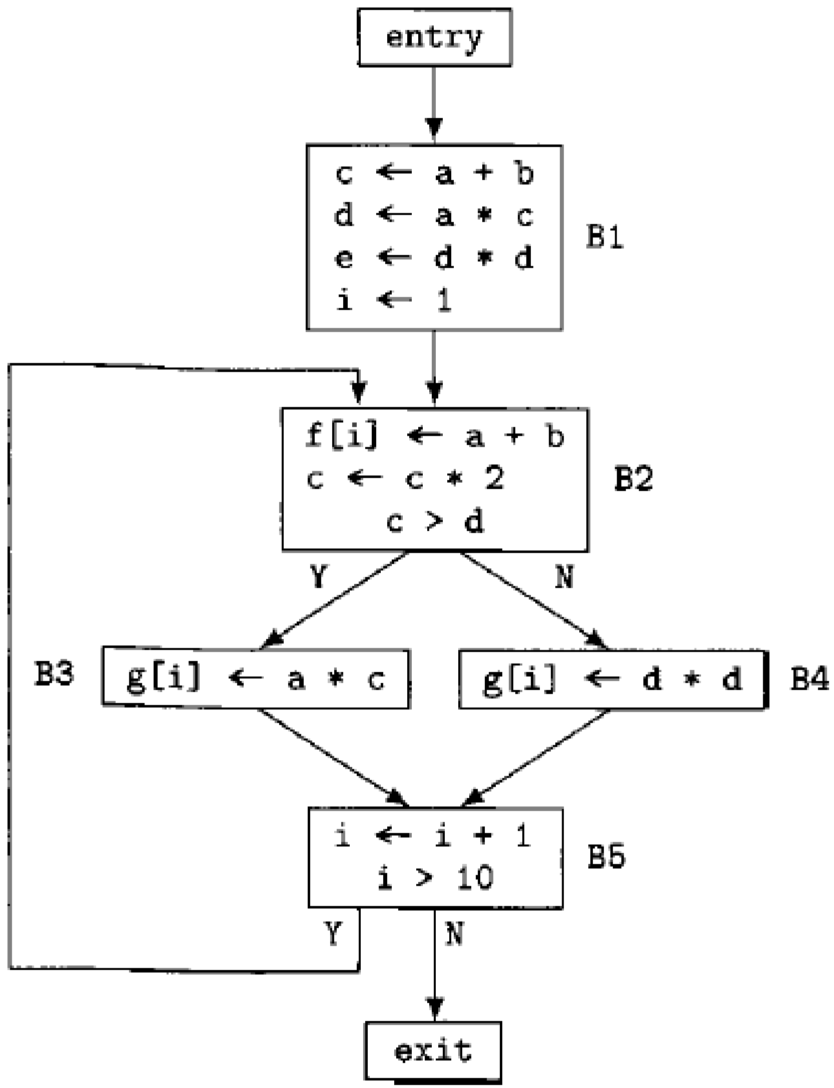
CC Lecture 7

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Example: Given a flow graph



Applying the iterative algorithm, the final values are:

- in(entry) = Ø
- in(B1) = Ø
- in(B2) = {a+b, a*c, d*d}
- in(B3) = {a+b, c>d, d*d}
- in(B4) = {a+b, c>d, d*d}
- in(B5) = {a+b, c>d, d*d}
- in(exit) = {i>10, a+b, c>d, d*d}

Global common-subexpression elimination using the AEin() data-flow function

- For simplicity, we assume that local common-subexpression elimination has already been done, so that only the first evaluation of an expression in a block is a candidate for global common-subexpression elimination.

Procedure

- For each block **i** and expression **exp ∈ AEin(i)** evaluated in block i,

1. Locate the first evaluation of **exp** in block **i**.
2. Search backward from the first occurrence to determine whether any of the operands of exp have been previously assigned to in the block.

   If so, this occurrence of **exp** is not a global common subexpression; proceed to another expression or another block as appropriate.

Procedure

3. Having found the first occurrence of **exp** in block **i** and determined that it is a global common subexpression,

   search backward in the flowgraph to find the occurrences of exp, such as in the context **v** $\leftarrow$ **exp**, that caused it to be in **AEin(i)**.

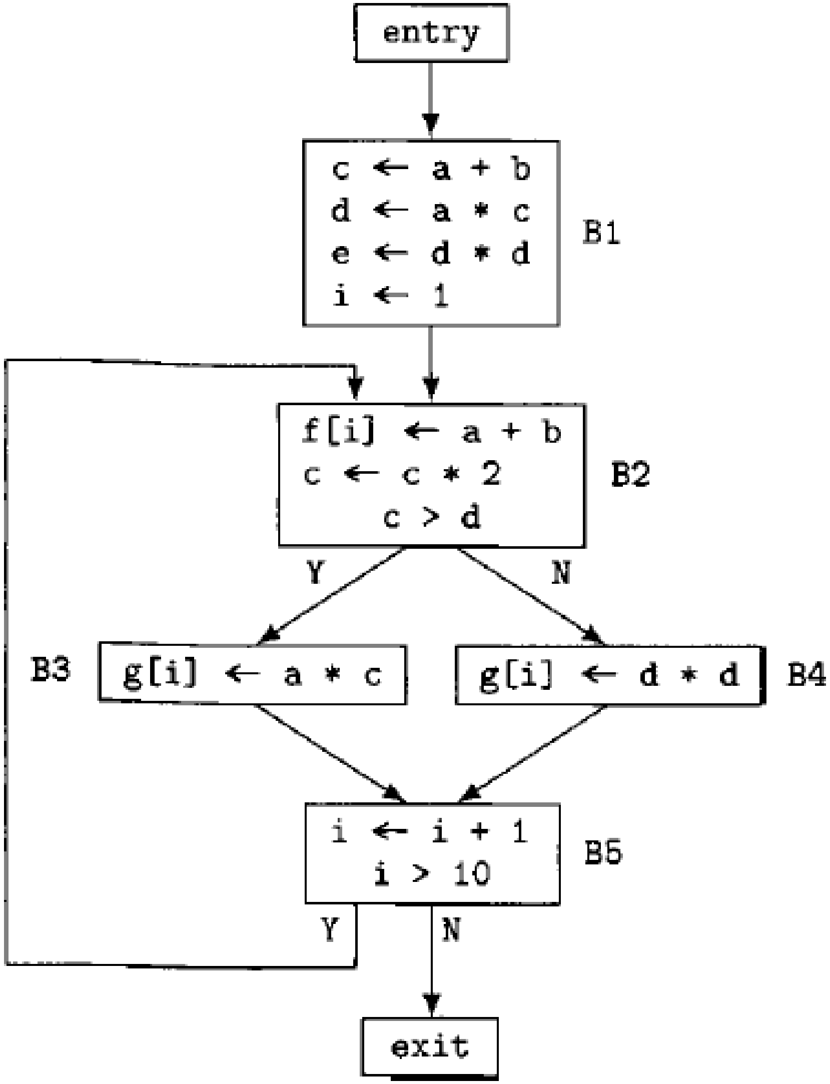
   These are the final occurrences of exp in their respective blocks; each of them must flow unimpaired to the entry of block i; and every flow path from the entry block to block i must include at least one of them.

Procedure

4. Select a new temporary variable **tj**.

   Replace the expression in the first instruction inst that uses exp in block i by tj and replace each instruction that uses exp identified in step (3) by **tj ← exp**

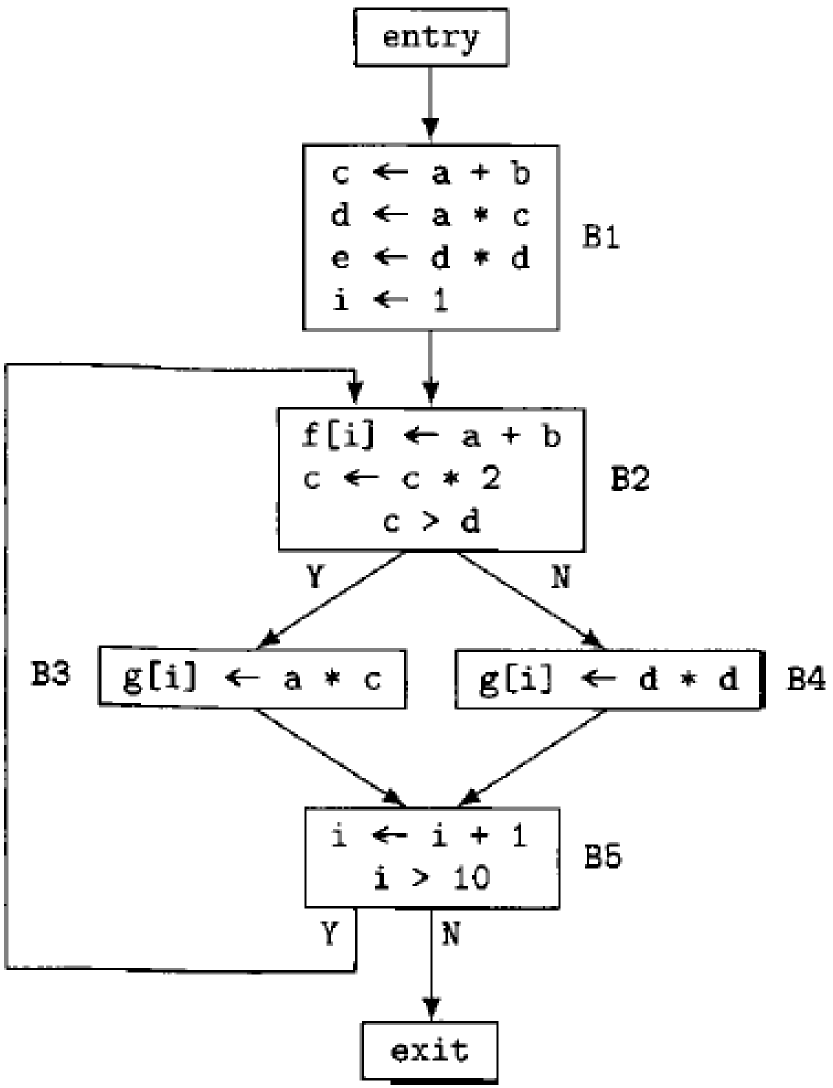
Applying the procedure to given flow graph



- in(entry) = Ø
- in(B1) = Ø

So, no expression suitable for global common subexpression elimination in B1.

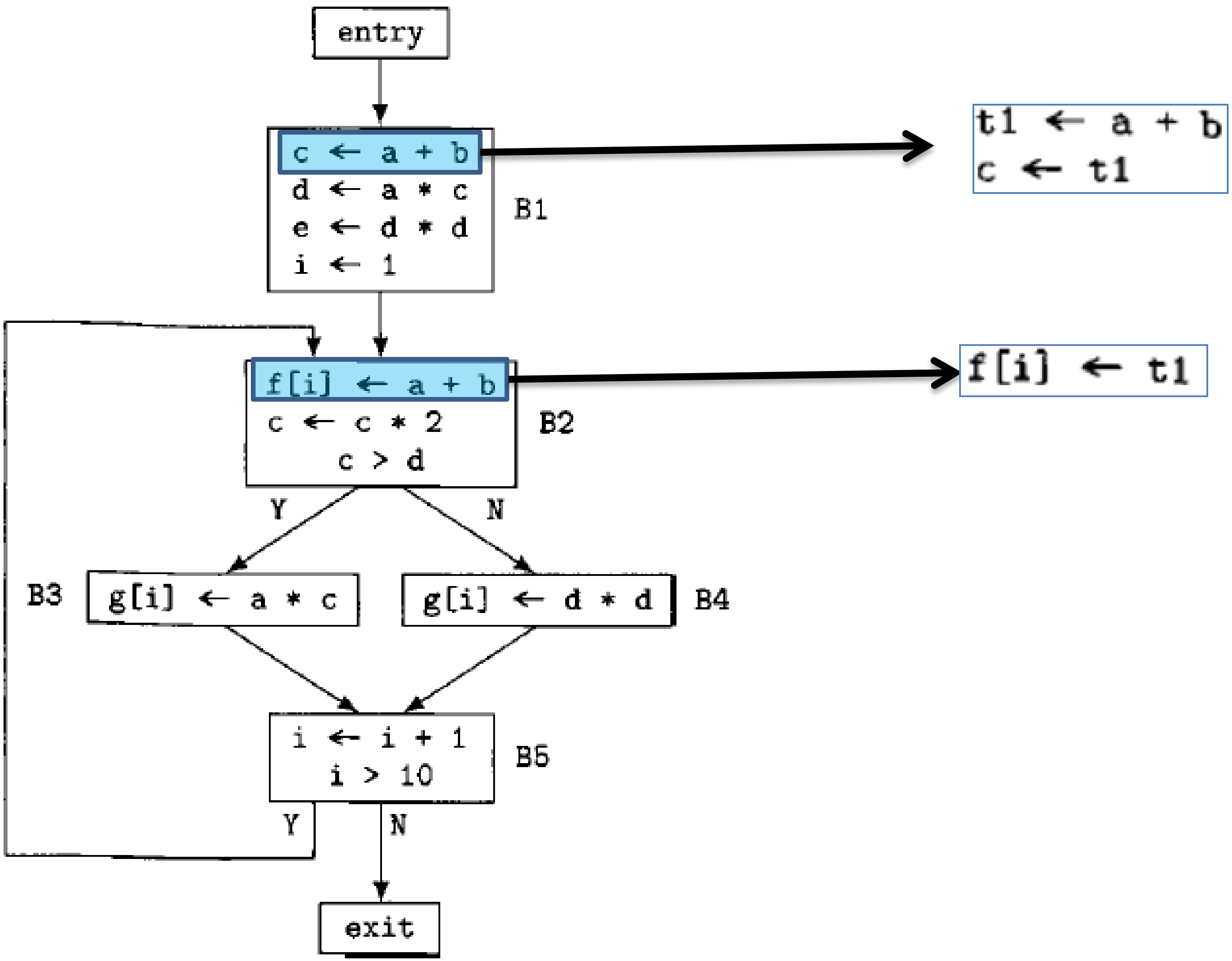
Applying the procedure to given flow graph



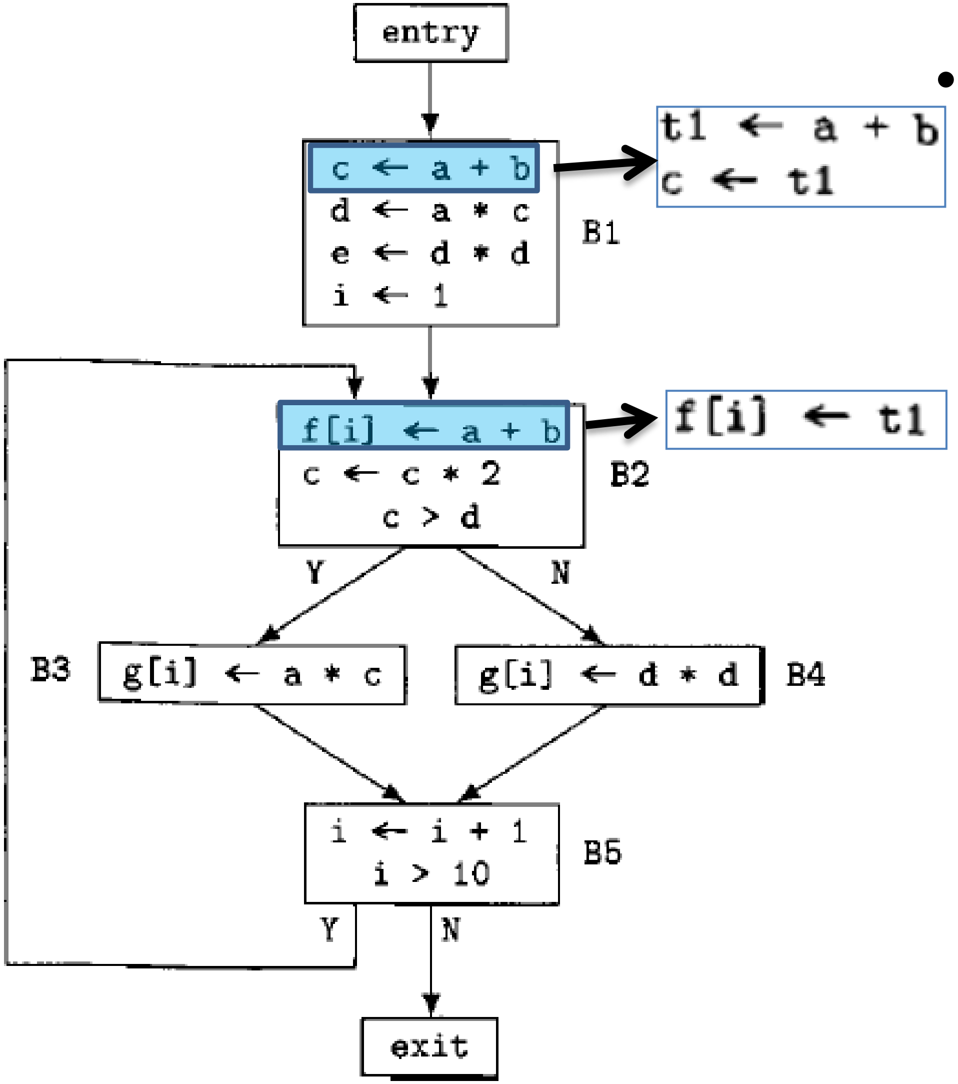
- in(B2) = {a+b, a*c, d*d}

1. a+b ∈ AEin(B2) and a+b is found/located in B2
2. a or b have not been assigned previously in the block.
3. Searching backward from it, we find the instruction **c ← a+b in B1**
4. replace it by **t1 ← a+b** and **c ← t1** and the instruction in block **B2** by **f [i] ← t1.**

Applying the procedure to given flow graph



Applying the procedure to given flow graph

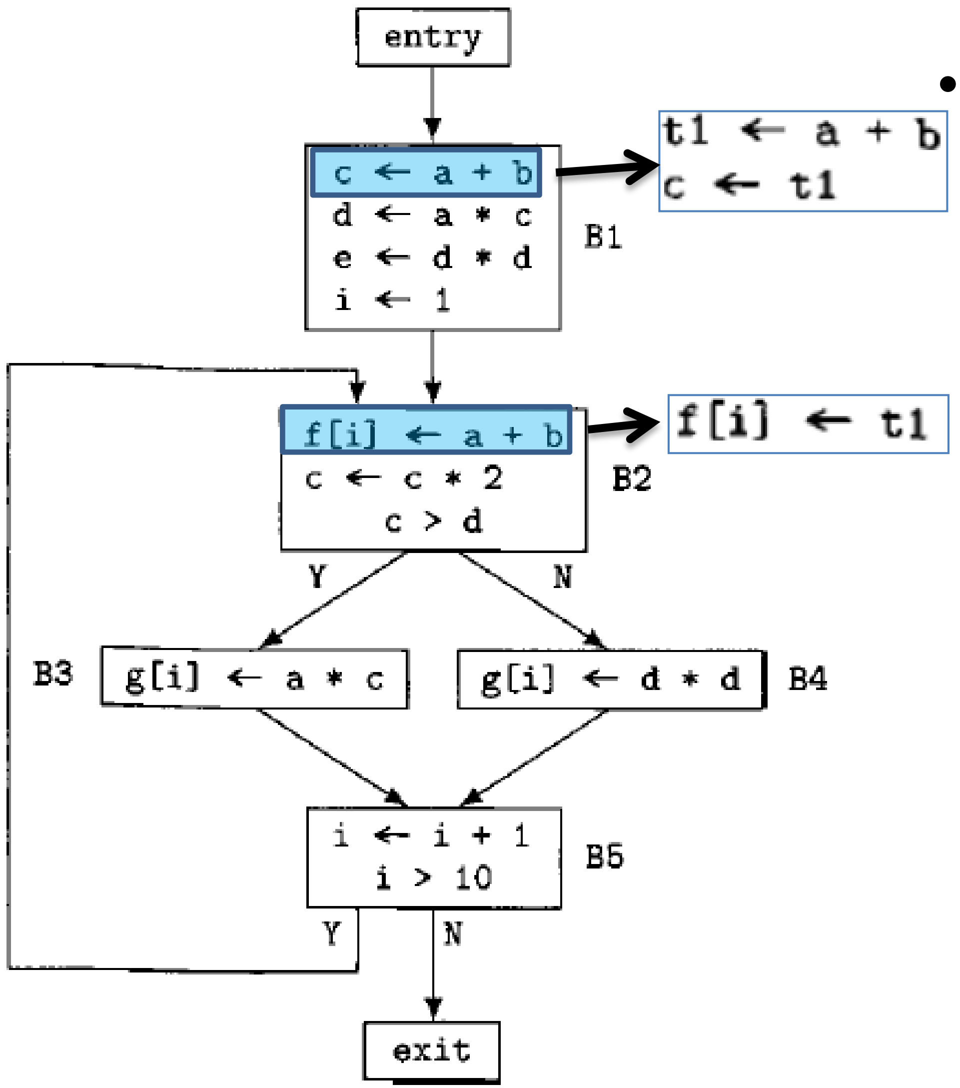


- in(B2) = {a+b, a*c, d*d}

  a*c ∈ AEin(B2) but a*c not found or located in B2

  d*d ∈ AEin(B2) but d*d not found or located in B2

Applying the procedure to given flow graph



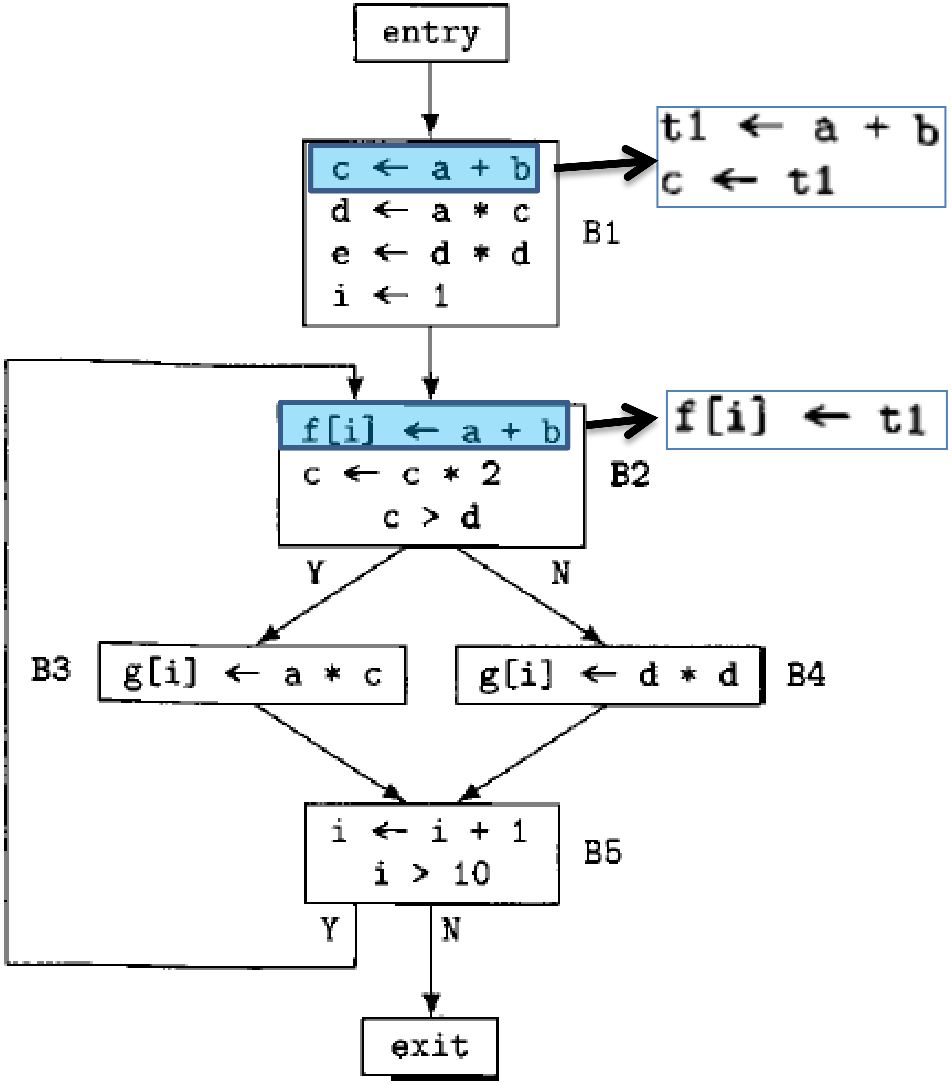
- in(B3) = {a+b, c>d, d*d}

  a+b ∈ AEin(B3) but a+b not found or located in B3

  c>d ∈ AEin(B3) but c>d not found or located in B3

  d*d ∈ AEin(B3) but d*d not found or located in B3

Applying the procedure to given flow graph

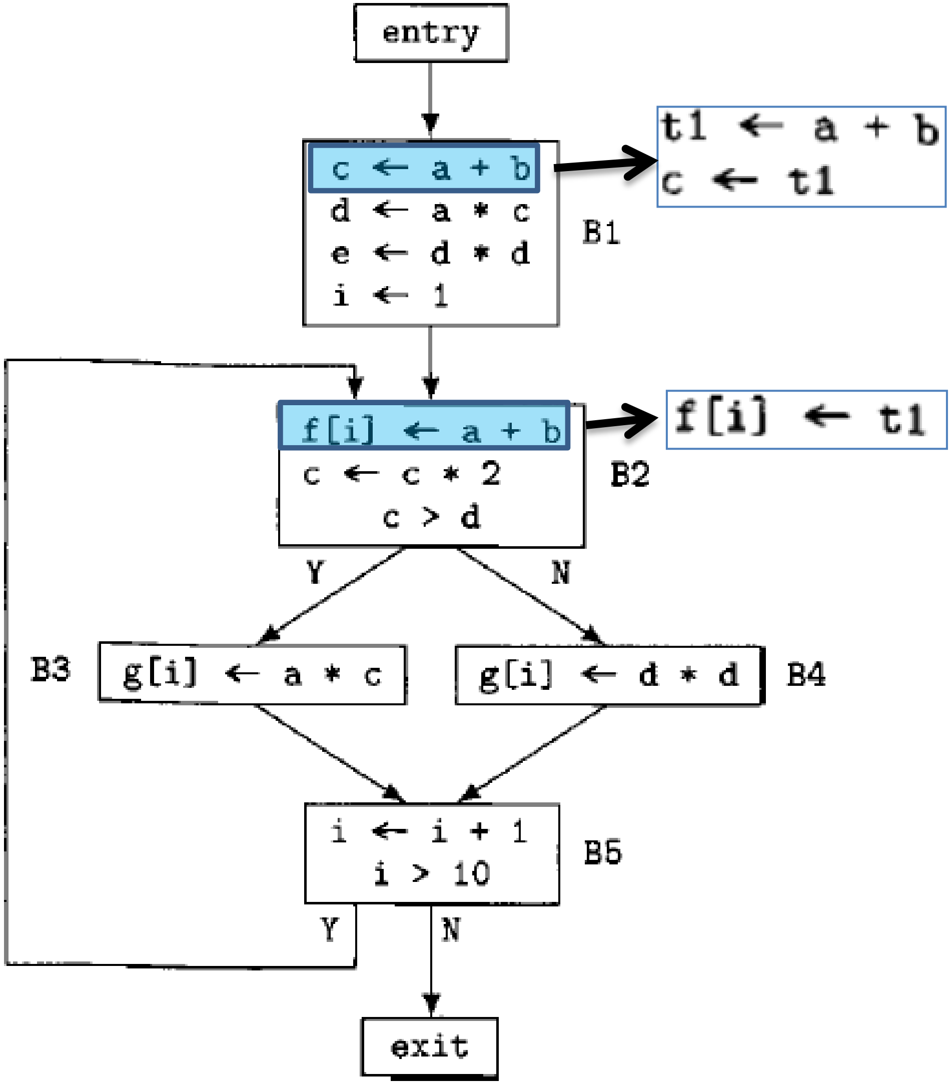


- in(B4) = {a+b, c>d, d*d}

a+b ∈ AEin(B4) but a+b not found or located in B4

c>d ∈ AEin(B4) but c>d not found or located in B4

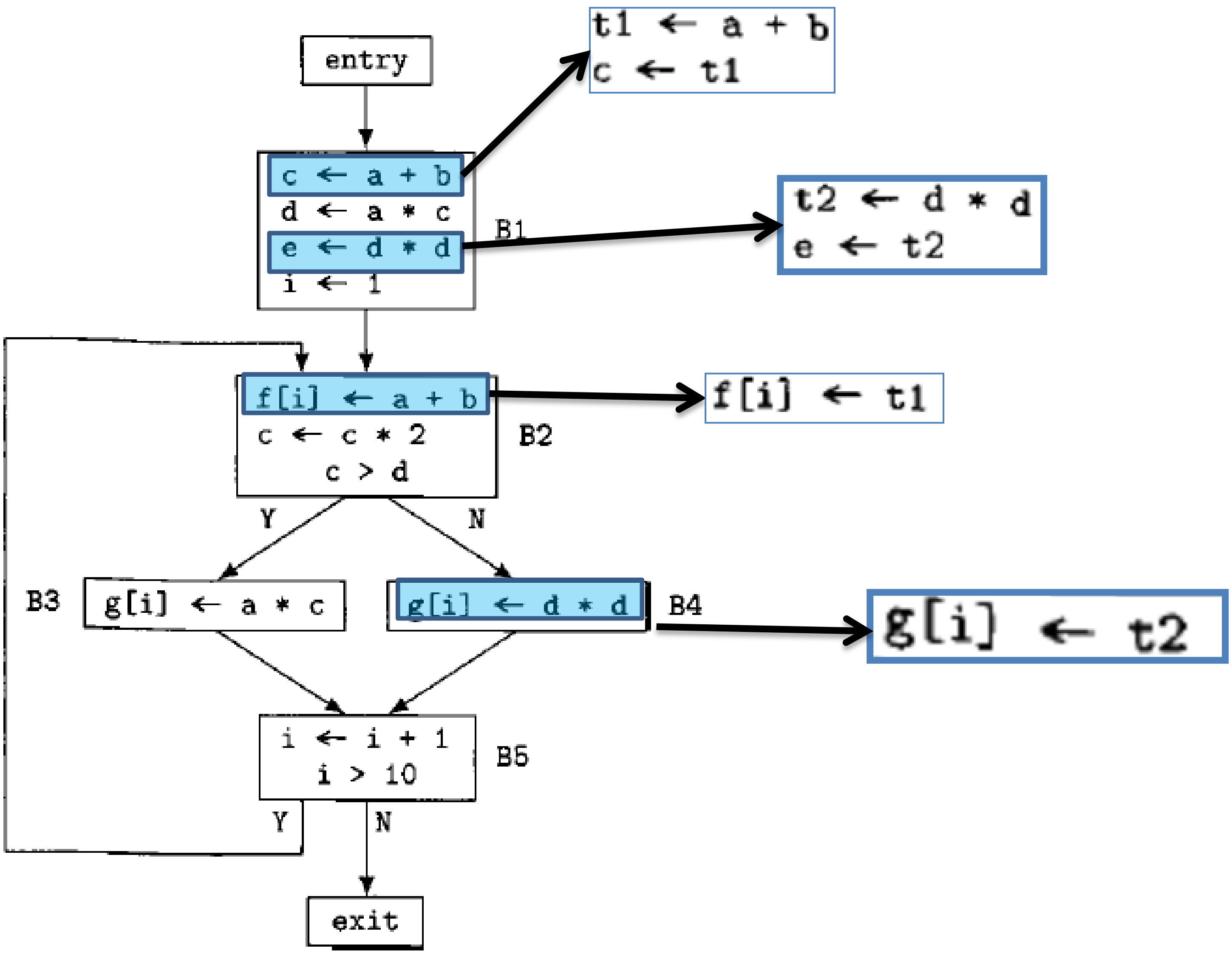
Applying the procedure to given flow graph



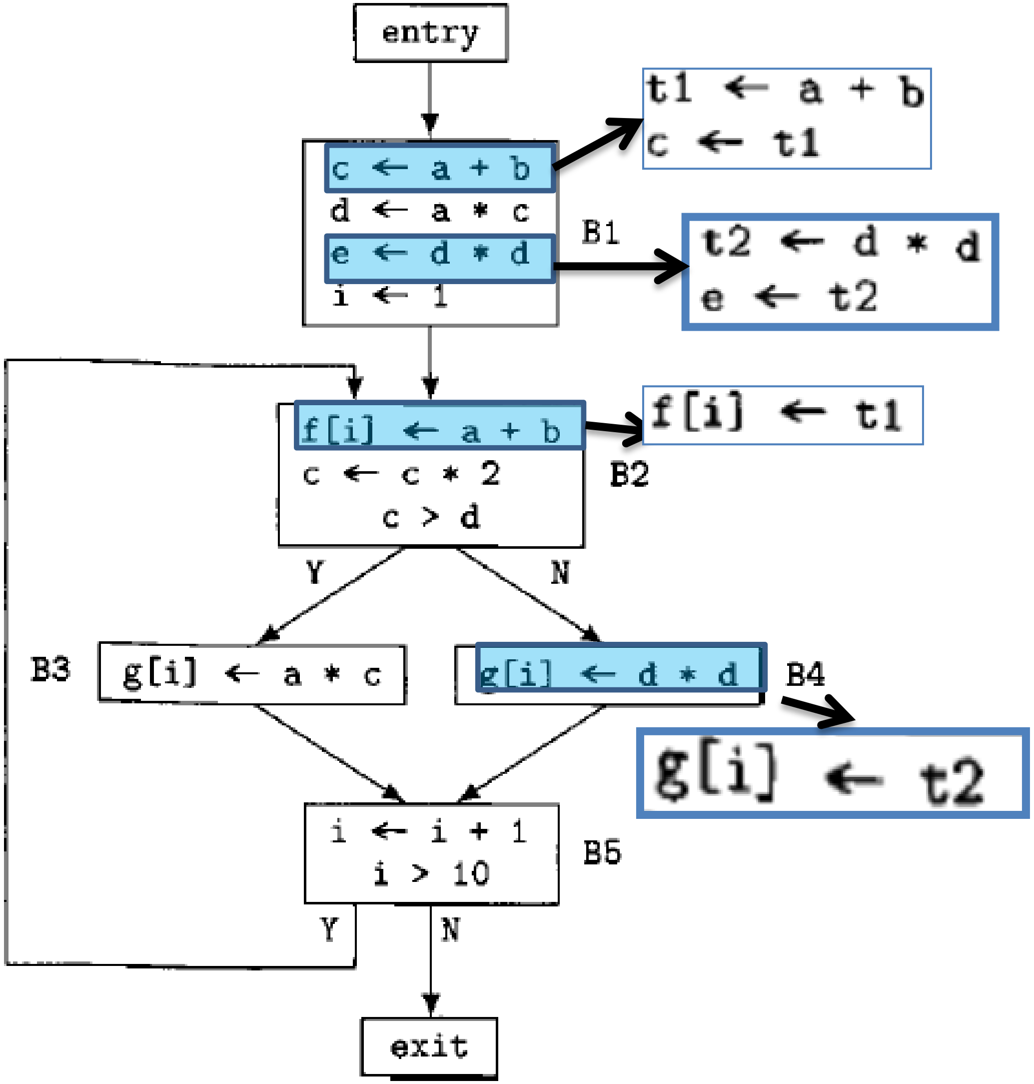
- in(B4) = {a+b, c>d, d*d}

1. d*d ∈ AEin(B4) and d*d is found/located in B4
2. d has not been assigned previously in the block.
3. Searching backward from it, we find the instruction **e ← d*d in B1**
4. replace it by **t2 ← d*d** and **e ← t2** and the instruction in block **B4** by **g[i] ← t2.**

Applying the procedure to given flow graph



Applying the procedure to given flow graph



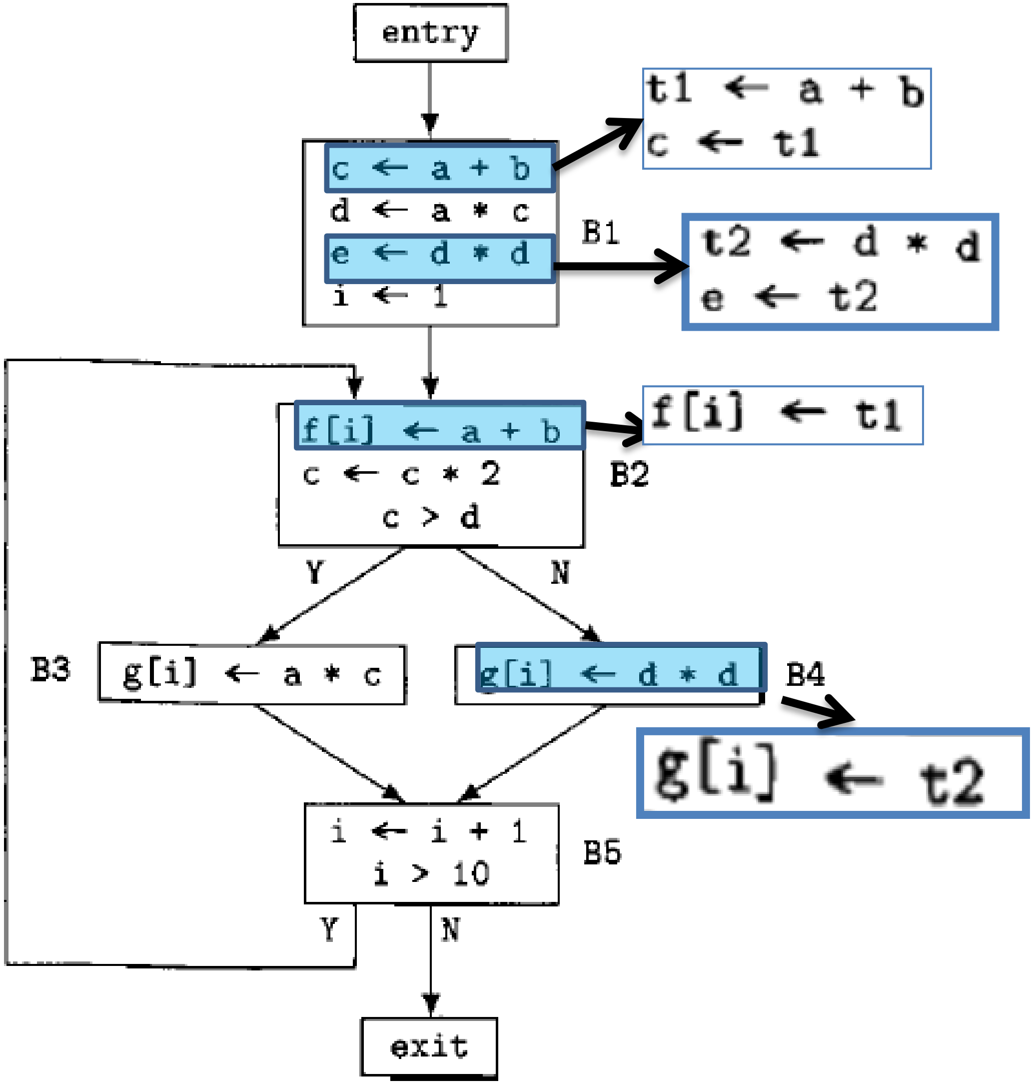
- in(B5) = {a+b, c>d, d*d}

  a+b ∈ AEin(B5) but a+b not found or located in B5

  c>d ∈ AEin(B5) but c>d not found or located in B5

  d*d ∈ AEin(B5) but d*d not found or located in B5

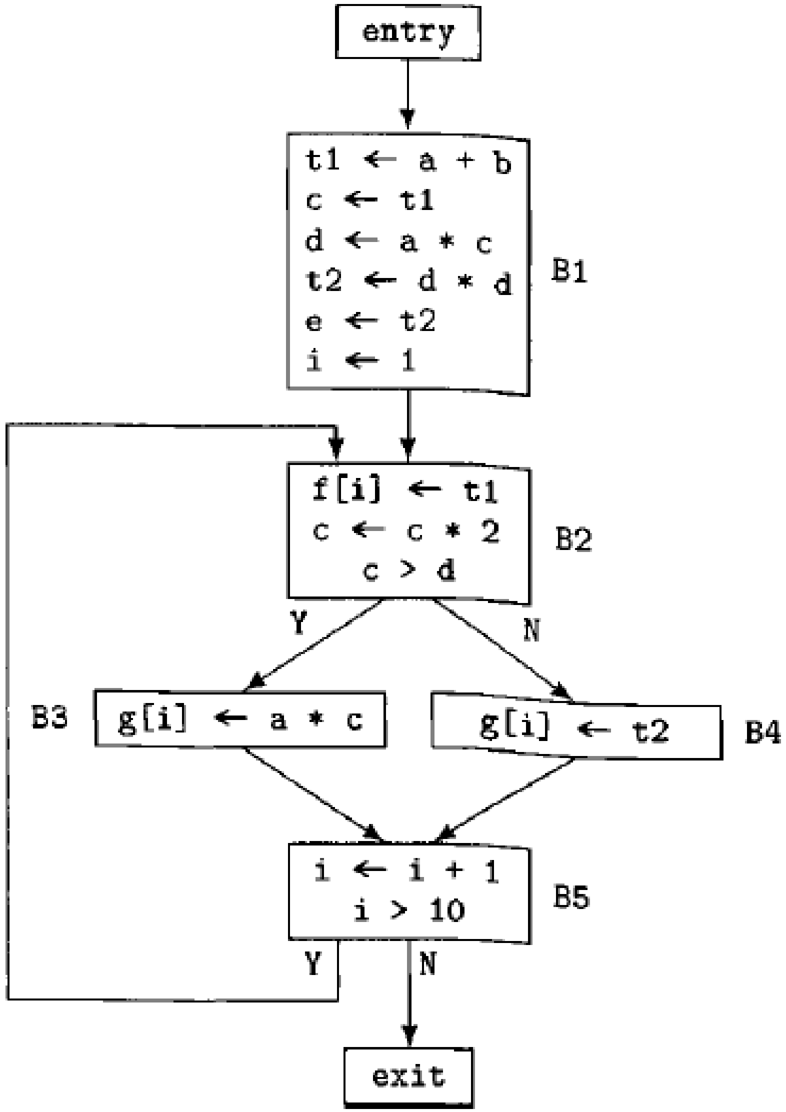
Applying the procedure to given flow graph



- in(exit) = {i>10, a+b, c>d, d*d}

  no instructions in exit block.

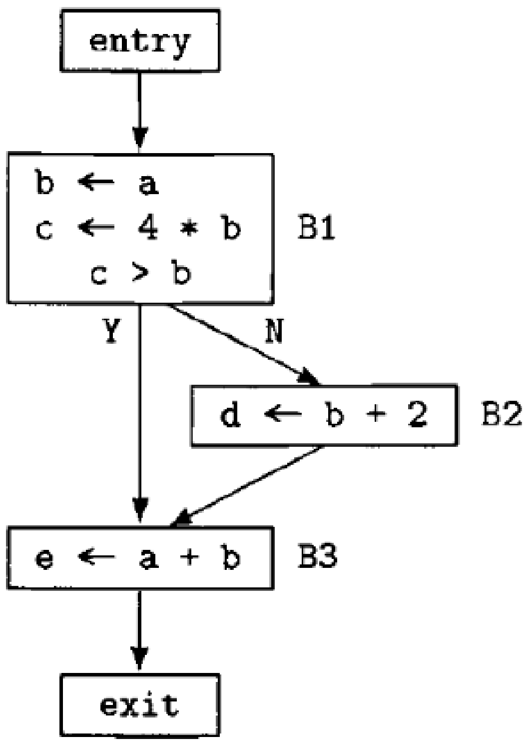
After global common subexpression elimination

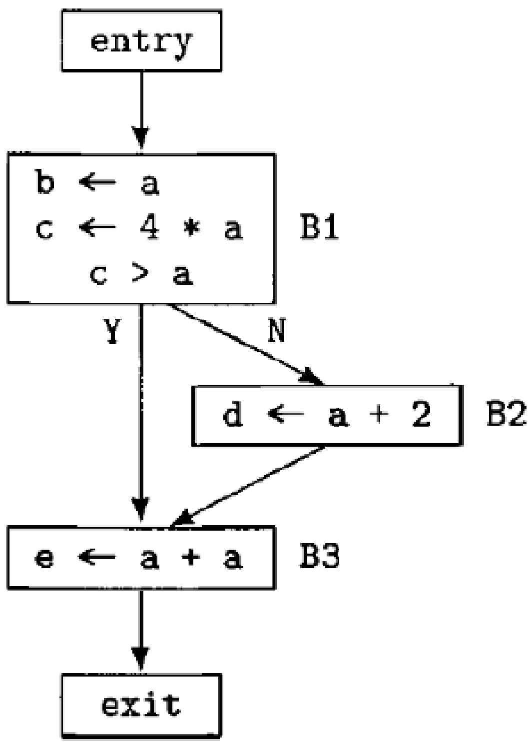


Copy Propagation

- Copy propagation is a transformation that, given an assignment **x ← y** for some variables x and y, replaces later uses of x with uses of y, as long as intervening instructions have not changed the value of **either x or y**.

Example of Copy Propagation





(a) Example of a copy assignment to propagate, namely, b ← a in **B1**

(b) the result of doing copy propagation on it.

Phases of Copy Propagation

- Copy propagation can reasonably be divided into **local** and **global** phases,
  - the first operating within individual basic blocks and
  - the latter across the entire flow- graph,
- or it can be accomplished in a single global phase.

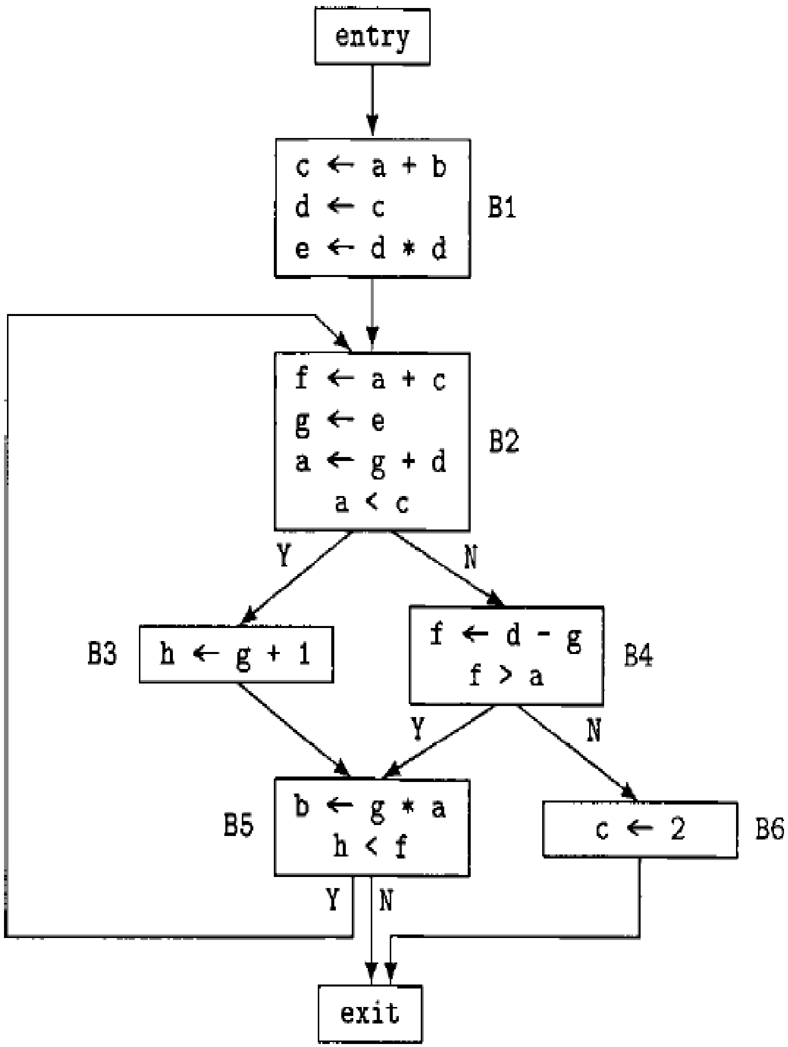
Example 1: Basic block of 5 instructions

Position	Code Before	ACP	Code After
		$\emptyset$	
1	b ← a		b ← a
		{⟨b,a⟩}	
2	c ← b + 1		c ← a + 1
		{⟨b,a⟩}	
3	d ← b		d ← a
		{⟨b,a⟩,⟨d,a⟩}	
4	b ← d + c		b ← a + c
		{⟨d,a⟩}	
5	b ← d		b ← a
		{⟨d,a⟩,⟨b,a⟩}	

- The first column shows the position
- The second column shows a basic block of five instructions before applying the ACP algorithm
- The third column shows the value of ACP at each step
- The fourth column shows the result of applying ACP

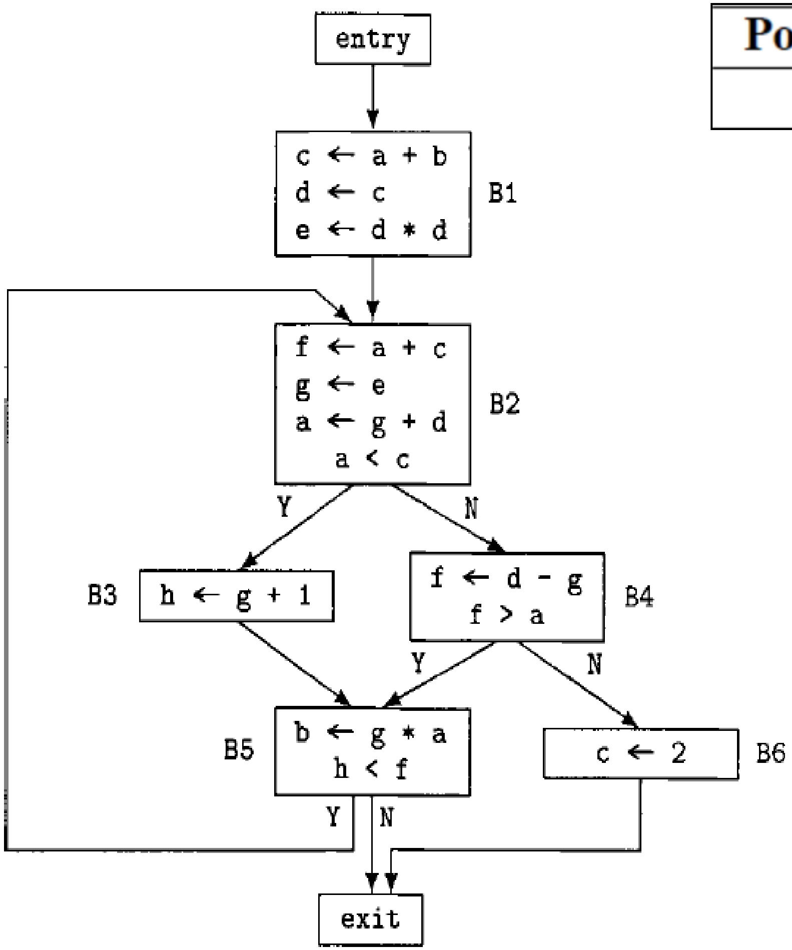
- ACP = Available Copy Propagation

Example 2



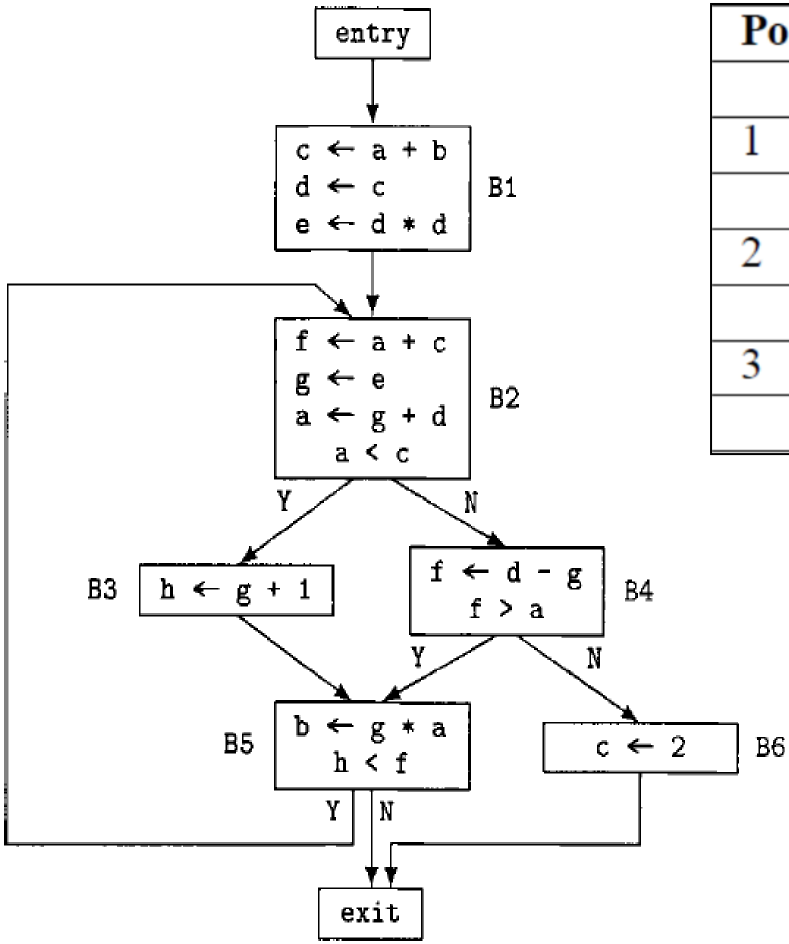
- This is the flow graph **before** copy propagation.

Local Copy Propagation on block entry



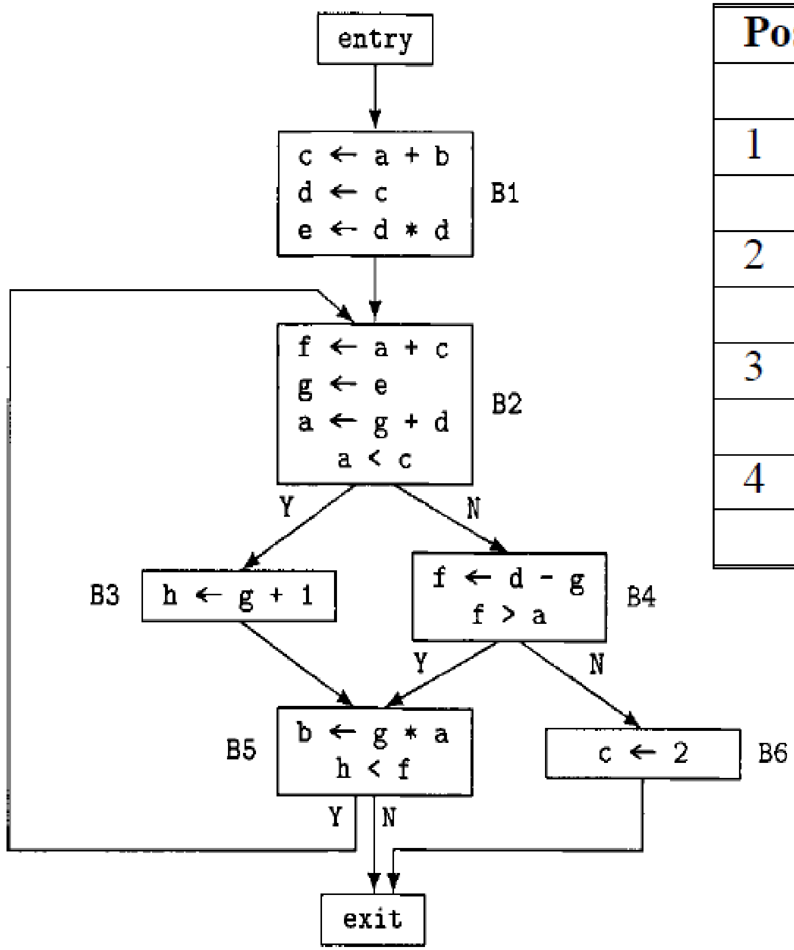
Position	Code Before	ACP	Code After
		$\emptyset$	

Local Copy Propagation on block B1



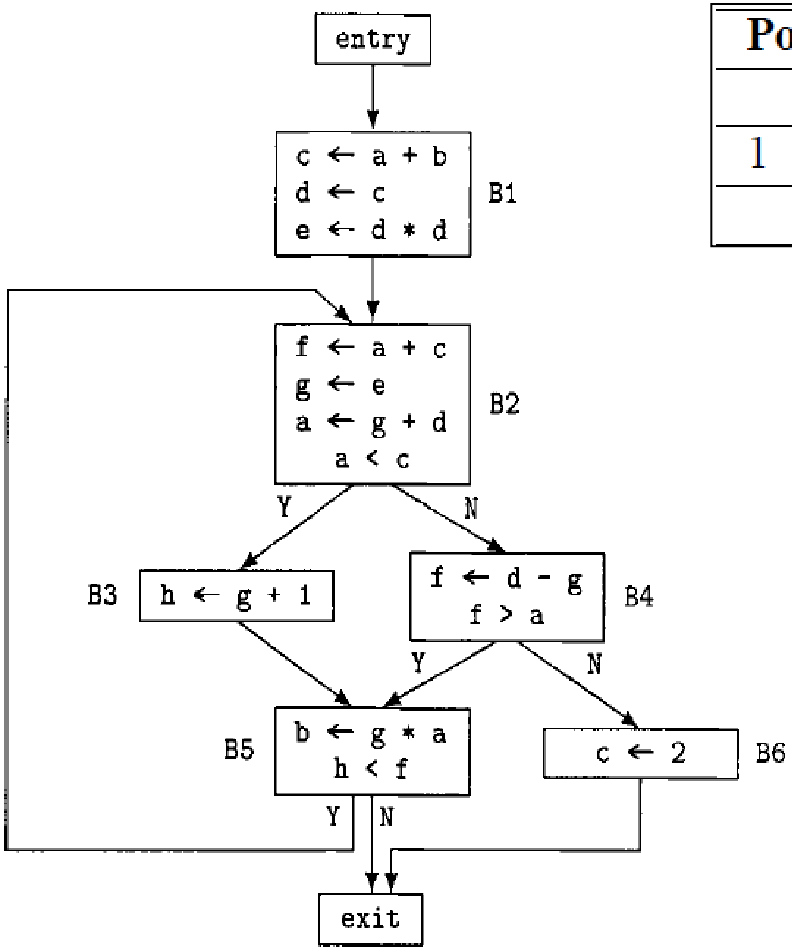
Position	Code Before	ACP	Code After
		$\emptyset$	
1	c ← a + b		c ← a + b
		$\emptyset$	
2	d ← c		d ← c
		$\{\langle d, c\rangle\}$	
3	e ← d * d		e ← c * c
		$\{\langle d, c\rangle\}$	

Local Copy Propagation on block B2



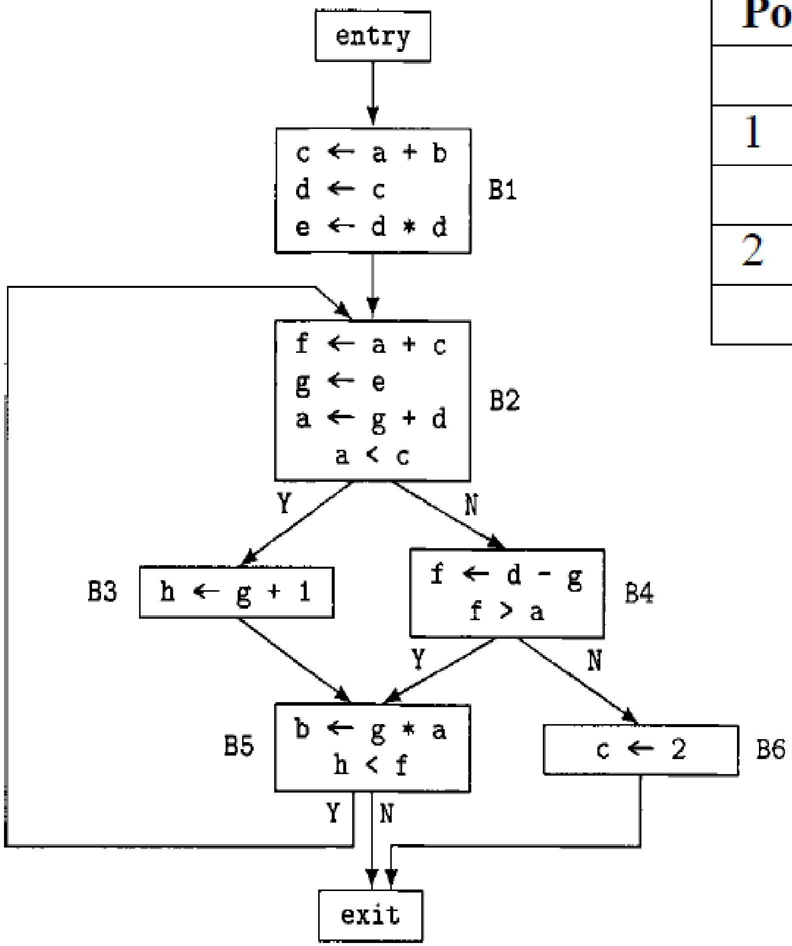
Position	Code Before	ACP	Code After
		$\emptyset$	
1	f ← a + c		f ← a + c
		$\emptyset$	
2	g ← e		g ← e
		$\{\langle g, e\rangle\}$	
3	a ← g + d		a ← e + d
		$\{\langle g, e\rangle\}$	
4	a < c		a < c
		$\{\langle g, e\rangle\}$	

Local Copy Propagation on block B3



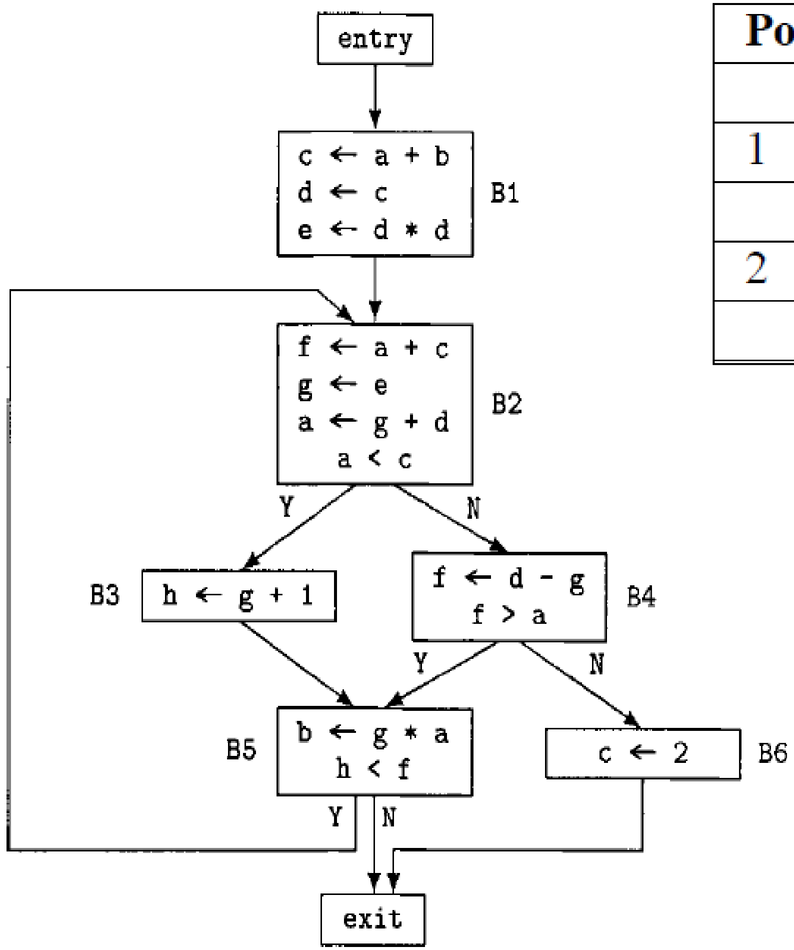
Position	Code Before	ACP	Code After
		∅	
1	h ← g + 1		h ← g + 1
		∅	

Local Copy Propagation on block B4



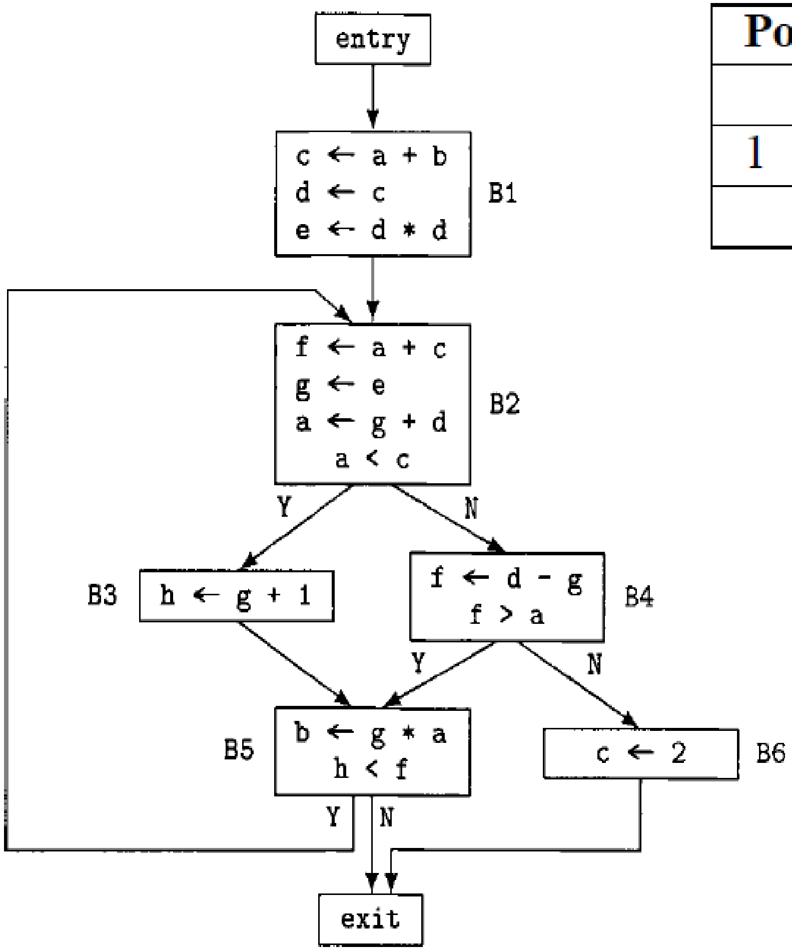
Position	Code Before	ACP	Code After
		$\emptyset$	
1	f ← d - g		f ← d - g
		$\emptyset$	
2	f < a		f < a
		$\emptyset$	

Local Copy Propagation on block B5



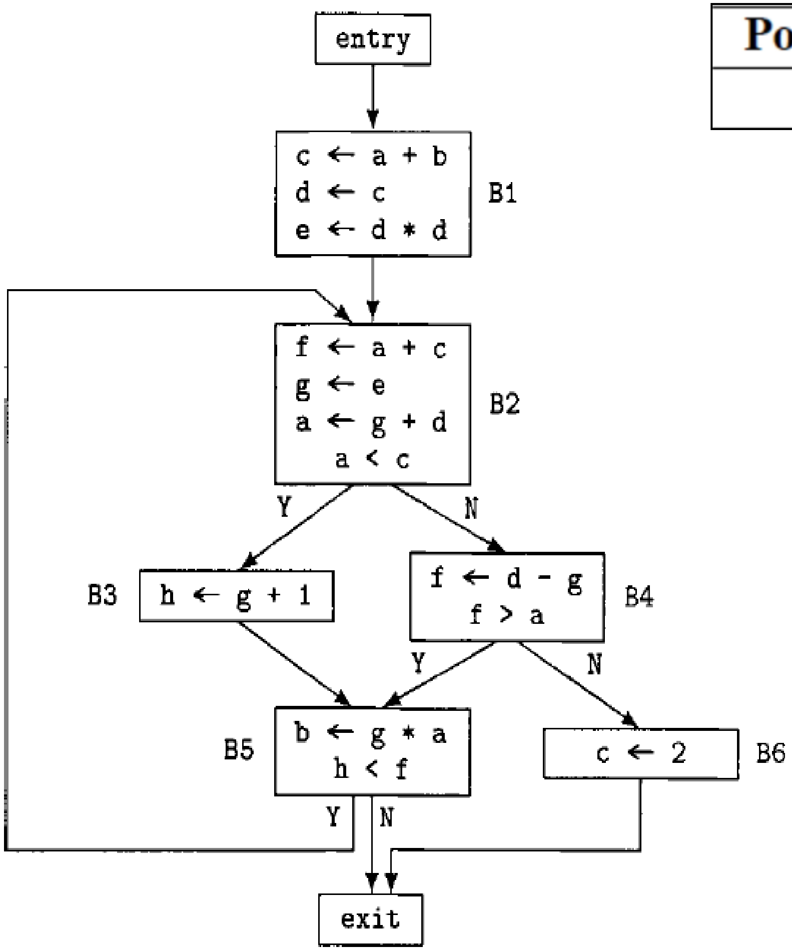
Position	Code Before	ACP	Code After
		$\emptyset$	
1	b ← g * a		b ← g * a
		$\emptyset$	
2	h < f		h < f
		$\emptyset$	

Local Copy Propagation on block B6



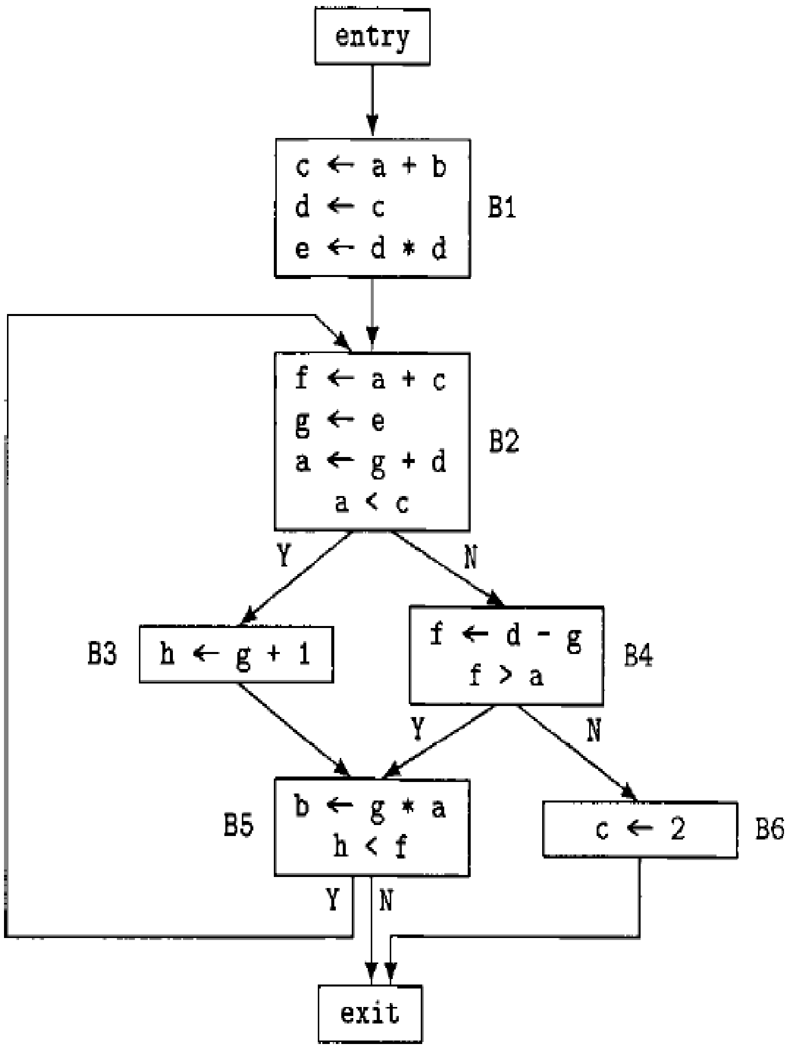
Position	Code Before	ACP	Code After
		$\emptyset$	
1	c ← 2		c ← 2
		$\emptyset$	

Local Copy Propagation on block exit



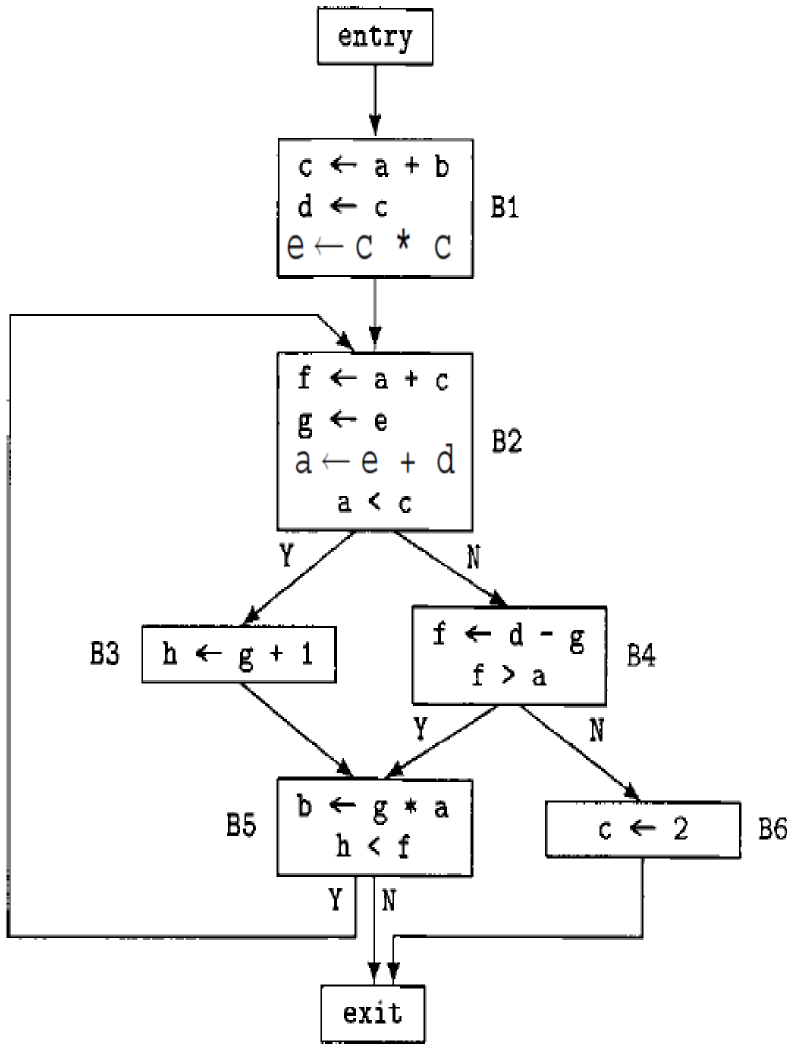
Position	Code Before	ACP	Code After
		$\emptyset$	

Before local copy propagation



- This is the flow graph **before** local copy propagation.

After local copy propagation



- This is the flow graph **after** local copy propagation.

Global Copy Propagation

- To perform global copy propagation, we first do a data-flow analysis to determine which copy assignments reach uses of their left-hand variables unimpaired, i.e., without having either variable redefined in between.

- We define the set **COPY(i)** to consist of the instances of copy assignments occurring in block i that reach the end of block i.

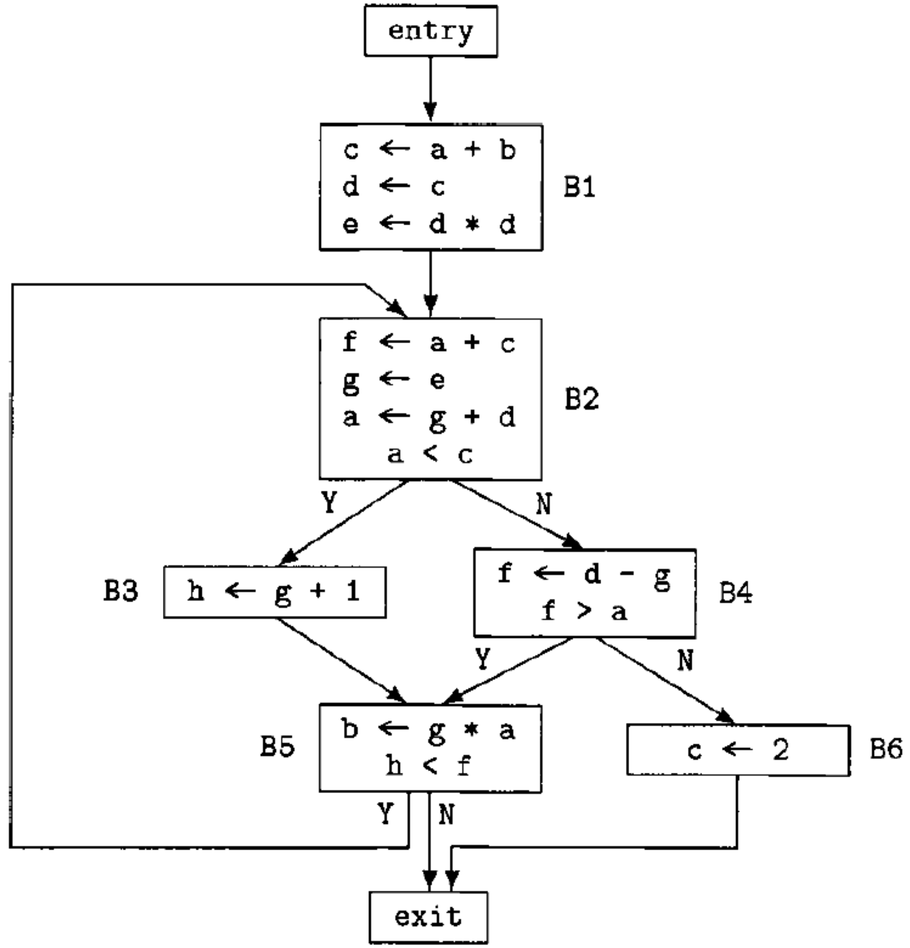
- We define **KILL(i)** to be the set of copy assignment instances killed by block i.

COPY(i) and KILL(i)

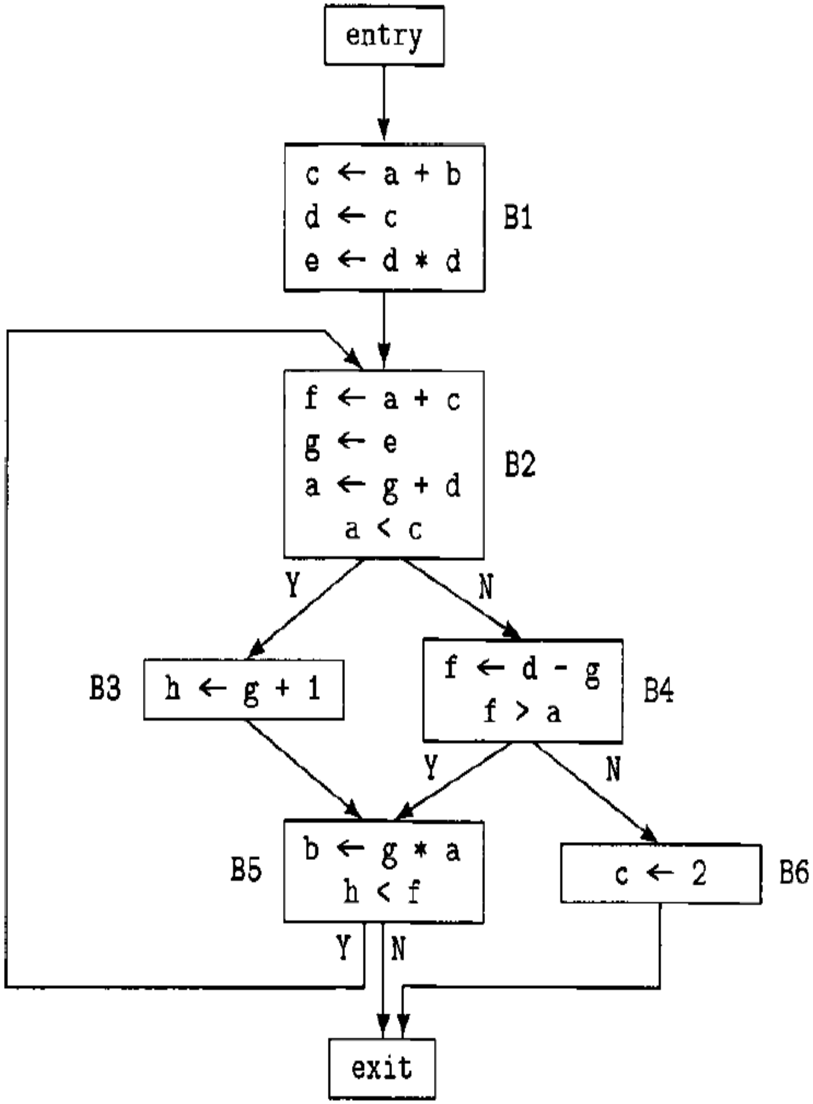
- COPY(i) is a set of quadruples **(u, v, i, pos)**,
  - such that **u ← v** is a copy assignment
  - and **pos** is the position in block **i** where the assignment occurs,
  - and neither **u** nor **v** is assigned to later in block **i**.

- KILL(i) is the set of quadruples **(u, v, blk, pos)**
  - such that **u ← v** is a copy assignment occurring at position **pos** in block **blk ≠ i**.

Find COPY(i) and KILL(i) for given flow graph

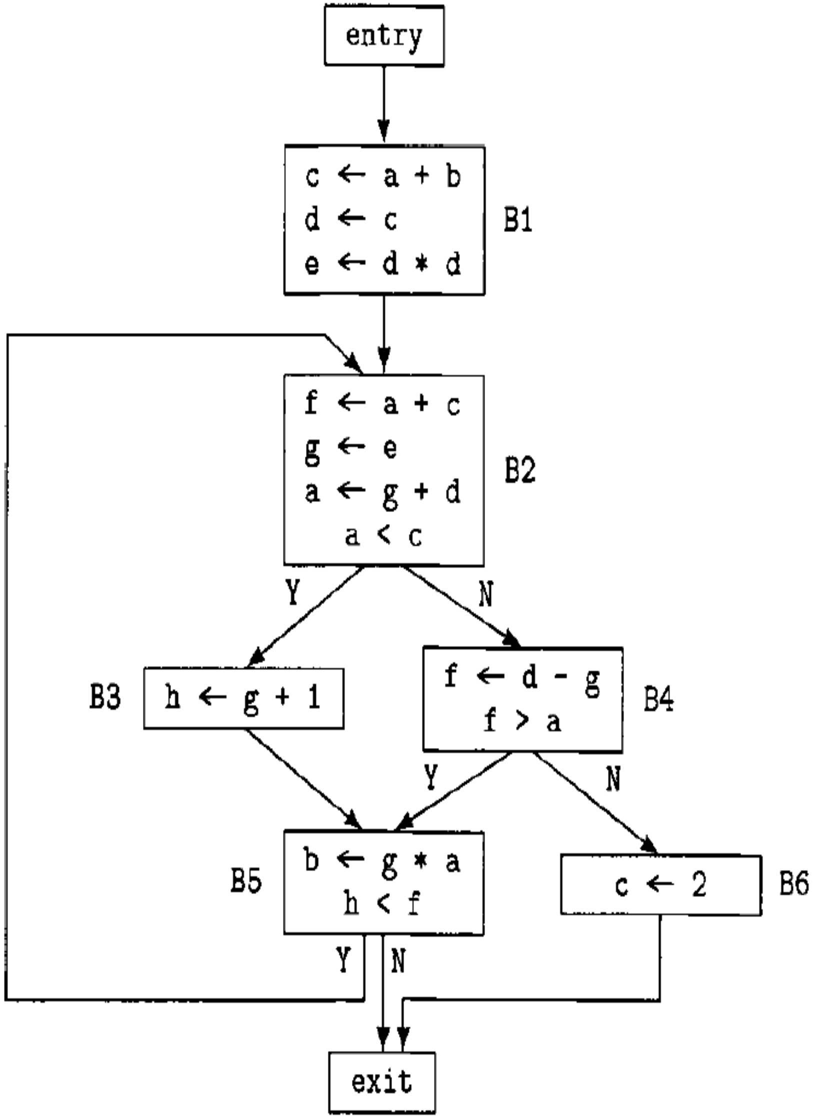


COPY(i) using set notation



- COPY(entry) = ∅
- COPY(B1) = {(d, c, B1, 2)}
- COPY(B2) = {(g, e, B2, 2)}
- COPY(B3) = ∅
- COPY(B4) = ∅
- COPY(B5) = ∅
- COPY(B6) = ∅
- COPY(exit) = ∅

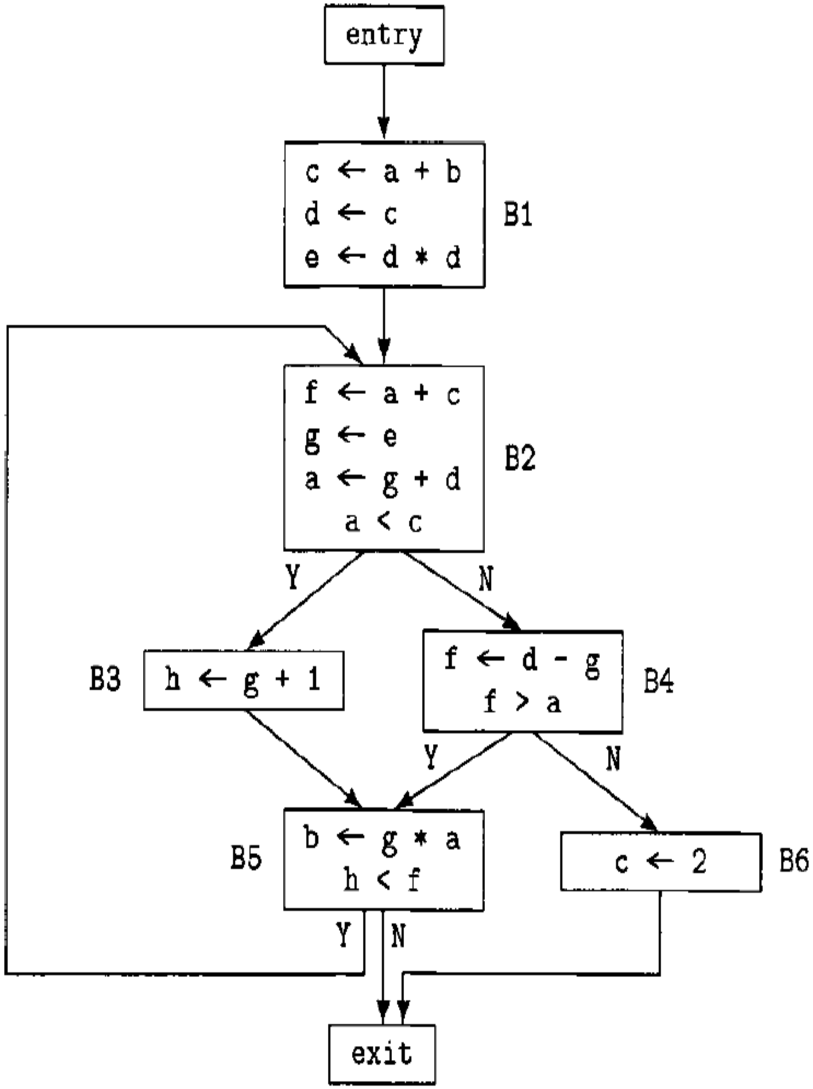
COPY(i) using vector representation



- COPY(entry) = <00>
- COPY(B1) = <10>
- COPY(B2) = <01>
- COPY(B3) = <00>
- COPY(B4) = <00>
- COPY(B5) = <00>
- COPY(B6) = <00>
- COPY(exit) = <00>

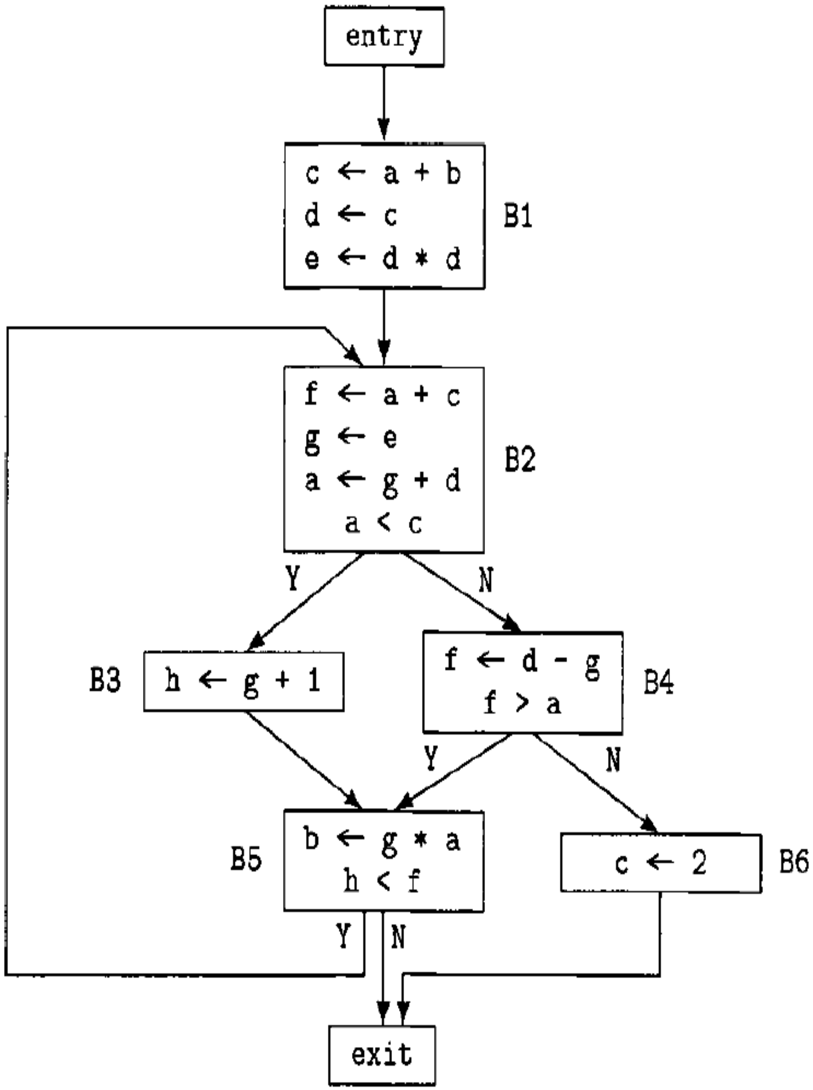
Bit position	COPY
1	{(d, c, B1, 2)}
2	{(g, e, B2, 2)}

KILL(i) using set notation



- KILL(entry) = Ø
- KILL(B1) = {(g, e, B2, 2)}
- KILL(B2) = Ø
- KILL(B3) = Ø
- KILL(B4) = Ø
- KILL(B5) = Ø
- KILL(B6) = {(d, c, B1, 2)}
- KILL(exit) = Ø

KILL(i) using vector representation



- KILL(entry) = <00>
- KILL(B1) = <01>
- KILL(B2) = <00>
- KILL(B3) = <00>
- KILL(B4) = <00>
- KILL(B5) = <00>
- KILL(B6) = <10>
- KILL(exit) = <00>

Bit position	COPY
1	{(d, c, B1, 2)}
2	{(g, e, B2, 2)}

Initialize CPin

- $CPin(x) = \emptyset$ if x = entry
- $CPin(x) = U$ otherwise, where $U = \cup\, COPY(i)$ for all i

CPin for all blocks

- CPin(entry) = Ø | <00>
- CPin(B1) = {(d, c, B1, 2),(g, e, B2, 2)} | <11>
- CPin(B2) = {(d, c, B1, 2),(g, e, B2, 2)} | <11>
- CPin(B3) = {(d, c, B1, 2),(g, e, B2, 2)} | <11>
- CPin(B4) = {(d, c, B1, 2),(g, e, B2, 2)} | <11>
- CPin(B5) = {(d, c, B1, 2),(g, e, B2, 2)} | <11>
- CPin(B6) = {(d, c, B1, 2),(g, e, B2, 2)} | <11>
- CPin(exit) = {(d, c, B1, 2),(g, e, B2, 2)} | <11>

Data-flow equations for CPin(i) and CPout(i)

- Next, we define data-flow equations for CPin(i) and CPout(i) that represent the sets of copy assignments that are available for copy propagation on entry to and exit from block i, respectively.

- A copy assignment is **available on entry** to block i if it is available on exit from all predecessors of block i, so the path-combining operator is intersection.

- A copy assignment is **available on exit** from block j if it is either in COPY(j) or it is available on entry to block j and not killed by block j, i.e., if it is in CPin(j) and not in KILL(j)

Data-flow equations

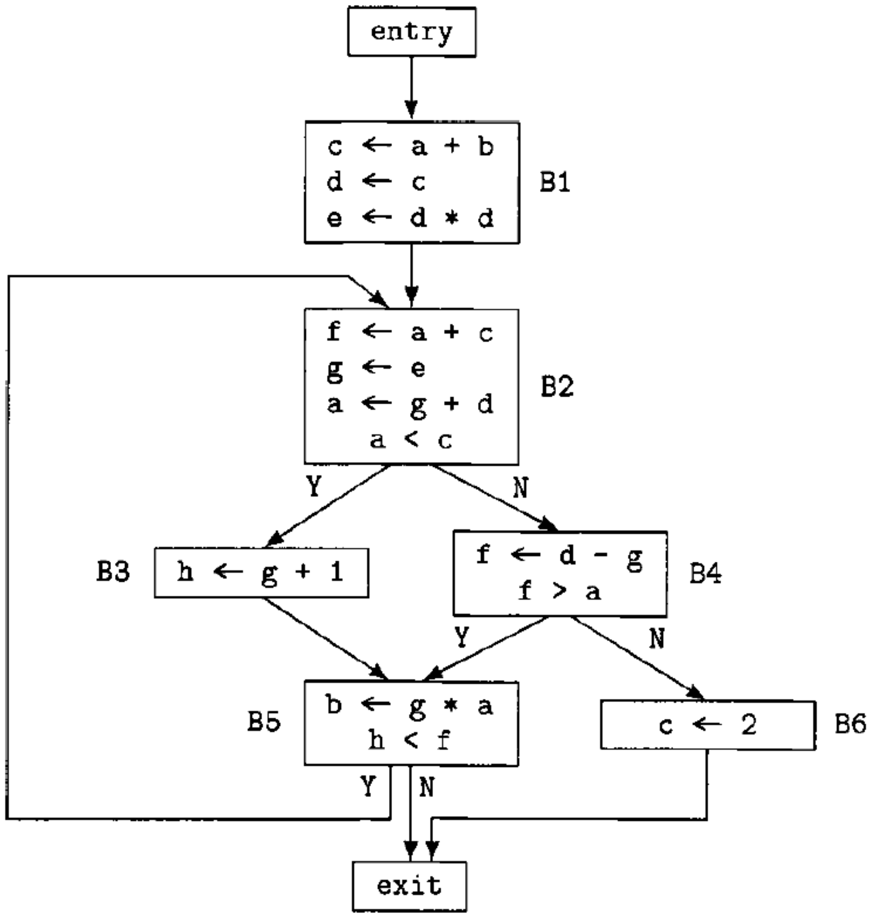
- CPin(i) = ∩ CPout(j) where j ∈ pred(i)
- CPout(i) = COPY(i) ∪ (CPin(i) - KILL(i))
- Equivalent:

$$CPout(i) = COPY(i) \bigcup (CPin(i) \bigcap \overline{KILL(i)})$$

- Substituting CPout into CPin, we obtain:

$$\boxed{CPin(i) = \bigcap_{j \in pred(i)} COPY(j) \bigcup (CPin(j) \bigcap \overline{KILL(j)})}$$

Our work-list order



- Since this is a forward problem, we manage our work-list in a reverse post-order (i.e. preorder means each block before its successors) order.

- One such order is **entry, B1, B2, B4, B6, B3, B5, exit**.

Applying iterative analysis for block entry

- CPin(entry) = <00>

- as per the equation as no predecessor is available.

Applying iterative analysis for block i = B1

$$CPin(i) = \bigcap_{j \in pred(i)} COPY(i) \bigcup (CPin(j) \bigcap \overline{KILL(j)})$$

- entry is predecessor of B1
- CPin(B1) = COPY(entry) ∪ (CPin(entry) - KILL(entry))
- CPin(B1) = <00> ∪ (<00> - <00>)
- **CPin(B1) = <00>**

Applying iterative analysis for block i = B2

$$CPin(i) = \bigcap_{j \in pred(i)} COPY(i) \bigcup (CPin(j) \bigcap \overline{KILL(j)})$$

- B1 and B5 are predecessors of B2

- CPin(B2) = (COPY(B1) ∪ (CPin(B1) - KILL(B1)))
  ∩ (COPY(B5) ∪ (CPin(B5) - KILL(B5)))
- CPin(B2) = (<10> ∪ (<11> - <01>))
  ∩ (<00> ∪ (<11> - <00>))
  = (<10> ∪ <10>) ∩ (<00> ∪ <11>)
  = <10> ∩ <11>
- **CPin(B2) = <10>**

Applying iterative analysis for block i = B4

$$CPin(i) = \bigcap_{j \in pred(i)} COPY(i) \bigcup (CPin(j) \bigcap \overline{KILL(j)})$$

- B2 is predecessor of B4
- CPin(B4) = COPY(B2) ∪ (CPin(B2) - KILL(B2))
- CPin(B4) = <01> ∪ (<11> - <00>)
  = <01> ∪ <11>
- **CPin(B4) = <11>**

Applying iterative analysis for block i = B6

$$CPin(i) = \bigcap_{j \in pred(i)} COPY(i) \cup (CPin(j) \cap \overline{KILL(j)})$$

- B4 is predecessor of B6
- CPin(B6) = COPY(B4) ∪ (CPin(B4) - KILL(B4))
- CPin(B6) = <00> ∪ (<11> - <00>)
  = <00> ∪ <11>
- **CPin(B6) = <11>**

Applying iterative analysis for block i = B3

$$CPin(i) = \bigcap_{j \in pred(i)} COPY(i) \cup (CPin(j) \cap \overline{KILL(j)})$$

- B2 is predecessor of B3
- CPin(B3) = COPY(B2) ∪ (CPin(B2) - KILL(B2))
- CPin(B3) = <01> ∪ (<11> - <00>)
  = <01> ∪ <11>
- **CPin(B3) = <11>**

Applying iterative analysis for block i = B5

$$CPin(i) = \bigcap_{j \in pred(i)} COPY(i) \cup (CPin(j) \cap \overline{KILL(j)})$$

- B3 and B4 are predecessors of B5

- CPin(B5) = (COPY(B3) ∪ (CPin(B3) - KILL(B3)))
  ∩ (COPY(B4) ∪ (CPin(B4) - KILL(B4)))

- CPin(B5) = (<00> ∪ (<11> - <00>)) ∩ (<00> ∪ (<11> - <00>))
  = (<00> ∪ <11>) ∩ (<00> ∪ <11>)
  = <11> ∩ <11>
- **CPin(B5) = <11>**

Applying iterative analysis for block i = exit

$$CPin(i) = \bigcap_{j \in pred(i)} COPY(i) \cup (CPin(j) \cap \overline{KILL(j)})$$

- B5 and B6 are predecessors of exit

- CPin(exit) = (COPY(B5) ∪ (CPin(B5) - KILL(B5)))
  ∩ (COPY(B6) ∪ (CPin(B6) - KILL(B6)))
- CPin(exit) = (<00> ∪ (<11> - <00>))
  ∩ (<00> ∪ (<11> - <10>))
  = (<00> ∪ <11>) ∩ (<00> ∪ <01>)
  = <11> ∩ <01>
- **CPin(exit) = <01>**

Cpin(i)

	Pass 1	Pass 2
CPin(entry)	<00>	**<00>**
CPin(B1)	<11>	**<00>**
CPin(B2)	<11>	**<10>**
CPin(B3)	<11>	**<11>**
CPin(B4)	<11>	**<11>**
CPin(B5)	<11>	**<11>**
CPin(B6)	<11>	**<11>**
CPin(exit)	<11>	**<01>**