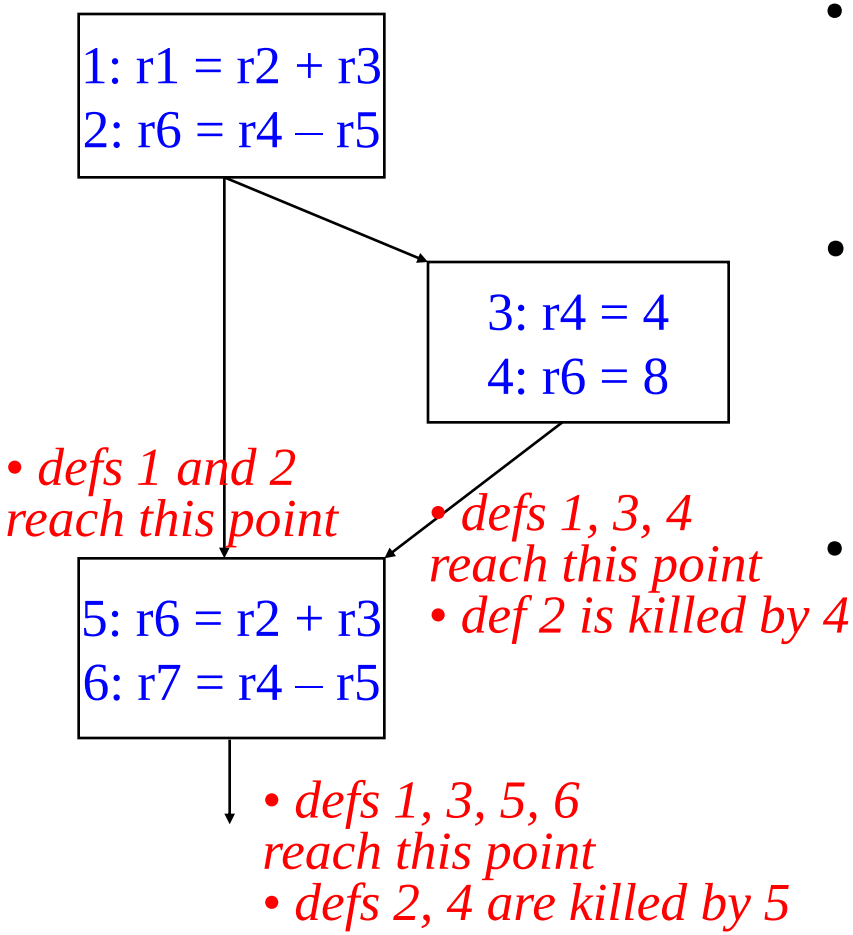

- liveness analysis = “특정 시점 (block 시작, block 끝)에서 어떤 register 들이 의미있는 값을 가지고 있는지 찾는 것”

# Class Problem




Compute liveness, ie
calculate USE/DEF
calculate IN/OUT

# Reaching Definition Analysis (rdefs)




- 변수의 definition
  - 해당 변수가 assignment 왼쪽에 나타나는 것을 말함
- "definition *d* 가 프로그램의 특정 위치 (point) *p*에 reach 함"
  - *d* 직후부터 *p*까지 *d*를 kill 하지 않는 path 가 존재한다.
- "definition *d* 가 두개의 포인트 사이에서 kill 됨"
  - 두 point 를 잇는 path 사이에 동일 변수에 대한 다른 definition 이 존재한다.
  - 예) r1=r2+ r3 등 은 앞서 나타난 r1 의 정의를 kill 한다.

# Reaching Definition Analysis – Algorithms

다음과 같은 데이터를 모으는 것

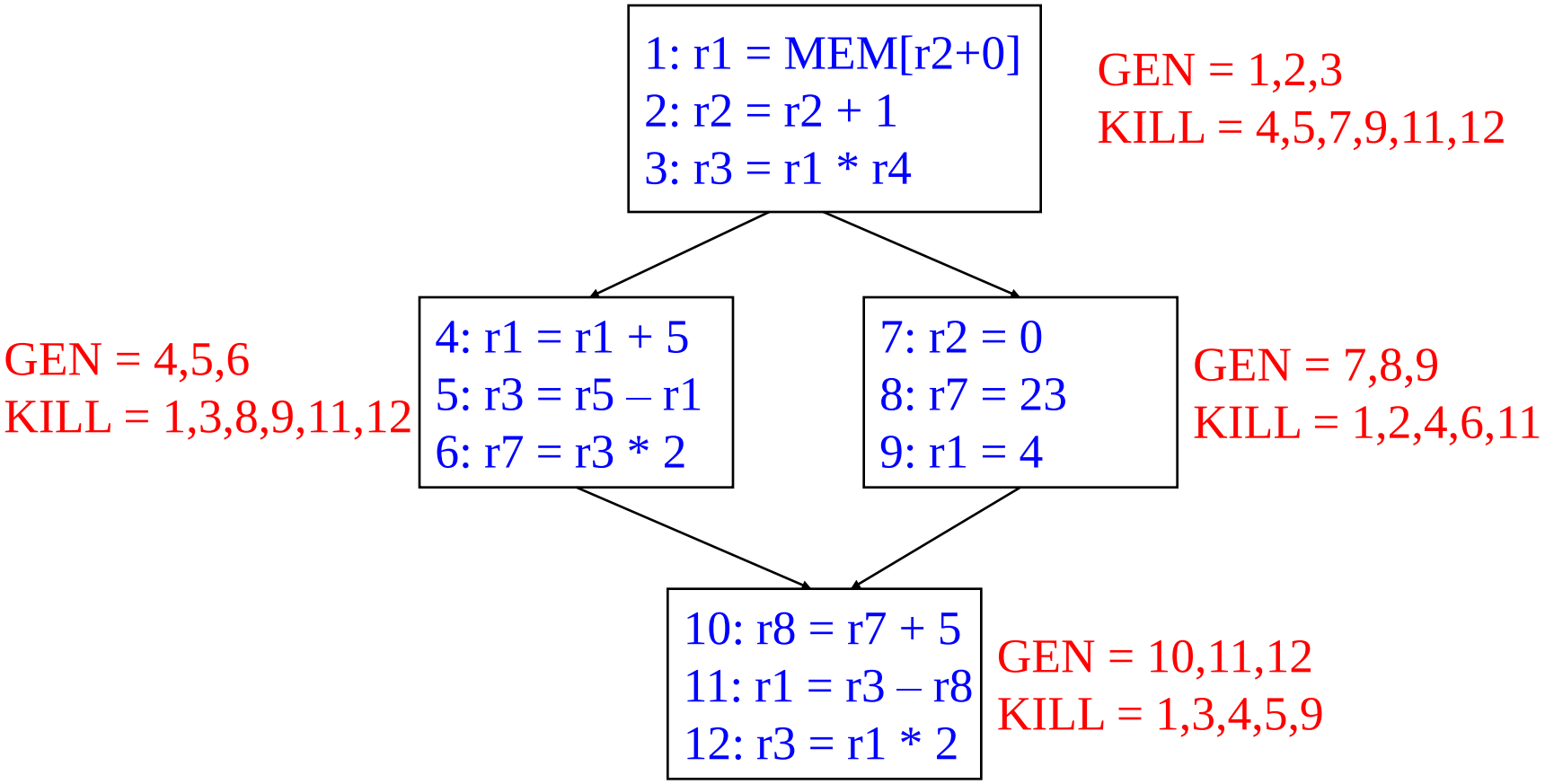
- GEN/KILL 한 block 내
  - GEN = set of *definitions* generated in the BB
  - KILL = set of definitions killed in the BB
- IN/OUT block들 간
  - IN = set of definitions reaching the BB entry
  - OUT = set of definitions reaching the BB exit
- *note)*
  - *IN/OUT 은 forward 로 계산하면 좋음*
  - *IN/OUT 은 GEN/KILL 를 써서 정의됨*

# Compute Rdef GEN/KILL Sets For *Each* BB

for each basic block *X*, do
    GEN(*X*) = ∅
    KILL(*X*) = ∅
    for each operation in sequential order in *X*, *op*, do
        for each destination operand of *op*, *dest*, do
            G = *op*
            K = {all *ops* which define *dest*}
            GEN(*X*) = G + (GEN(*X*) – K)
            KILL(*X*) = K + (KILL(*X*) – G)
        endfor
    endfor
endfor

*dest* : = $src_1$+$src_2$

# Example Rdef GEN/KILL Calculation

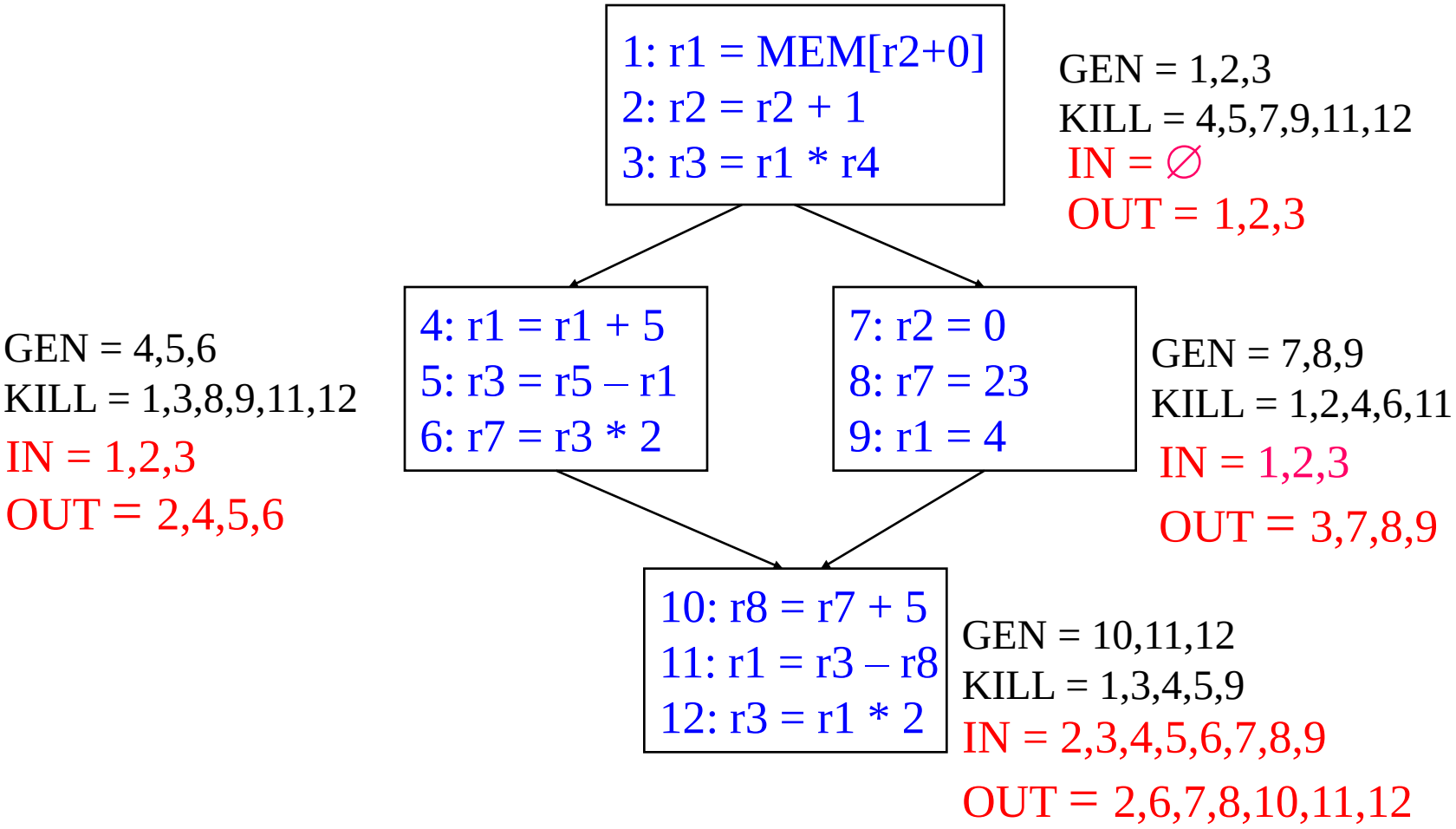


# Compute Rdef IN/OUT Sets for all BBs

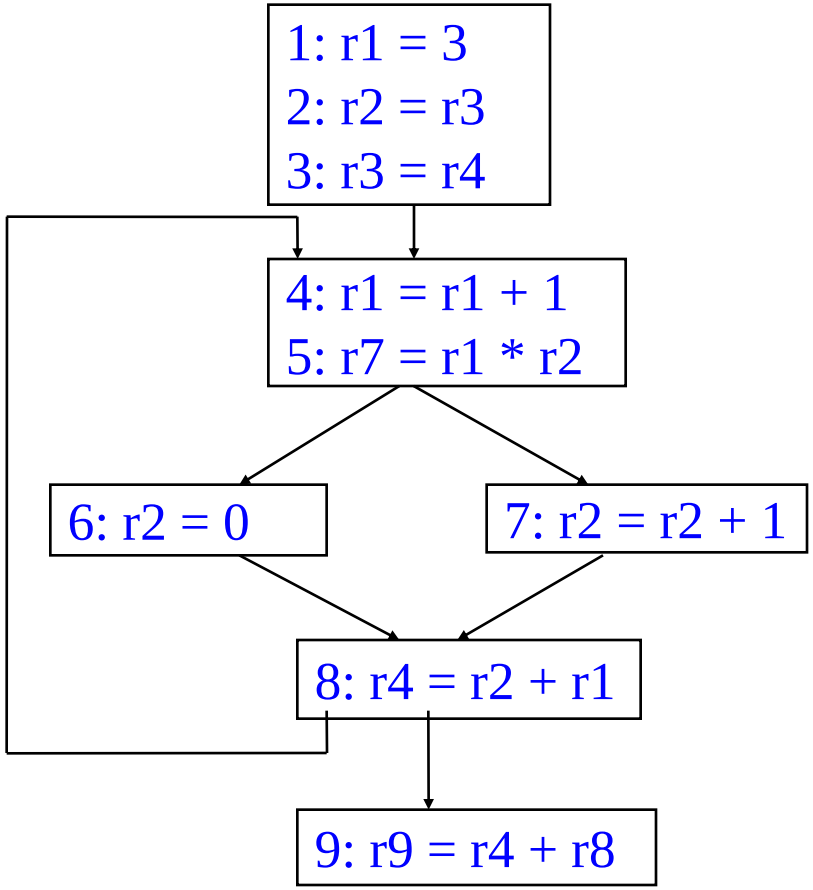
initialize IN($X$) = ∅ for all basic blocks $X$
initialize OUT($X$) = GEN($X$) for all basic blocks $X$
*change* = 1
while (*change*) do
  *change* = 0
  for each basic block $X$, do
    old_OUT = OUT($X$)
    IN($X$) = $\cup_{Y \in \text{predessor}(X)}$ OUT($Y$)
    OUT($X$) = GEN($X$) + (IN($X$) – KILL($X$))
    if (old_OUT != OUT($X$)) then
      *change* = 1
    endif
  endfor
endfor

IN = set of definitions reaching the entry of BB
OUT = set of definitions leaving BB

# Example Rdef IN/OUT Calculation



# Class Problem




Reaching definitions

- Calculate GEN/KILL
- Calculate IN/OUT