CC Week 3

Prepared for: 7th Sem, CE, DDU
Prepared by: Niyati J. Buch

Ref. Book 1: Compiler: Principles, techniques and tools by Aho, Ullman and Sethi, 2nd Ed., Pearson Education
Ref. Book 2: Advanced Compiler Design & Implementation By Steven S Muchnick

Contents

Common Subexpression Elimination

- **Common subexpression elimination** finds computations that are always performed at least twice on a given execution path and eliminates the second and later occurrences of them.

- An occurrence of an expression in a program is a common subexpression:-
  - if there is another occurrence of the expression whose evaluation always precedes this one in execution order
  - and if the operands of the expression remain unchanged between the two evaluations.

Common Subexpression Elimination

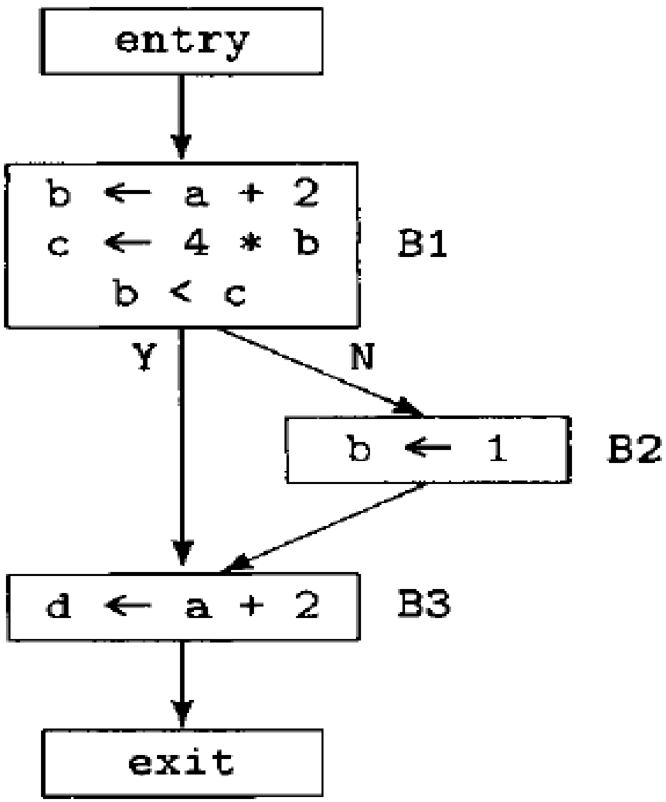
- **Common- subexpression elimination** is a transformation that removes the re-computations of common subexpressions and replaces them with the uses of saved values.

- Also, note that common-subexpression elimination may not always be worthwhile.

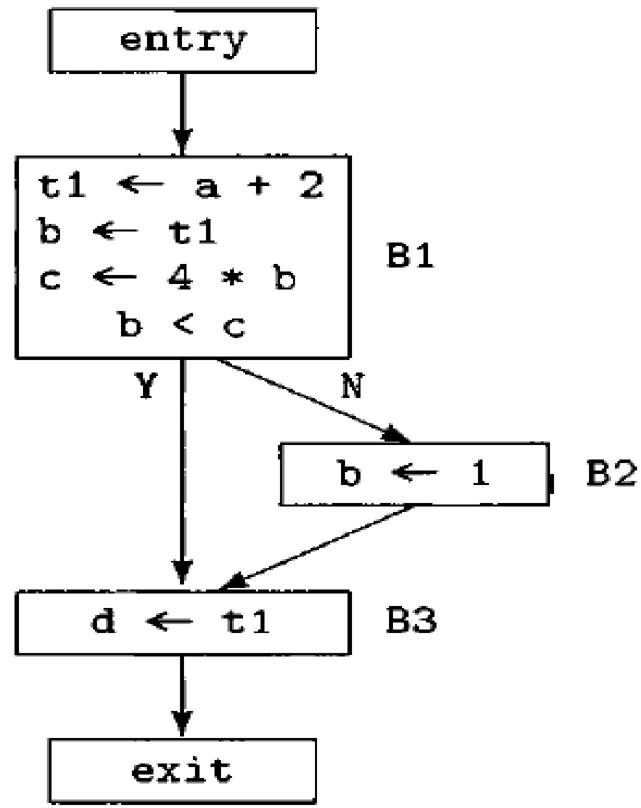
- Optimizers frequently divide common-subexpression elimination into two phases,
  1. **local**: done within each basic block,
  2. **global:** done across an entire flow graph.

Common-Subexpression Elimination

Example of a common subexpression, namely, **a + 2**,



The result of doing common-subexpression elimination on it.



To do local common-subexpression elimination

- we iterate through the basic block
- adding entries to and removing them from **AEB** (Available Expression Block) as appropriate
- inserting instructions to save the expressions' values in temporaries
- modifying the instructions to use the temporaries instead.

Available Expressions

- **Available expressions** is an analysis algorithm that determines for each point in the program, the set of expressions that need not be recomputed.

- Those expressions are said to be ***available*** at such a point.

- To be available on a program point, the operands of the expression should not be modified on any path from the occurrence of that expression to the program point.

Algorithm using AEB

- For each instruction inst at position i, we determine whether it computes a binary expression or not and then execute one of two cases accordingly.

- The (nontrivial) binary case is as follows:

1. We compare inst's operands and operator with those in the quintuples in AEB.

   If we find a match, say, **(pos, opd1, opr, opd2, tmp),** we check whether **tmp is nil**.

Algorithm using AEB

If it is, we

(a) generate a new temporary variable name **ti** and replace the nil in the identified triple by it,

(b) insert the instruction **ti ← opd1 opr opd2** immediately before the instruction at position pos, and

(c) replace the expressions in the instructions at positions **pos** and **i by ti**.

Algorithm using AEB

If we found a match with **tmp = ti**, where ti ≠ nil, we replace the expression in **inst by ti**.

If we did not find a match for inst's expression in AEB, we insert a quintuple for it, with tmp = nil, into AEB.

2. We check whether the **result variable** of the current instruction, if there is one, occurs as an operand in any element of AEB.

   If it does, we **remove all such quintuples** from AEB.

Example: basic block before local common-subexpression elimination.

Position	Instruction
1	c ← a + b
2	d ← m & n
3	e ← b + d
4	f ← a + b
5	g ← - b
6	h ← b + a
7	a ← j + a
8	k ← m & n
9	j ← b + d
10	a ← - b
11	If m & n goto L2

Position	Instruction
1	c ← a + b
2	d ← m & n
3	e ← b + d
4	f ← a + b
5	g ← - b
6	h ← b + a
7	a ← j + a
8	k ← m & n
9	j ← b + d
10	a ← - b
11	If m & n goto L2

Entry: AEB = ∅

Position 1 :
AEB = {**<1, a, +, b, nil>**}

Position 2 :
AEB = { <1, a, +, b, nil>,
<2, m, &, n, nil>}

Position 3 :
AEB = { <1, a, +, b, nil>,
<2, m, &, n, nil>,
<3, b, +, d, nil>}

Position	Instruction
1	c ← a + b
2	d ← m & n
3	e ← b + d
4	f ← a + b
5	g ← - b
6	h ← b + a
7	a ← j + a
8	k ← m & n
9	j ← b + d
10	a ← - b
11	If m & n goto L2

Position 4 :

f ← a + b matches with the first quintuple in AEB.

AEB = { <1, a, +, b, nil>,
<2, m, &, n, nil>,
<3, b, +, d, nil>}

So, insert **t1** into that quintuple in place of nil, generate the instruction

t1 ← a + b before position 1 and renumber the entries in AEB,

replace the instruction that was in position 1 but that is now in position 2 by **c ← t1**, set i = 5, and replace the instruction in position 5 by

f ← t1.

Position	Instruction
1	**t1 ← a + b**
2	c ← **t1**
3	d ← m & n
4	e ← b + d
5	f ← **t1**
6	g ← - b
7	h ← b + a
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 4 :

f ← a + b matches with the first quintuple in AEB.

AEB = { <1, a, +, b, nil>,
<2, m, &, n, nil>,
<3, b, +, d, nil>}

So, insert **t1** into that quintuple in place of nil, generate the instruction

t1 ← a + b before position 1 and renumber the entries in AEB,

replace the instruction that was in position 1 but that is now in position 2 by **c ← t1**, set i = 5, and replace the instruction in position 5 by

f ← t1.

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← b + a
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 5 :

AEB = { <1, a, +, b, **t1**>,
<3, m, &, n, nil>,
<4, b, +, d, nil>}

Position 6:

AEB = { <1, a, +, b, t1>,
<3, m, &, n, nil>,
<4, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← b + a
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 7:
h ← b + a, matches (commutative property)

AEB = { <1, a, +, b, t1>,
<3, m, &, n, nil>,
<4, b, +, d, nil>}

[no change]

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← **t1**
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 7:
h ← b + a, matches
(commutative property)

AEB = { <1, a, +, b, t1>,
<3, m, &, n, nil>,
<4, b, +, d, nil>}
[no change]

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← t1
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 8:
a ← j + a
here variable matches in operand of quintuple so remove **<1, a, +, b, t1>** from AEB

AEB = {<3, m, &, n, nil>,
<4, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← t1
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 9:
m & n is recognized as common sub expression
So,

AEB = {<3, m, &, n, **t2**>,
<**5**, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	d ← m & n
4	e ← b + d
5	f ← t1
6	g ← - b
7	h ← t1
8	a ← j + a
9	k ← m & n
10	j ← b + d
11	a ← - b
12	If m & n goto L2

Position 9:
m & n is recognized as common sub expression
So,

AEB = {<3, m, &, n, **t2**>,
<**5**, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	**t2 ← m & n**
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← - b
8	h ← t1
9	a ← j + a
10	k ← **t2**
11	j ← b + d
12	a ← - b
13	If m & n goto L2

Position 9:
m & n is recognized as common sub expression
So,

AEB = {<3, m, &, n, **t2**>,
<**5**, b, +, d, nil>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← - b
8	h ← t1
9	a ← j + a
10	k ← t2
11	j ← b + d
12	a ← - b
13	If m & n goto L2

New Position 11:
b + d is recognized as common sub expression

AEB = {<3, m, &, n, t2>,
<5, b, +, d, **t3**>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← - b
8	h ← t1
9	a ← j + a
10	k ← t2
11	j ← b + d
12	a ← - b
13	If m & n goto L2

New Position 11:
b + d is recognized as common sub expression

AEB = {<3, m, &, n, t2>,
<5, b, +, d, **t3**>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	e ← b + d
6	f ← t1
7	g ← - b
8	h ← t1
9	a ← j + a
10	k ← t2
11	j ← b + d
12	a ← - b
13	If m & n goto L2

New Position 11:
b + d is recognized as common sub expression

AEB = {<3, m, &, n, t2>,
<5, b, +, d, **t3**>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	t3 ← b + d
6	e ← t3
7	f ← t1
8	g ← - b
9	h ← t1
10	a ← j + a
11	k ← t2
12	j ← t3
13	a ← - b
14	If m & n goto L2

New Position 11:
b + d is recognized as common sub expression

AEB = {<3, m, &, n, t2>,
<5, b, +, d, **t3**>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	t3 ← b + d
6	e ← t3
7	f ← t1
8	g ← - b
9	h ← t1
10	a ← j + a
11	k ← t2
12	j ← t3
13	a ← - b
14	If m & n goto L2

Position 13:
AEB = {<3, m, &, n, t2>,
<5, b, +, d, t3>}
[no change]

Position 14 :
m & n is a common sub expression found
AEB = {<3, **m, &, n**, t2>,
<5, b, +, d, t3>}

Position	Instruction
1	t1 ← a + b
2	c ← t1
3	t2 ← m & n
4	d ← t2
5	t3 ← b + d
6	e ← t3
7	f ← t1
8	g ← - b
9	h ← t1
10	a ← j + a
11	k ← t2
12	j ← t3
13	a ← - b
14	If **t2** goto L2

Position 13:

AEB = {<3, m, &, n, t2>,
<5, b, +, d, t3>}

[no change]

Position 14 :

m & n is a common sub expression found

AEB = {<3, **m, &, n**, t2>,
<5, b, +, d, t3>}

Conclusion

- In the original form of this code there are 11 instructions, 12 variables, and 9 binary operations performed,
- while in the final form there are 14 instructions, 15 variables, and 4 binary operations performed.

Conclusion

- Assuming all the variables occupy registers and that each of the register-to-register operations requires only a single cycle, as in any RISC and the more advanced CICSs, the original form is to be preferred, since it has fewer instructions and uses fewer registers.
- On the other hand, if some of the variables occupy memory locations or the redundant operations require more than one cycle, the result of the optimization is to be preferred.
- Thus, **whether an optimization actually improves the performance of a block of code depends on both the code and the machine it is executed on.**

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.

EVAL(i)

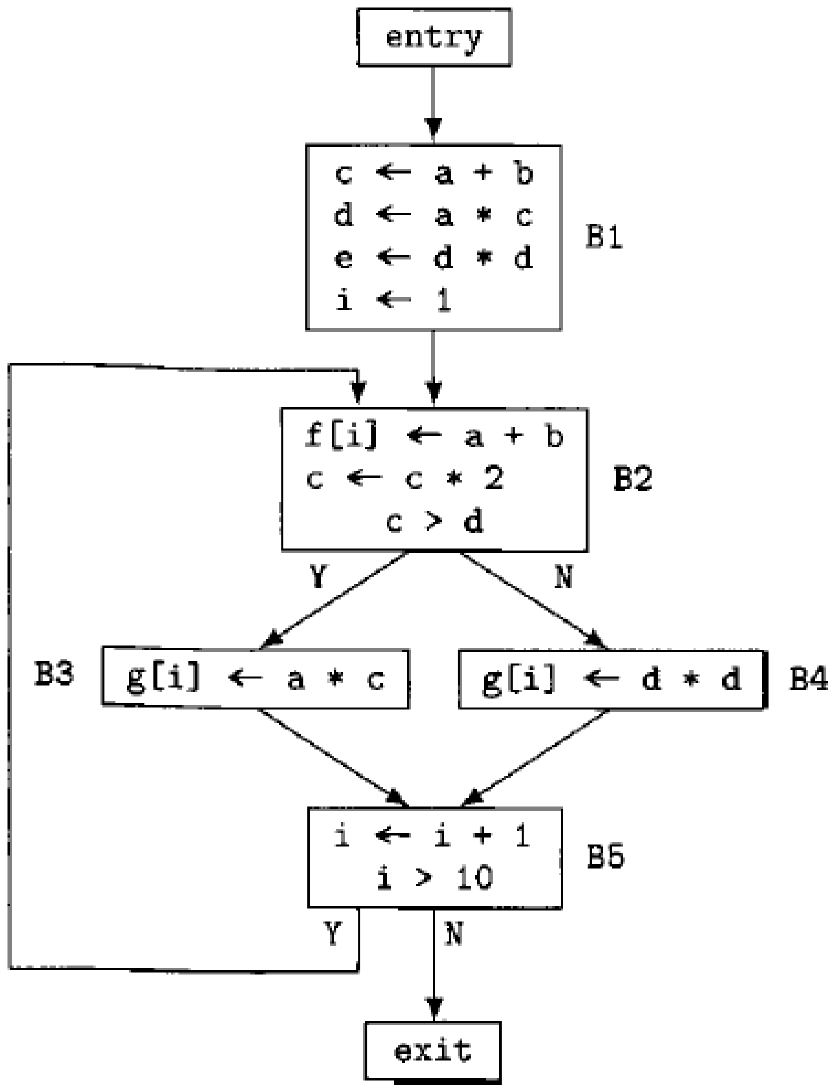
- To compute **EVAL(i)**,
  - we scan block i from beginning to end,
  - accumulating the expressions evaluated in it and
  - deleting those whose operands are later assigned new values in the block.

Note: An assignment such as a $\leftarrow$ a + b, in which the variable on the left-hand side occurs also as an operand on the right-hand side, does not create an available expression because the assignment happens after the expression evaluation.

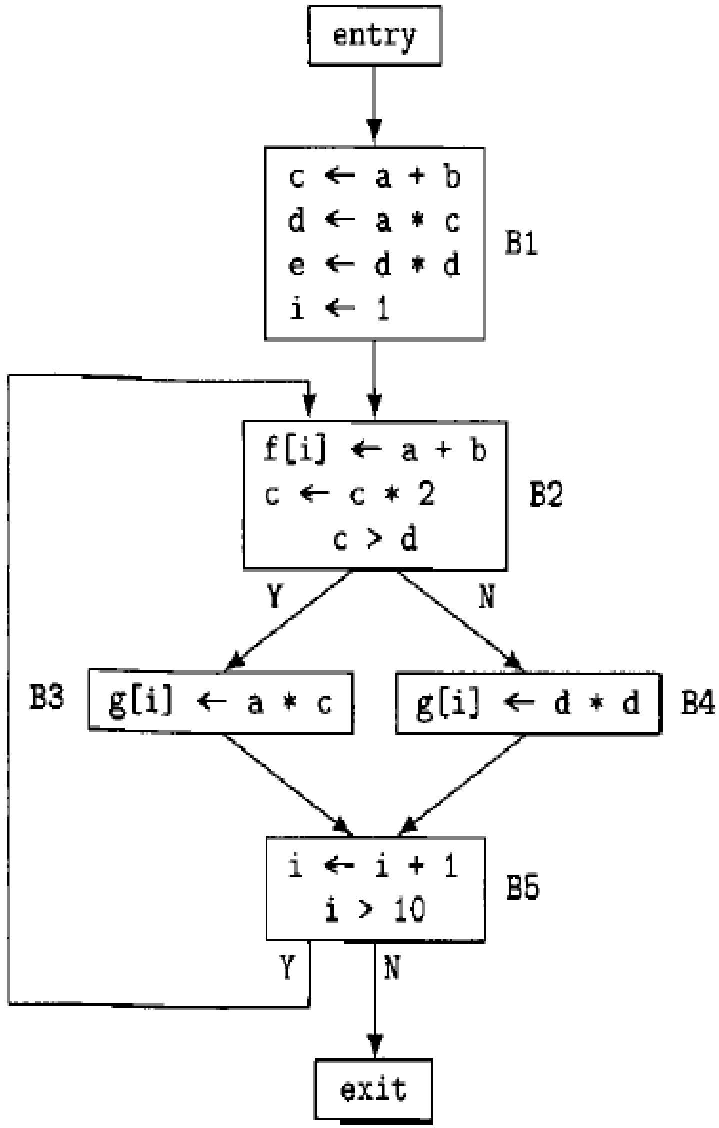
KILL(i)

- **KILL(i)** is the set of all expressions
  - evaluated in **other blocks** such that one or more of their operands are **assigned to in block** i,
  - or that are evaluated in **block** i and subsequently have an operand **assigned to in block** i.

Example: Given a flow graph



Find EVAL(i) sets for the basic blocks



- EVAL(entry) = ?
- EVAL(B1) = ?
- EVAL(B2) = ?
- EVAL(B3) = ?
- EVAL(B4) = ?
- EVAL(B5) = ?
- EVAL(exit) = ?

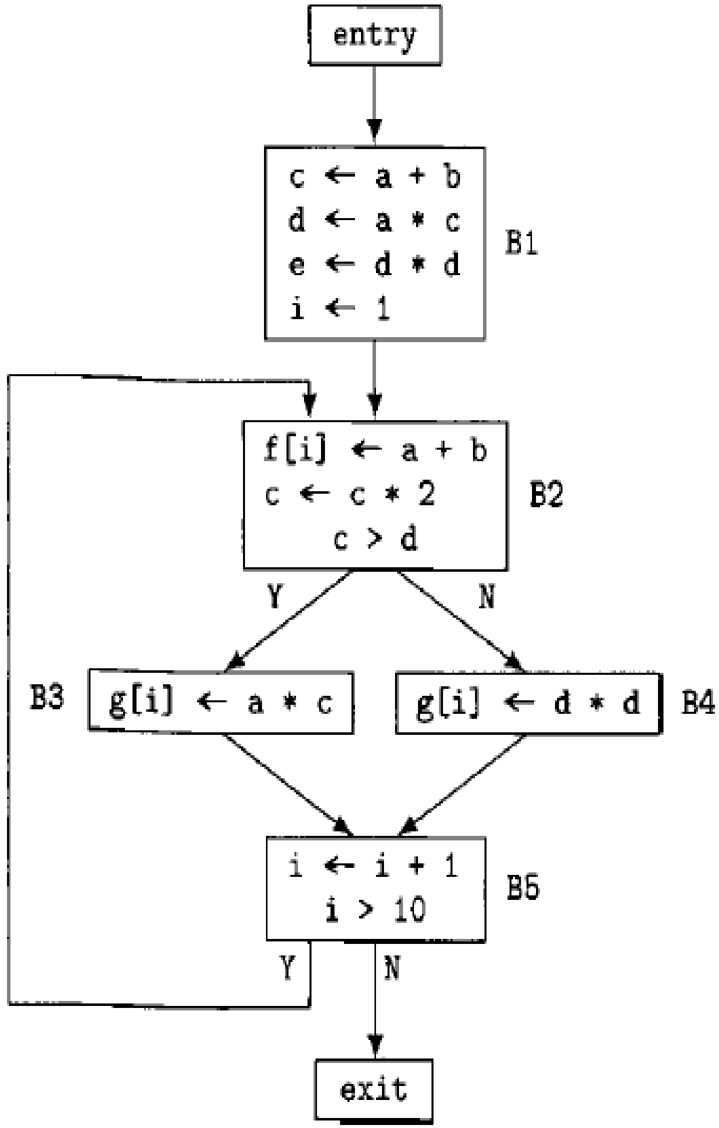
To compute **EVAL(i)**,

— we scan block i from beginning to end,

— accumulating the expressions evaluated in it and

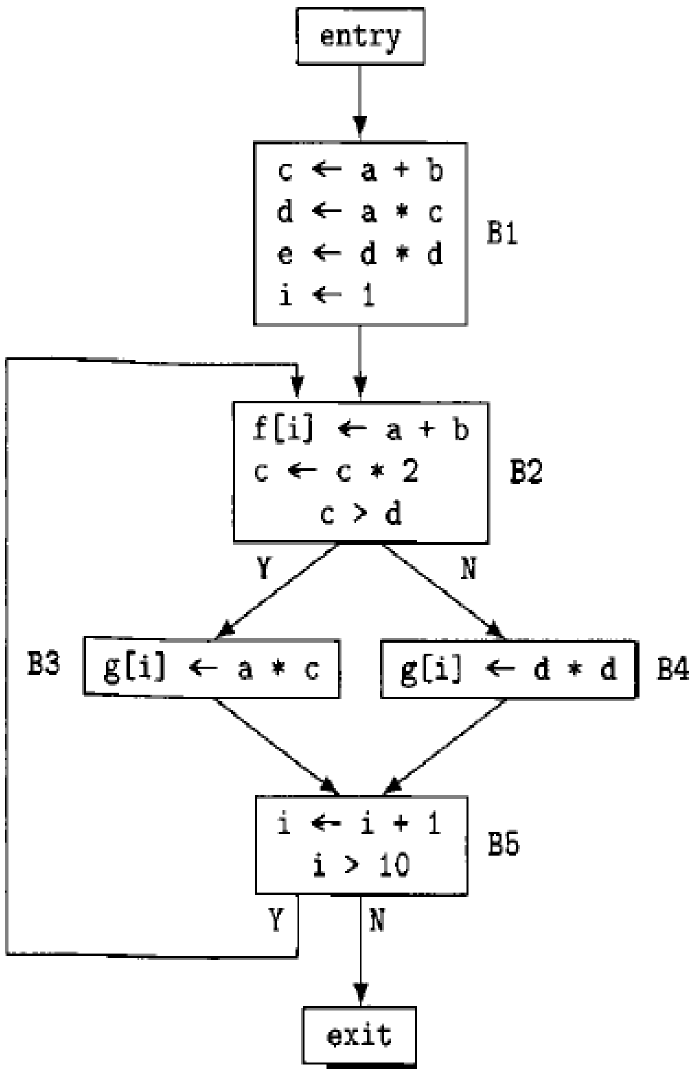
— deleting those whose operands are later assigned new values in the block.

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

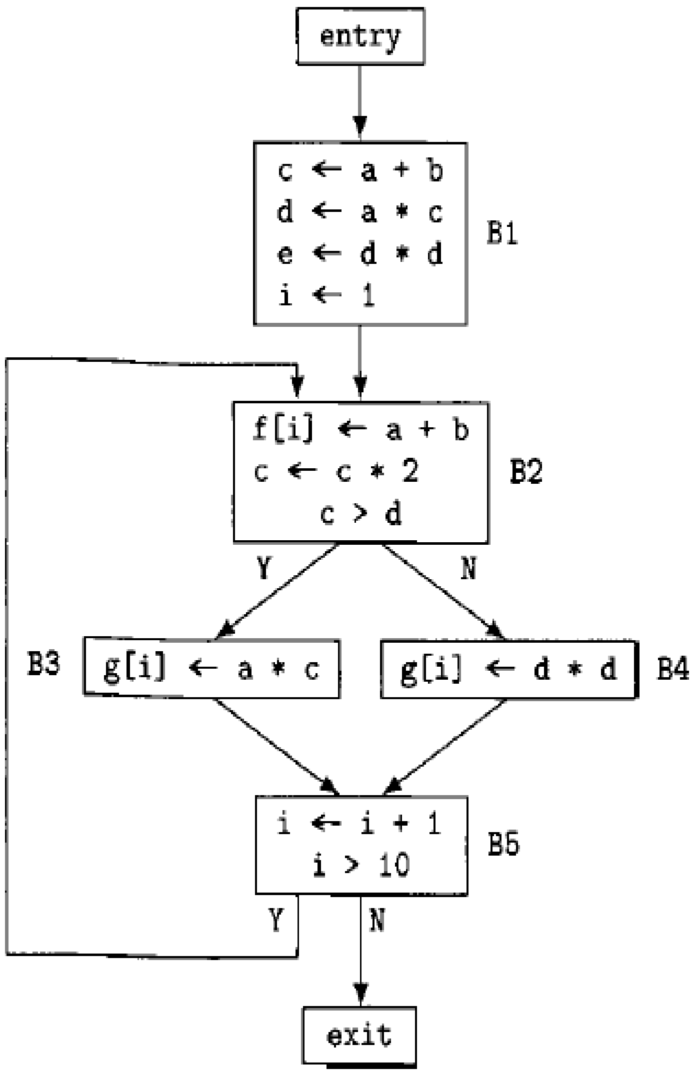
Find the KILL(i) sets for the basic blocks



- KILL(entry) = ?
- KILL(B1) = ?
- KILL(B2) = ?
- KILL(B3) = ?
- KILL(B4) = ?
- KILL(B5) = ?
- KILL(exit) = ?

- **KILL(i)** is the set of all expressions
  - evaluated in **other blocks** such that one or more of their operands are **assigned to in block i**,
  - or that are evaluated in **block i** and subsequently have an operand **assigned to in block i**.

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 $i \neq entry$**

- **$U = \cup\ EVAL(i)$** //union for all i
- **U = {a+b, a*c, d*d, c>d, i>10}**

- **$in(i) = \cap\ out(j)$ $j \in Pred(i)$**

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) = 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

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

- $in(B2) = out(B1) \cap out(B5)$

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

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

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

- $out(B2) = EVAL(B2) \cup (in(B2) - KILL(B2))$

  $= \{a+b, c>d\} \cup (\{a+b, a*c, d*d\} - \{a*c, c*2\})$

  $= \{a+b, c>d\} \cup \{a+b, d*d\}$

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

- $oldout(B2) \neq out(B2)$ [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.

2nd 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} - Ø)
  = {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).

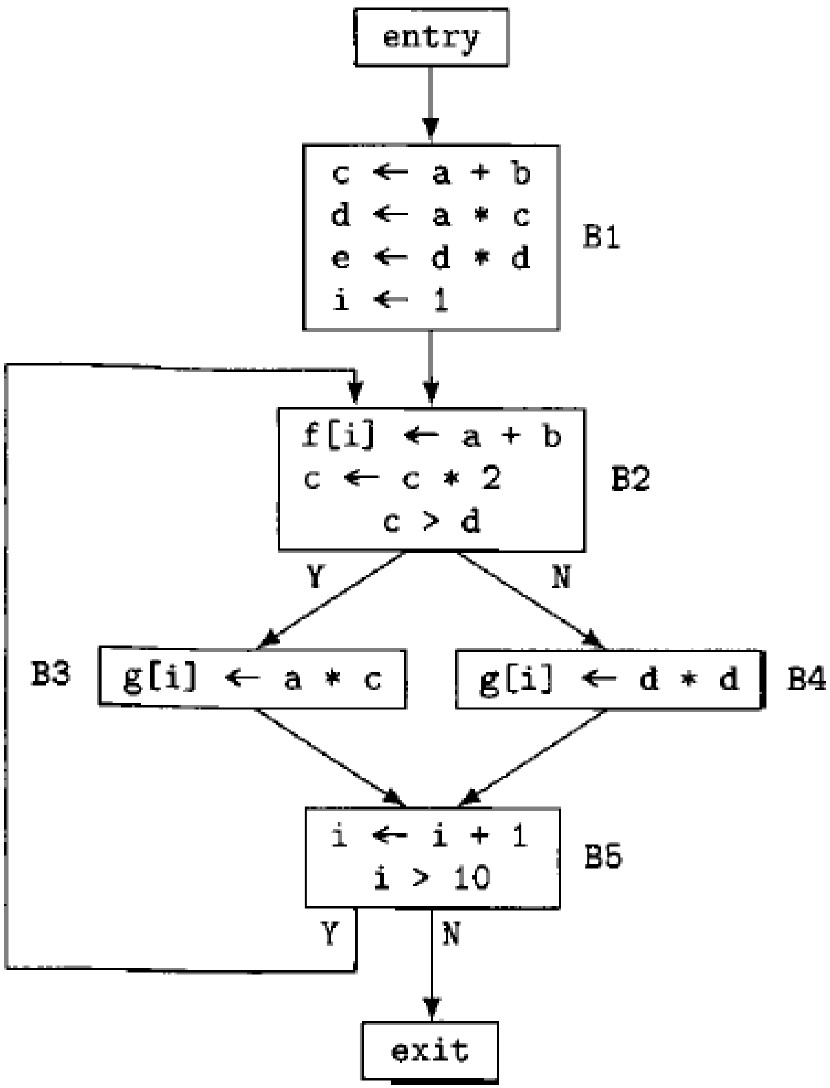
   These are the final occurrences of exp in their respective blocks; each of them must flow unimpaired to the entry of block i; and every flow path from the entry block to block i must include at least one of them.

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

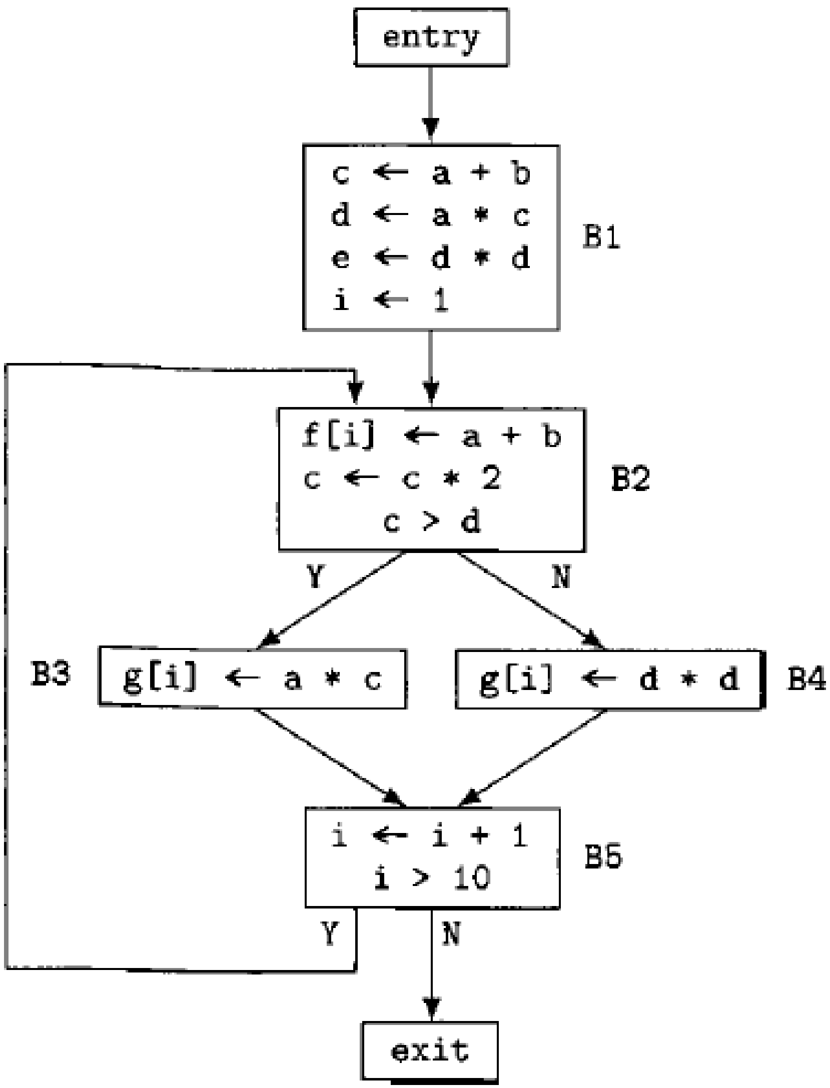
Applying the procedure to given flow graph



- in(entry) = Ø
- in(B1) = Ø

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

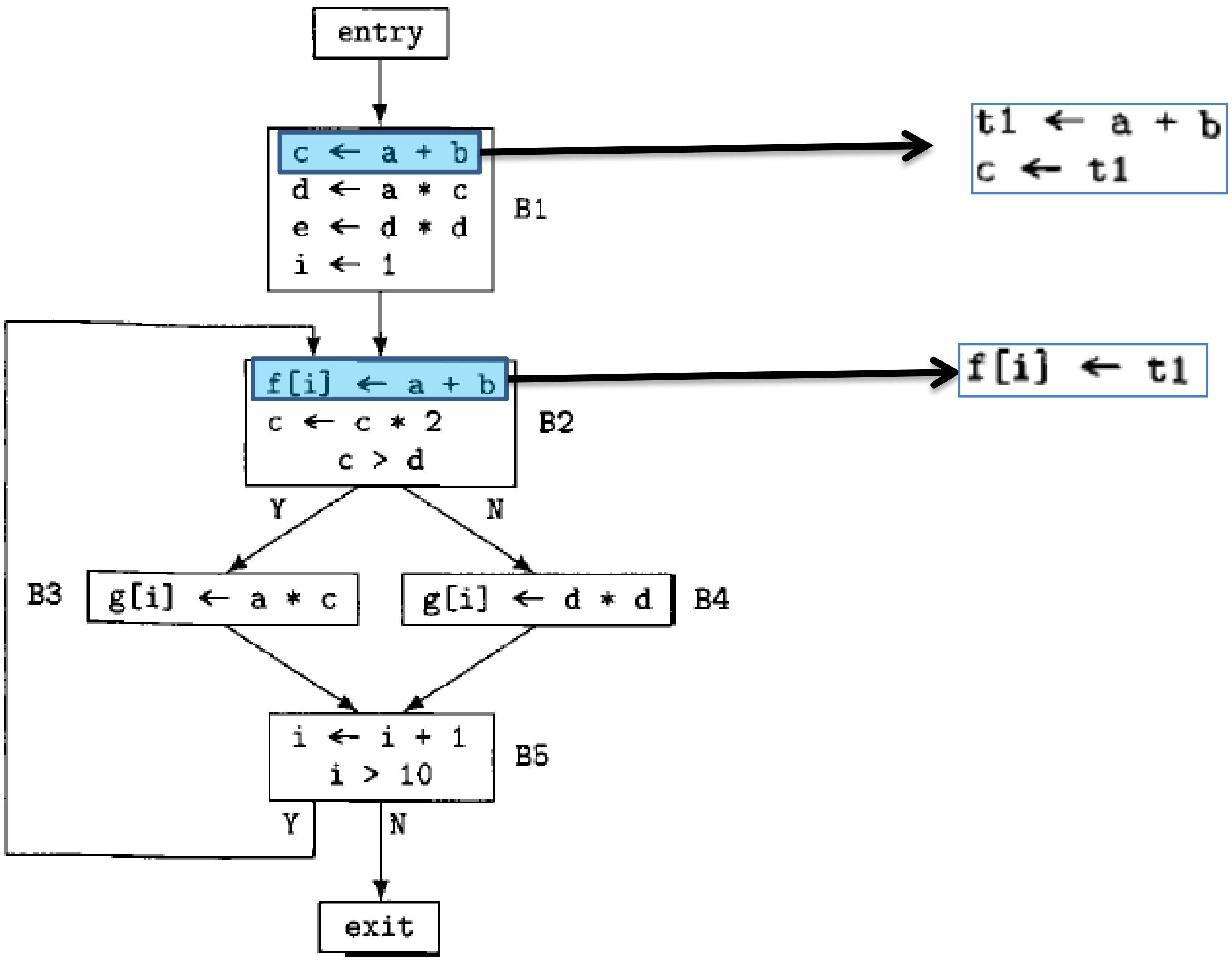
Applying the procedure to given flow graph



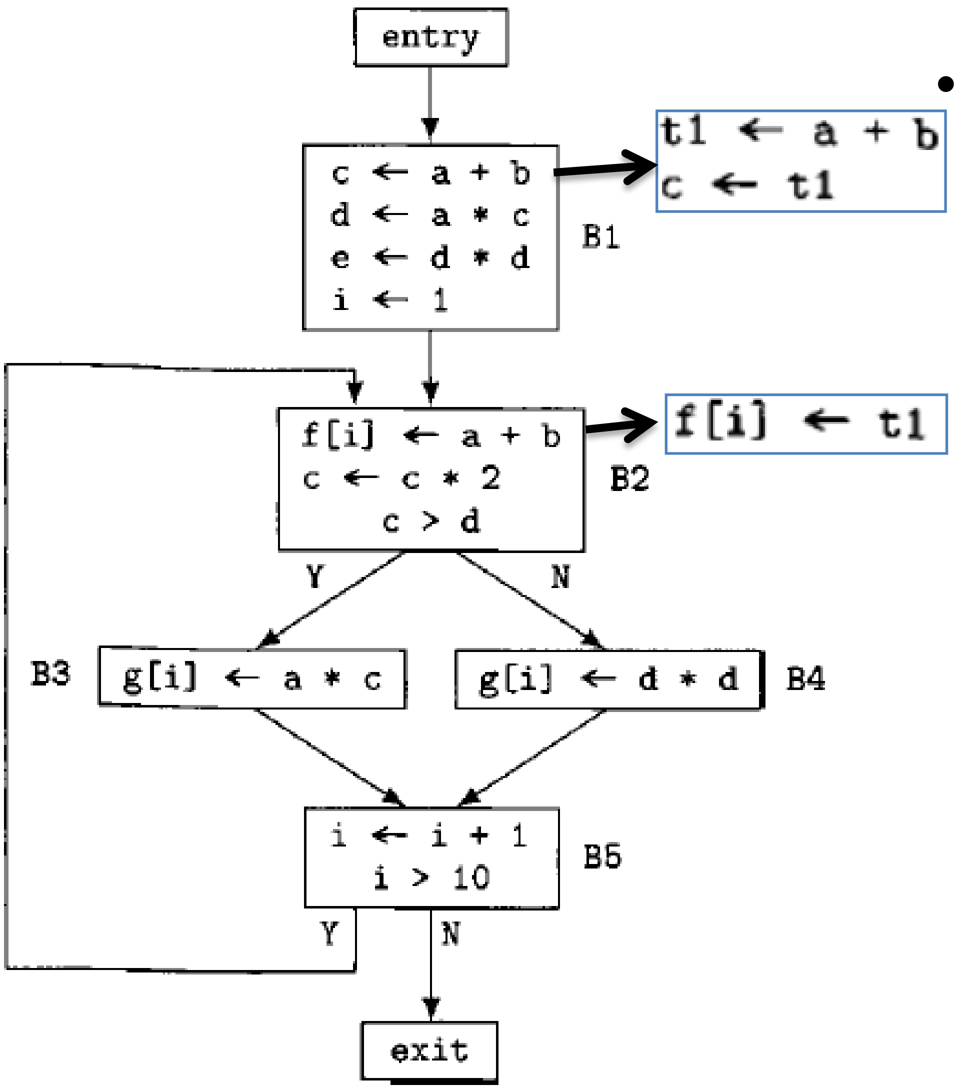
- in(B2) = {a+b, a*c, d*d}

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

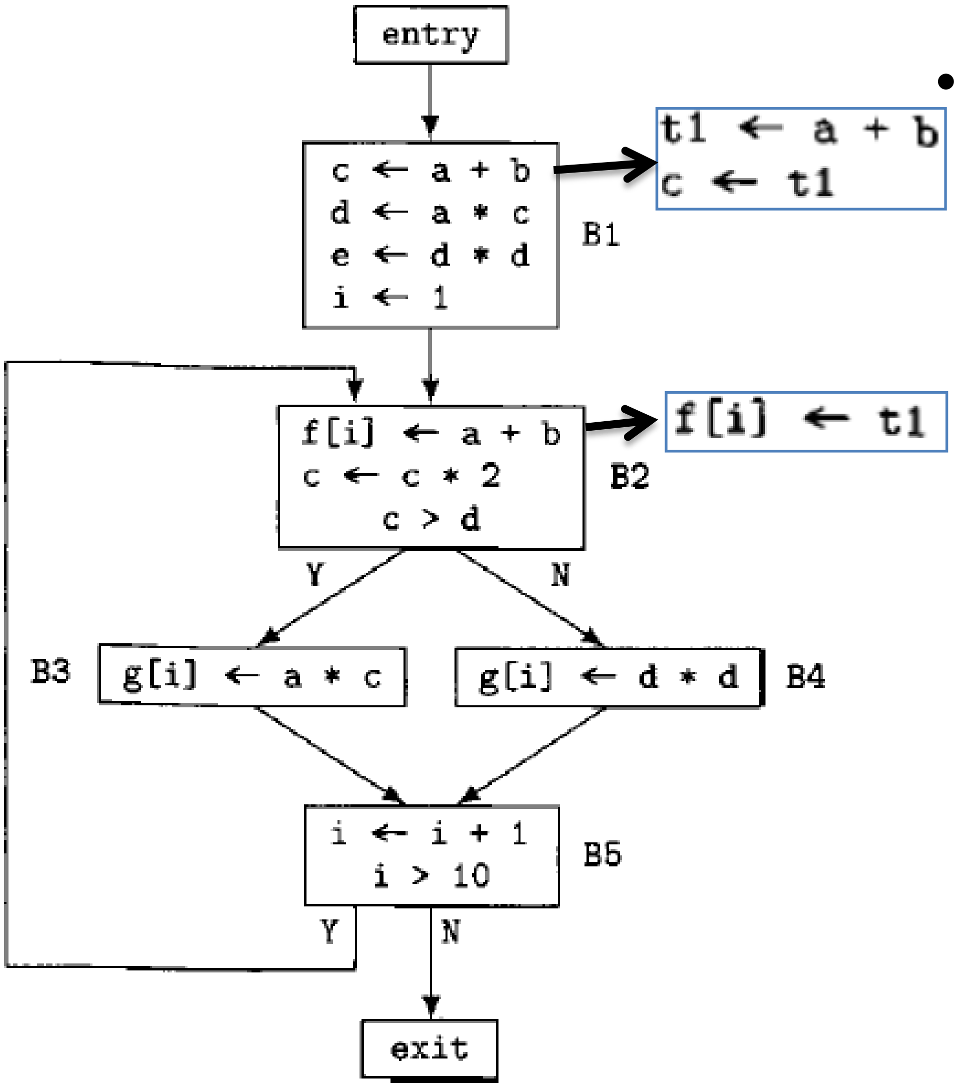


- in(B2) = {a+b, a*c, d*d}

  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



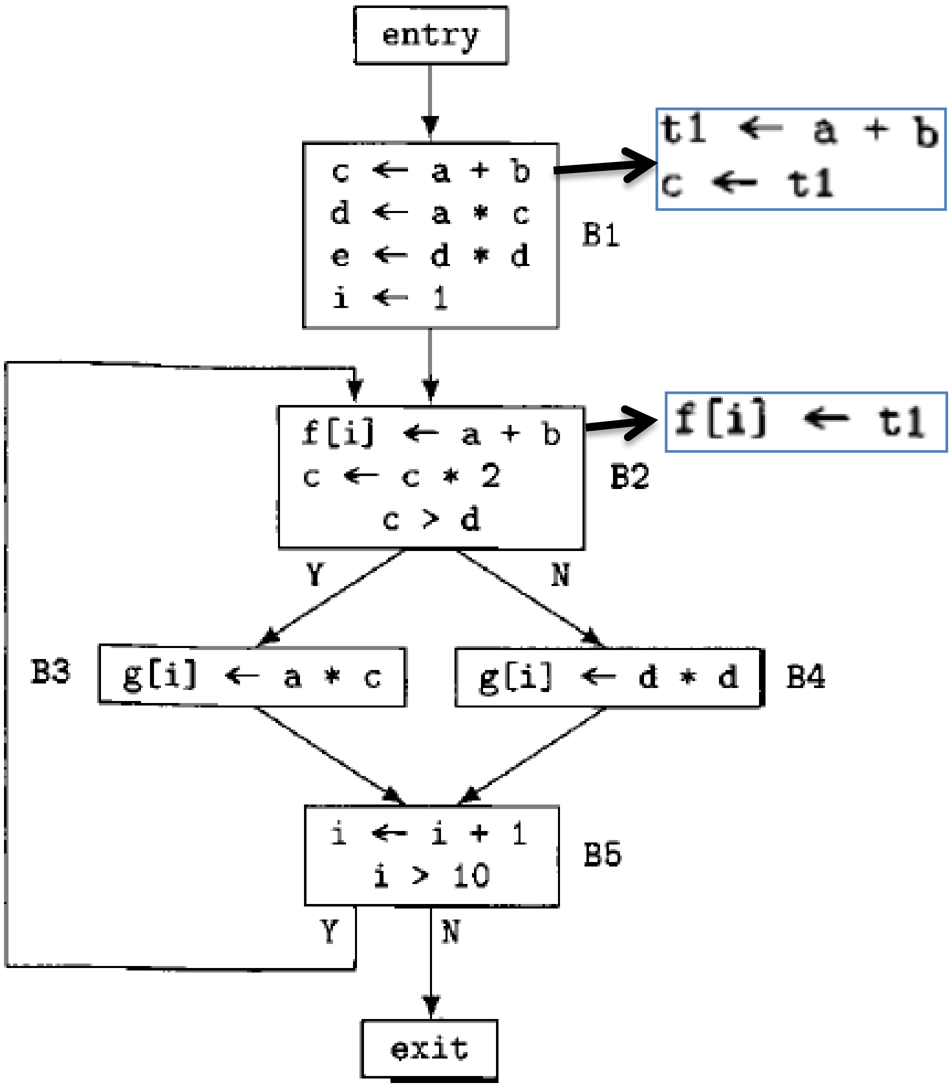
- in(B3) = {a+b, c>d, d*d}

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



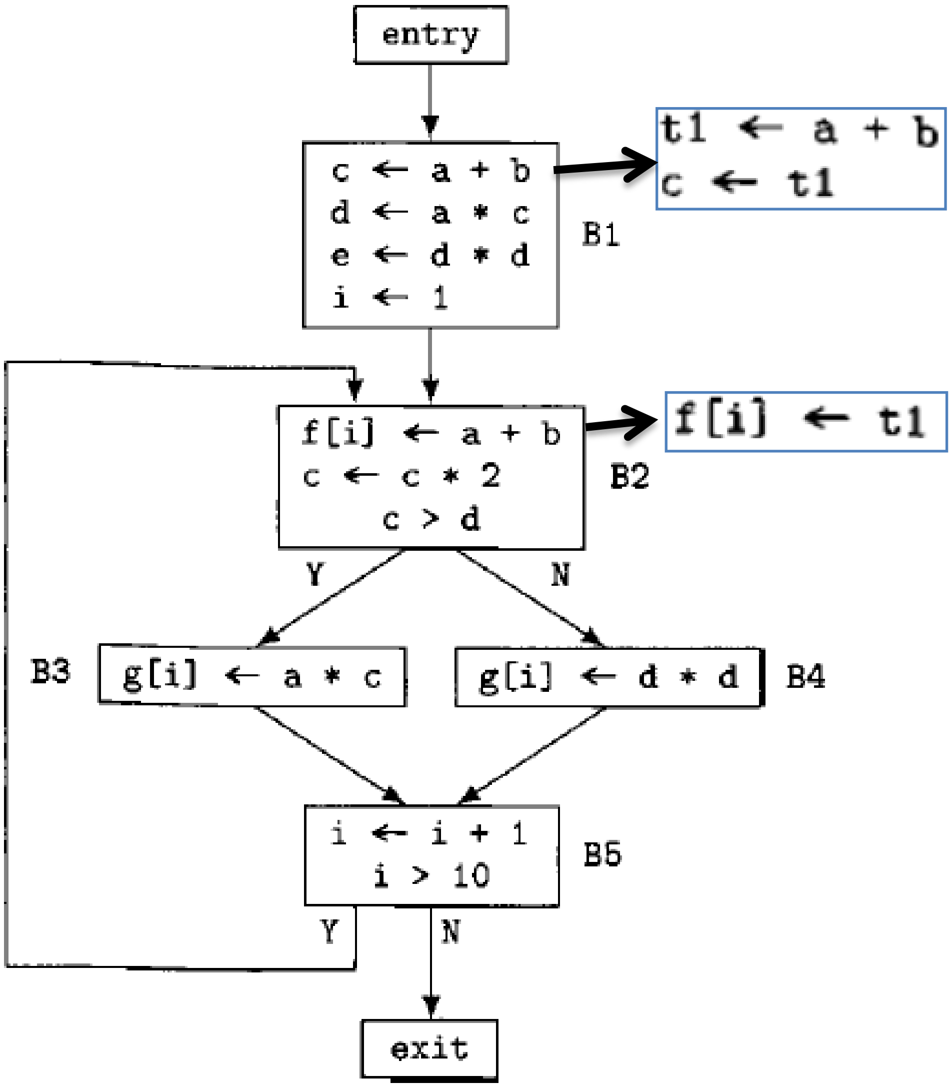
in(B4) = {a+b, c>d, d*d}

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

d*d ∈ AEin(B4) and d*d is found/located in B4

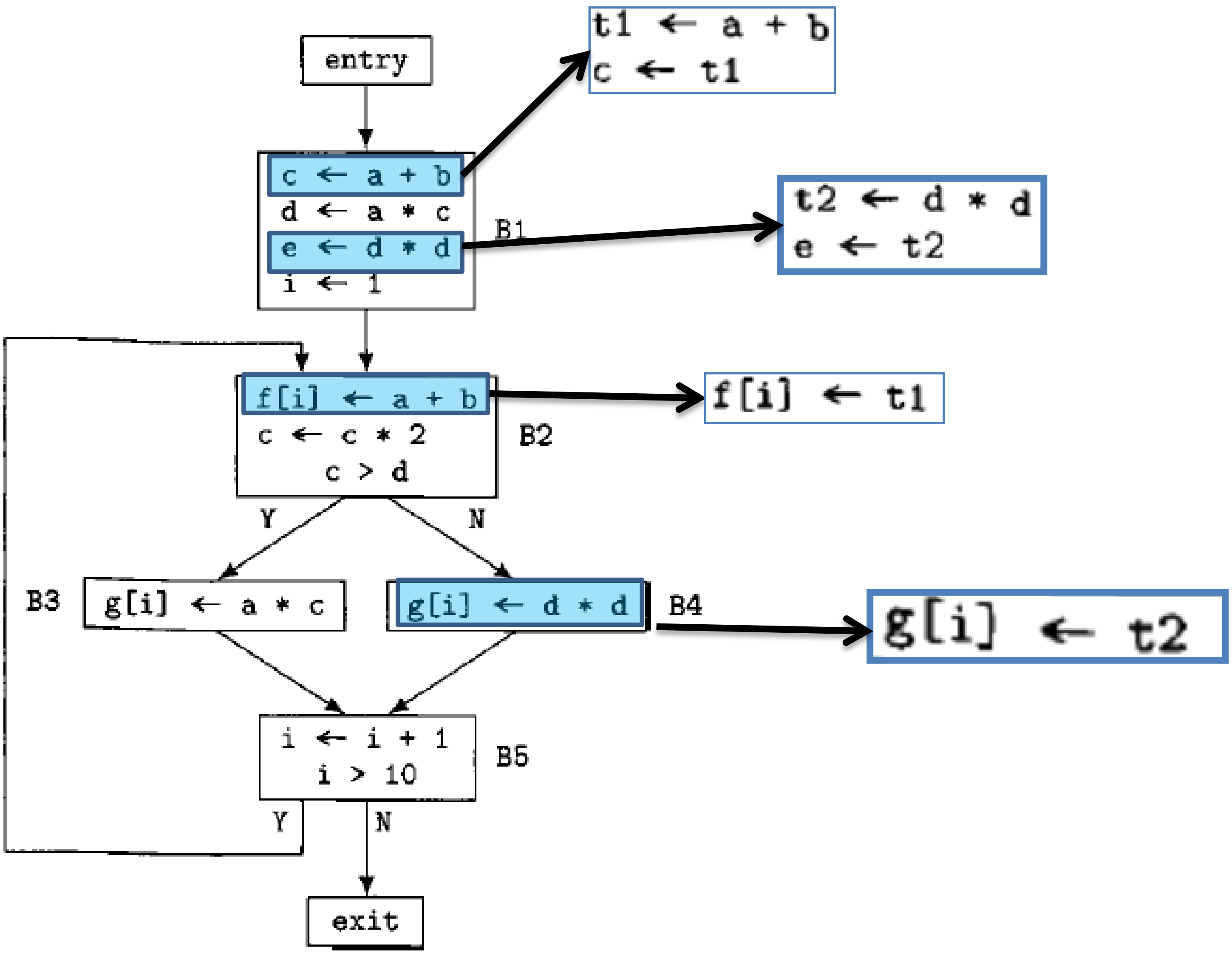
Applying the procedure to given flow graph



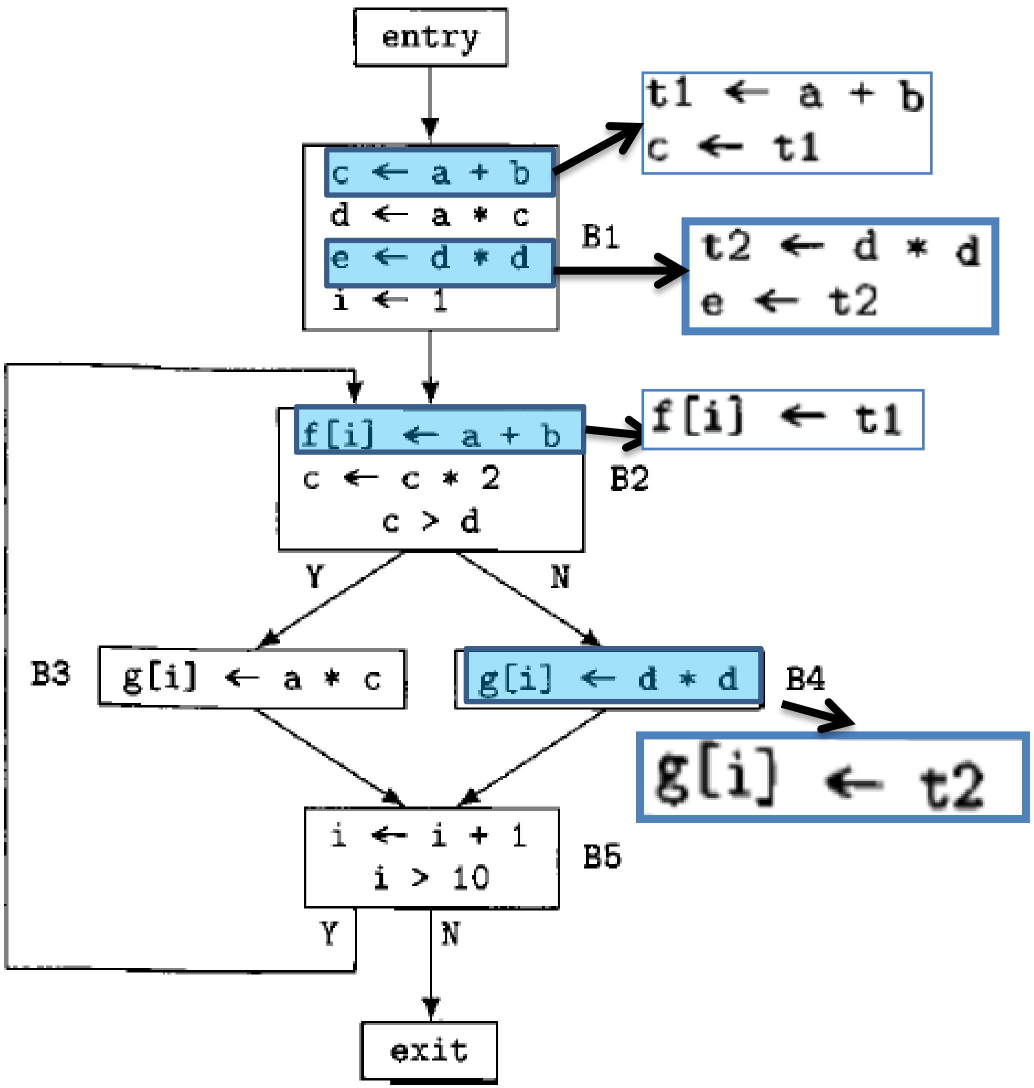
- in(B4) = {a+b, c>d, d*d}

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



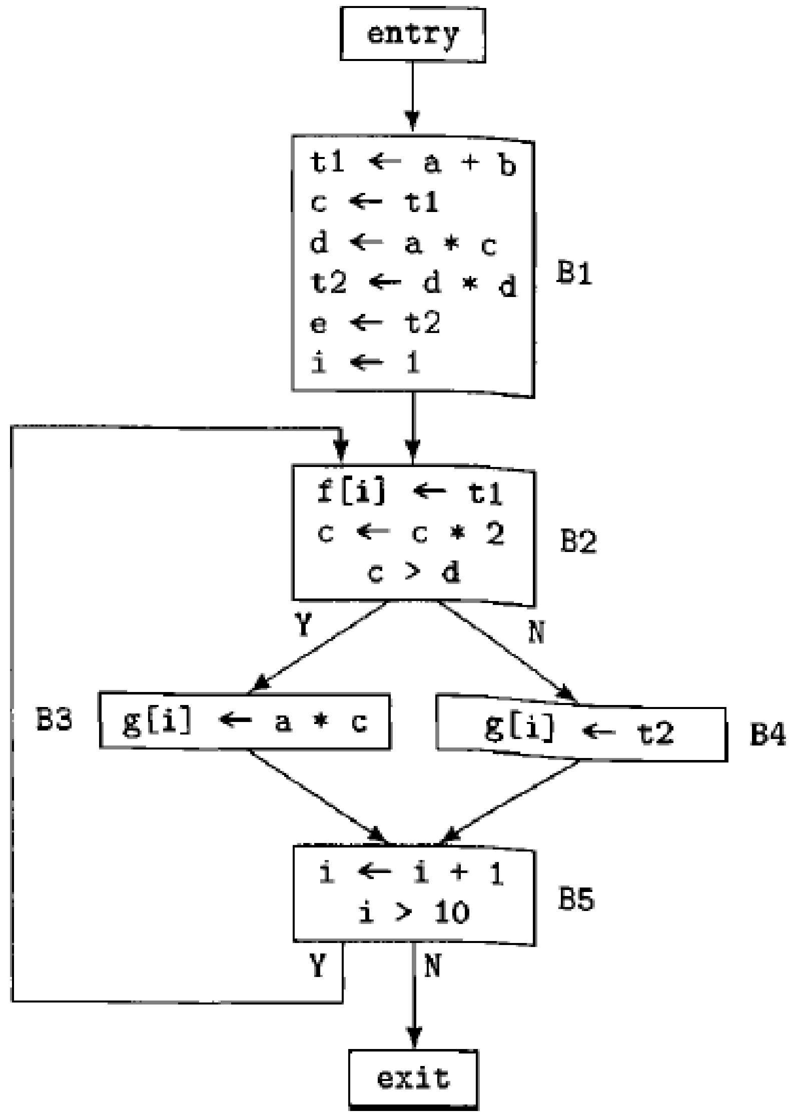
in(B5) = {a+b, c>d, d*d}

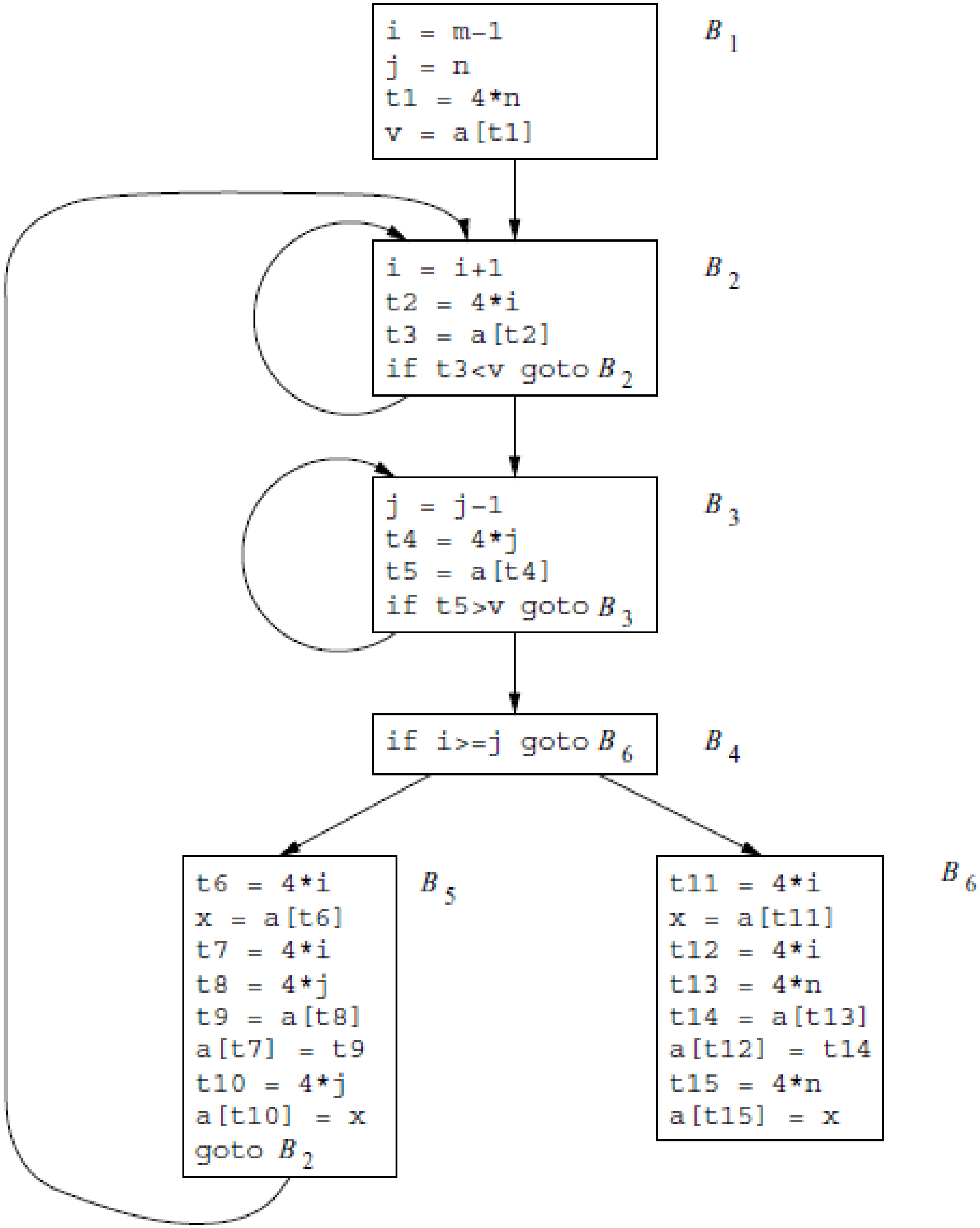
a+b ∈ AEin(B5) but a+b not found or located in B5

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

After global common subexpression elimination





Apply local and global common subexpression elimination to **quicksort code flowgraph**.