Data Flow Analysis

Data Flow Analysis

- These are techniques that derive information about the flow of data along program execution paths
- An *execution path* (or *path*) from point p_1 to point p_n is a sequence of points $p_1, p_2, ..., p_n$ such that for each $i = 1, 2, ..., n-1$, either
  1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
  2. p_i is the end of some block and p_{i+1} is the beginning of a successor block

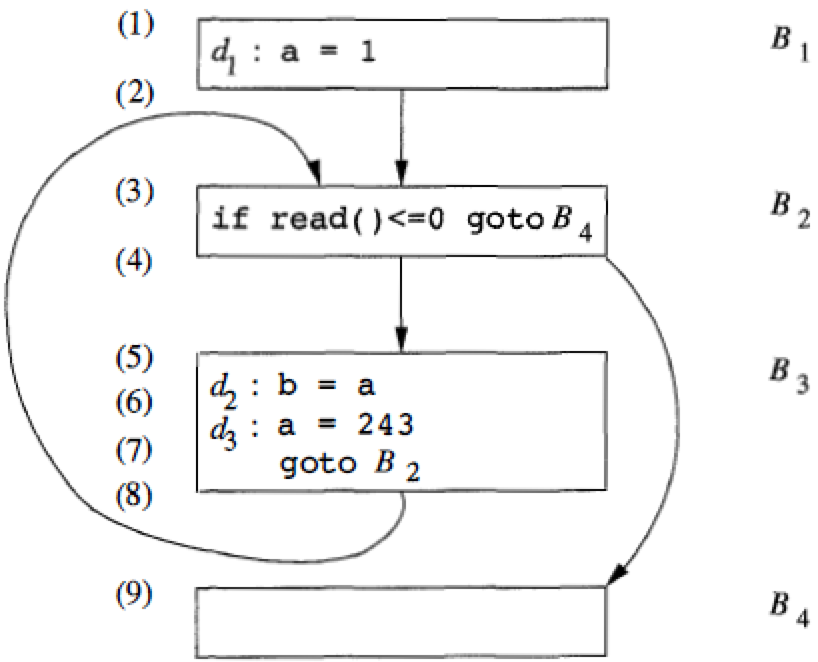
pi

x=y+z

P(i+1)

Data Flow Analysis

Different execution paths of the program



Not entering the loop at all, the shortest complete execution path consists of the program points (1, 2, 3 ,4,9).

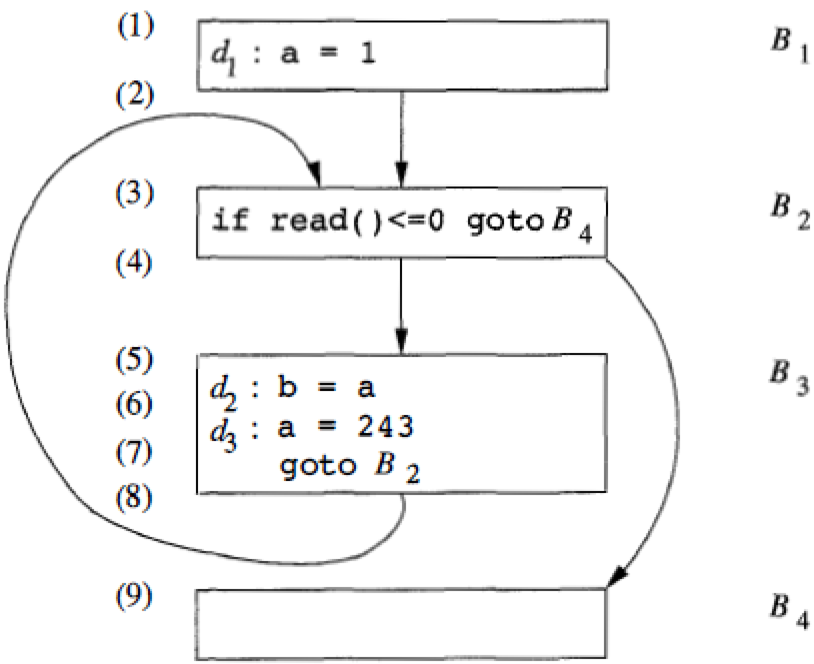
The next shortest path executes one iteration of the loop and consists of the points (1, 2, 3, 4 ,5,6,7,8,3,4,9).

Flow of data value

- For example, the first time program point (5) is executed, the value of a is **1** due to definition d1.
  - We say that **d1 reaches point (5)** in the first iteration.

In subsequent iterations, d3 reaches point (5) and the value of a is 243.

Data Flow Analysis



To help users debug their programs, we may wish to find out what are all the values a variable may have at a program point, and where these values may be defined. For instance, we may summarize all the program states at point (5) by saying that the value of a is one of $\{1, 243\}$, and that it may be defined by one of $\{d_1, d_3\}$. The definitions that *may* reach a program point along some path are known as *reaching definitions.*

Data Flow Analysis

* We denote the data-flow values before and after each statement s by IN[S] and OUT[S] , respectively.
* The data-flow problem is to find a solution to a set of constraints on the IN [S] 'S and OUT[S] 'S, for all statements s.
* There are two sets of constraints:
* (a) “Transfer functions” (based on the semantics of the statements)
* (b) Flow of control functions.

(a) Transfer Functions

- The data-flow values before and after a statement are constrained by the **TF** (semantics of the statement)

p(i)

b=a

p(i+1)

- For example, suppose data-flow analysis involves **determining the value** of variables at points.
- If variable **a** has value **v** **before executing statement b = a**, then both a and b will have the value **v after the statement**.
- This **relationship** between the data-flow values **before** and **after** the assignment statement is known as a **transfer function.**

(a) Transfer Functions

- Transfer functions come in two flavors: **information may propagate forward** along execution paths,
- Or it may **flow backwards** up the execution paths.
- In **a forward-flow problem,** the **transfer function fs** of a statement **s,**
- (i) **takes** the data-flow value **before the statement** and
- (ii) **produces** a new data-flow value **after the statement**

(a) Transfer Functions

usually denote f_s, takes the data-flow value before the statement and produces a new data-flow value after the statement. That is,

$$\text{OUT}[s] = f_s(\text{IN}[s]).$$

Conversely, in a backward-flow problem, the transfer function f_s for statement s converts a data-flow value after the statement to a new data-flow value before the statement. That is,

$$\text{IN}[s] = f_s(\text{OUT}[s]).$$

(b) Control-Flow Constraints

The second set of constraints on data-flow values is derived from the flow of control. Within a basic block, control flow is simple. If a block B consists of statements $s_1, s_2, \ldots, s_n$ in that order, then the control-flow value out of s_i is the same as the control-flow value into s_{i+1}. That is,

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \text{ for all } i = 1, 2, \ldots, n-1.$$

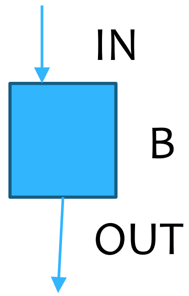
- However, **control-flow edges between basic blocks** create more complex constraints between the **last statement of one basic block** and the **first statement of the following block**.

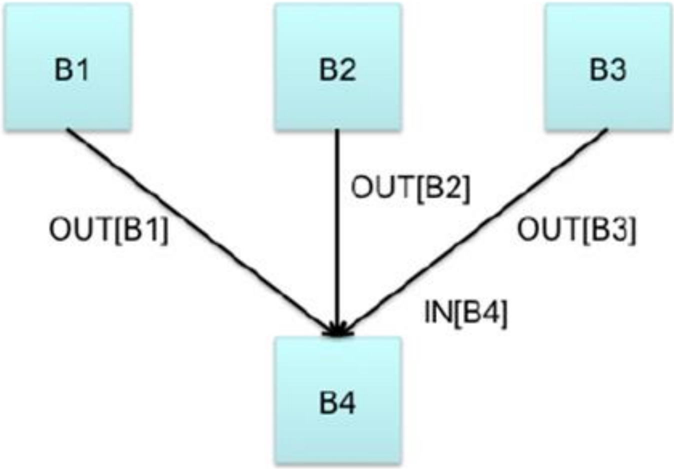
- For example, if we wish to collect **all the definitions that may reach a program point,**
- Then the set of definitions reaching the leader statement of a basic block is the
- **union** of the definitions after the last statements of each of the predecessor blocks.

(b) Control-Flow Constraints

Suppose block B consists of statements $s_1, \ldots, s_n$, in that order. If s_1 is the first statement of basic block B, then $\text{IN}[B] = \text{IN}[s_1]$, Similarly, if s_n is the last statement of basic block B, then $\text{OUT}[B] = \text{OUT}[s_n]$. The transfer function of a basic block B, which we denote f_B, can be derived by composing the transfer functions of the statements in the block. That is, let f_{s_i} be the transfer function of statement s_i. Then $f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$. The relationship between the beginning and end of the block is

$$\text{OUT}[B] = f_B(\text{IN}[B]).$$



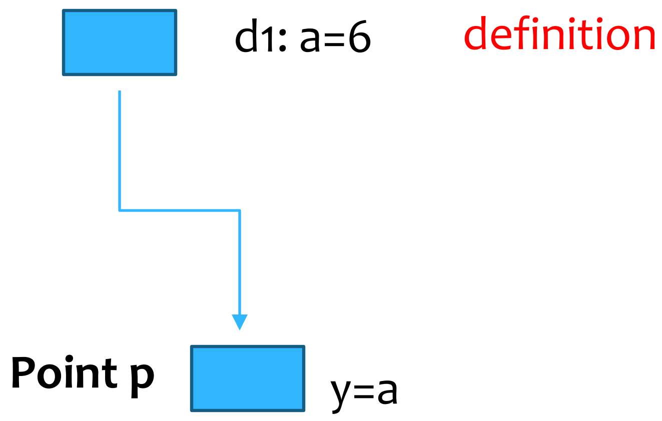


DFA Steps

- A *data-flow value* for a program point represents an abstraction of the set of all possible program states that can be observed for that point
- The set of all possible data-flow values is the *domain* for the application under consideration
  - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of definitions in the program
  - A particular data-flow value is a set of definitions
- $IN[s]$ and $OUT[s]$: data-flow values *before* and *after* each statement s **May extend for blocks**
- The *data-flow problem* is to find a solution to a set of constraints on $IN[s]$ and $OUT[s]$, for all statements s

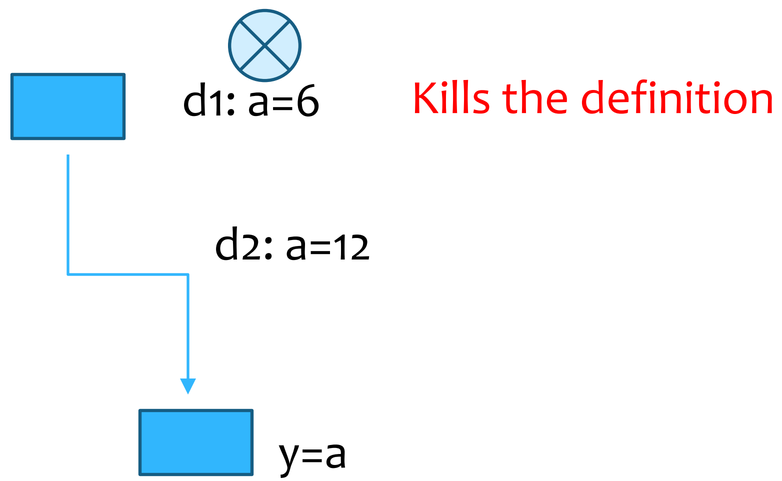
Reaching Definitions (RD) Problem

A definition d reaches a point p, if there is a path from the point immediately following d to p, such that d is not *killed* along that path



Reaching Definitions (RD) Problem

- We *kill* a definition of a variable *a*, if between two points along the path, there is an assignment to *a*
- A definition *d* reaches a point *p*, if there is a path from the point immediately following *d* to *p*, such that *d* is not *killed* along that path



Motivation: Usage

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

```
a[t2] = t5
a[t4] = t3
goto B2
```

```
if (debug) print ...
```

```
debug = FALSE
```

Copy propagation

- Drop this code segment
- Constant folding

RD Problem

- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
  - If a variable x has two or more defintions in a basic block, then only the last definition of x is downwards exposed; all others are not visible outside the block
- $KILL[B]$ = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in B
  - If a variable x has a definition d_i in a basic block, then d_i kills all the definitions of the variable x in the program, except d_i

RD Analysis: GEN and KILL

In other blocks:

d5: b = a+4
d6: f = e+c
d7: e = b+d
d8: d = a+b
d9: a = c+f
d10: c = e+a

d1: a = f + 1
d2: b = a + 7
d3: c = b + d
d4: a = d + c

B

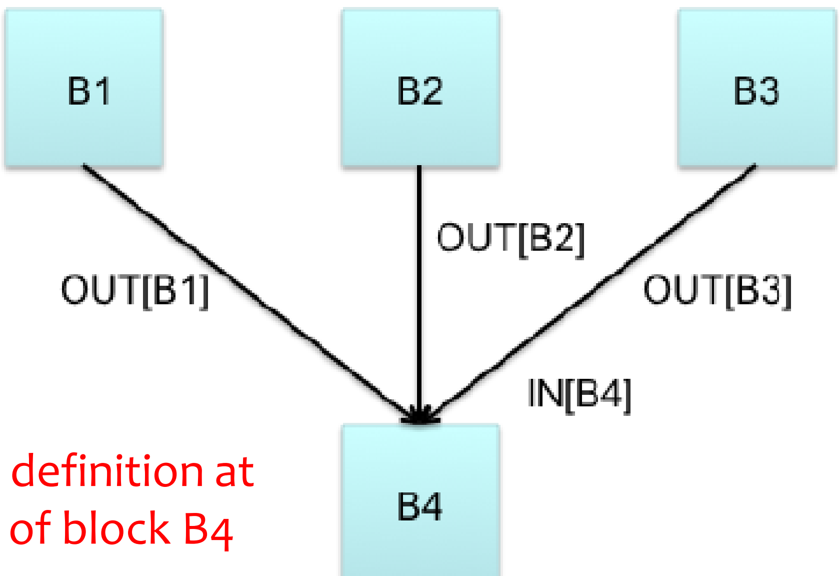
Set of all definitions = {d1,d2,d3,d4,d5,d6,d7,d8,d9,10}

GEN[B] = {d2,d3,d4}

Kills(d9, d5, d10, d1)

RD Analysis: DF Equations

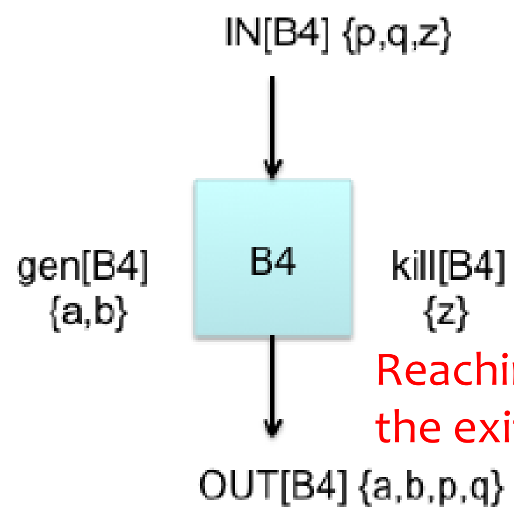
Control flow Eqs



Reaching definition at the entry of block B4

IN[B4] = OUT[B1] U OUT[B2] U OUT[B3]

Transfer Eqs



Reaching definition at the exit of block B4

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

OUT[B4] = gen[B4] U (IN[B4] – kill[B4])

RD Problem

- The data-flow equations (constraints)

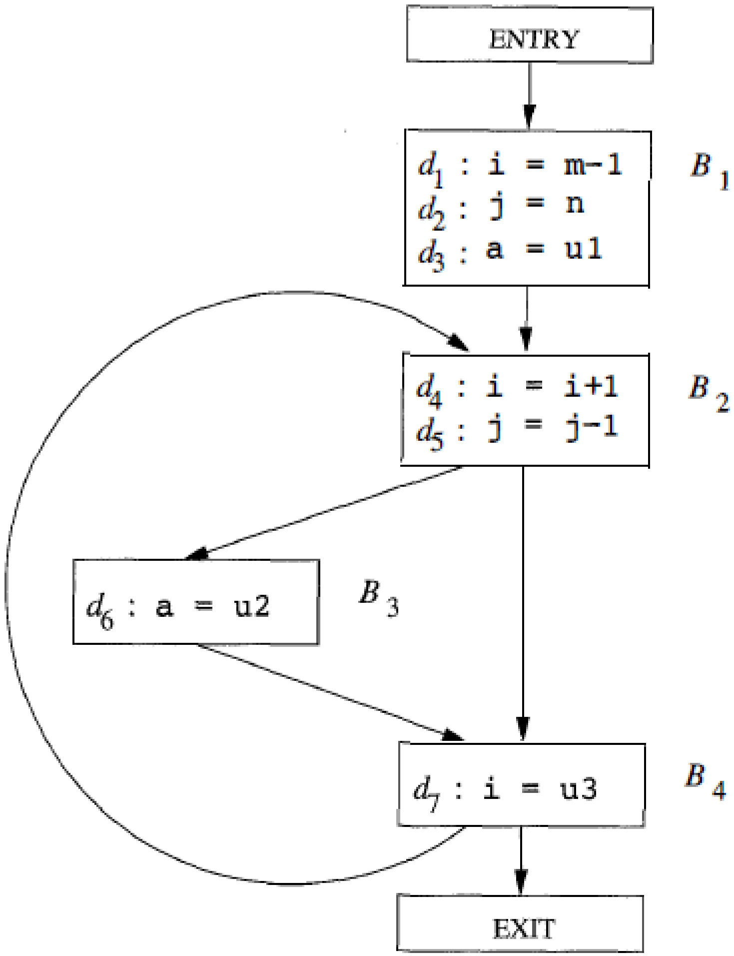
$$
\begin{aligned}
IN[B] &= \bigcup_{P \text{ is a predecessor of } B} OUT[P] \\
OUT[B] &= GEN[B] \bigcup (IN[B] - KILL[B]) \\
IN[B] &= \phi, \text{ for all } B \text{ (initialization only)}
\end{aligned}
$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
  - Direction of flow does not imply traversing the basic blocks in a particular order
  - The final result does not depend on the order of traversal of the basic blocks

RD algorithm

OUT[ENTRY] $= \emptyset$;
for (each basic block B other than ENTRY) OUT[B] $= \emptyset$;
while (changes to any OUT occur)
  for (each basic block B other than ENTRY) {
    $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
    $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
  }

Block B	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111



$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

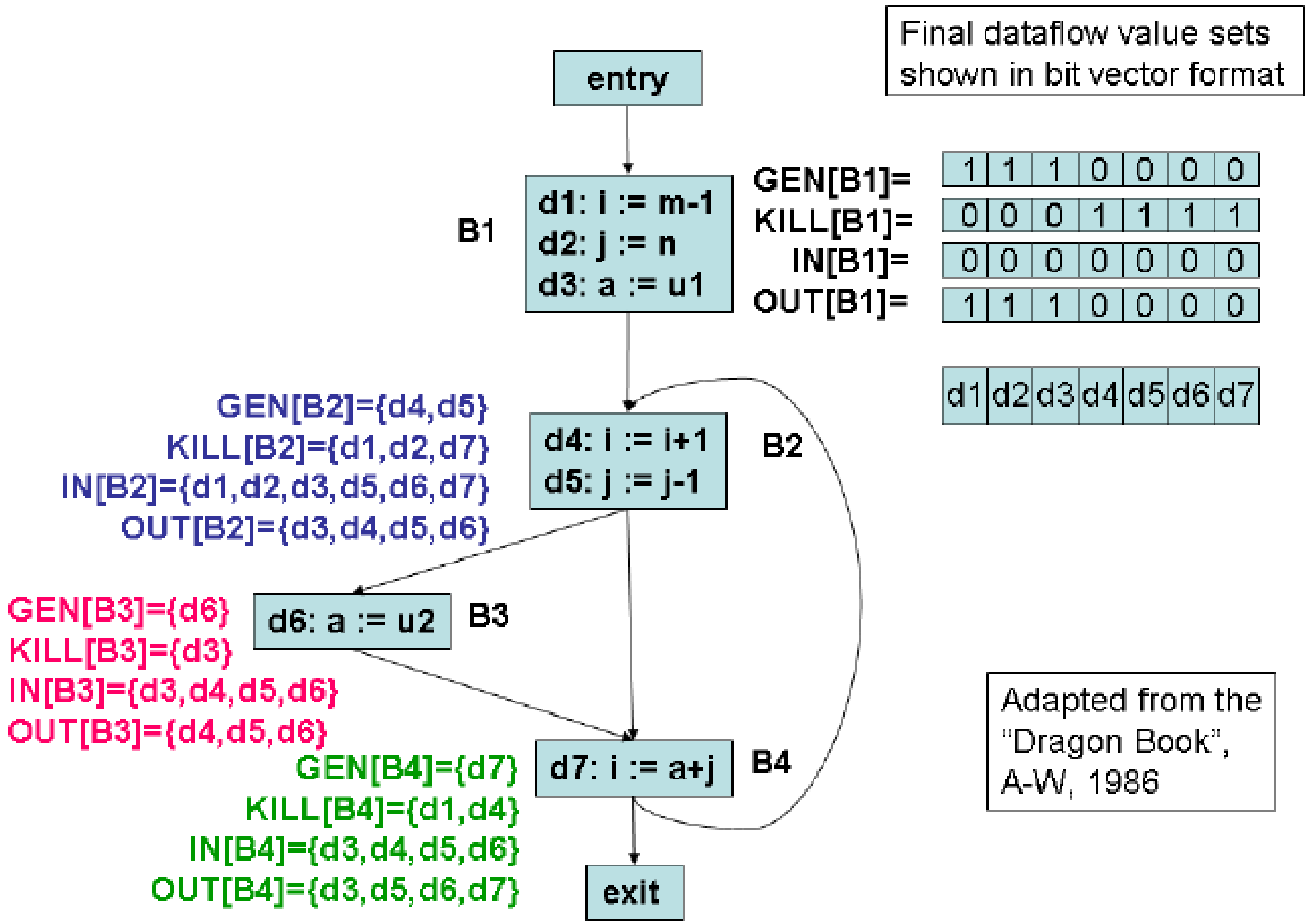
$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

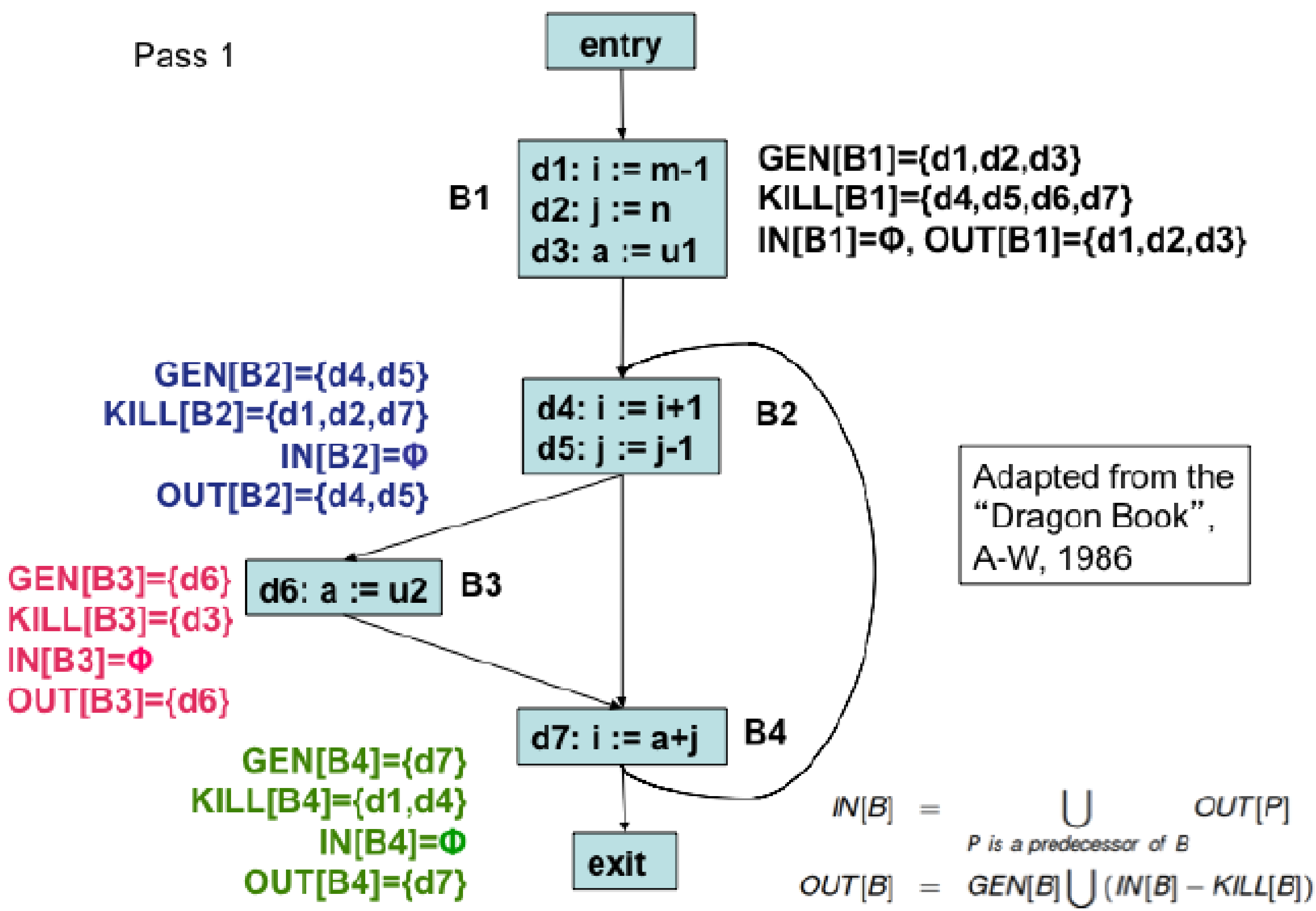
$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

RD: Bit vector representation



RD Analysis: An example



RD algorithm

$\text{OUT}[\text{ENTRY}] = \emptyset;$

for (each basic block B other than ENTRY) $\text{OUT}[B] = \emptyset;$

while (changes to any OUT occur)

  for (each basic block B other than ENTRY) {

    $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P];$

    $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B);$

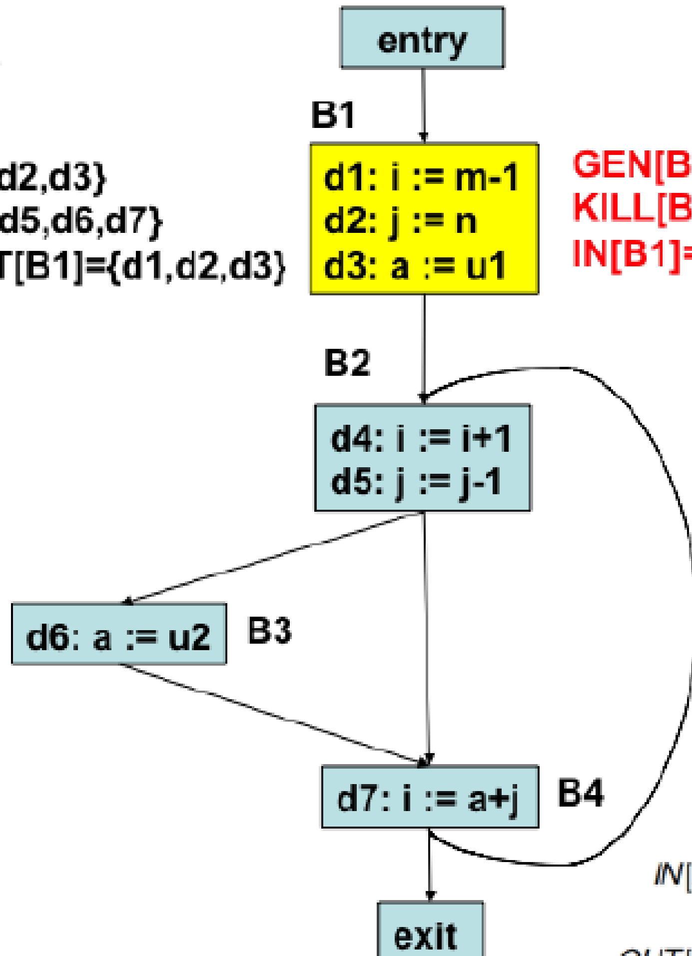
  }

Block B	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

RD Analysis: An example

Pass 2

GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}



GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

RD algorithm

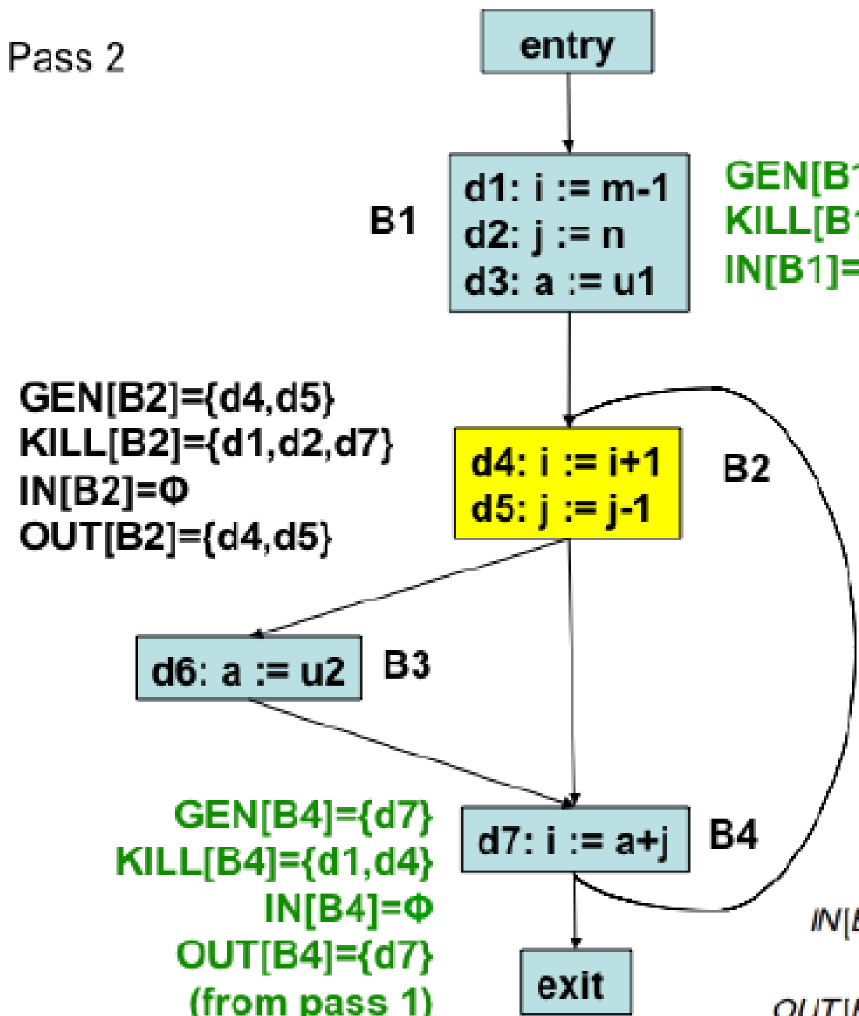
1) $\text{OUT}[\text{ENTRY}] = \emptyset;$
2) **for** (each basic block B other than ENTRY) $\text{OUT}[B] = \emptyset;$
3) **while** (changes to any OUT occur)
4) **for** (each basic block B other than ENTRY) {
5) $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P];$
6) $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B);$

}

Next iteration

Block B	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

RD Analysis: An example



GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d7}
OUT[B2]={d3,d4,d5}

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

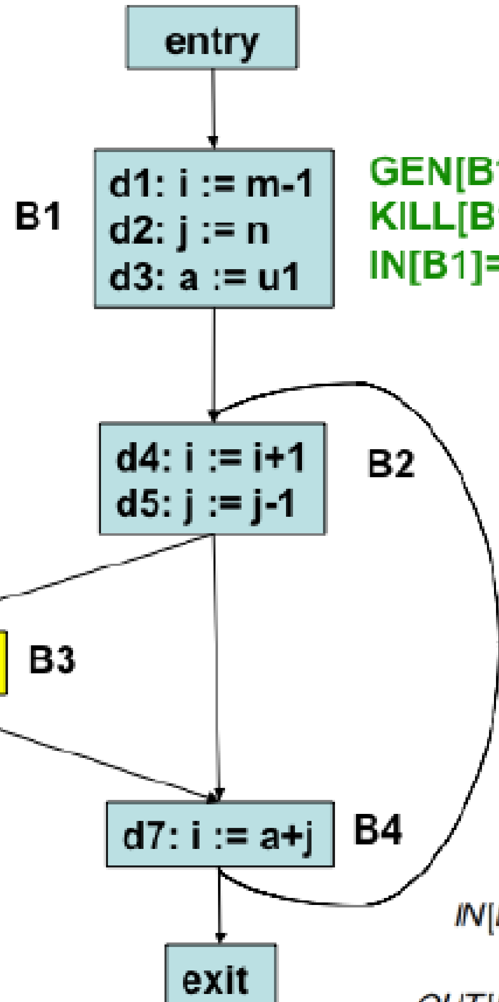
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

RD Analysis: An example

Pass 2

GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]=Φ
OUT[B3]={d6}



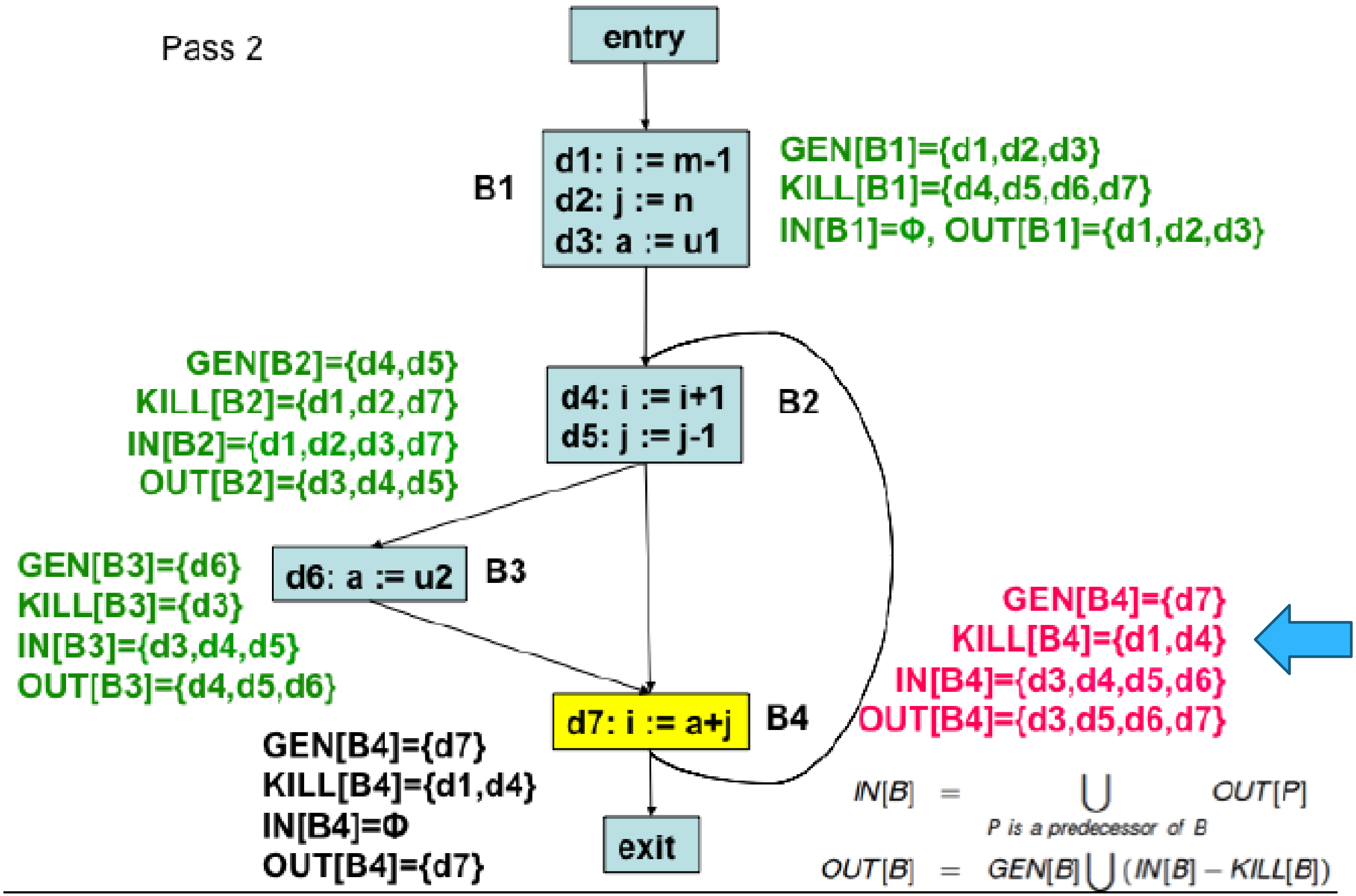
GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d7}
OUT[B2]={d3,d4,d5}

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5}
OUT[B3]={d4,d5,d6}

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

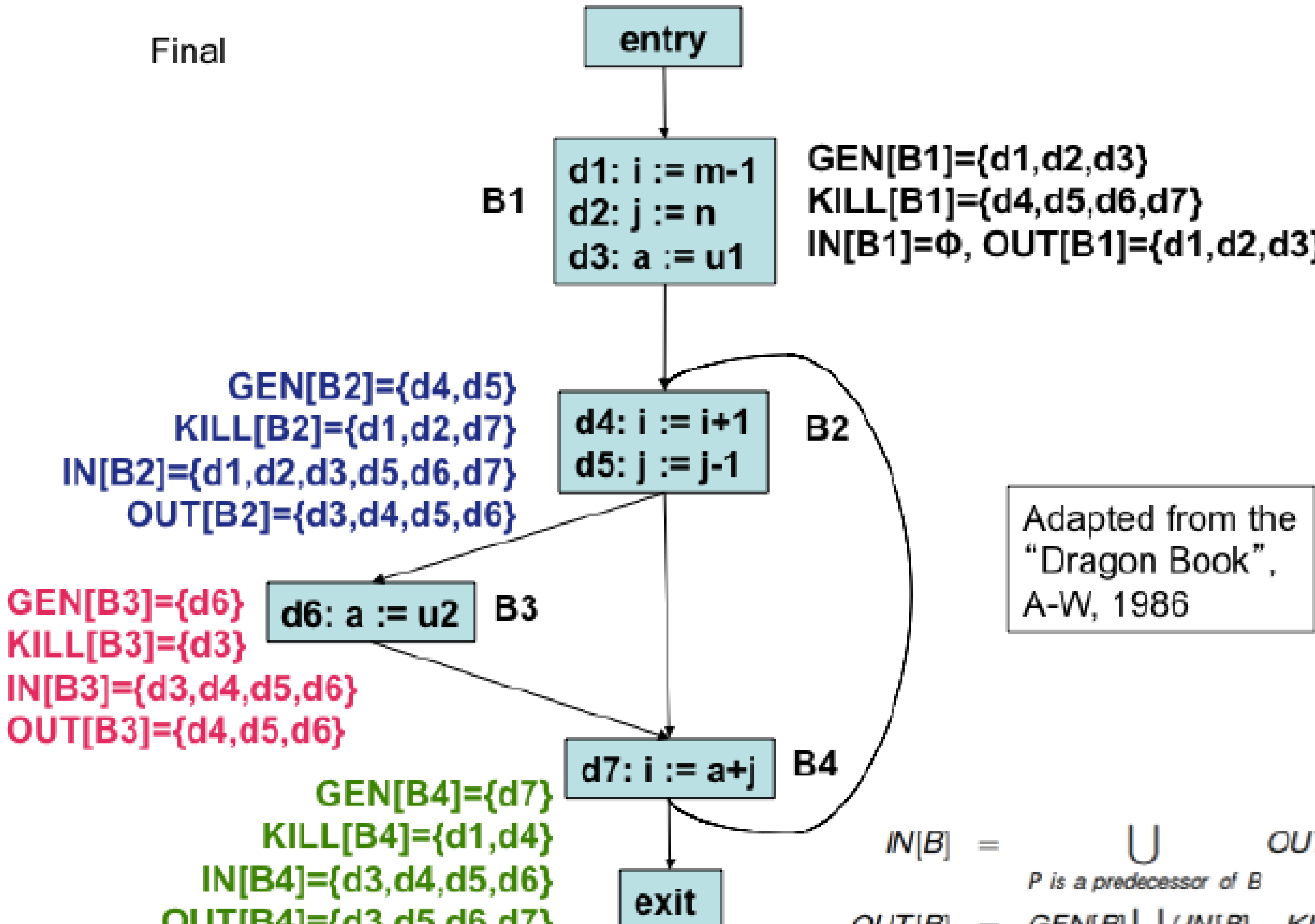
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

RD Analysis: An example



RD Analysis: An example

Final



Adapted from the "Dragon Book", A-W, 1986

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

RD algorithm

1) $\text{OUT}[\text{ENTRY}] = \emptyset;$
2) **for** (each basic block B other than ENTRY) $\text{OUT}[B] = \emptyset;$
3) **while** (changes to any OUT occur)
4) **for** (each basic block B other than ENTRY) {
5) $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P];$
6) $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B);$
}

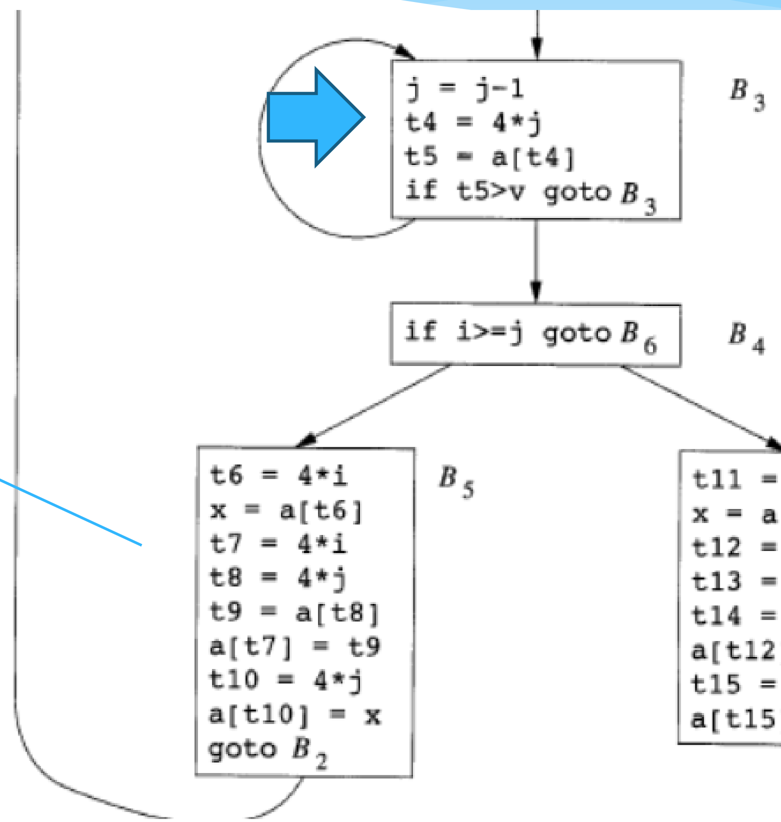
Block B	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Available expression

Global Common Subexpressions

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

B_5



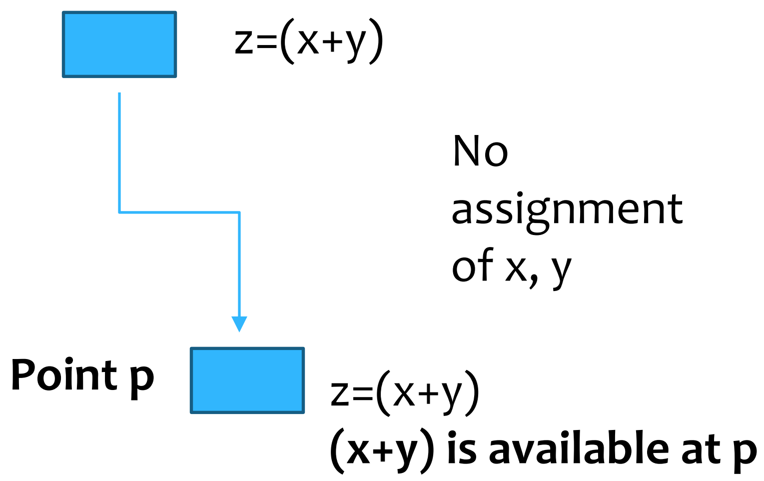

```
t8 = 4*j
t9 = a[t8]
a[t8] = x
```

in B_5 can be replaced by

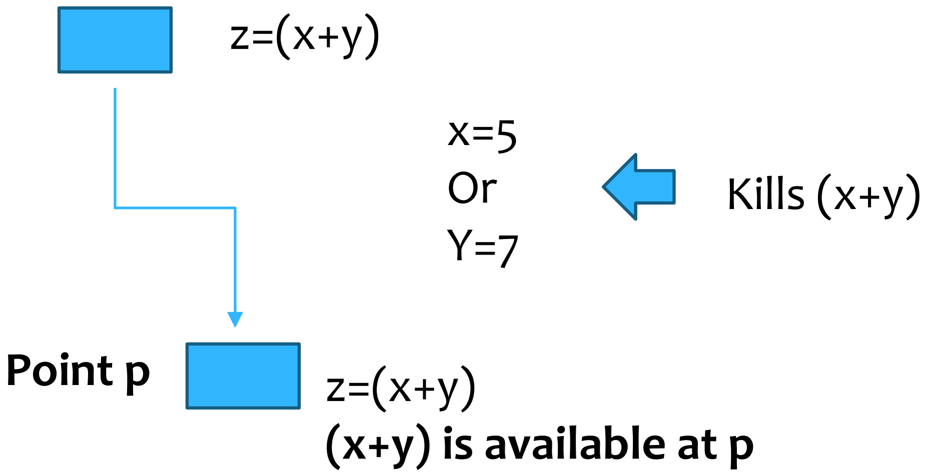
```
t9 = a[t4]
a[t4] = x
```

- Control passes from the evaluation of 4 * j in B3 to B5,
- No change to j and no change to t4, so t4 can be used if 4 * j is needed.

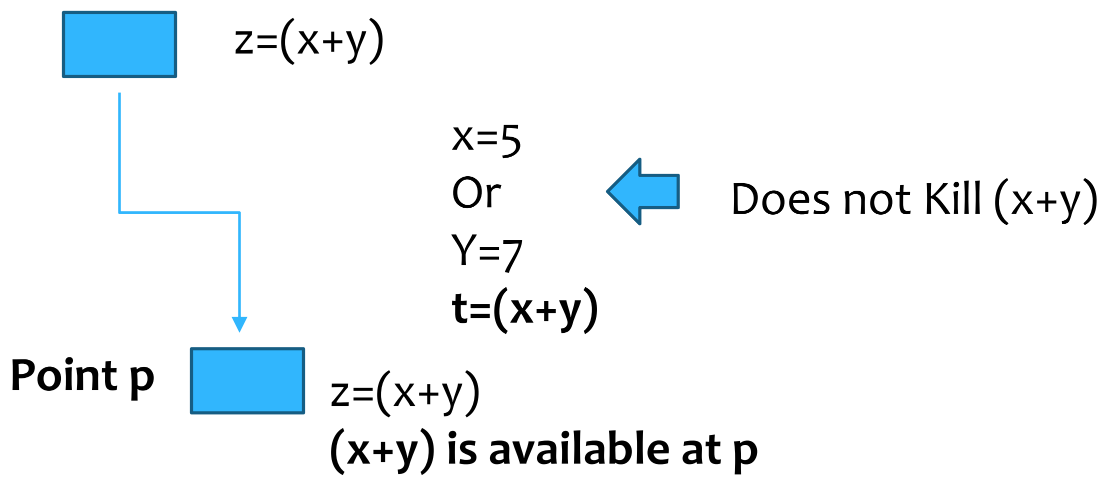
Available expression



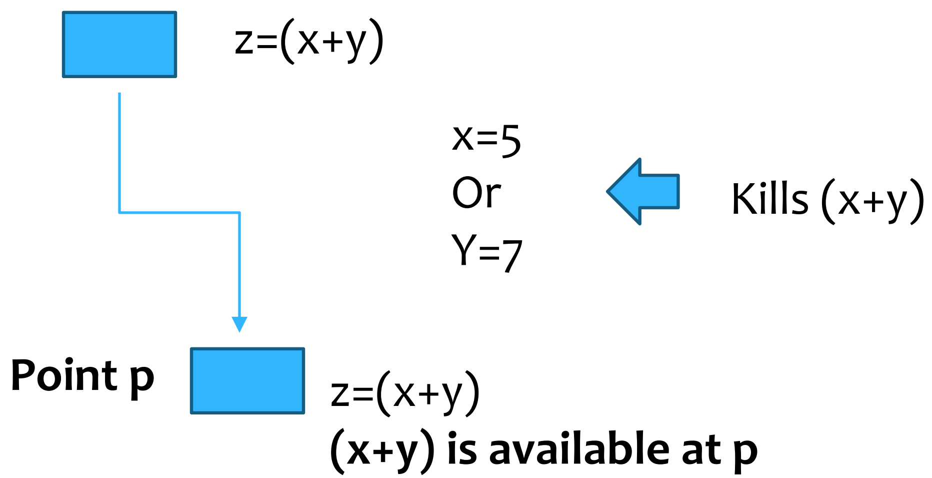
Available expression



Available expression



Available expression

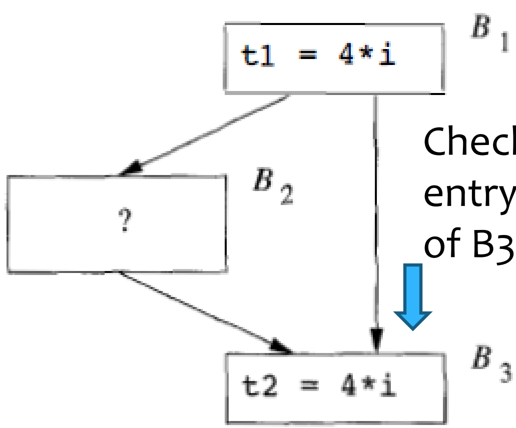


- A block **generates expression x + y** if it (a) evaluates x + y, (b) and does not subsequently define x or y
- We say that a block **kills expression x + y** if it (a) assigns x or y and (b) does not subsequently recompute x + y.

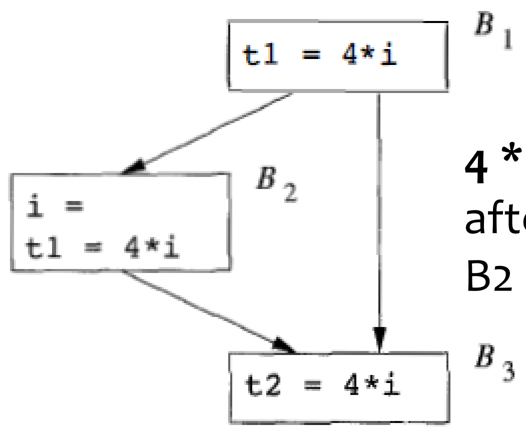
Available expression

Usage: Detecting global common subexpression

Not assigned to i



Check the entry point of B3



4 * i is recomputed after i is assigned in B2

Is **4*i** available expression in B3?

Generated available expressions

p: expressions S is available

x=y+z

q: expressions available ??

Basic block

We can compute the set of generated expressions for each point in a block, working from beginning to end of the block. At the point prior to the block, no expressions are generated. If at point p set S of expressions is available, and q is the point after p, with statement **x = y+z** between them, then we form the set of expressions available at q by the following two steps.

1. Add to S the expression $y + z$.

2. Delete from S any expression involving variable x.

Generated available expressions

Example

Statement	Available Expressions
	$\emptyset$
`a = b + c`	
	$\{b + c\}$
`b = a - d`	
	$\{a - d\}$
`c = b + c`	
	$\{a - d\}$
`d = a - d`	
	$\emptyset$

DFA: Available expression
(e_gen and e_kill)

- For statements of the form $x = a$, step 1 below does not apply
- The set of all expressions appearing as the RHS of assignments in the flow graph is assumed to be available and is represented using a hash table and a bit vector

e_gen[q] = A   **q ▪**
x = y + z
p ▪

Computing e_gen[p]

1. A = A **U** {y+z}
2. A = A – {all expressions involving x}
3. e_gen[p] = A

e_kill[q] = A   **q ▪**
x = y + z
p ▪

Computing e_kill[p]

1. A = A – {y+z}
2. A = A **U** {all expressions involving x}
3. e_kill[p] = A

DFA: Available expression
(e_gen and e_kill)

In other blocks:

d5: b = a+4
d6: f = e+c
d7: e = b+d
d8: d = a+b
d9: a = c+f
d10: c = e+a

d1: a = f + 1
d2: b = a + 7
d3: c = b + d
d4: a = d + c

B

Set of all expressions = {f+1,a+7,b+d,d+c,a+4,e+c,a+b,c+f,e+a}

EGEN[B] = {f+1,b+d,d+c}
EKILL[B] = {a+4,a+b,e+a,e+c,c+f,a+7}

Equations for available exp

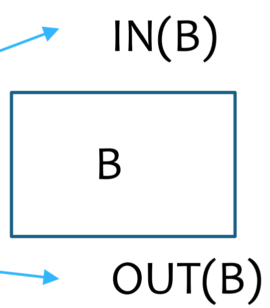
We can find available expressions in a manner reminiscent of the way reaching definitions are computed. Suppose U is the "universal" set of all expressions appearing on the right of one or more statements of the program. For each block B, let IN$[B]$ be the set of expressions in U that are available at the point just before the beginning of B. Let OUT$[B]$ be the same for the point following the end of B. Define e_gen_B to be the expressions generated by B and e_kill_B to be the set of expressions in U killed in B. Note that IN, OUT, e_gen, and e_kill can all be represented by bit vectors. The following equations relate the unknowns IN and OUT to each other and the known quantities e_gen and e_kill:

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

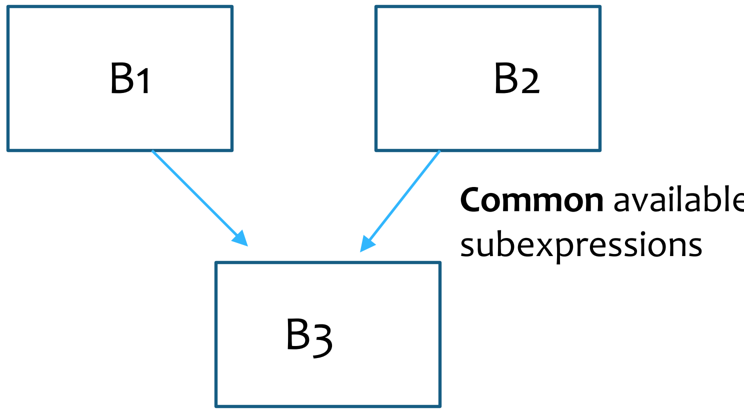
and for all basic blocks B other than ENTRY,

$$\text{OUT}[B] = e\_gen_B \cup (\text{IN}[B] - e\_kill_B)$$
$$\text{IN}[B] = \bigcap_{P \text{ a predecessor of } B} \text{OUT}[P].$$

Transfer Eq

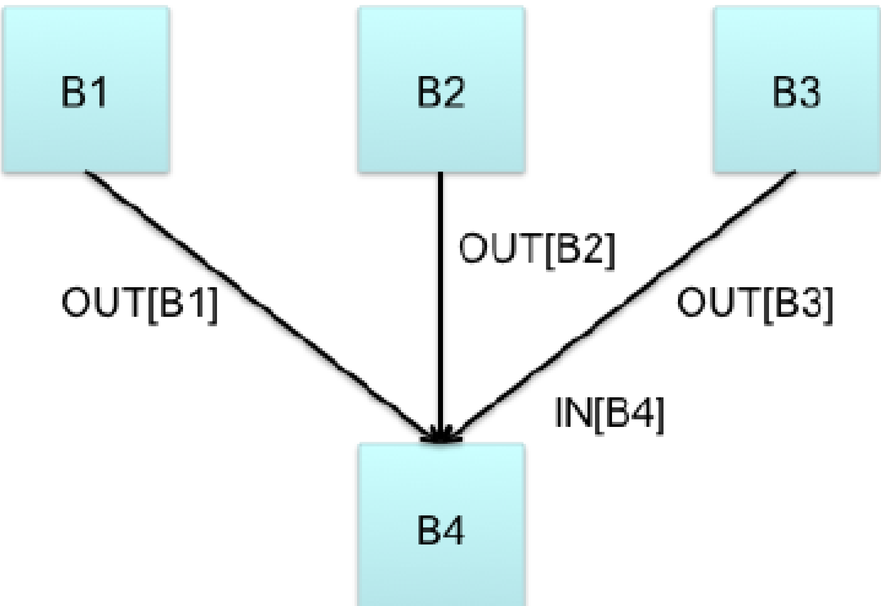


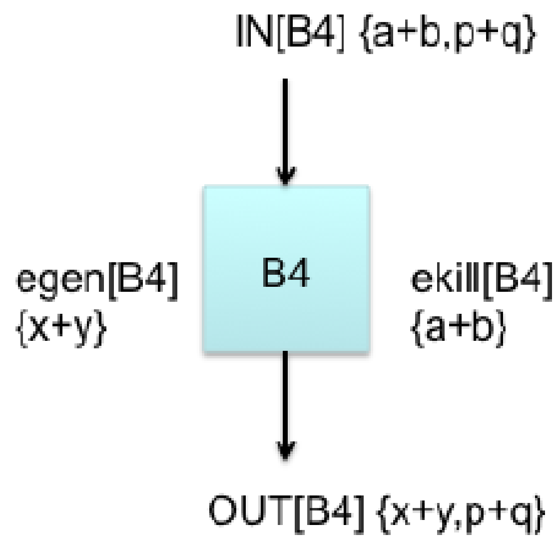
Control flow Eq



Intersection: **E**xpression is **available** at the beginning of a block only if it is available at the end of **all its predecessors.**

Equations for available exp





IN[B4] = OUT[B1] $\bigcap$ OUT[B2] $\bigcap$ OUT[B3]

OUT[B4] = egen[B4] $\bigcup$ (IN[B4] – ekill[B4])

$$IN[B] = \bigcap_{P \text{ is a predecessor of } B} OUT[P], \; B \text{ not initial}$$

$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$$

Equations for available exp

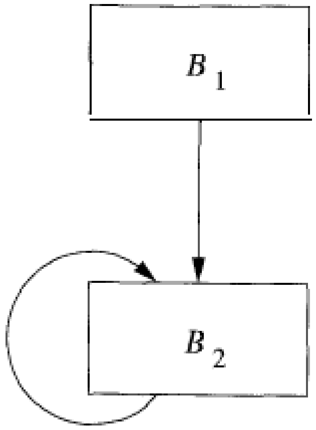
Algorithm 9.17: Available expressions.

INPUT: A flow graph with e_kill_B and e_gen_B computed for each block B. The initial block is B_1.

OUTPUT: IN$[B]$ and OUT$[B]$, the set of expressions available at the entry and exit of each block B of the flow graph.

METHOD: Execute the algorithm of Fig. 9.20. The explanation of the steps is similar to that for Fig. 9.14. □

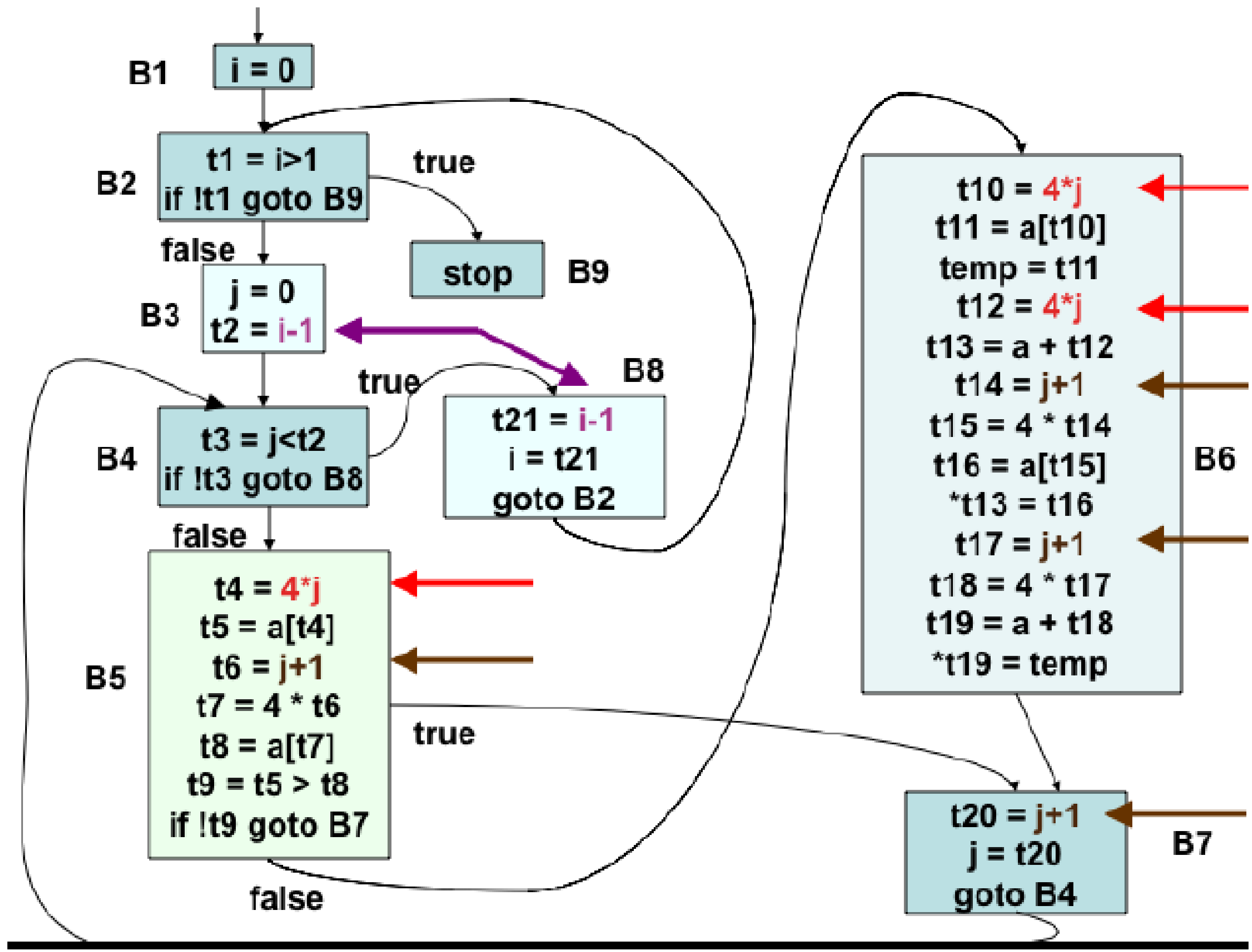
OUT[ENTRY] $= \emptyset$;
for (each basic block B other than ENTRY) OUT$[B] = U$;
while (changes to any OUT occur)
    for (each basic block B other than ENTRY) {
        IN$[B] = \bigcap_{P \text{ a predecessor of } B}$ OUT$[P]$;
        OUT$[B] = e_gen_B \cup ($IN$[B] - e_kill_B)$;
    }



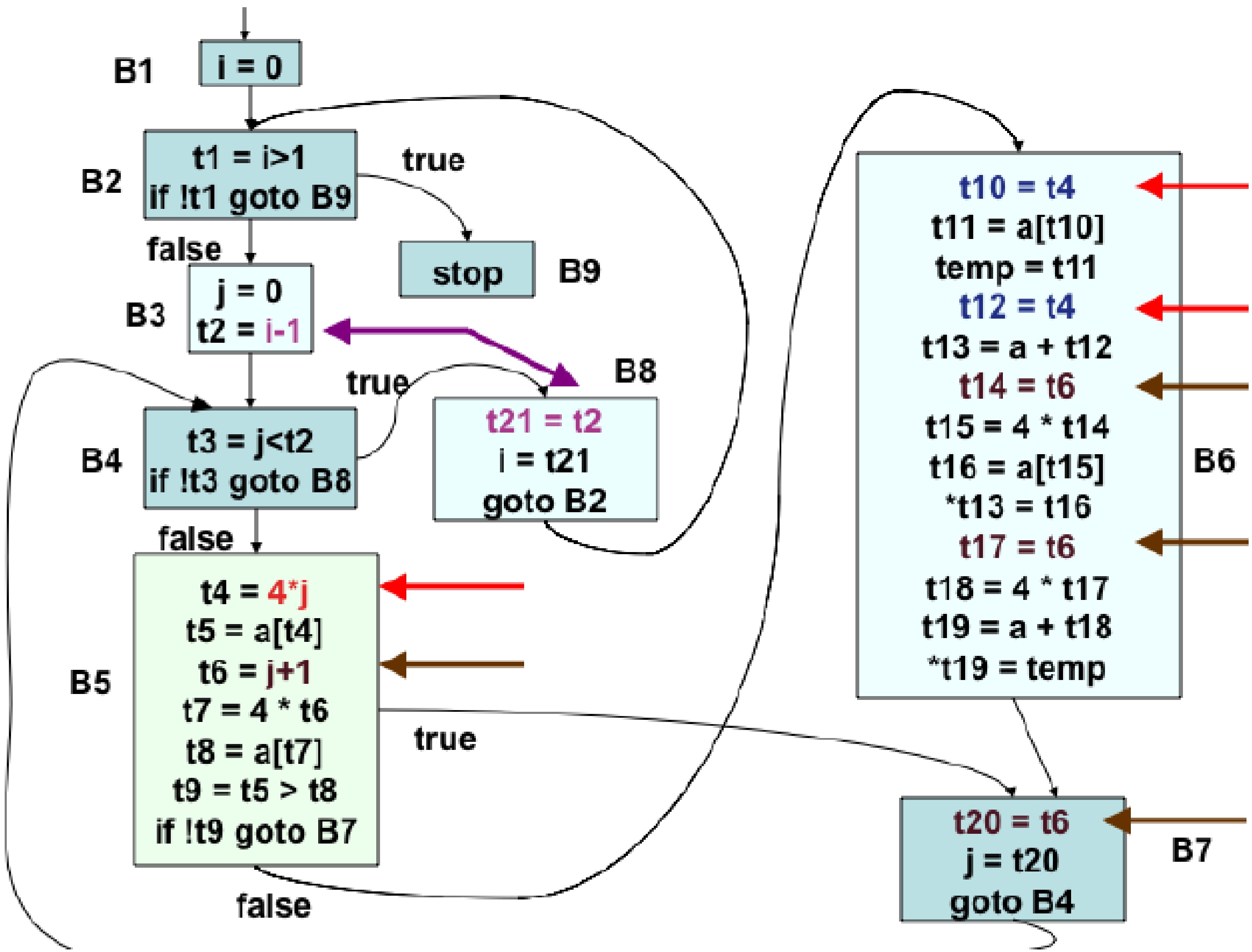
Out(B2): NULL??
Out(B2): U??

$$\text{IN}[B_2] = \text{OUT}[B_1] \cap \text{OUT}[B_2]$$

DFA: Available expression (Example)



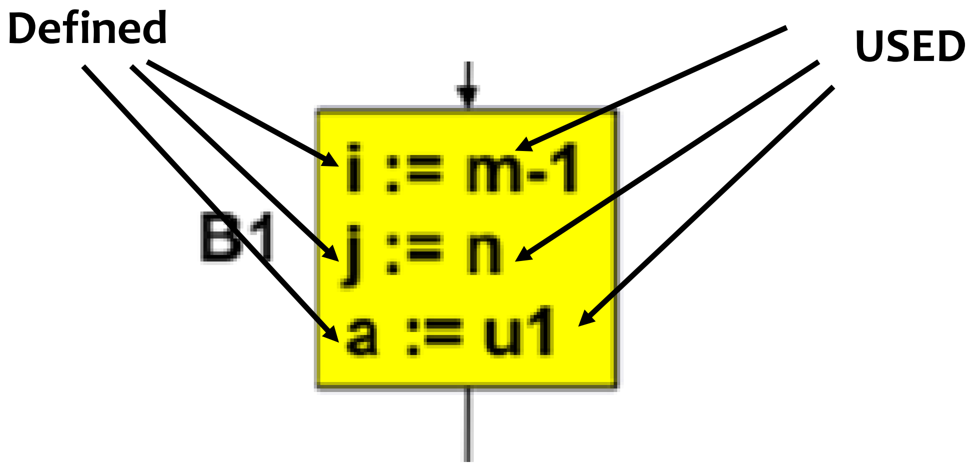
DFA: Available expression (Example)



DFA: Live Variables

- The variable x is *live* at the point p, if the value of x at p could be used along some path in the flow graph, starting at p; otherwise, x is *dead* at p
- Sets of variables constitute the domain of data-flow values
- Backward flow problem, with confluence operator $\bigcup$
- $IN[B]$ is the set of variables live at the beginning of B
- $OUT[B]$ is the set of variables live just after B
- $DEF[B]$ is the set of variables definitely assigned values in B, prior to any use of that variable in B
- $USE[B]$ is the set of variables whose values may be used in B prior to any definition of the variable

DEF(B) and USE(B)



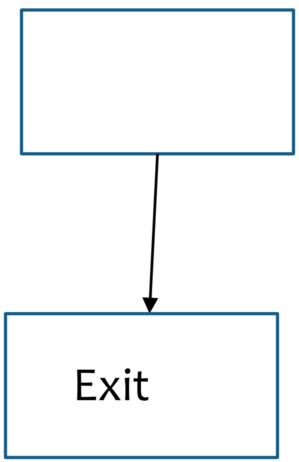
DEF(B) : Set of variables **defined** in B prior to any use of that variable in B

DEF(B1):{i, j, a}

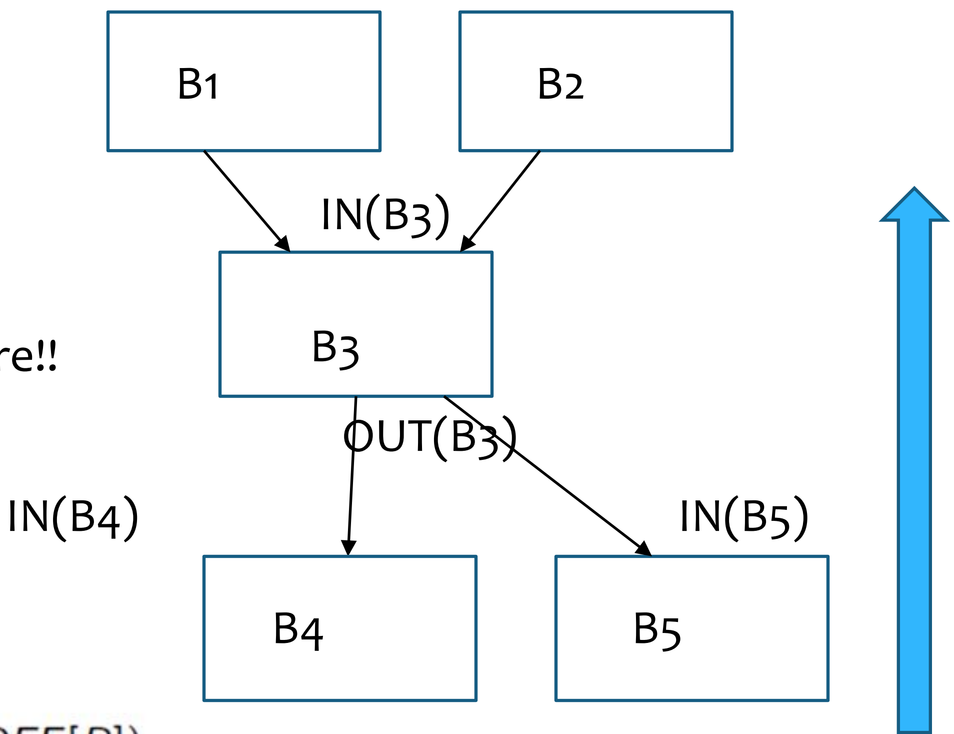
USE(B): Set of variables whose values may be **used** in B prior to any definition of the variable.

USE(B1): {m, n, u1}

DFA: Live Variables



Nothing is live here!!



Compute upwards

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

$$IN[B] = \phi, \text{ for all } B \text{ (initialization only)}$$

Live Variable Algorithm

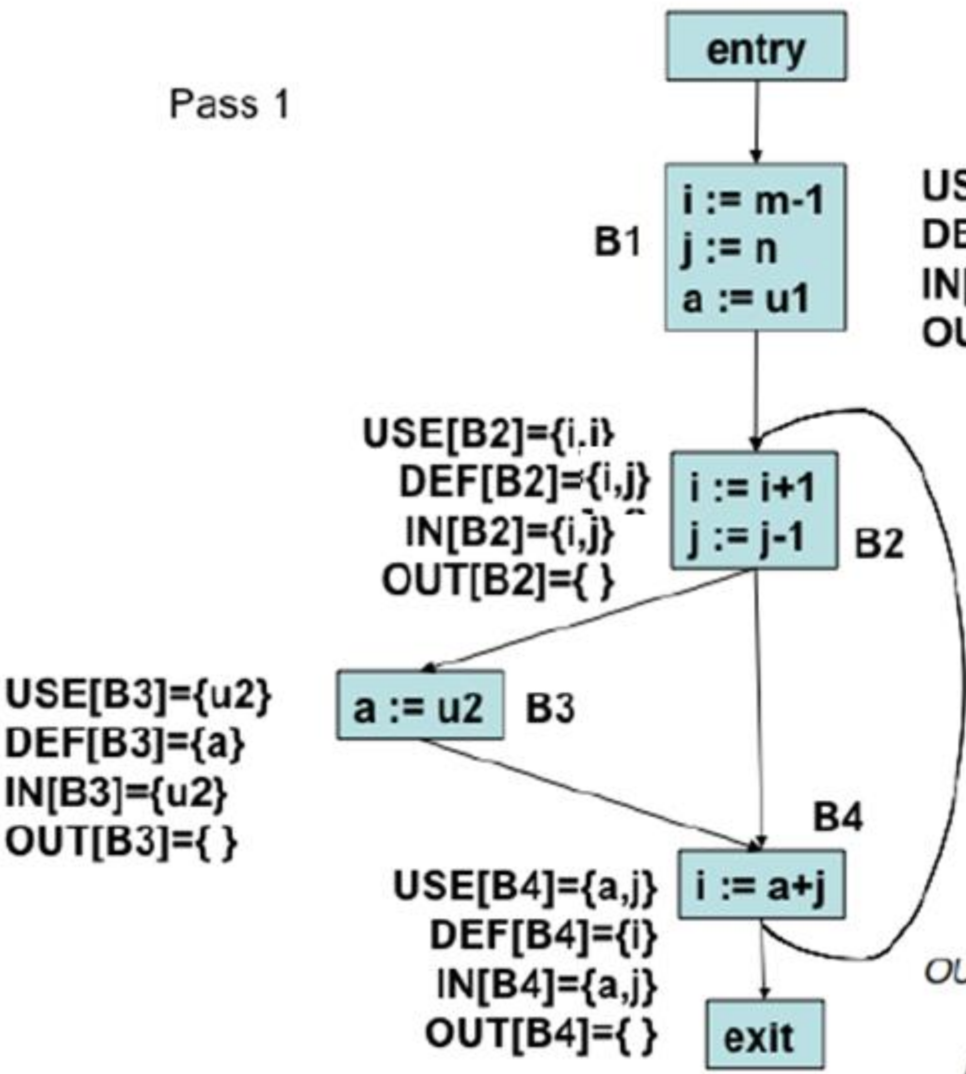
INPUT: A flow graph with *def* and *use* computed for each block.

OUTPUT: $\text{IN}[B]$ and $\text{OUT}[B]$, the set of variables live on entry and exit of each block B of the flow graph.

METHOD: Execute the program

$\text{IN}[\text{EXIT}] = \emptyset;$
for (each basic block B other than EXIT) $\text{IN}[B] = \emptyset;$
while (changes to any IN occur)
  for (each basic block B other than EXIT) {
    $\text{OUT}[B] \; = \; \bigcup_{S \text{ a successor of } B} \text{IN}[S];$
    $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B);$
  }

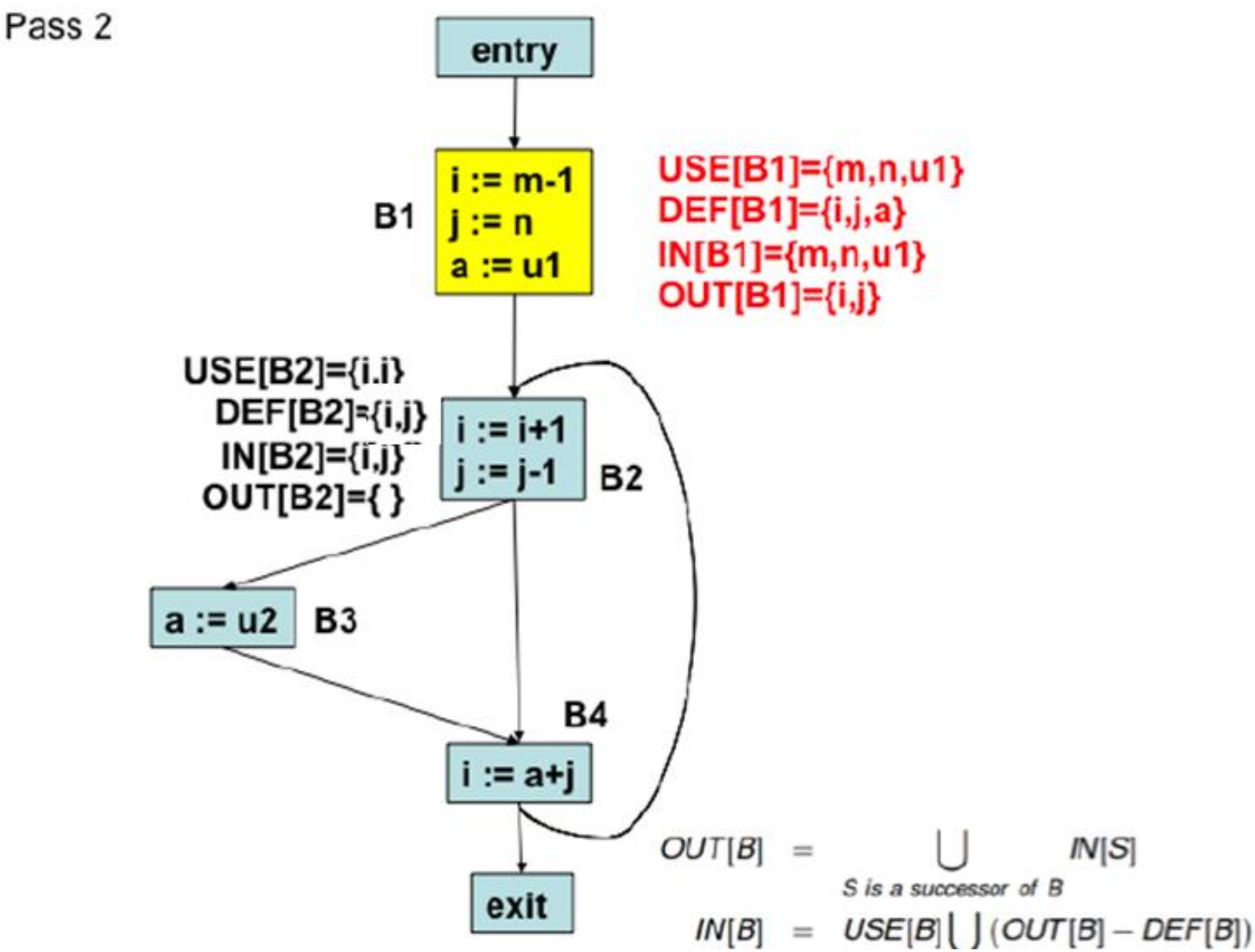
Live Variable Example



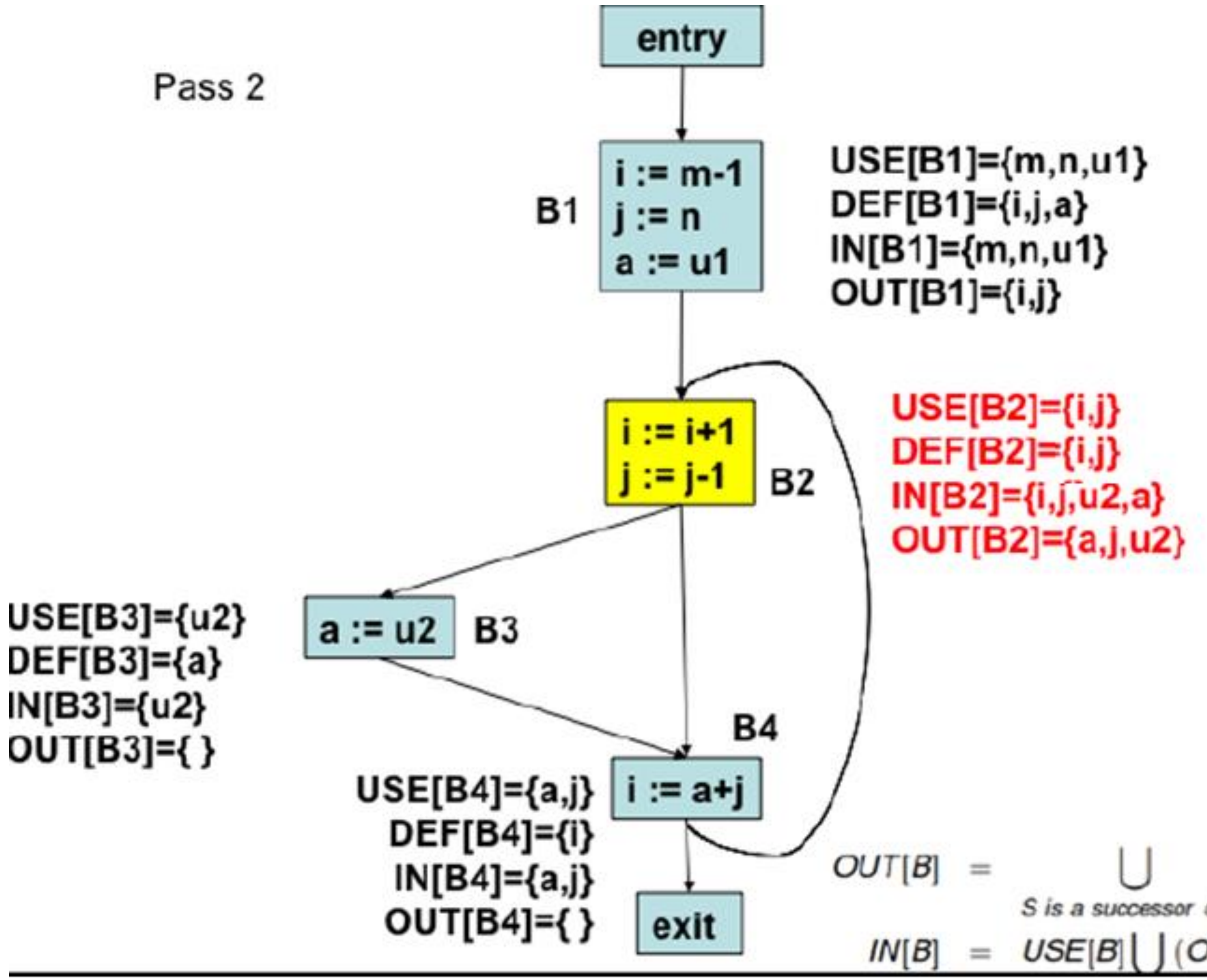
$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Example



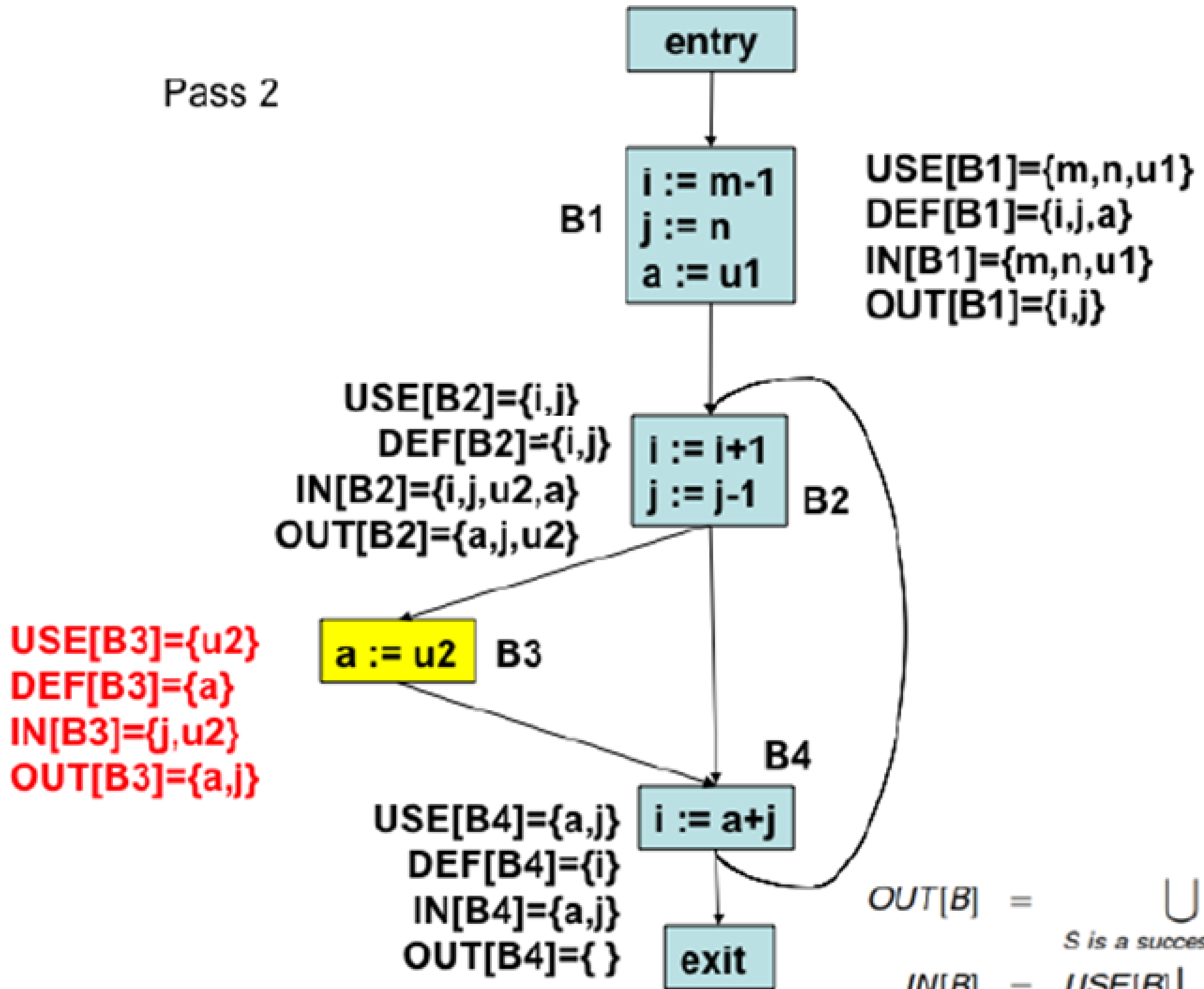
Live Variable Example



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

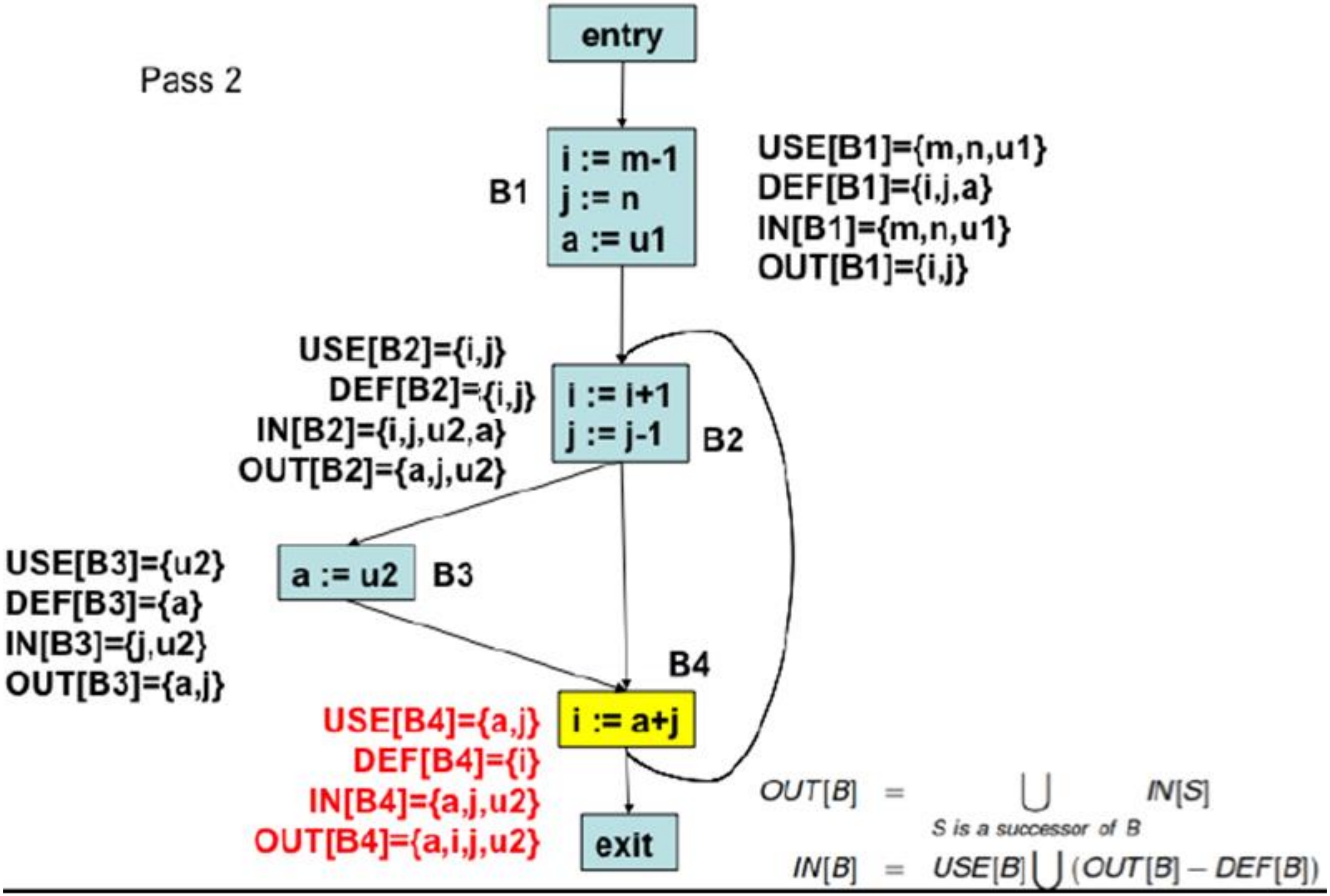
Live Variable Example



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

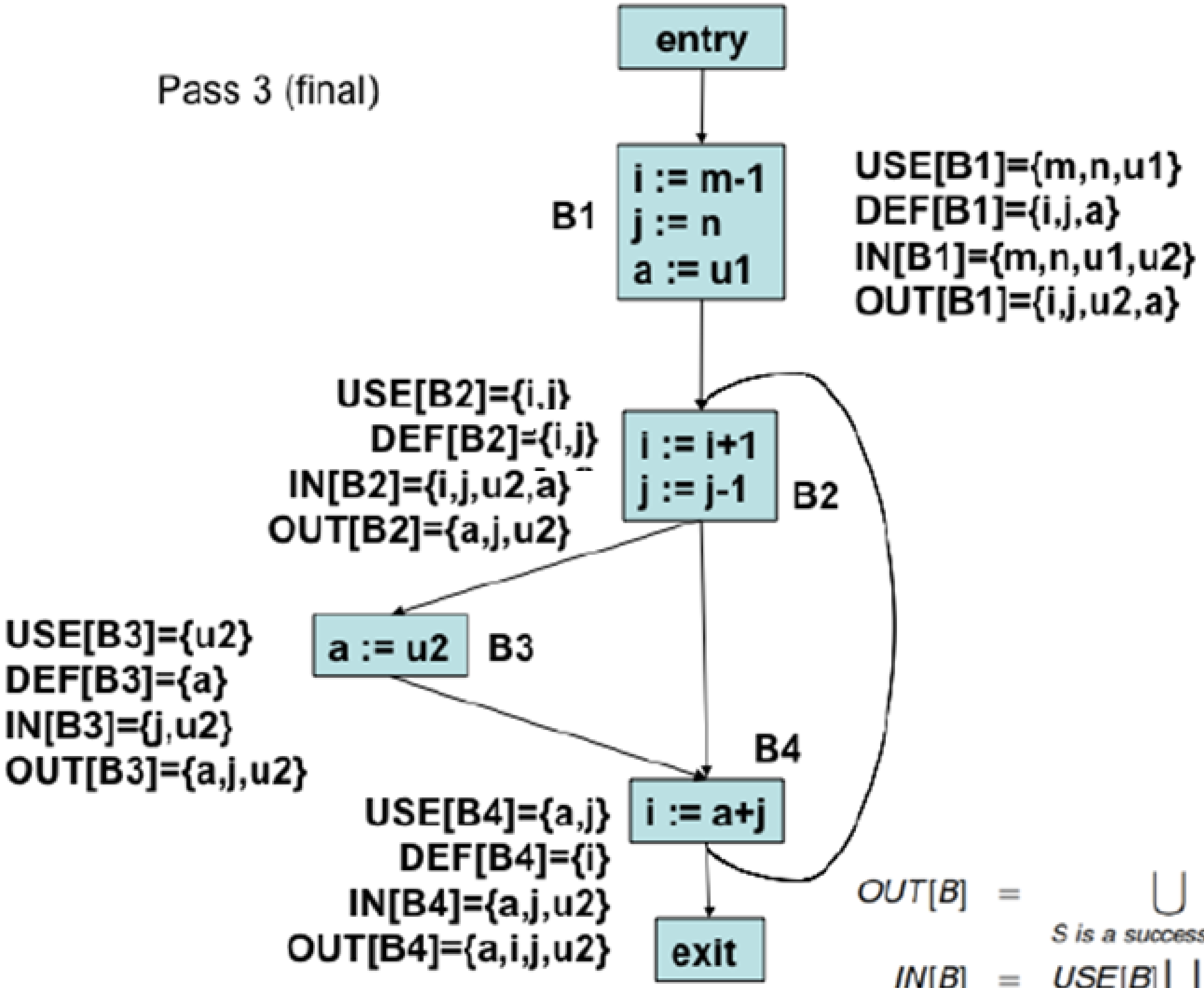
$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Example



Live Variable Example

Pass 3 (final)



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$