# CC Lecture 6

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

# Optimizing Transformations

1. Compile time evaluation
2. **Elimination of common subexpression**
3. Dead code elimination
4. Frequency reduction
5. Strength reduction

# Global Common Subexpression Elimination

- **Global common-subexpression elimination** takes as its scope a flowgraph representing a procedure.

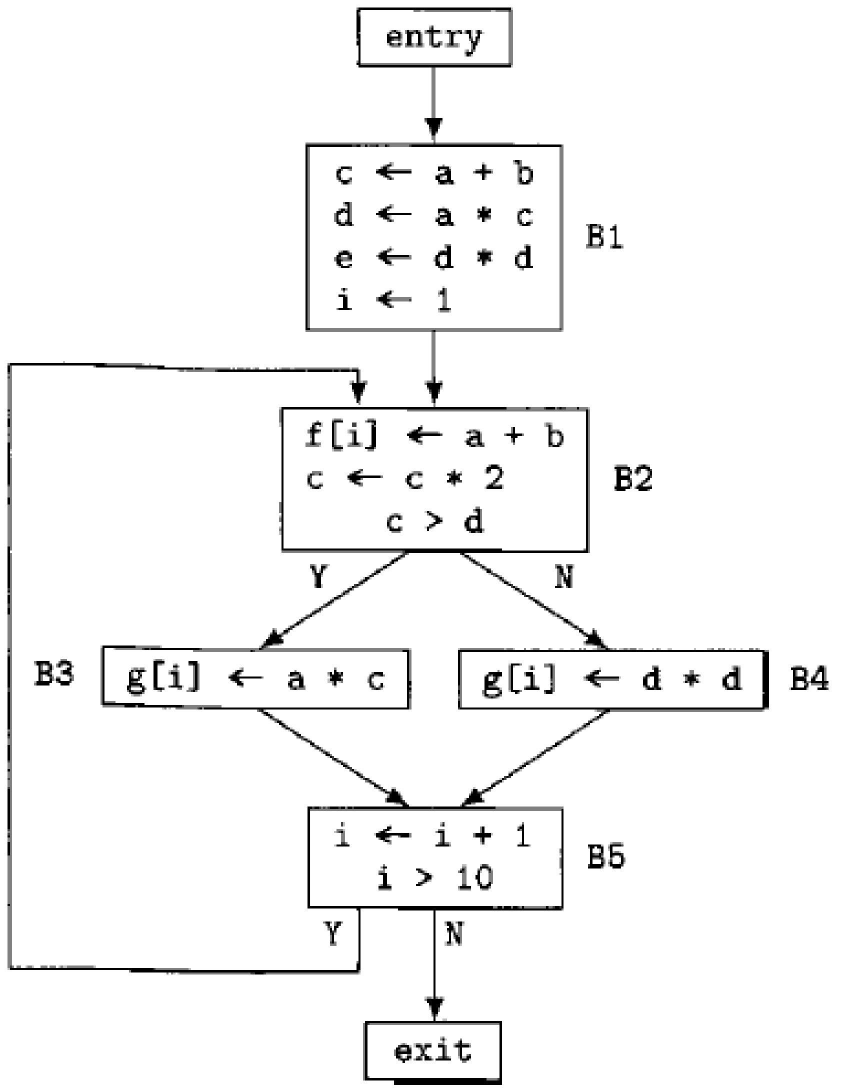
- It solves the data-flow problem known as **available expressions**.

- An **expression exp** is said to be **available** at the entry to a basic block if along every control-flow path from the entry block to this block there is an evaluation of exp that is not subsequently killed by having one or more of its operands assigned a new value.

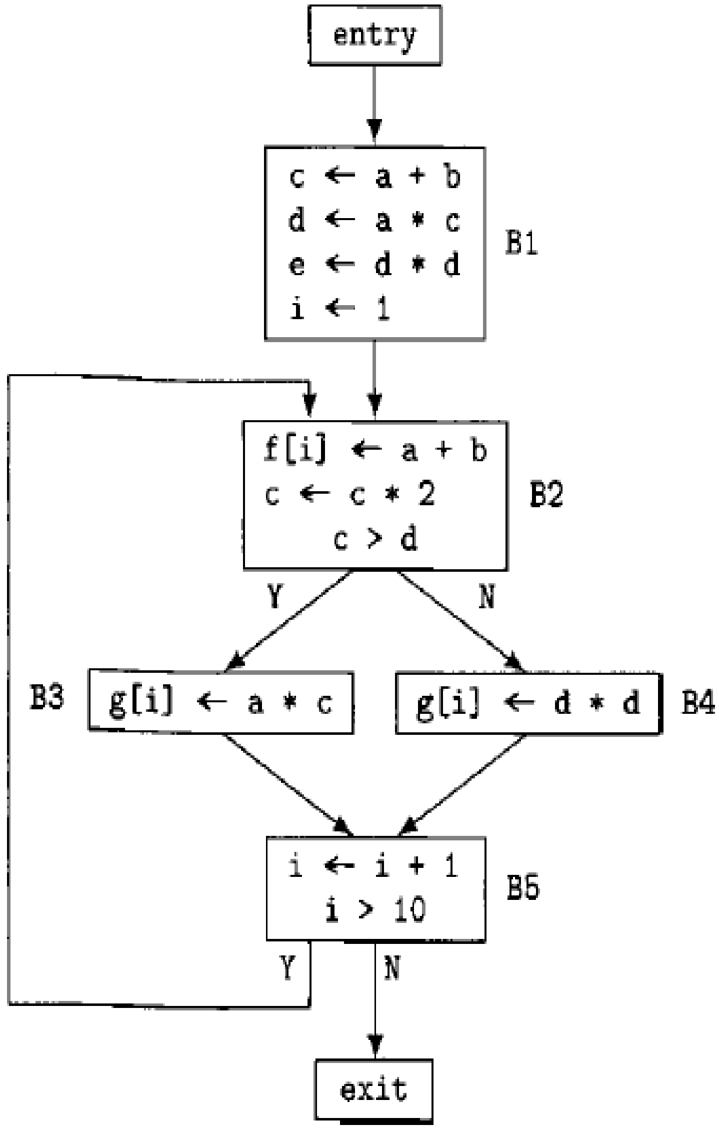
# Global Common Subexpression Elimination

- In determining what **expressions are available**, we use
  - **EVAL(i)** to denote the set of expressions evaluated in block i that are still available at its exit
  - **KILL(i)** to denote the set of expressions that are killed by block i.

# Example: Given a flow graph

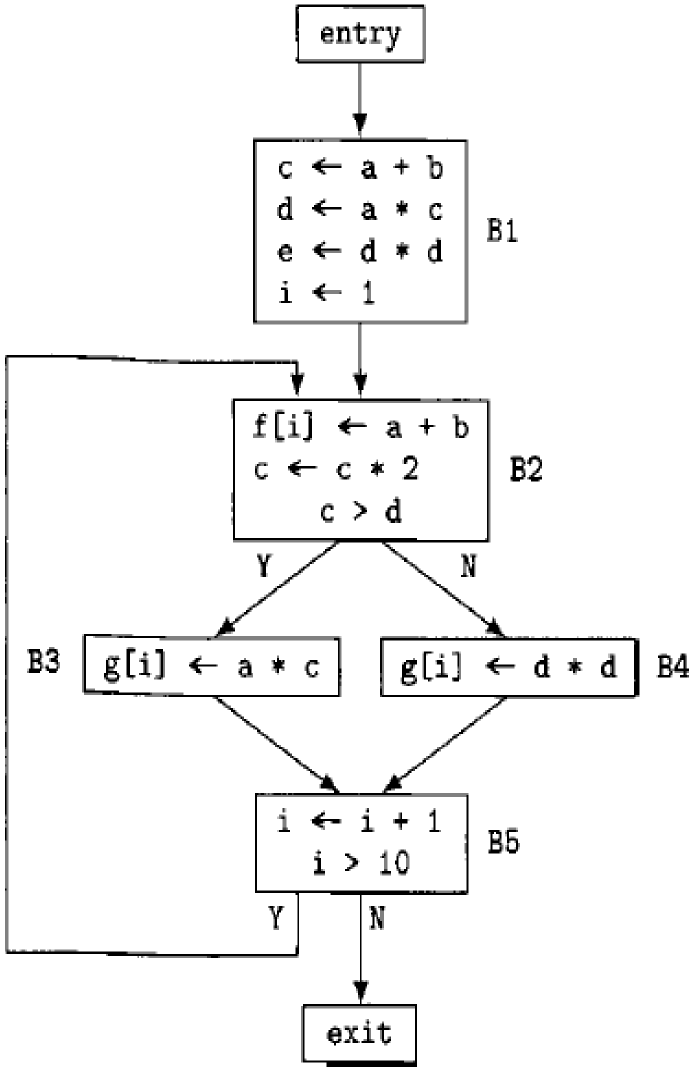


# The EVAL(i) sets for the basic blocks




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

# The KILL(i) sets for the basic blocks




- KILL(entry) = Ø
- KILL(B1) = {c*2, c>d ,a*c, d*d, i+1, i>10}
- KILL(B2) = {a*c, c*2}
- KILL(B3) = Ø
- KILL(B4) = Ø
- KILL(B5) = {i +1}
- KILL(exit) = Ø

The equation system for the data-flow analysis can be constructed as follows:

- This is a forward-flow problem.
- We use **in(i)** and **out(i)** to represent the sets of expressions that are available on entry to and exit from block i, respectively.
- An **expression is available on entry** to block i if it is available at the exits of all predecessor blocks, so the path-combining operator is set intersection.
- An **expression is available at the exit** from a block if it is either evaluated in the block and not subsequently killed in it, or if it is available on entry to the block and not killed in it.

The system of data-flow equations is:

- **out(i) = U - KILL(i)** **for all i ≠ entry**
- **U = ∪ EVAL(i)** //union for all i
- **U = {a+b, a*c, d*d, c>d, i>10}**
- **in(i) = ∩ out(j)** **j∈ Pred(i)**

# An Iterative Algorithm for Computing Available Expressions

for each block $B \neq B1$ do $\{OUT[B] = U - e\_kill[B];\}$
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
$change = true$;
while $change$ do { $change = false$;
  for each block $B \neq B1$ do {

$$
\begin{aligned}
IN[B] &= \bigcap_{P \text{ a predecessor of } B} OUT[P]; \\
oldout &= OUT[B]; \\
OUT[B] &= e\_gen[B] \bigcup (IN[B] - e\_kill[B]);
\end{aligned}
$$

    if $(OUT[B] \neq oldout)$ $change = true$;
  }
}

# For all blocks, calculate out(i)

- out(i) = U - KILL(i)    for all i ≠ entry
- U = ∪ EVAL(i)  //union for all i        U={a+b, a*c, d*d, c>d, i>10}

- KILL(entry) = ∅
- KILL(B1) = {c*2, c>d ,a*c, d*d, i+1, i>10}
- KILL(B2) = {a*c, c*2}
- KILL(B3) = ∅
- KILL(B4) = ∅
- KILL(B5) = {i +1}
- KILL(exit) = ∅

# For all blocks, out(i) is as follows:

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

# Applying the algorithm: for i=entry

- **in(i) = ∩ out(j) j∈ Pred(i)**

- in(entry) = Ø

- Simply, because entry has no predecessors.

# Applying the algorithm: for i = B1

- in(B1) = out(entry) = Ø

- oldout(B1) = out(B1) = {a+b}

- out(B1) = EVAL(B1) ∪ (in(B1) - KILL(B1))
  = {a+b, a*c, d*d} ∪ (Ø - {c*2, c>d ,a*c, d*d, i+1, i>10})
  = {a+b, a*c, d*d} ∪ Ø
  = {a+b, a*c, d*d}

- oldout(B1) ≠ out(B1) [change = true]

# Applying the algorithm: for i = B2

- in(B2) = out(B1) ∩ out(B5)

  = {a+b, a*c, d*d} ∩ {a+b, a*c, d*d, c>d, i>10}

  = {a+b, a*c, d*d}

- oldout(B2) = out(B2) = {a+b, d*d, c>d, i>10}

- out(B2) = EVAL(B2) ∪ (in(B2) - KILL(B2))

  = {a+b, c>d} ∪ ({a+b, a*c, d*d} - {a*c, c*2})

  = {a+b, c>d} ∪ {a+b, d*d}

  = {a+b, c>d, d*d}

- oldout(B2) ≠ out(B2) [change = true]

# Applying the algorithm: for i = B3

- in(B3) = out(B2) = {a+b, c>d, d*d}

- oldout(B3) = out(B3) = {a+b, a*c, d*d, c>d, i>10}

- out(B3) = EVAL(B3) ∪ (in(B3) - KILL(B3))
  = {a*c} ∪ ({a+b, c>d, d*d} - Ø )
  = {a*c, a+b, c>d, d*d}

- oldout(B3) ≠ out(B3)     [change = true]

# Applying the algorithm: for i = B4

- in(B4) = out(B2)= {a+b, c>d, d*d}

- oldout(B4) = out(B4)

  = {a+b, a*c, d*d, c>d, i>10}

- out(B4) = EVAL(B4) ∪ (in(B4) - KILL(B4))

  = {d*d} ∪ ({a+b, c>d, d*d} - Ø )

  = {a+b, c>d, d*d}

- oldout(B4) ≠ out(B4) [change = true]

# Applying the algorithm: for i = B5

- in(B5) = out(B3) ∩ out(B4)

  = {a*c, a+b, c>d, d*d} ∩ {a+b, c>d, d*d}

  = {a+b, c>d, d*d}
- oldout(B5) = out(B5) = {a+b, a*c, d*d, c>d, i>10}

- out(B5) = EVAL(B5) ∪ (in(B5) - KILL(B5))

  = {i>10} ∪ ({a+b, c>d, d*d} - {i+1})

  = {i>10} ∪ {a+b, c>d, d*d}

  = {i>10, a+b, c>d, d*d}
- oldout(B5) ≠ out(B5) [change = true]

# Applying the algorithm: for i = exit

- in(exit) = out(B5) = {i>10, a+b, c>d, d*d}
- oldout(exit) = out(exit) = {a+b, a*c, d*d, c>d, i>10}
- out(exit) = EVAL(exit) ∪ (in(exit) - KILL(exit))

  = Ø ∪ ({i>10, a+b, c>d, d*d} - Ø )

  = {i>10, a+b, c>d, d*d}
- out(exit) is not required as there is no block after it.
- oldout(exit) ≠ out(exit) [change = true]

Second iteration will start with following values of out(i):

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

# $2^{nd}$ iteration of algorithm: for i = entry

- in(entry) = Ø

- Simply, because entry has no predecessors.

# $2^{nd}$ iteration of algorithm: for i = B1

- in(B1) = out(entry) = Ø

- oldout(B1) = out(B1) = {a+b, a*c, d*d}

- out(B1) = EVAL(B1) ∪ (in(B1) - KILL(B1))

  ={a+b, a*c, d*d} (Ø - {c*2, c>d ,a*c, d*d, i+, i>10})

  = {a+b, a*c, d*d} ∪ Ø

  = {a+b, a*c, d*d}

- oldout(B1) = out(B1) So, no change.

# $2^{nd}$ iteration of algorithm: for i = B2

- in(B2) = out(B1) ∩ out(B5)
  = {a+b, a*c, d*d} ∩ {i>10, a+b, c>d, d*d}
  = {a+b, a*c, d*d}

- oldout(B2) = out(B2) = {a+b, c>d, d*d}

- out(B2) = EVAL(B2) ∪ (in(B2) - KILL(B2))
  = {a+b, c>d} ∪ ({a+b, a*c, d*d} - {a*c, c*2})
  = {a+b, c>d} ∪ {a+b, d*d}
  = {a+b, c>d, d*d}
- oldout(B2) = out(B2) So, no change.

# $2^{nd}$ iteration of algorithm: for i = B3

- in(B3) = out(B2) = {a+b, c>d, d*d}

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

- out(B3) = EVAL(B3) ∪ (in(B3) - KILL(B3))
  = {a*c} ∪ ({a+b, c>d, d*d} - ∅ )
  = {a*c, a+b, c>d, d*d}

- oldout(B3) = out(B3)   So, no change

# $2^{nd}$ iteration of algorithm: for i = B4

- in(B4) = out(B2) = {a+b, c>d, d*d}

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

- out(B4) = EVAL(B4) ∪ (in(B4) - KILL(B4))
  = {d*d} ∪ ({a+b, c>d, d*d} - Ø )
  = {a+b, c>d, d*d}

- oldout(B4) = out(B4)    So, no change.

# $2^{nd}$ iteration of algorithm: for i = B5

- in(B5) = out(B3) ∩ out(B4)

  = {a*c, a+b, c>d, d*d} ∩ {a+b, c>d, d*d}

  = {a+b, c>d, d*d}

- oldout(B5) = out(B5) = {i>10, a+b, c>d, d*d}

- out(B5) = EVAL(B5) ∪ (in(B5) - KILL(B5))

  = {i>10} ∪ ({a+b, c>d, d*d} - {i+1})

  = {i>10} ∪ {a+b, c>d, d*d}

  = {i>10, a+b, c>d, d*d}
- oldout(B5) = out(B5) So, no change

# $2^{nd}$ iteration of algorithm: for i = exit

- in(exit) = out(B5) = {i>10, a+b, c>d, d*d}

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

- out(exit) = EVAL(exit) ∪ (in(exit) - KILL(exit))
  = Ø ∪ ({i>10, a+b, c>d, d*d} - Ø )
  = {i>10, a+b, c>d, d*d}
- out(exit) is not required as there is no block after it.

- oldout(exit) = out(exit)     So, no change

## As we have no change for all blocks, no further iterations are to be done.

Thus 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 ← exp, that caused it to be in AEin(i).

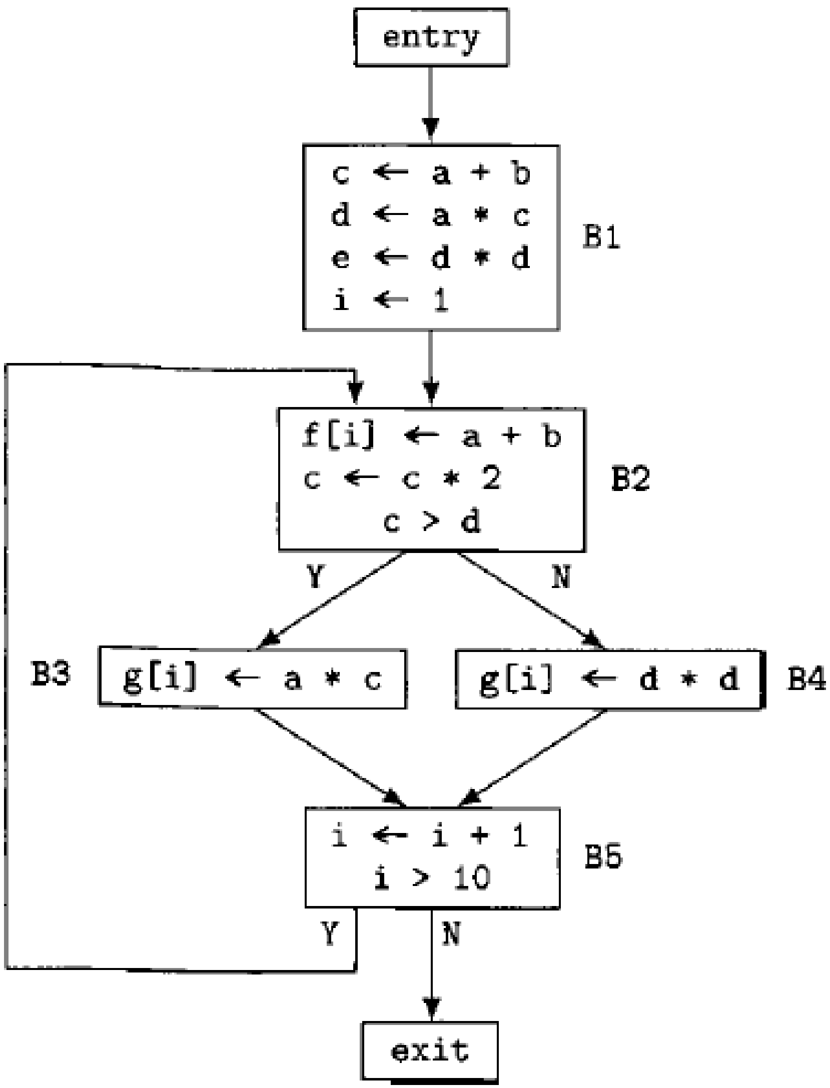
   These are the final occurrences of exp in their respective blocks; each of them must flow unimpaired to the entry of block i; and every flow path from the entry block to block i must include at least one of them.

# 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

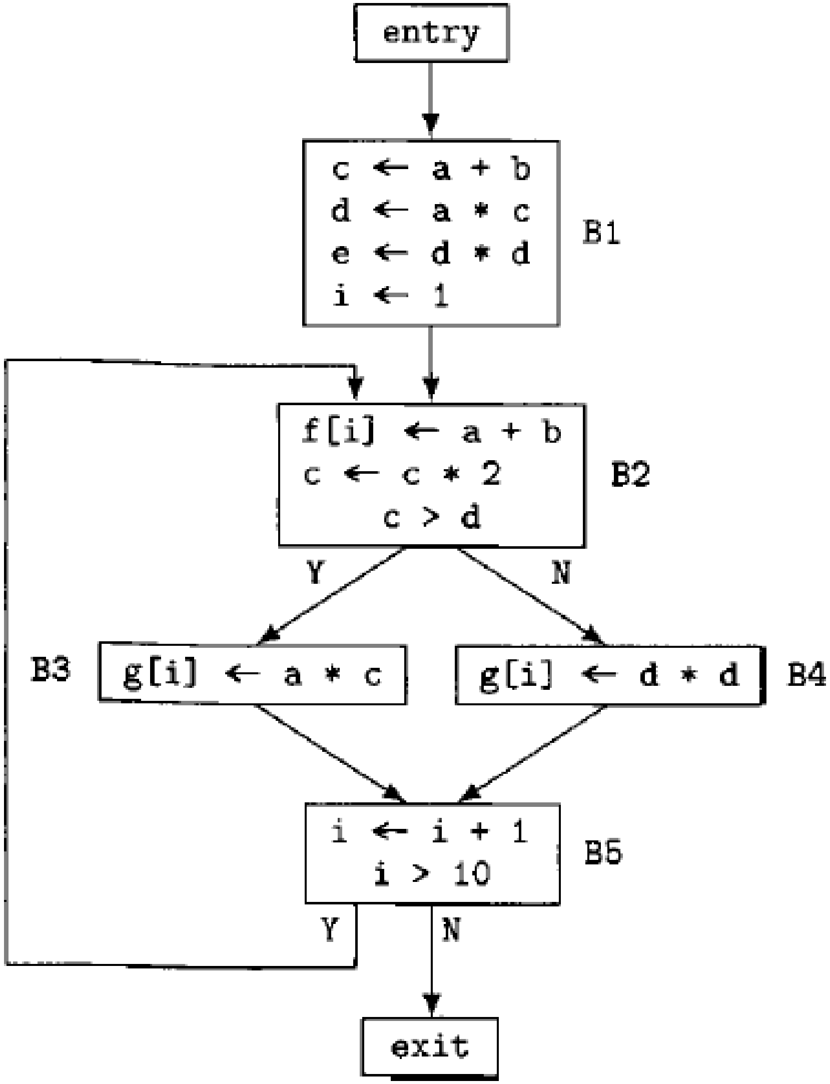
# Applying the procedure to given flow graph




- in(entry) = ∅
- in(B1) = ∅

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

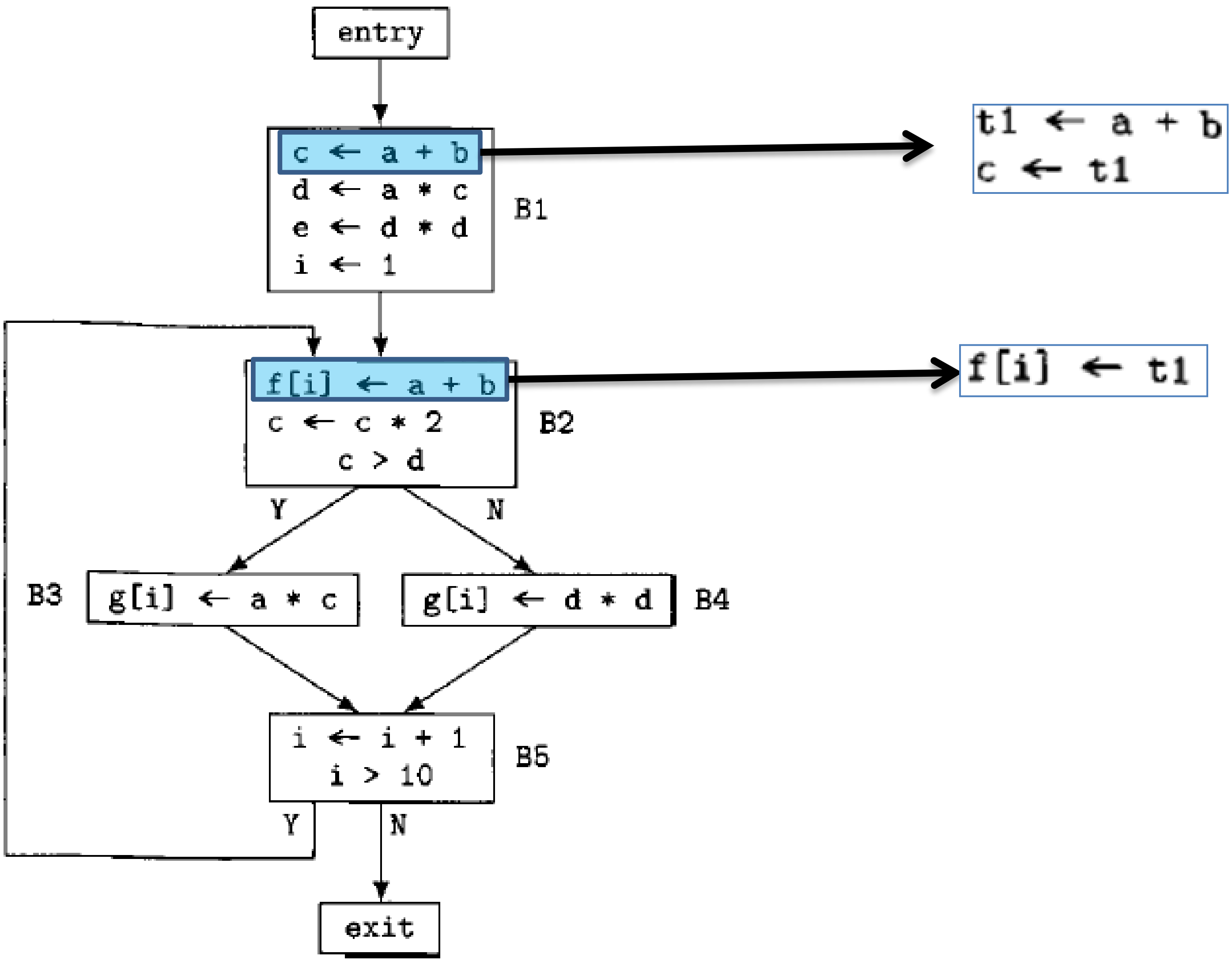
# Applying the procedure to given flow graph



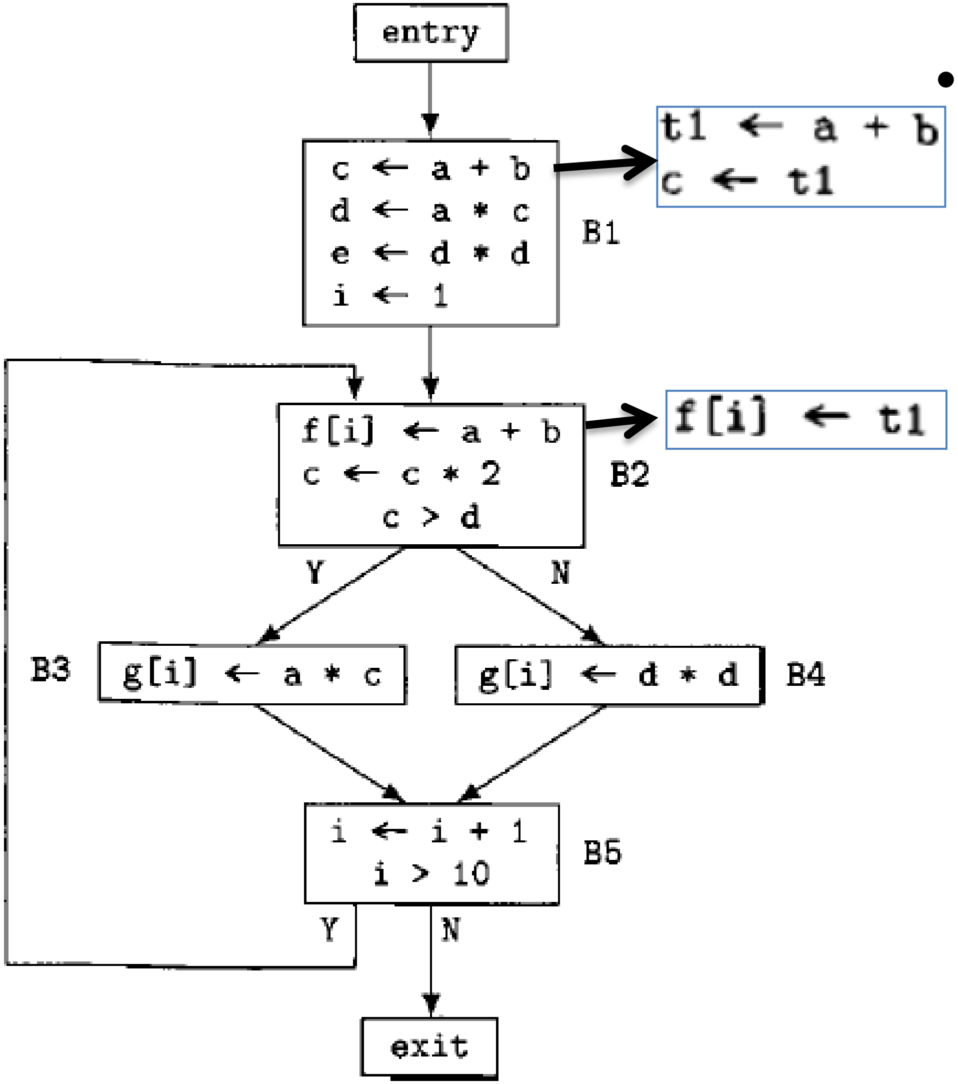

- in(B2) = {a+b, a*c, d*d}

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

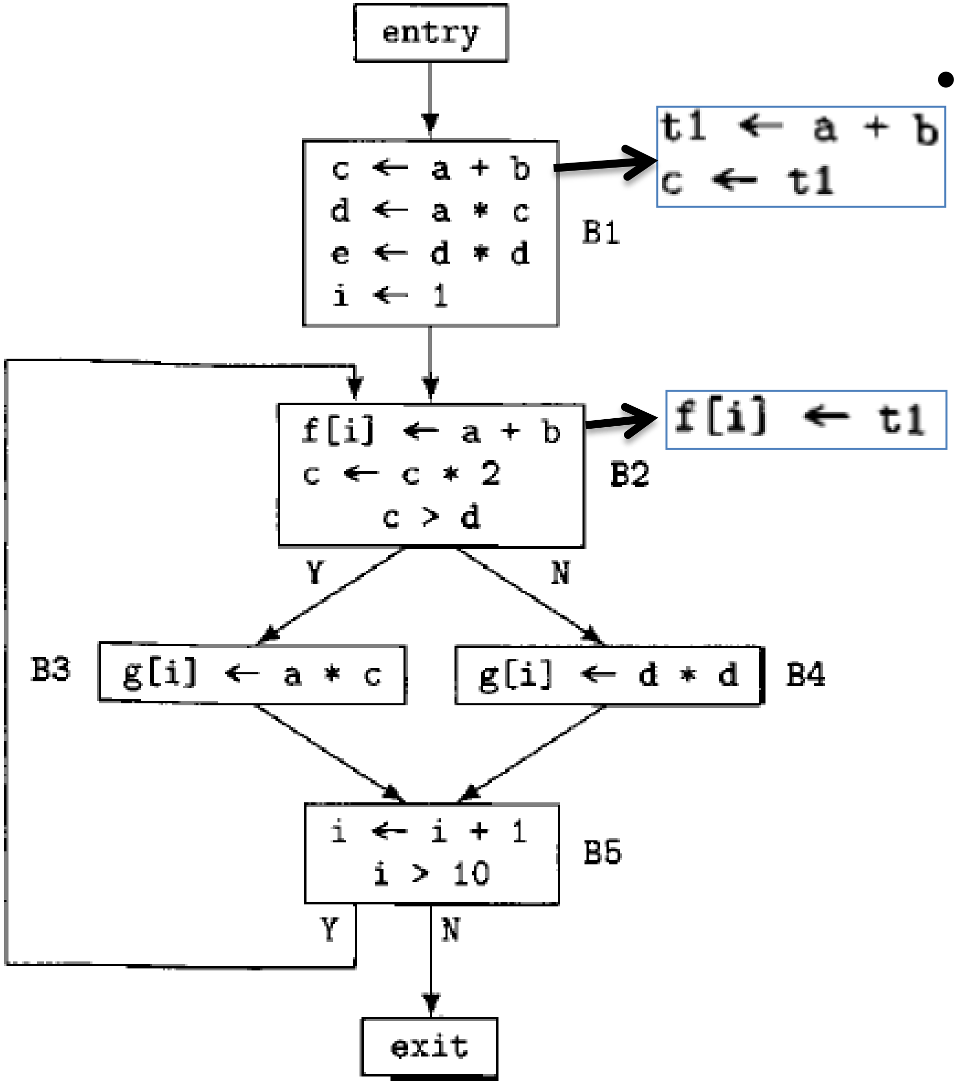



- in(B2) = {a+b, a*c, d*d}

  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, a*c, d*d}

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

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

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