PARTIAL REDUNDANCY ELIMINATION

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

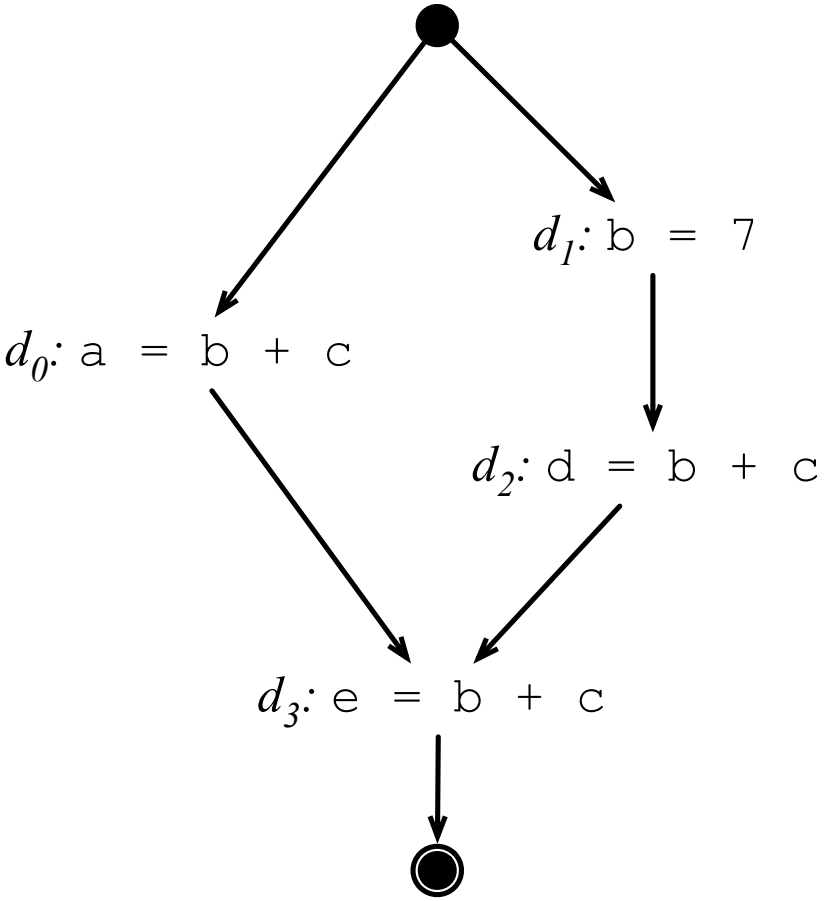
The material in these slides have been taken from the slides "Lazy Code Motion" 2011, by Keith D. Cooper & Linda Torczon.
The running example was taken from the Dragon Book, 3rd Edition, Section 9.5.

Partial Redundancy Elimination

- Partial Redundancy Elimination is one of the most complex classic compiler optimizations.
  - Includes many dataflow analyses
  - Subsumes several compiler optimizations:
    - Common subexpression elimination
    - Loop invariant code motion
- We shall be using "Lazy Code Motion", a technique developed by *Knoop et al*.
  - But, instead of using the original equations, which are explained in the Dragon Book, we shall use a simplification by *Drechsler* and *Stadel*♤.

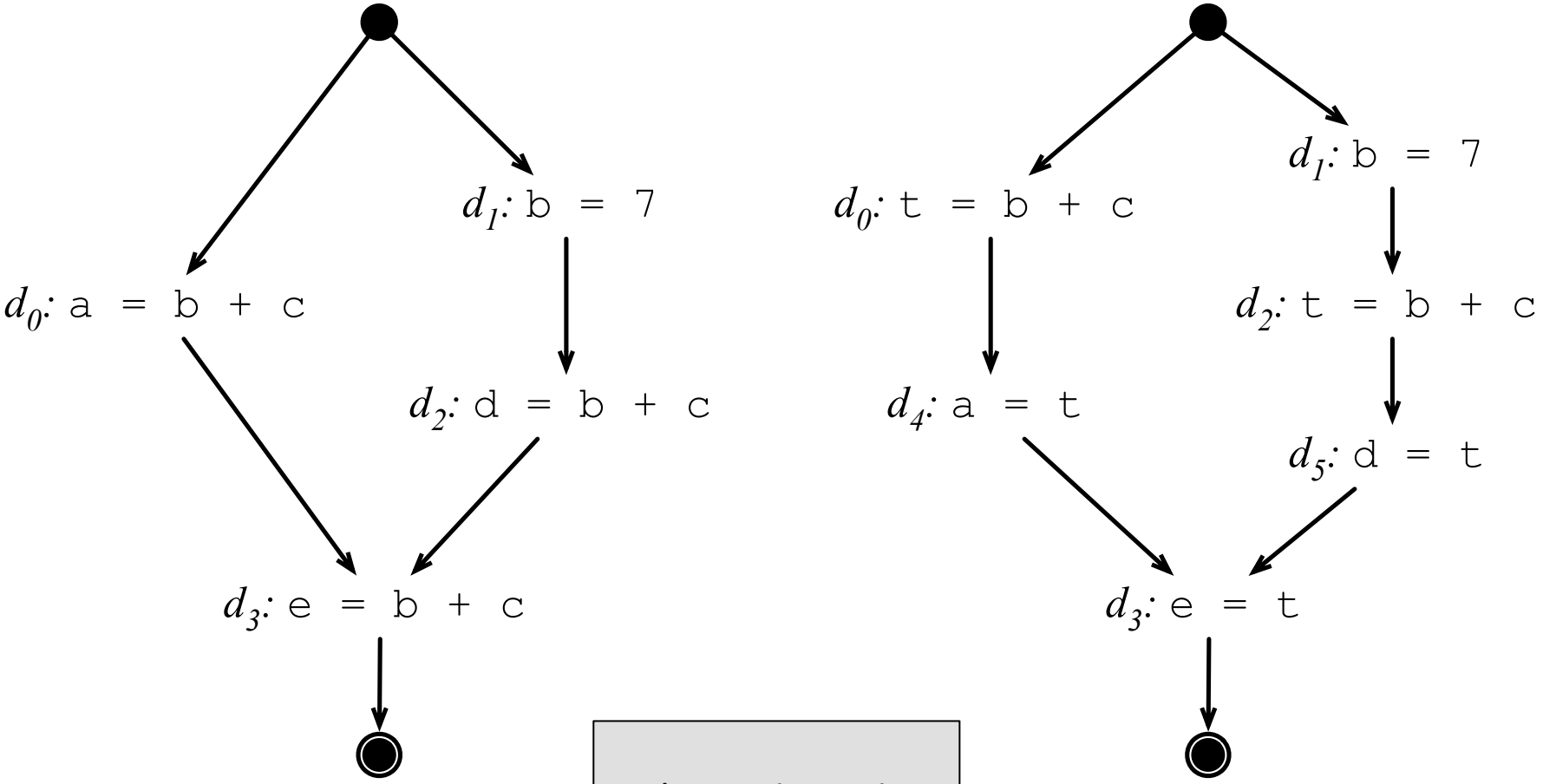
♤: See "A Bit of History" at the end of this set of slides.

Global Common Expression



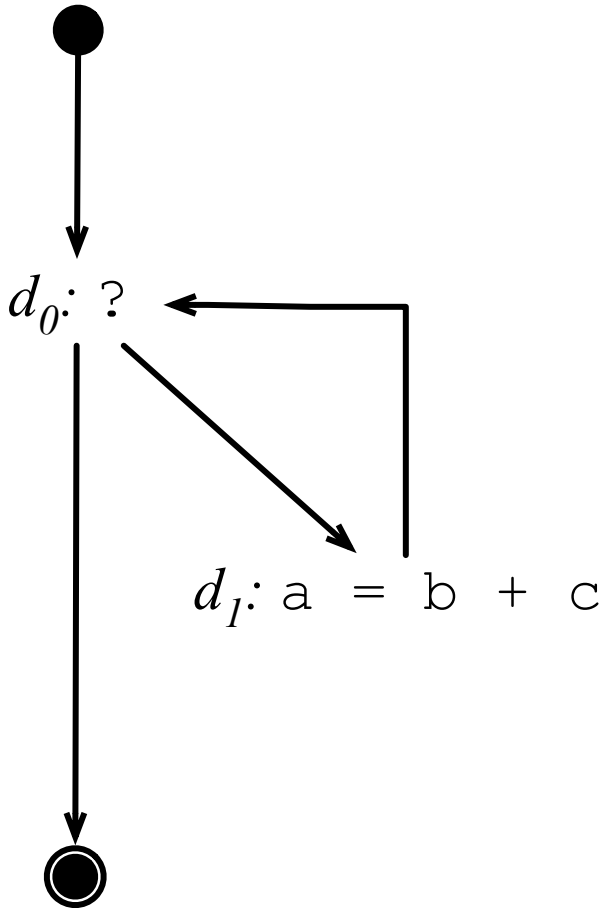
- Which computations are redundant in the example in the left?
- How could we optimize this program?
- Is the optimized program always better than the original program?

Global Common Expression



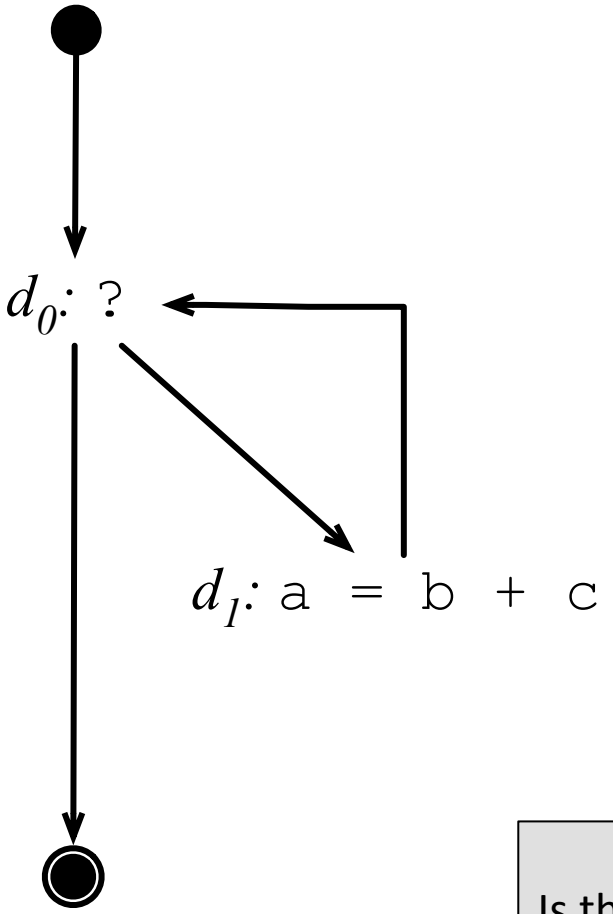
Is this optimization always profitable?

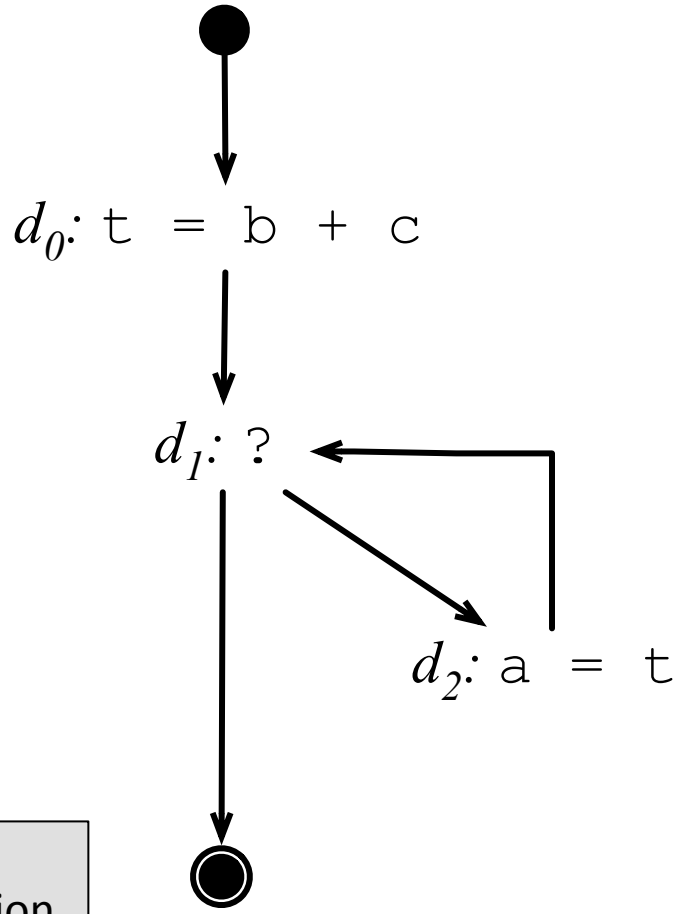
Loop Invariant Code Motion



- Which computations are redundant in the example in the left?
- How could we optimize this program?
- Will the optimized program always be better than the original program?

Loop Invariant Code Motion

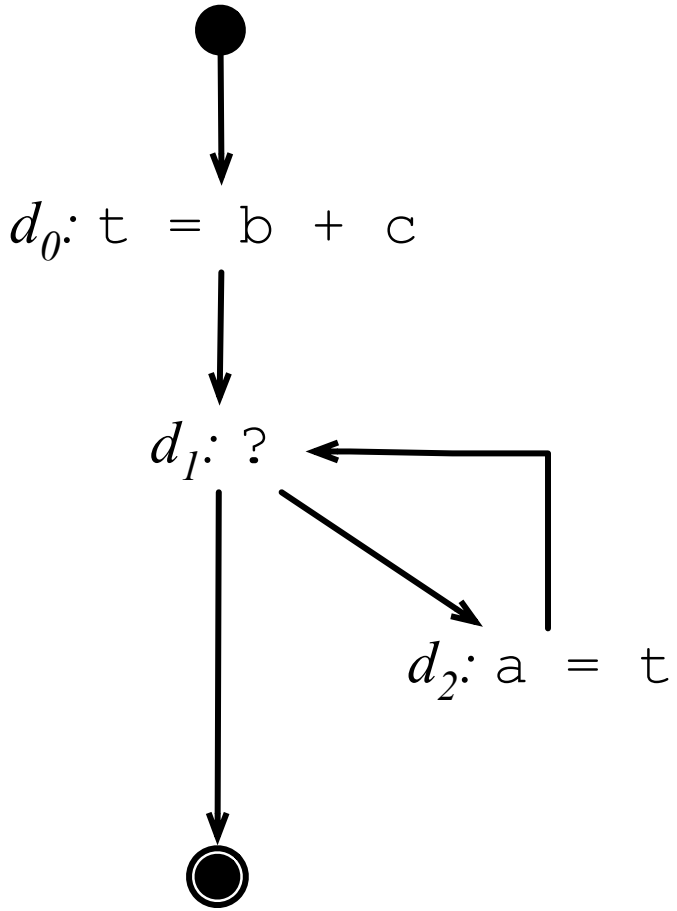




Is this optimization always profitable?

Unsafe Optimization

- An optimization is unsafe, in our context, if it may result in slower code sometimes.



1) This case of loop invariant code motion is unsafe. Why?

2) Would it be possible to transform the loop, in such a way that the optimization is always safe?

Loop Inversion

```
while c {
    S;
}
```

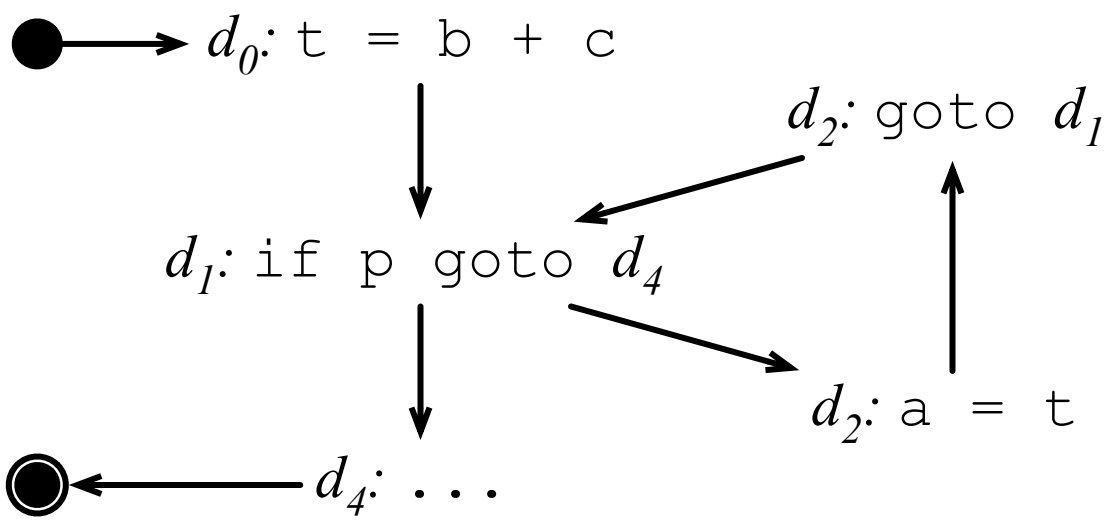
```
if c {
    do {
        S;
    } while c
}
```

- Which loop is faster?
- Is this optimization safe?
- How does it help in loop invariant code motion?

Loop Inversion

```
t = b + c

while p {

    a = t

}
```

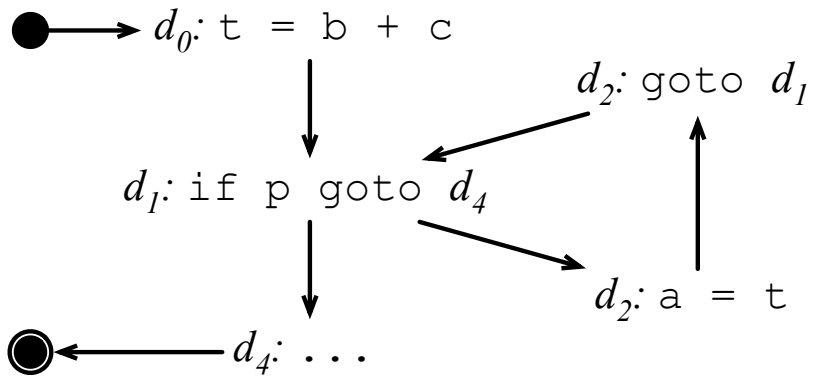
Can you transform **this** while in a do-while loop?



How the corresponding CFG would look like?

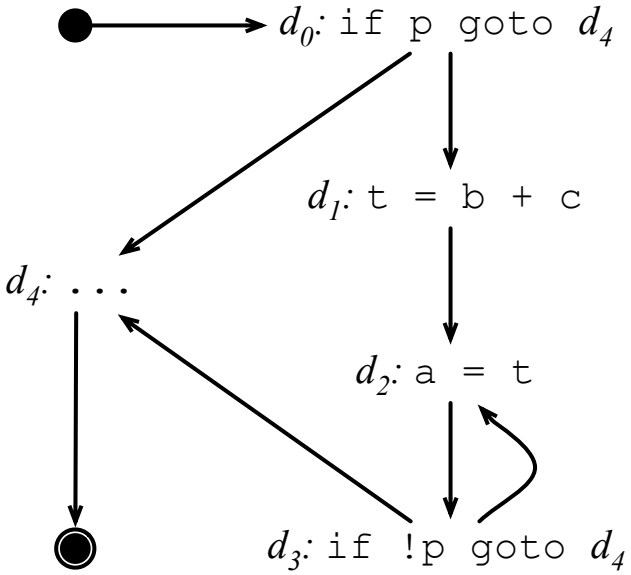
Loop Inversion

```
t = b + c

while p {

    a = t

}
```



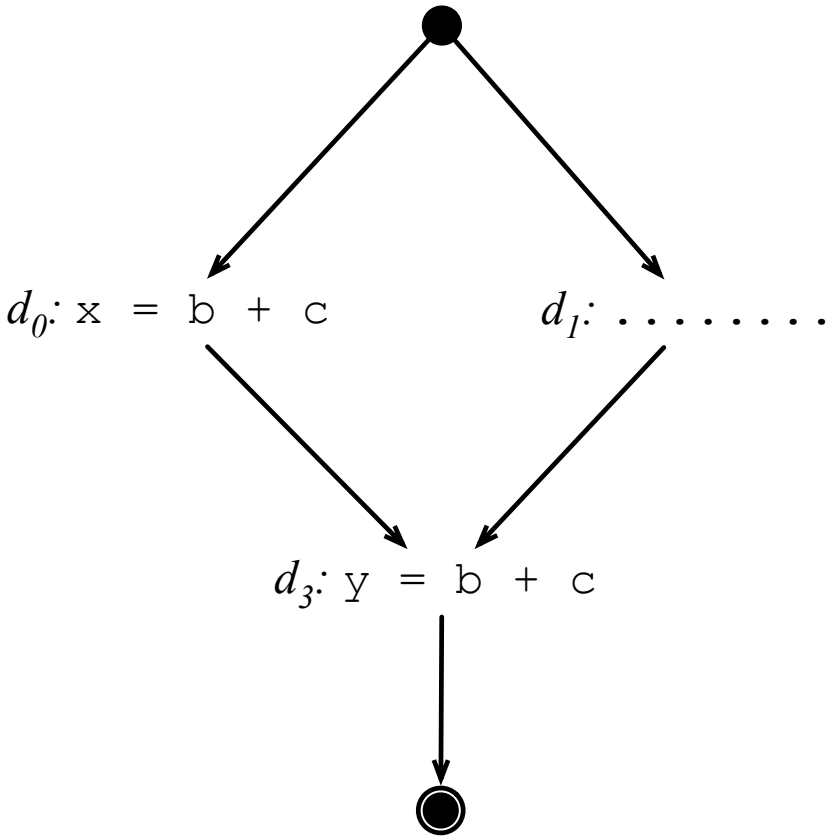
Which loop is faster?

```
if p {

    t = b + c

    do {

        a = t

    } while !p

}
```



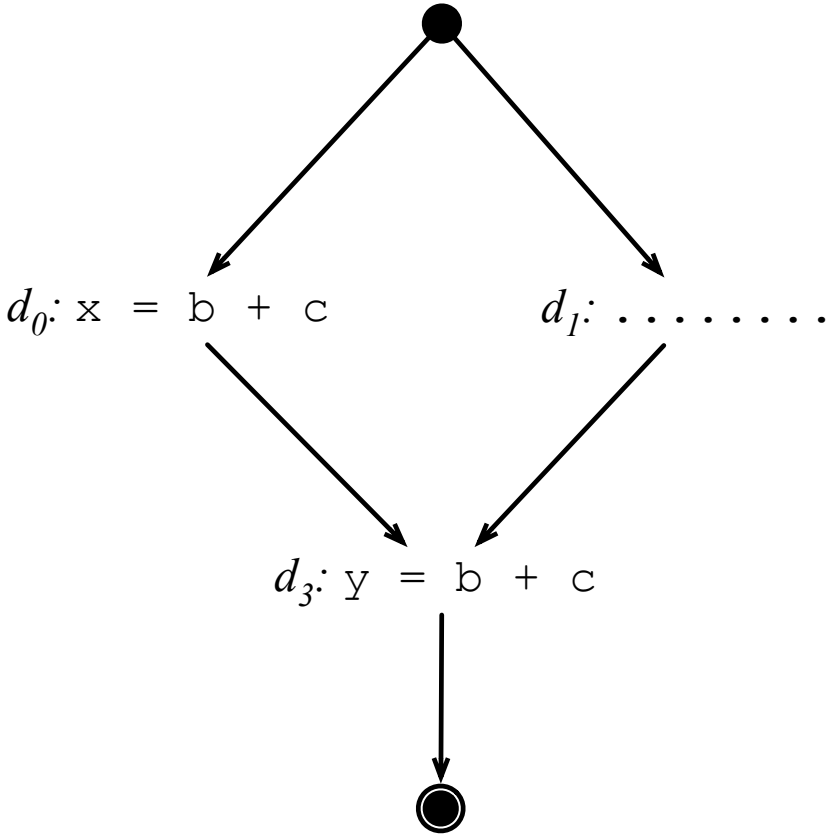
Is loop invariant code motion safe now?

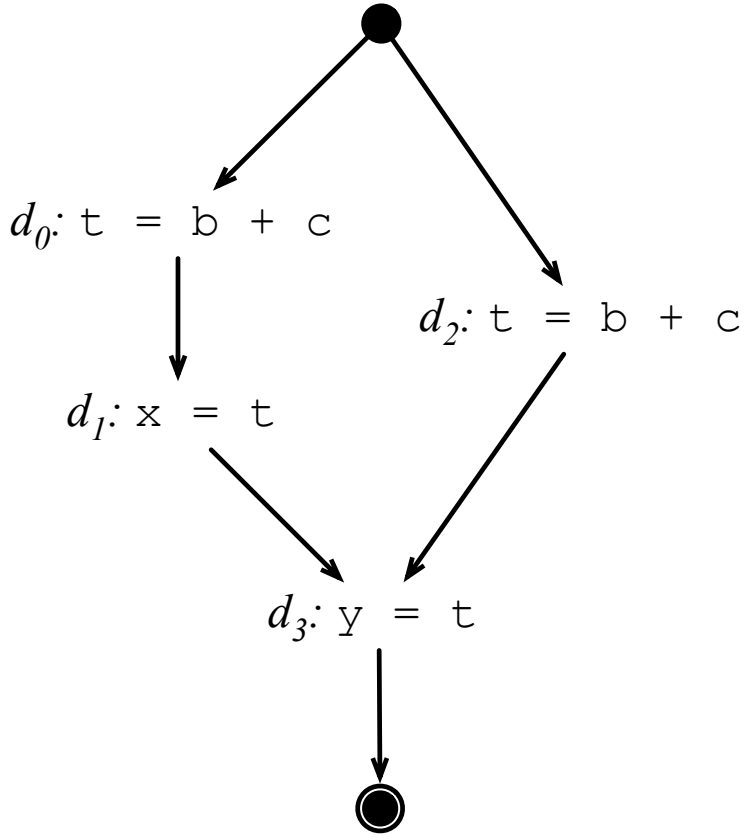
Partial Redundancy



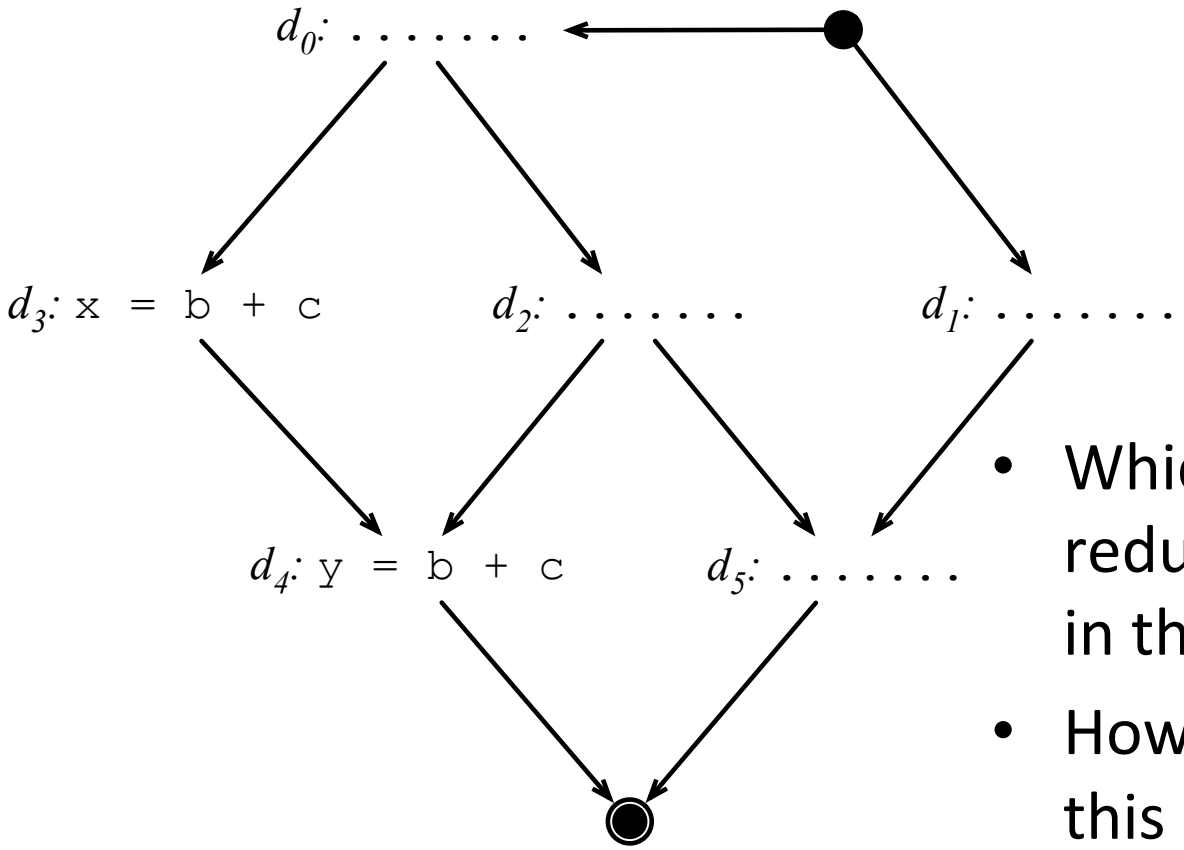
- Which computations are redundant in the example in the left?
- How could we optimize this program?
- Will the optimized program always be better than the original program?

Partial Redundancy



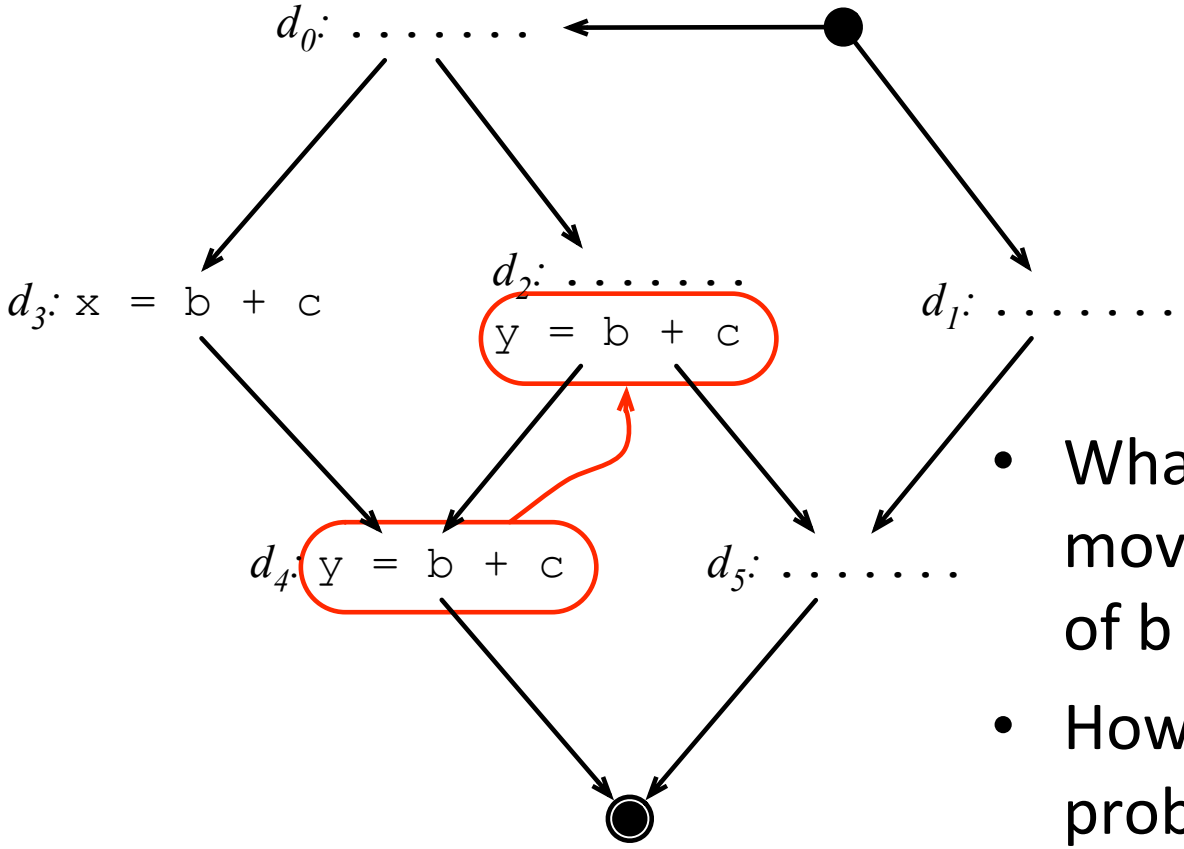


Critical Edges



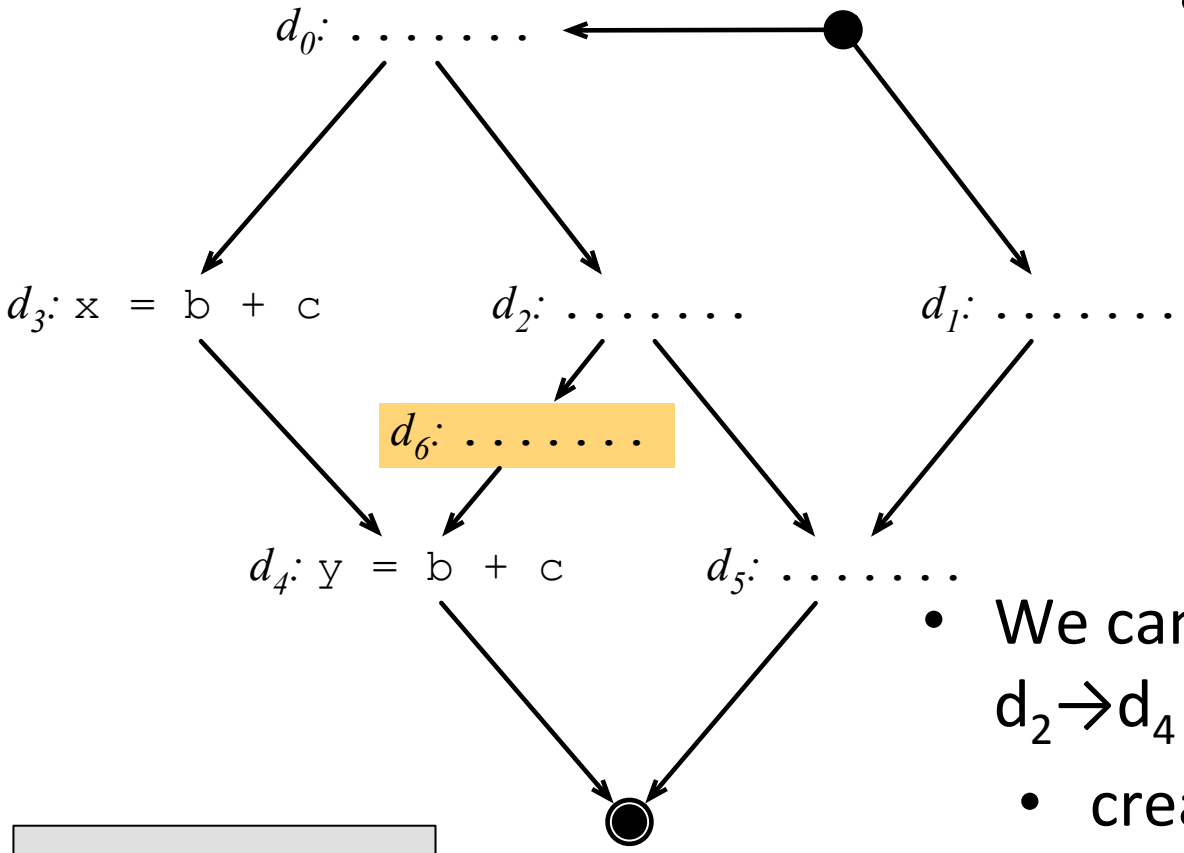
- Which computations are redundant in the example in the left?
- How could we optimize this program?
- Will the optimized program always be better than the original program?

Critical Edges



- What is the problem of moving the computation of b + c to d_2?
- How to solve this problem?

Critical Edges

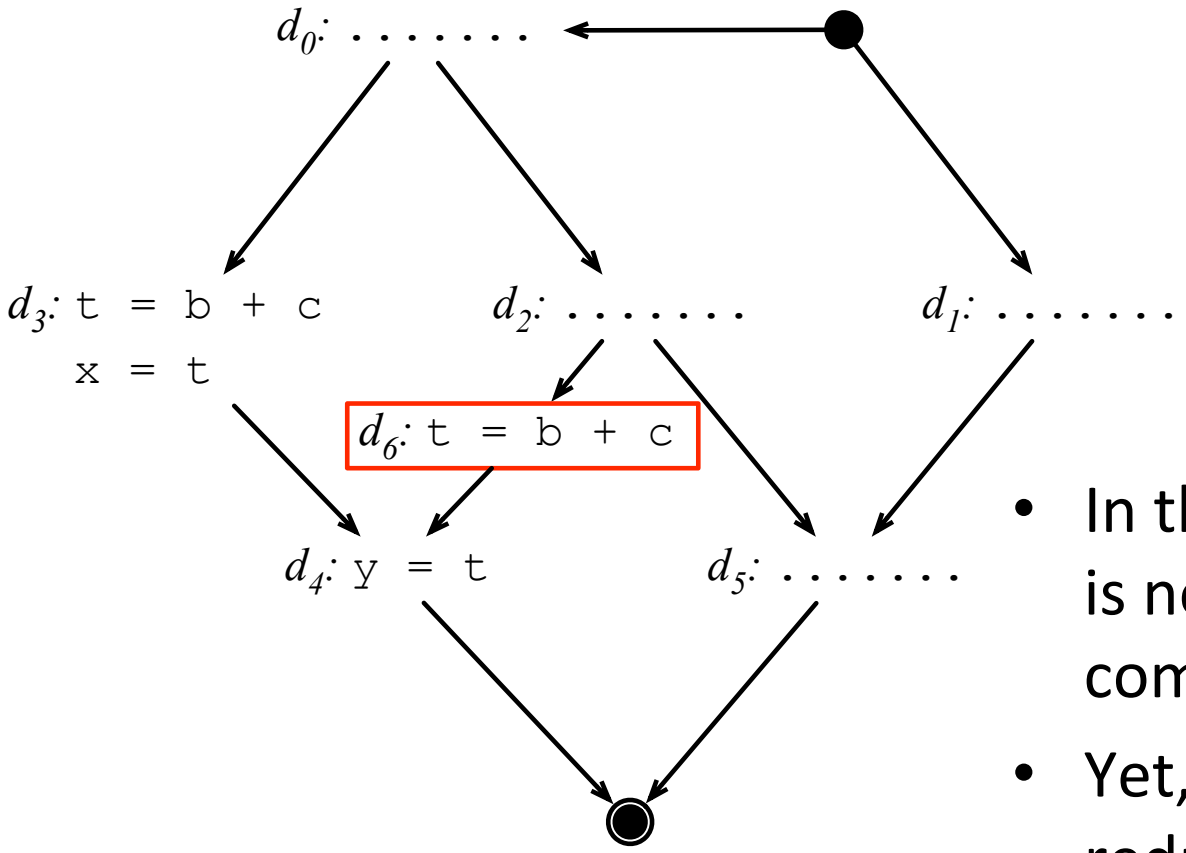


- If an edge links a block with multiple successors to a block with multiple predecessors, then this edge is called a *Critical Edge*.

- We can split a critical edge $d_2 \rightarrow d_4$ if we:
  - create a basic block d_x
  - create edge $d_2 \rightarrow d_x$
  - create edge $d_x \rightarrow d_4$
  - remove edge $d_2 \rightarrow d_4$

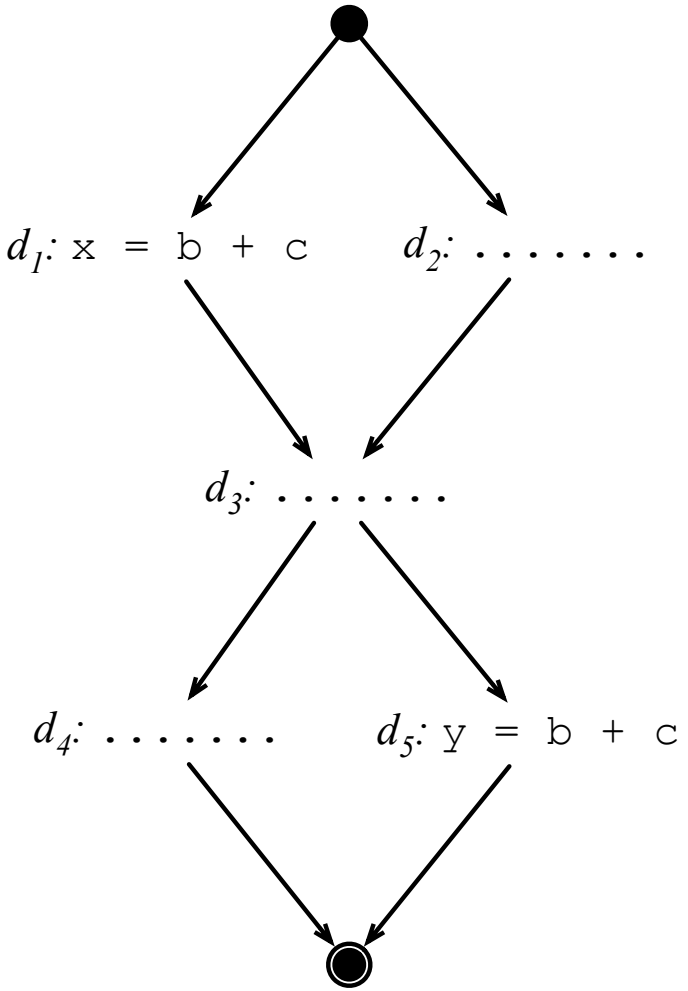
And how could we remove the redundancy in this new CFG?

Critical Edges



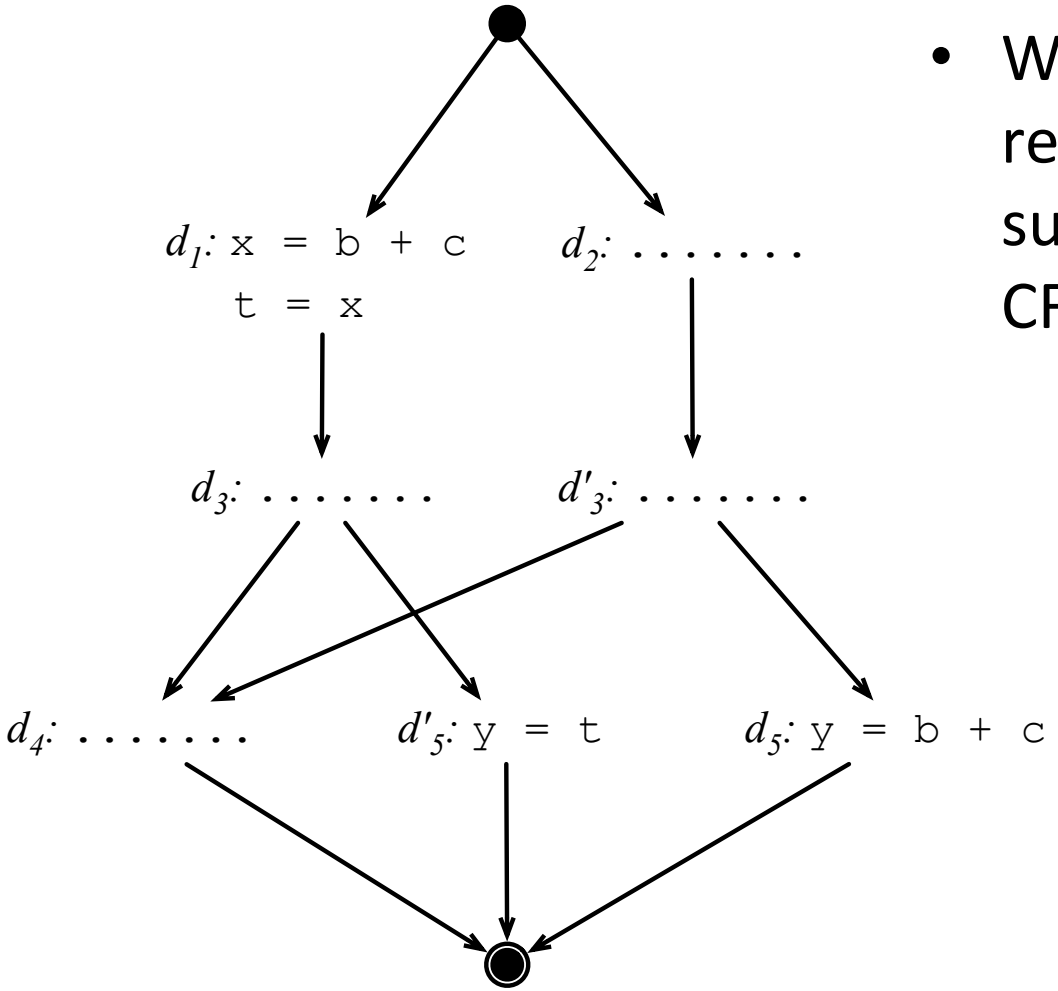
- In this new program there is no redundant computation .
- Yet, there are redundancies which are harder to eliminate…

Harder Redundancies



- What is the redundant computation in this program?
- How could we eliminate it?

Harder Redundancies



- We can only eliminate this redundancy replicating a substantial part of the CFG.

The number of paths is exponential in the number of branches; thus, eliminating every redundant expression might increase the program size exponentially.

Henceforth, the only program transformation that we will allow will be the elimination of critical edges.

Lazy Code Motion

- Lazy Code Motion is a technique developed by Knoop *et al.* to eliminate redundancies in programs.
- Lazy Code Motion has the following properties:
  1. All redundant computations of expressions that can be eliminated without code duplication are eliminated.
  2. The optimized program does not perform any computation that is not in the original program execution.
  3. Expressions are computed at the latest possible time.
     - That is why it is called *lazy*.

Why is item 3 important and desirable?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

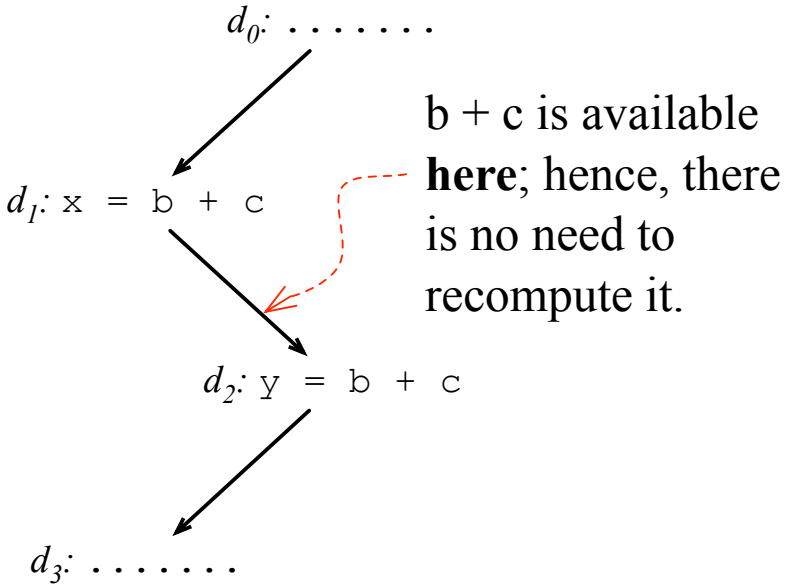
Lazy Code Motion

- LCM is solved via a combination of dataflow analyses:
  - Available Expressions
  - Very Busy (*Anticipable*) Expressions
- From the result of these dataflow analyses, we remove redundancies:
  - Compute earliest place where each expression can be computed without creating new redundancies.
  - From each of these places, try to push computation down on the CFG, to meet item 3 of the LCM problem.

Available Expressions

- An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.
- Computing the points where an expression is available is important because computations of an expression at points of availability are redundant.

We had seen available expressions before. Can you remember the dataflow equations?



Available Expressions

- An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.
- Computing the points where an expression is available is important because computations of an expression at points of availability are redundant.

How should the IN and OUT sets be initialized?

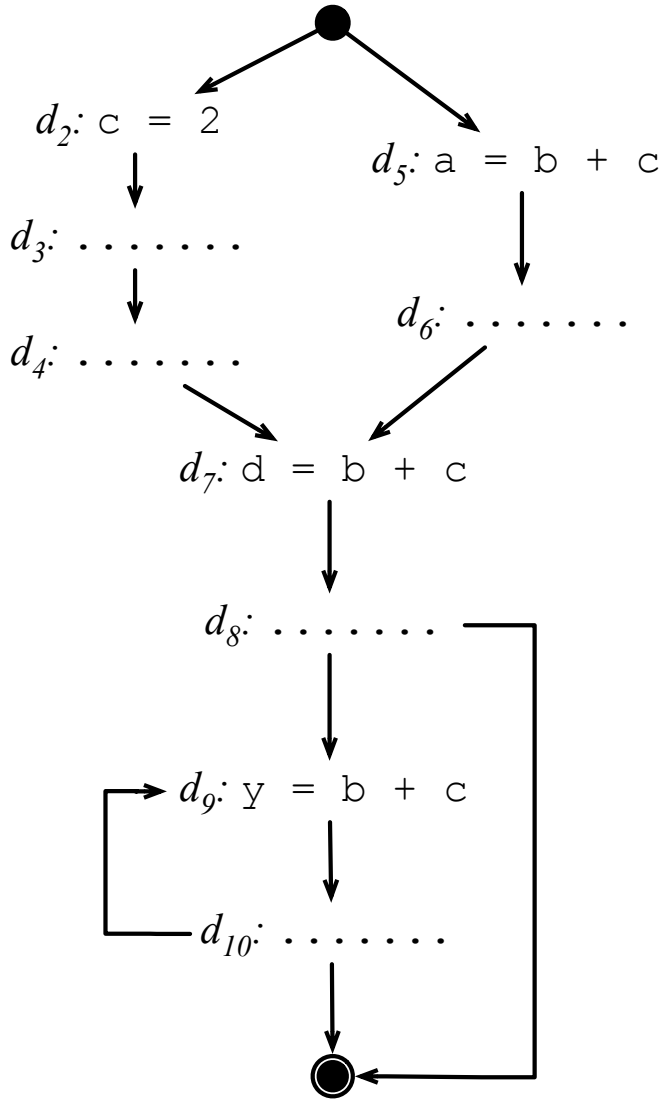
We had seen available expressions before. Can you remember the dataflow equations?

$$p : v = E$$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$

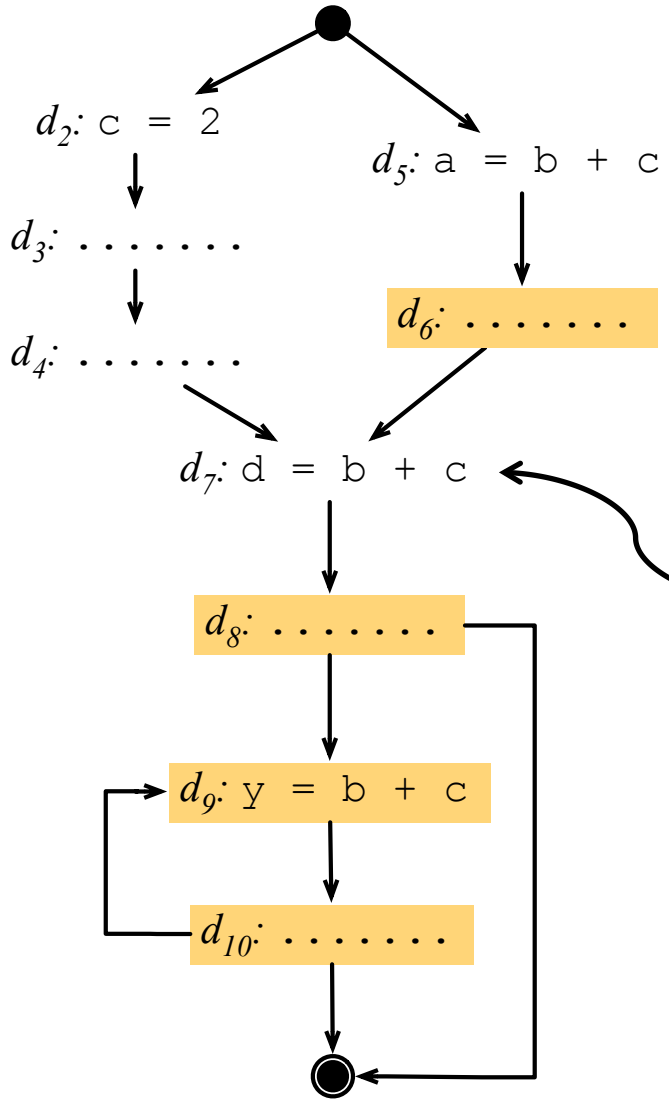
Available Expressions



- An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.

Where is the expression b + c available in the program on the left?

Available Expressions

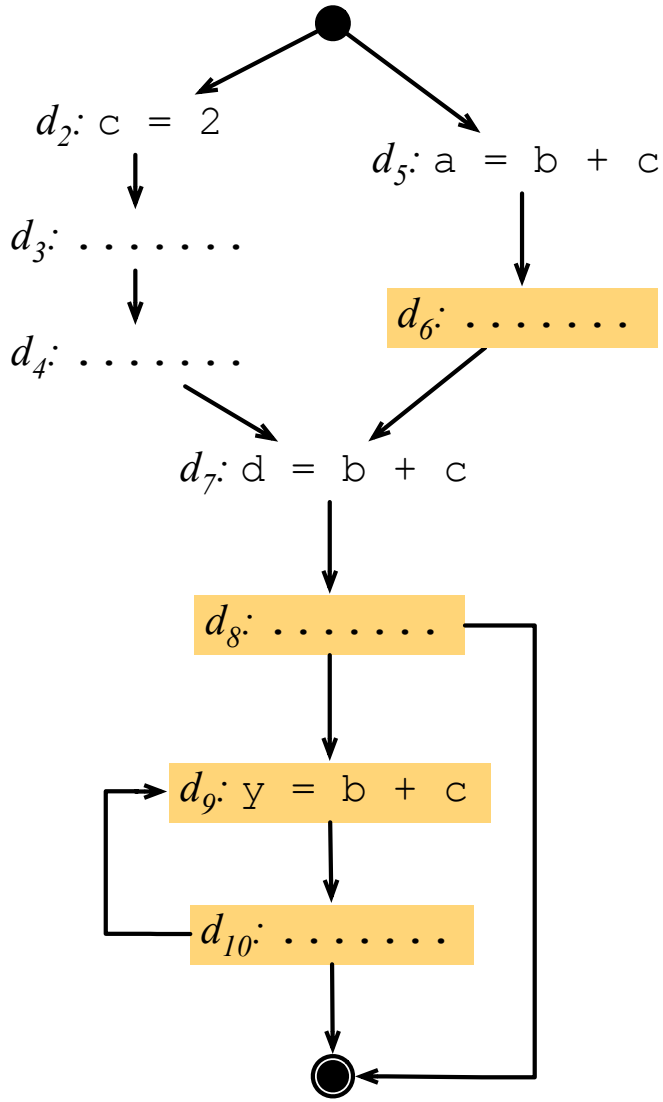


- An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.

Why is b + c not available at d_7?

We are marking the program points p such that $IN_{AVAILABLE}[p] \supseteq \{b + c\}$

A bit of intuition for Availability



- The set of available expressions tells us the program points where computations are redundant.
- e ∈ $OUT_{AVAILABLE}(b)$:
  - evaluating *e* at the exit of *b* gives same result; thus, the computation of *e* could move to the exit of *b*.
- e ∈ $IN_{AVAILABLE}(b)$:
  - An evaluation of *e* at the entry of *b* is redundant, because *e* is available from every predecessor of *b*.

Anticipable (Very Busy) Expressions

- An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined until p_{end}.
- It is safe to move an expression to a basic block where that expression is anticipable.
  - By "safe" we mean "performance safe", i.e., no extra computation will be performed.
- Notice that if an expression e is computed at a basic block where it is both available and anticipable, then that computation is clearly redundant.

We had seen very busy expressions before. Can you remember the dataflow equations?

Anticipable (Very Busy) Expressions

- An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined until its computation.
- It is safe, i.e., no extra computation will be performed, to move an expression to a basic block where that expression is anticipable.

How should the IN and OUT sets be initialized?

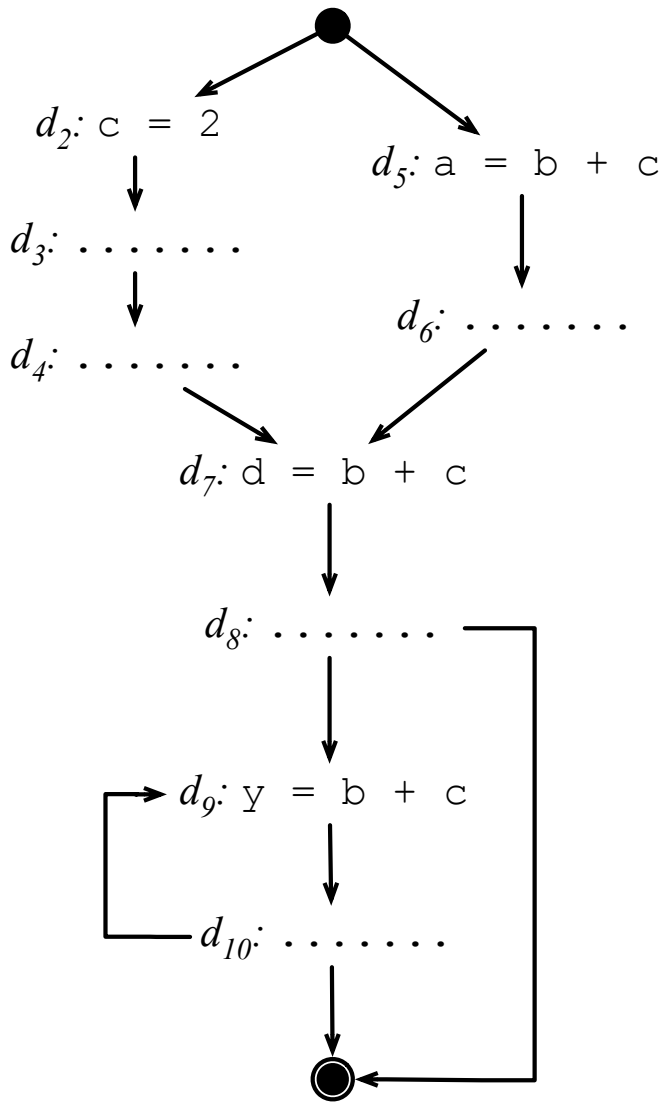
We had seen very busy expressions before. Can you remember the dataflow equations?

$$p : v = E$$

$$IN(p) = (OUT(p) \setminus \{Expr(v)\}) \cup \{E\}$$

$$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$$

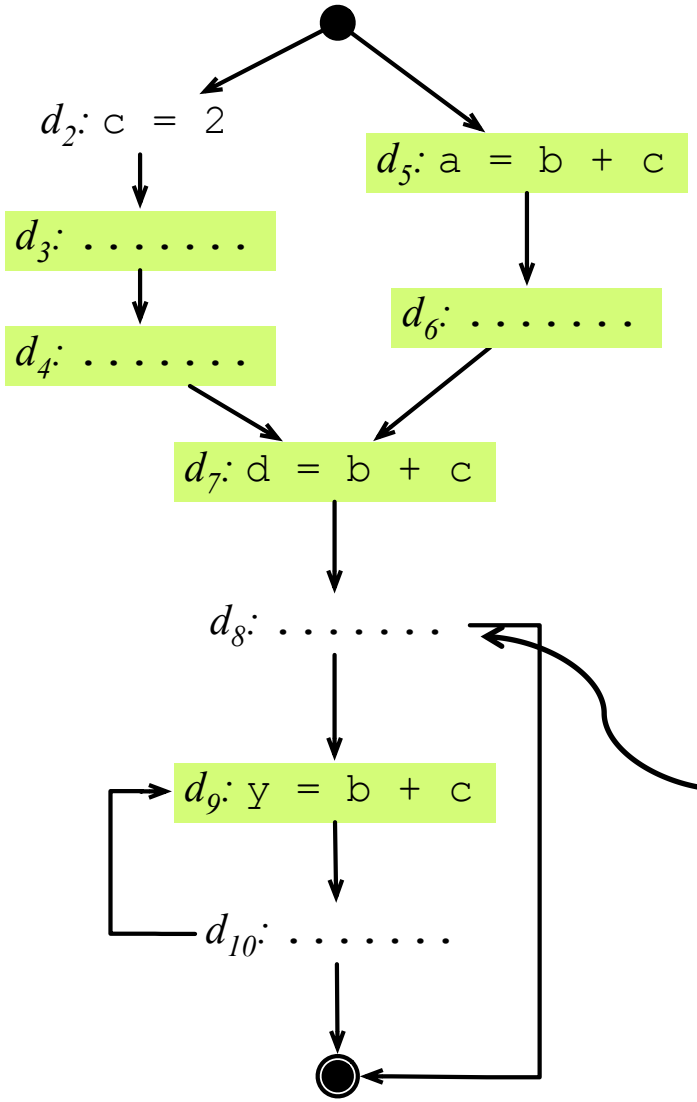
Anticipable (Very Busy) Expressions



- An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined between *p* and the point of computation.

Where is the expression b + c anticipable in the program on the left?

Anticipable (Very Busy) Expressions

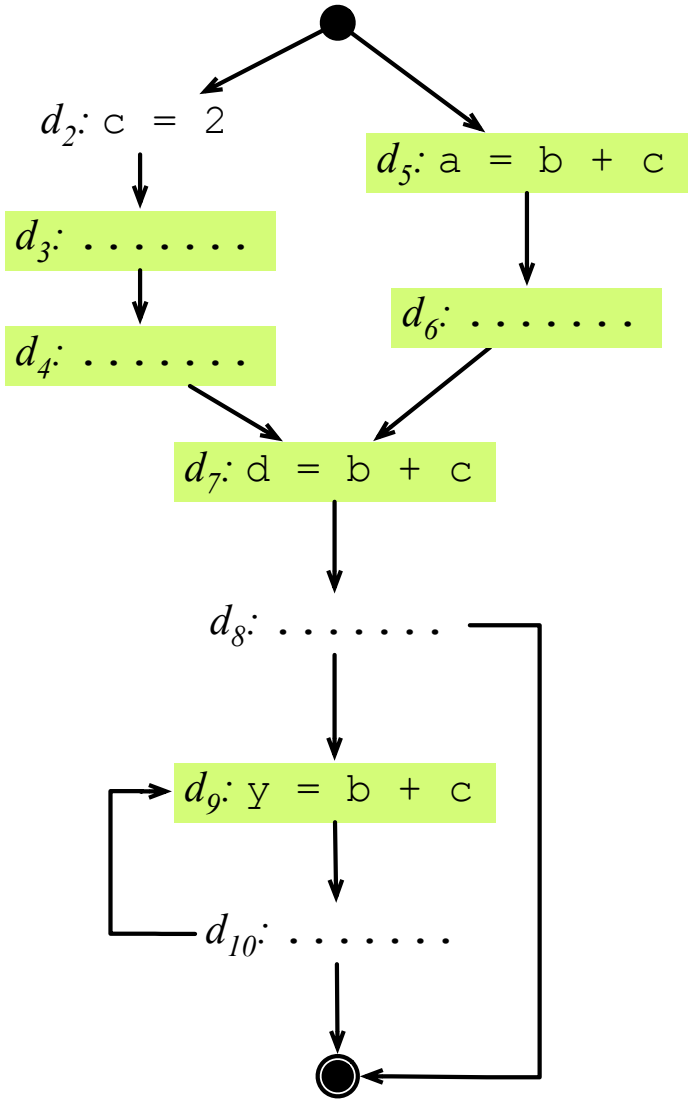


- An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined between *p* and the point of computation.

Why is a + b not very busy at this point **here**?

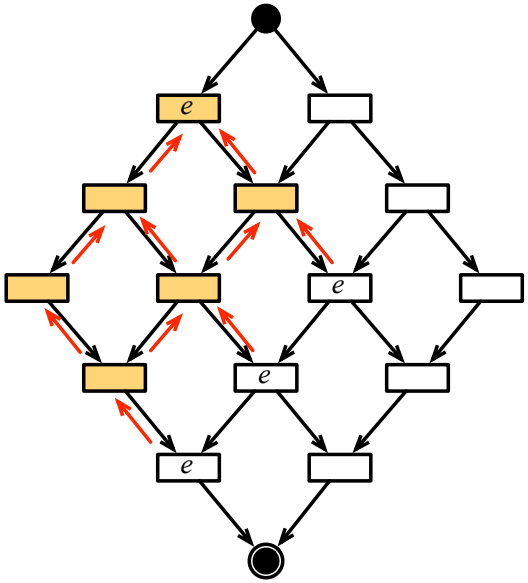
We are marking the program points p such that $IN_{ANTICIPABLE}[p] \supseteq \{b + c\}$

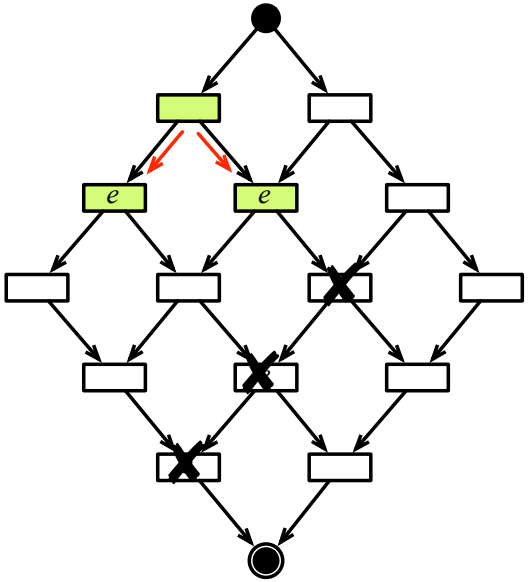
A bit of intuition on Anticipability



- The set of anticipable expressions tells us the program points where computations are necessary.
- e ∈ $IN_{ANTICIPABLE}(b)$:
  - evaluating *e* at the entry of *b* gives same result; thus, the computation of *e* could move to the entry of *b*.
- e ∈ $OUT_{ANTICIPABLE}(b)$:
  - An evaluation of *e* at any successor of *b* could move to the end of *b*, because *e* is used along every path that leaves *b*.

Pushing up, Pulling down





- There are now two steps that we must perform:
  - First, we find the earliest places in which we can move the computation of an expression without adding unnecessary computations to the CFG. This step is like pushing the computation of the expressions up.
  - Second, we try to move these computations down, closer to the places where they are necessary, without adding redundancies to the CFG. This phase is like pulling these computations down the CFG.

Why do we care about moving computations down?

Earliest Placement

- We must now find the earliest possible places where we can compute the target expressions.
  - Earliest in the sense that p_1 comes before p_2 if p_1 precedes p_2 in any topological ordering of the CFG.

$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

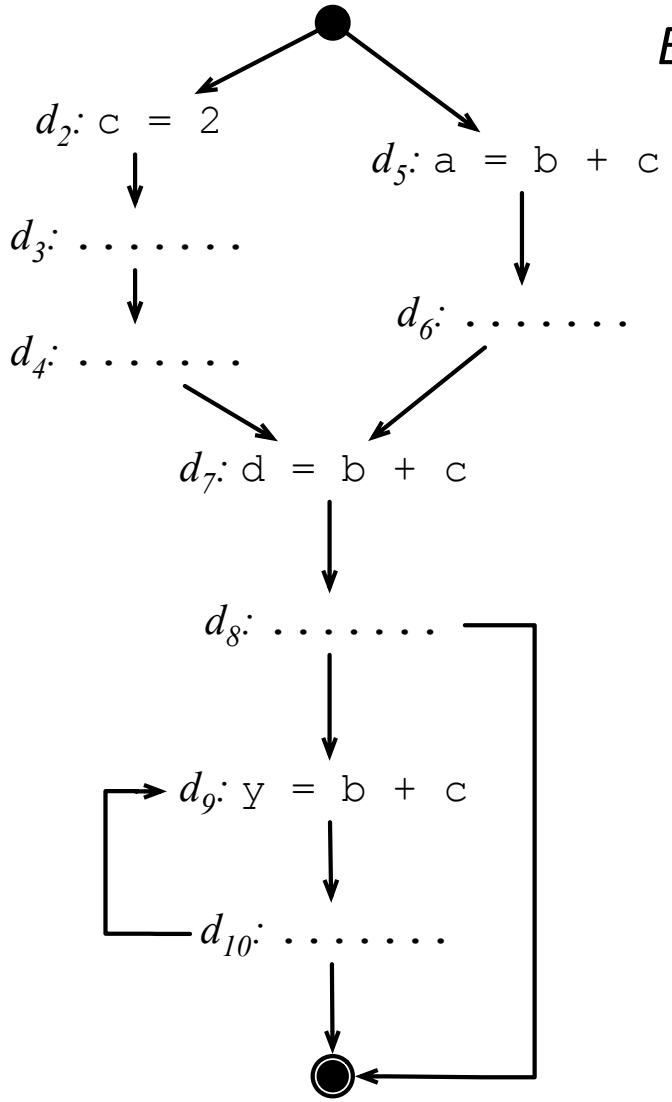
- An expression *e* is in *KILL(i)* if any term of *e* is redefined at block *i*.
- This predicate is computed for edges, not for nodes.
  - We can move an expression *e* to and edge *ij* only if *e* is anticipabled at the entrance of *j*. Why?
  - If the expression is available at the beginning of the edge, than we should not move it there. Why?

Earliest Placement

$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

- We have gotten an intuition about **this** first part, right?
  - We can move an expression *e* to and edge *ij* only if *e* is anticipabled at the entrance of *j*.
  - If the expression is available at the beginning of the edge, than we should not move it there.
- But the **second** part is trickier.
  - If an expression is anticipable at *i*, then we should not move it to *ij*, because we can move it to before *i*.
  - On the other hand, if *i* kills the expression, then it cannot be anticipable before *i*.

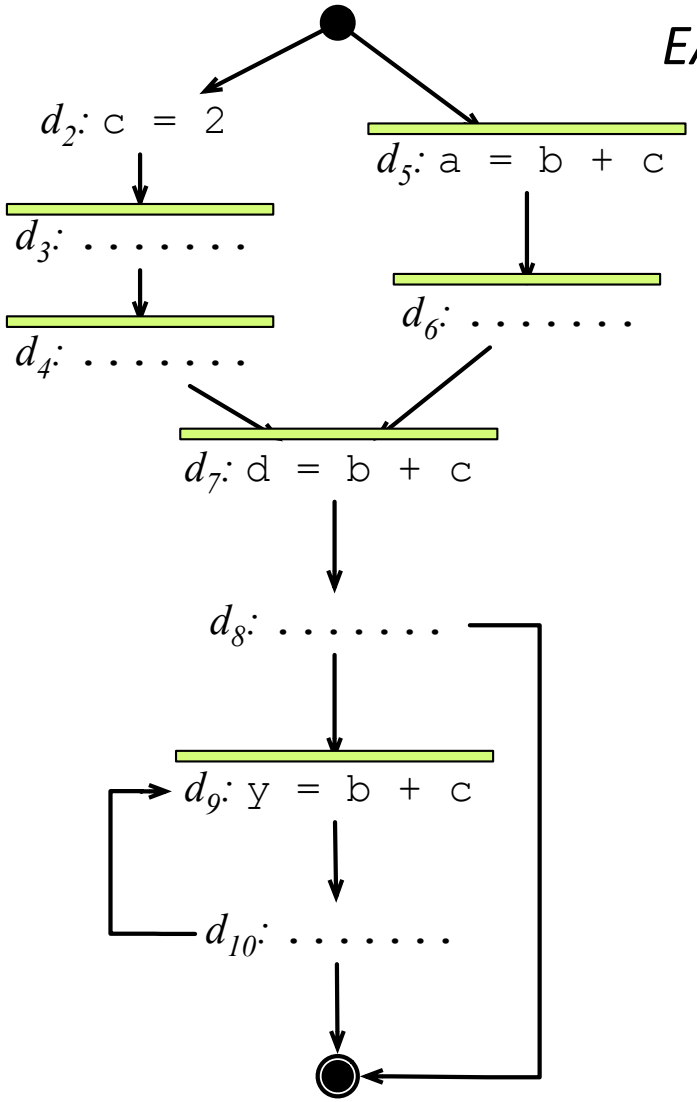
Earliest Placement Points



$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

Which are the basic blocks that have b + c in their $IN_{ANTICIPAPLE}$ sets?

Earliest Placement Points

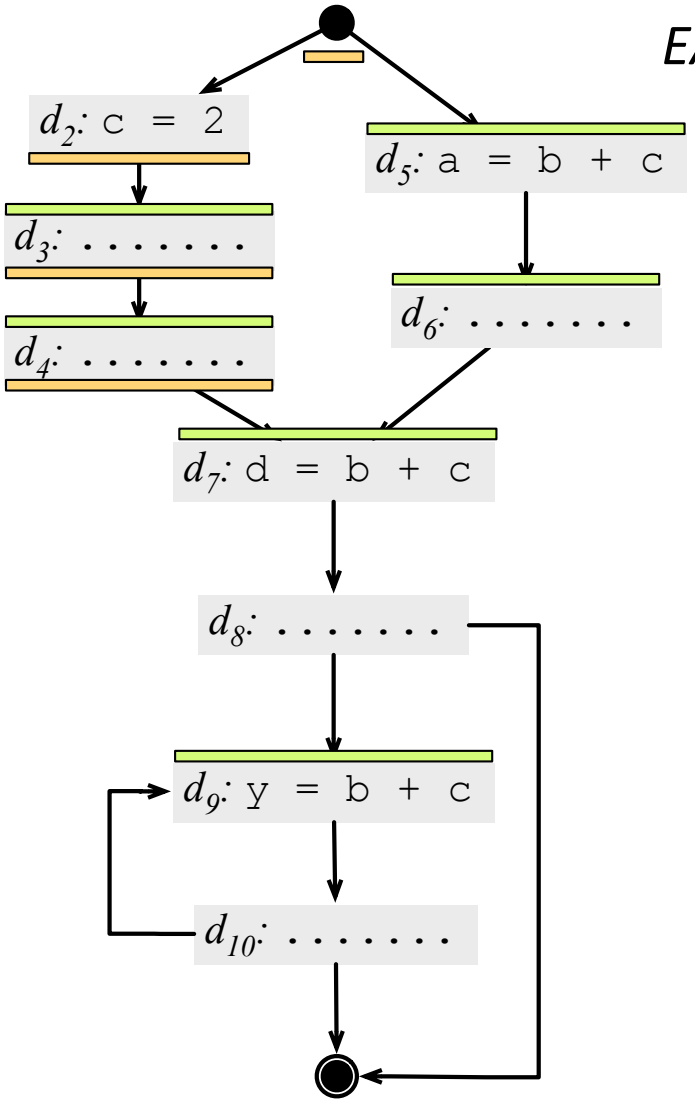


$$EARLIEST(i, j) = \mathbf{IN_{ANTICIPABLE}(j)} \cap \overline{OUT_{AVAILABLE}(i)}$$

$$\cap\ (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

Which are the basic blocks that do not have b + c in their $OUT_{AVAILABLE}$ sets?

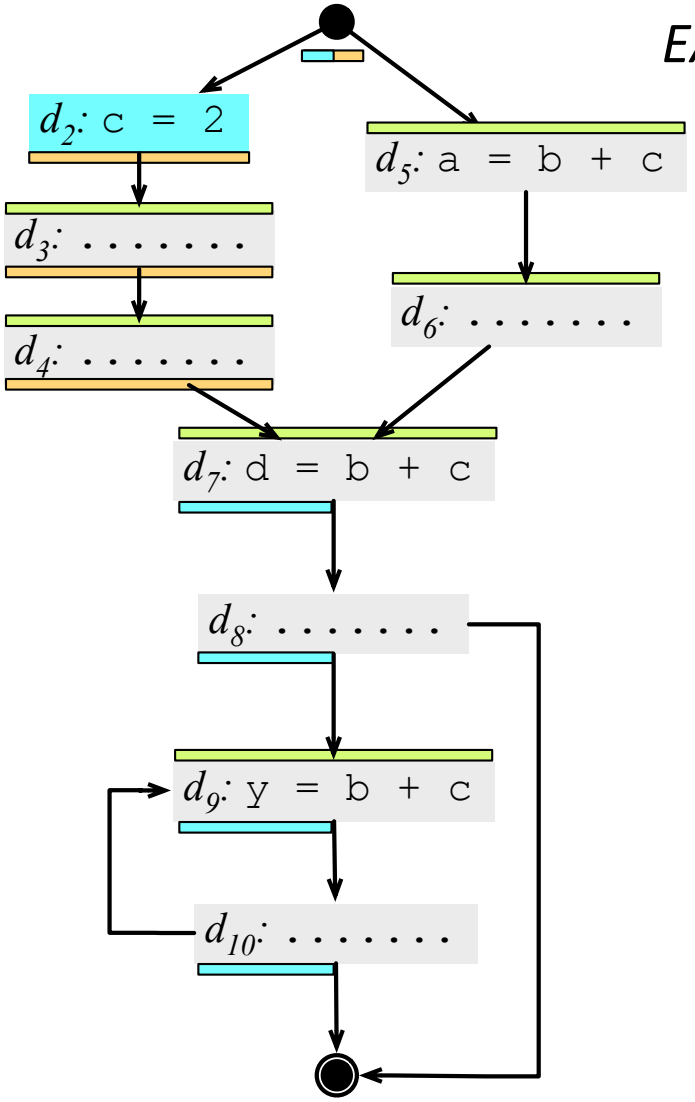
Earliest Placement Points



$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

Which are the basic blocks that do not have b + c in their $OUT_{ANTICIPABLE}$ sets, or that kill b + c?

Earliest Placement Points

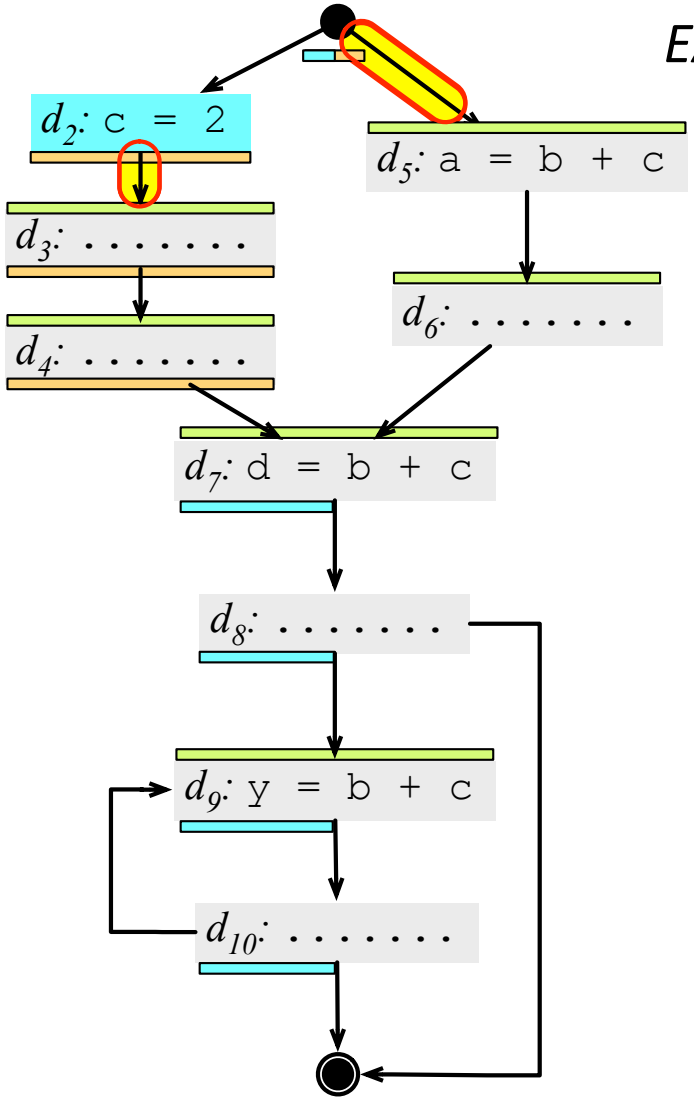


$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)}$$

$$\cap \; (\mathbf{KILL(i)} \cup \overline{\mathbf{OUT_{ANTICIPABLE}(i)}})$$

So, which are the earliest placement edges for b + c?

Earliest Placement Points



$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

- The earliest placement points represent the places where we can push the computation of the expression up, without causing further redundancies.
- But now we want to push these points down, so that we can, for instance, reduce register pressure.

Latest Placement

$$IN_{LATER}(j) = \cap_{i \in pred(j)} LATER(i, j)$$

$$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$$

- *LATER(i, j)* is true if we can move the computation of the expression down the edge *ij*.
- An expression *e* is in *EXPR(i)* if *e* is computed at *i*.
- This predicate is also computed for edges, although we have IN_{LATER} being computed for nodes.

Latest Placement

$$IN_{LATER}(j) = \cap_{i \in pred(j)} LATER(i, j)$$

$$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$$

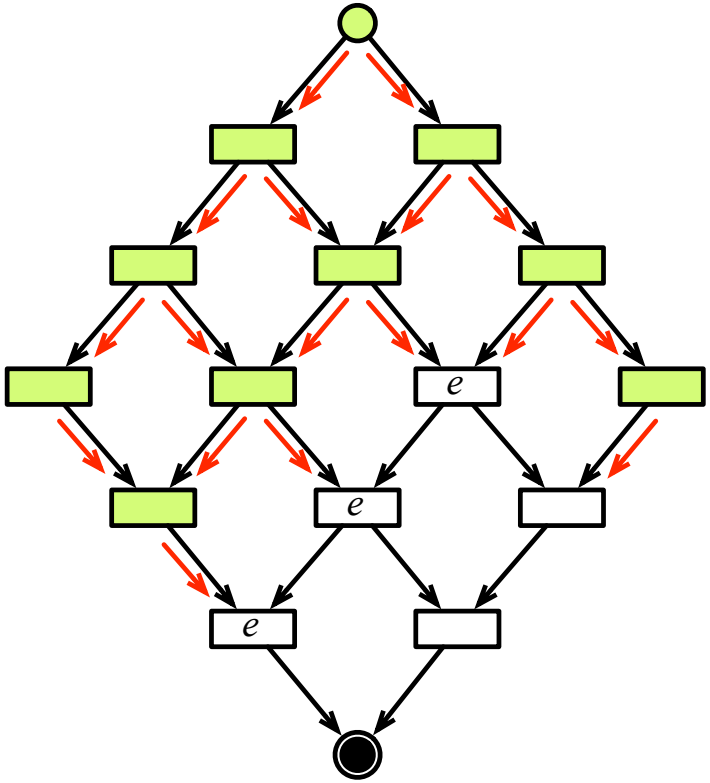
- If *EARLIEST(i, j)* is true, then *LATER(i, j)* is also true, as we can move the computation of *e* to edge *ij* without causing redundant computations.
- If $IN_{LATER}(i)$ is true, and the expression is not used at *i*, then *LATER(i, j)* is true.
  - If the expression is used at *i*, then there is no point in computing it at *ij*, because it will be recomputed at *i* anyway.

Latest Placement

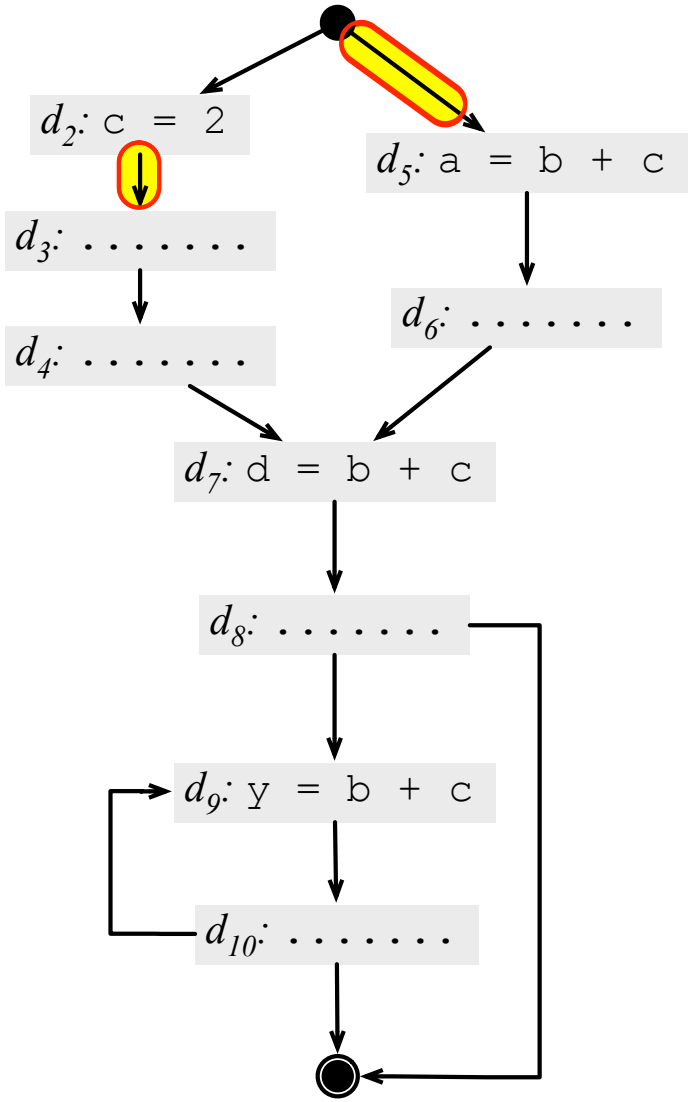
$$IN_{LATER}(j) = \cap_{i \in pred(i)} LATER(i, j)$$

$$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$$

- $IN_{LATER}(j)$ is a condition that we propagate down.
  - If all the predecessors of a node *j* accept the expression as non-redundant, then we can compute the expression down on *j*.



Earliest Placement Points

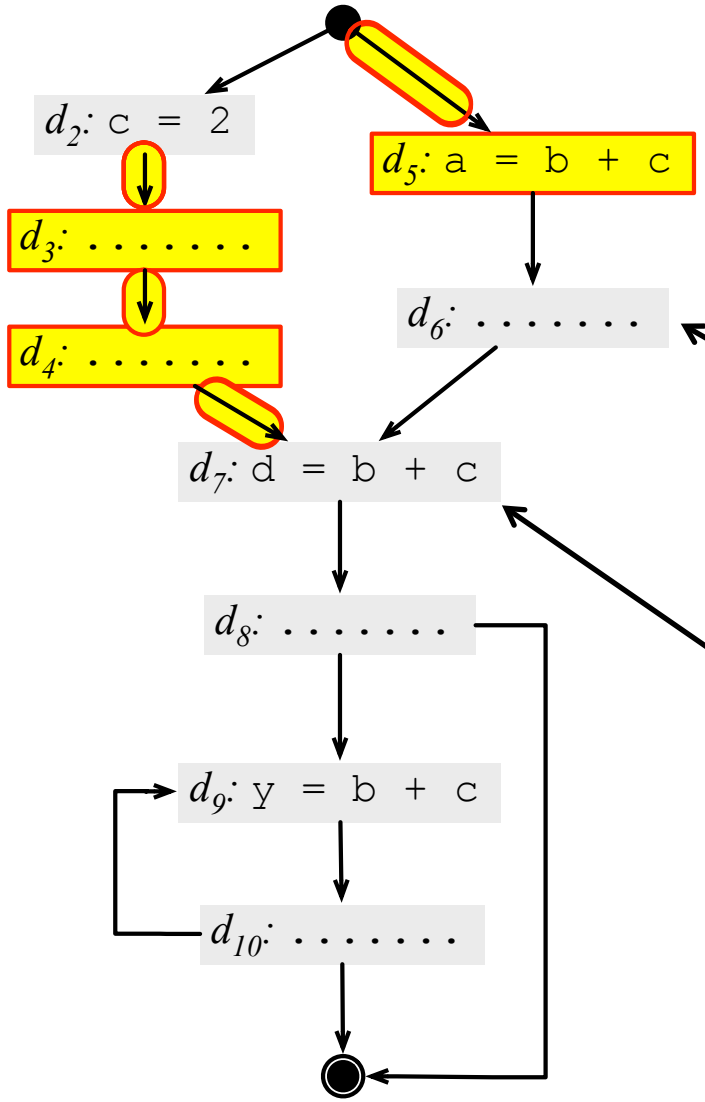


$IN_{LATER}(j) = \cap_{i \in pred(i)} LATER(i, j)$

$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$

Given these two earliest placement points, what are the latest placement edges and blocks?

Earliest Placement Points



$IN_{LATER}(j) = \cap_{i \in pred(i)} LATER(i, j)$

$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$

1) Why we do not have $LATER(d_5, d_6)$?

2) Why we do not have $LATER(d_7, d_8)$?

3) Considering this program, where should we compute b + c?

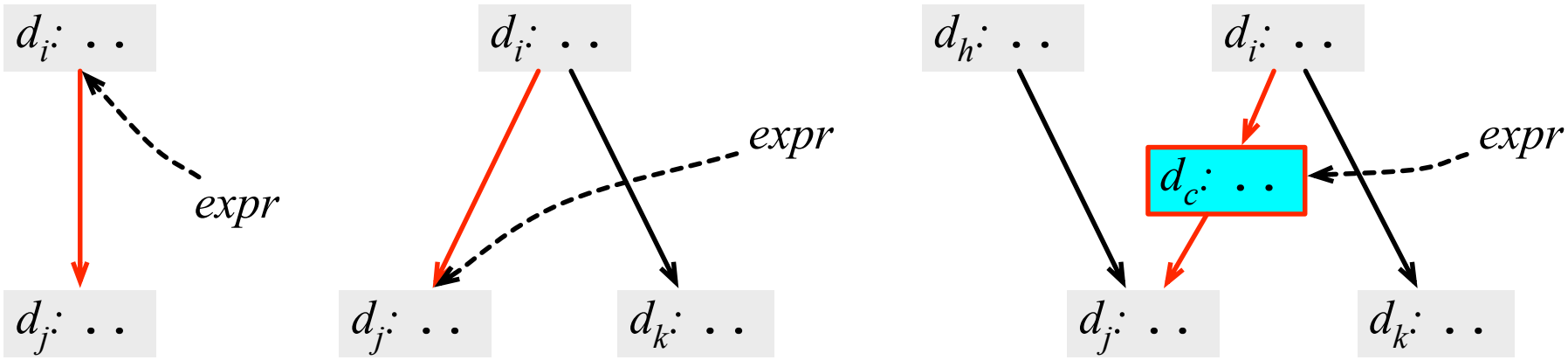
4) Can you come up with rules to compute the expressions?

Where to Insert Computations?

- We insert the new computations at the latest possible place.

$$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$$

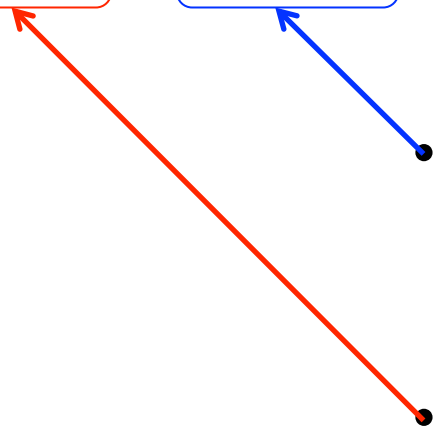
- There are different insertion points, depending on the structure of the CFG, if $x \in INSERT(i, j)$:



Which Computations to Remove?

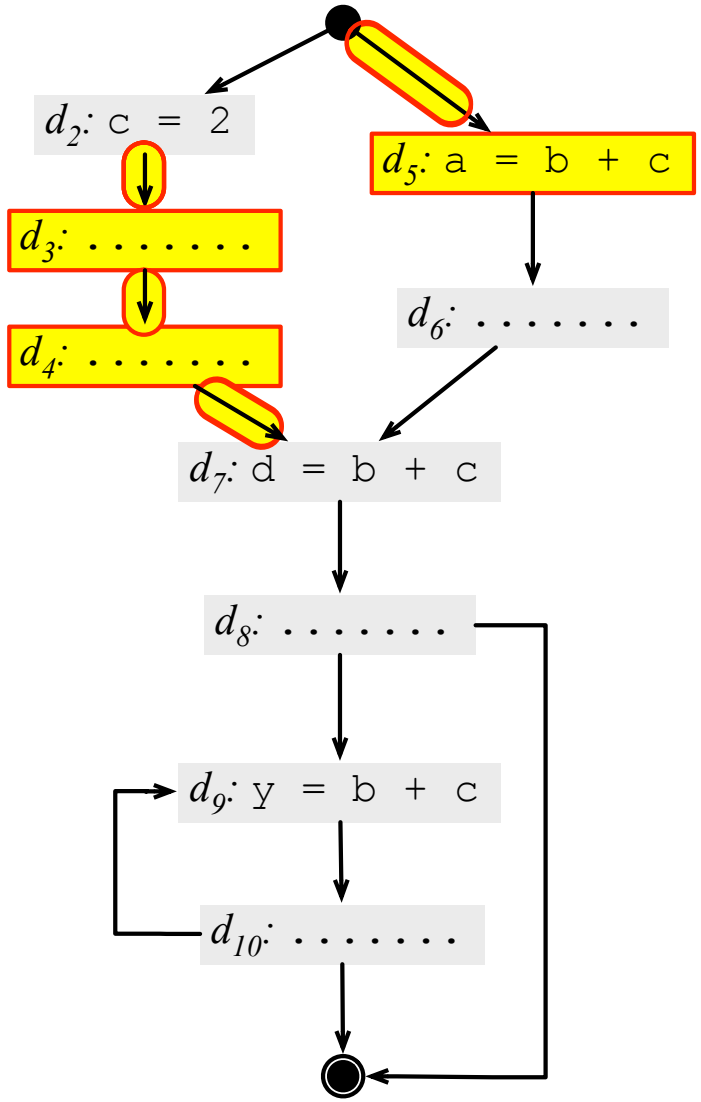
- We remove computations that are already covered by the latest points, and that we cannot use later on.

$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$



- The expression may not be a computation that is necessary later on.
- And, of course, the expression must be used in the block, otherwise we would have nothing to delete.

Modifying the CFG

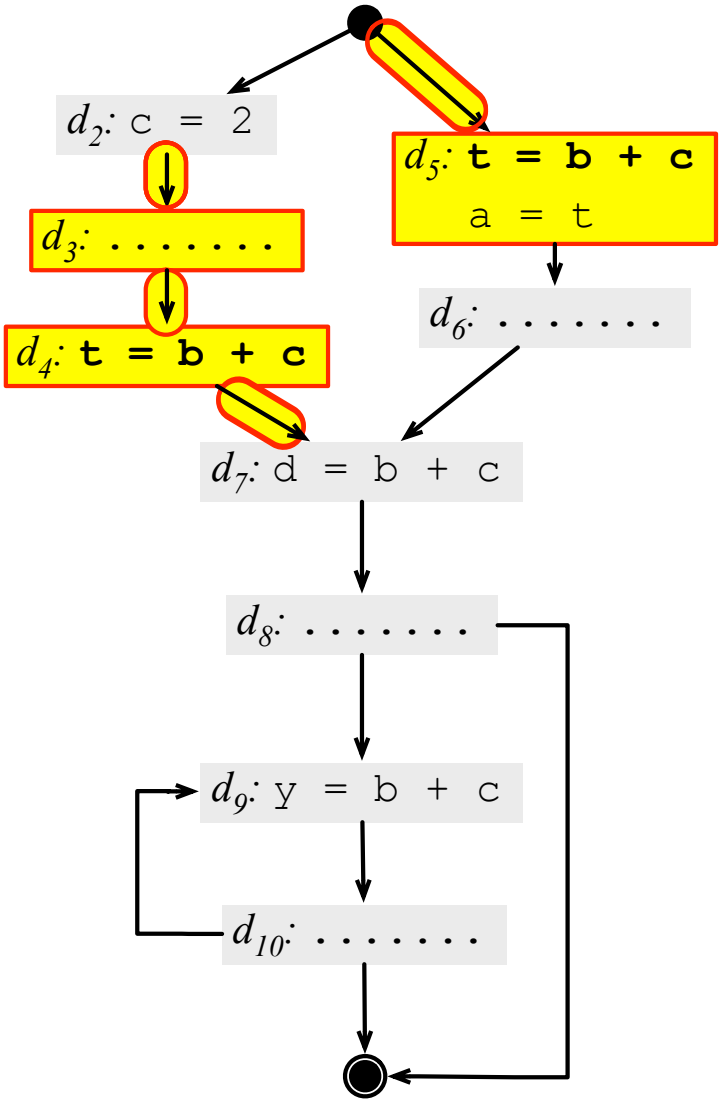


$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

$$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$$

Considering these latest placement points, which expressions should we insert?

Modifying the CFG



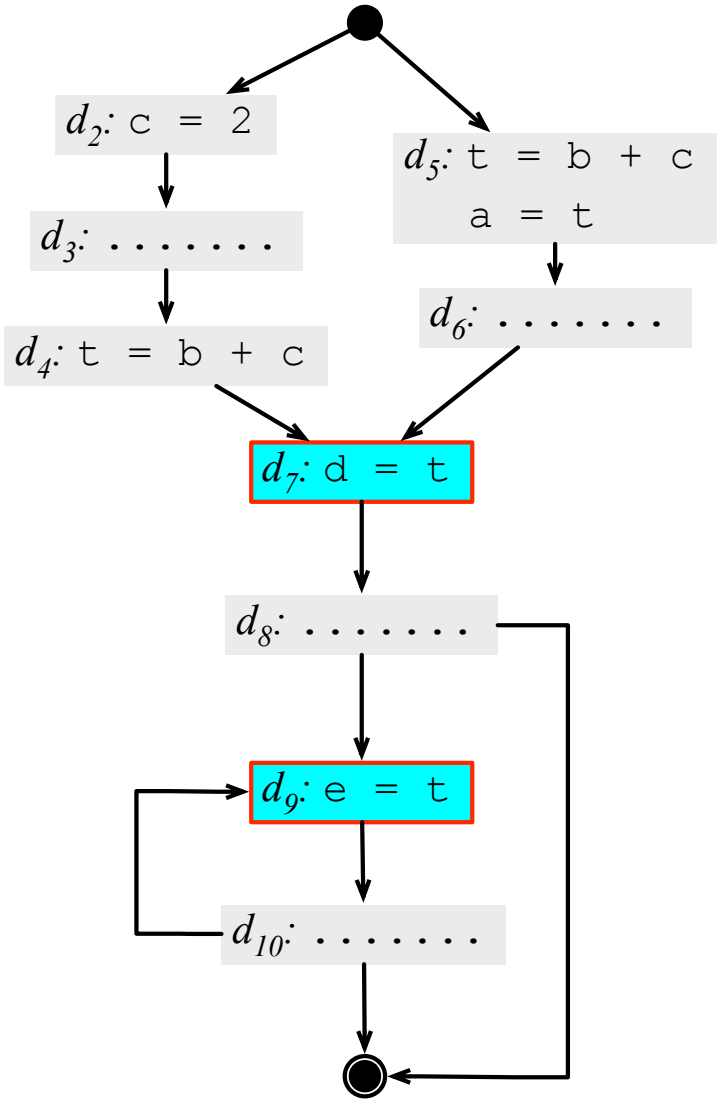
$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

$$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$$

- We need to find a new name for every computation of b + c.
  - In this example, we chose the name t, e.g., t = b + c

And which computations of b + c should we delete?

Modifying the CFG



$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

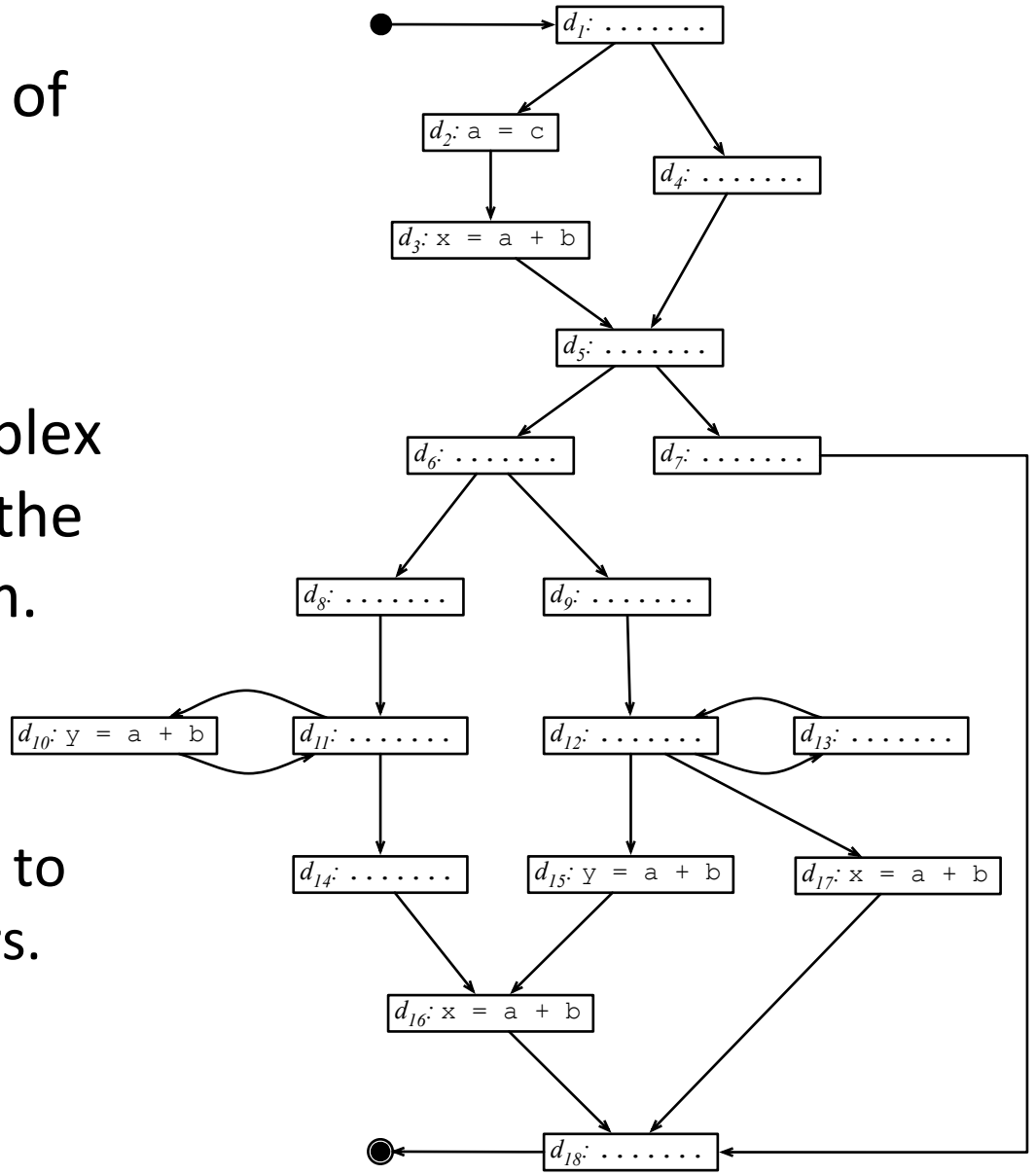
$$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$$

A FINAL EXAMPLE

DCC 888

An Example to Conquer them All

- The original formulation of Lazy Code Motion was published in a paper by Knoop *et al*[◇].
- The authors used a complex example to illustrate all the phases of their algorithm.
- Many papers are build around examples.
  - That is a good strategy to convey ideas to readers.

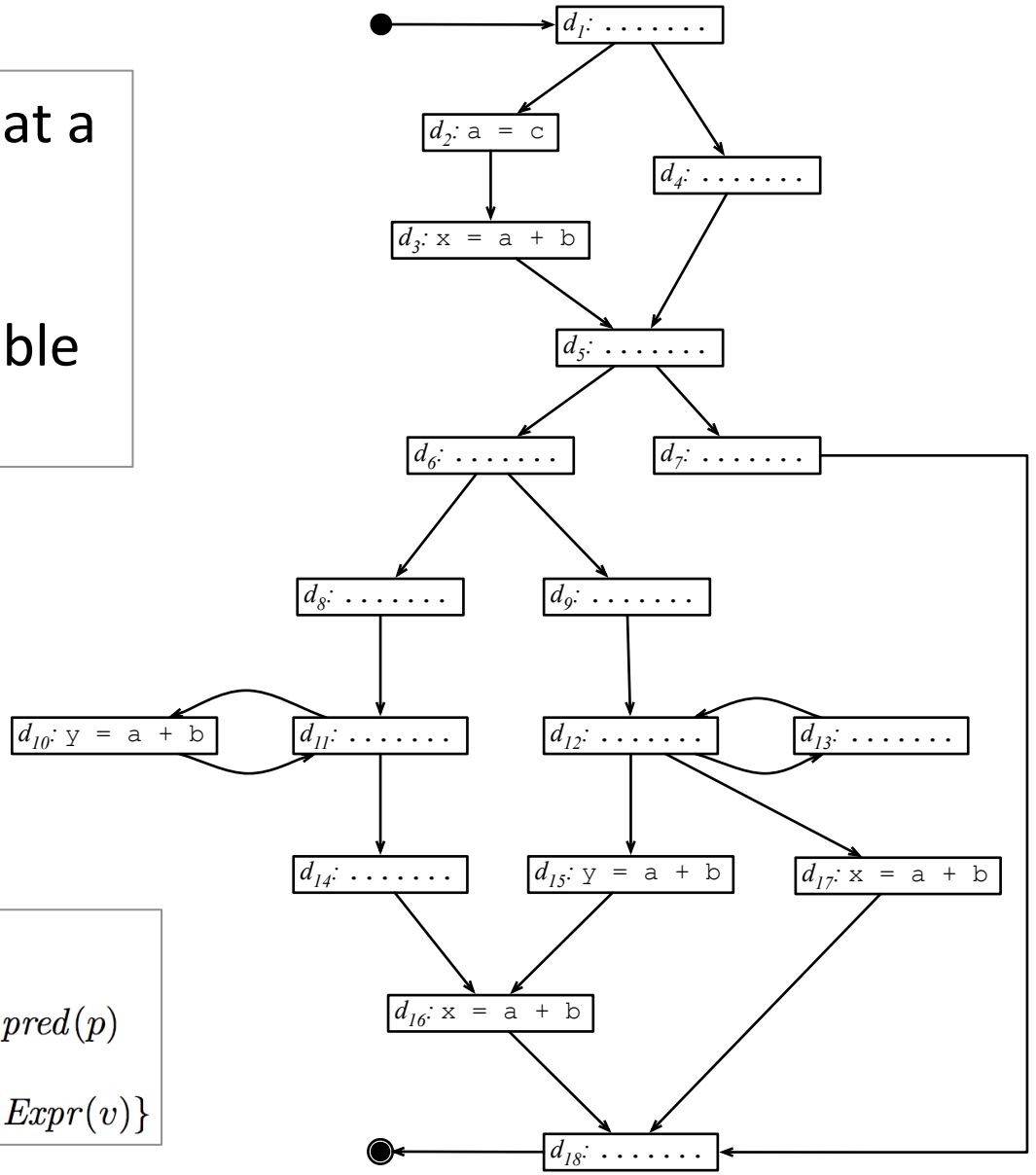


◇: Lazy Code Motion, PLDI (1992)

Available Expressions

An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.

What is the OUT set of available expressions in the example?



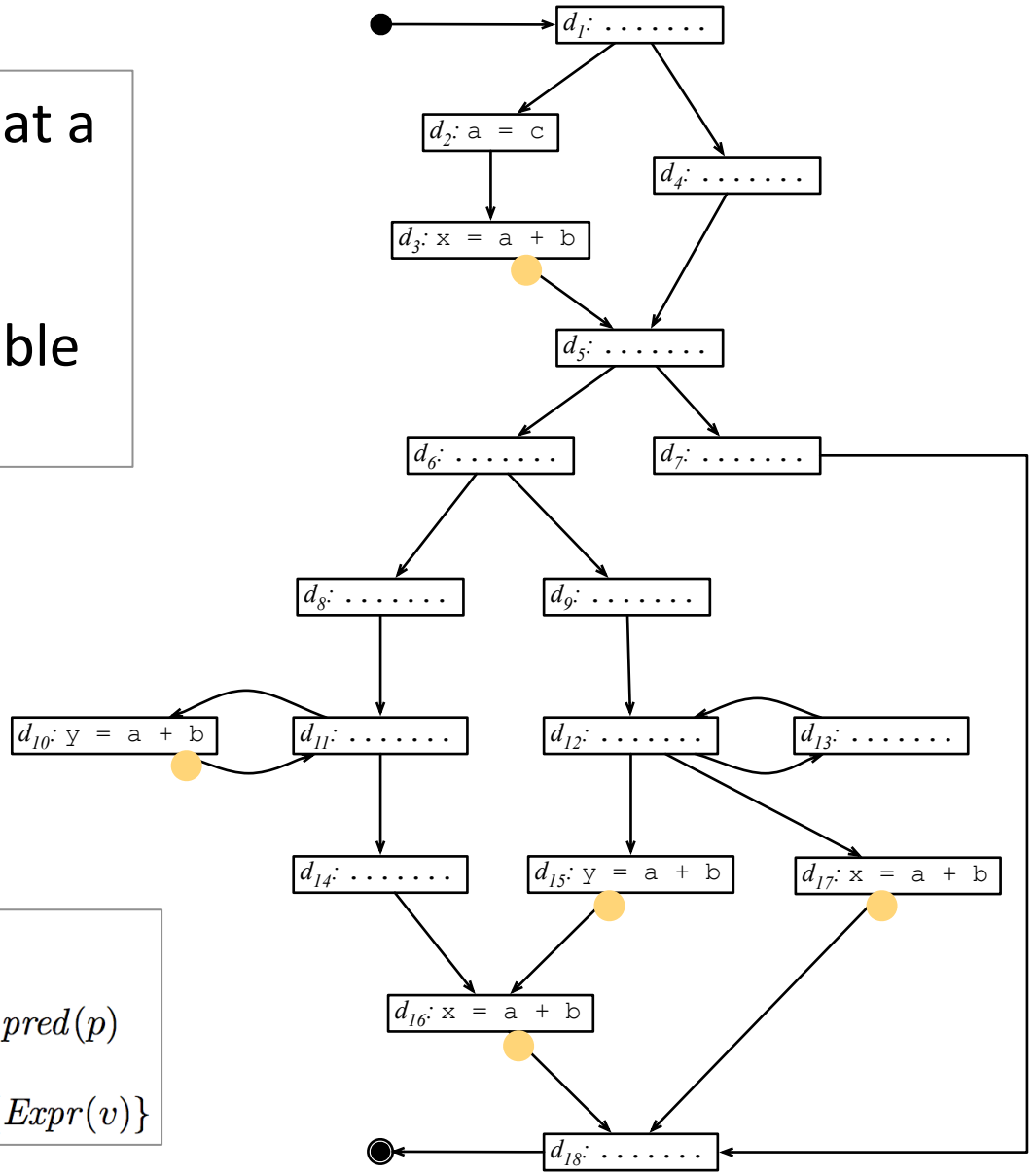
$$p : v = E$$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$

Available Expressions

An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.



$p : v = E$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$

Anticipable Expressions

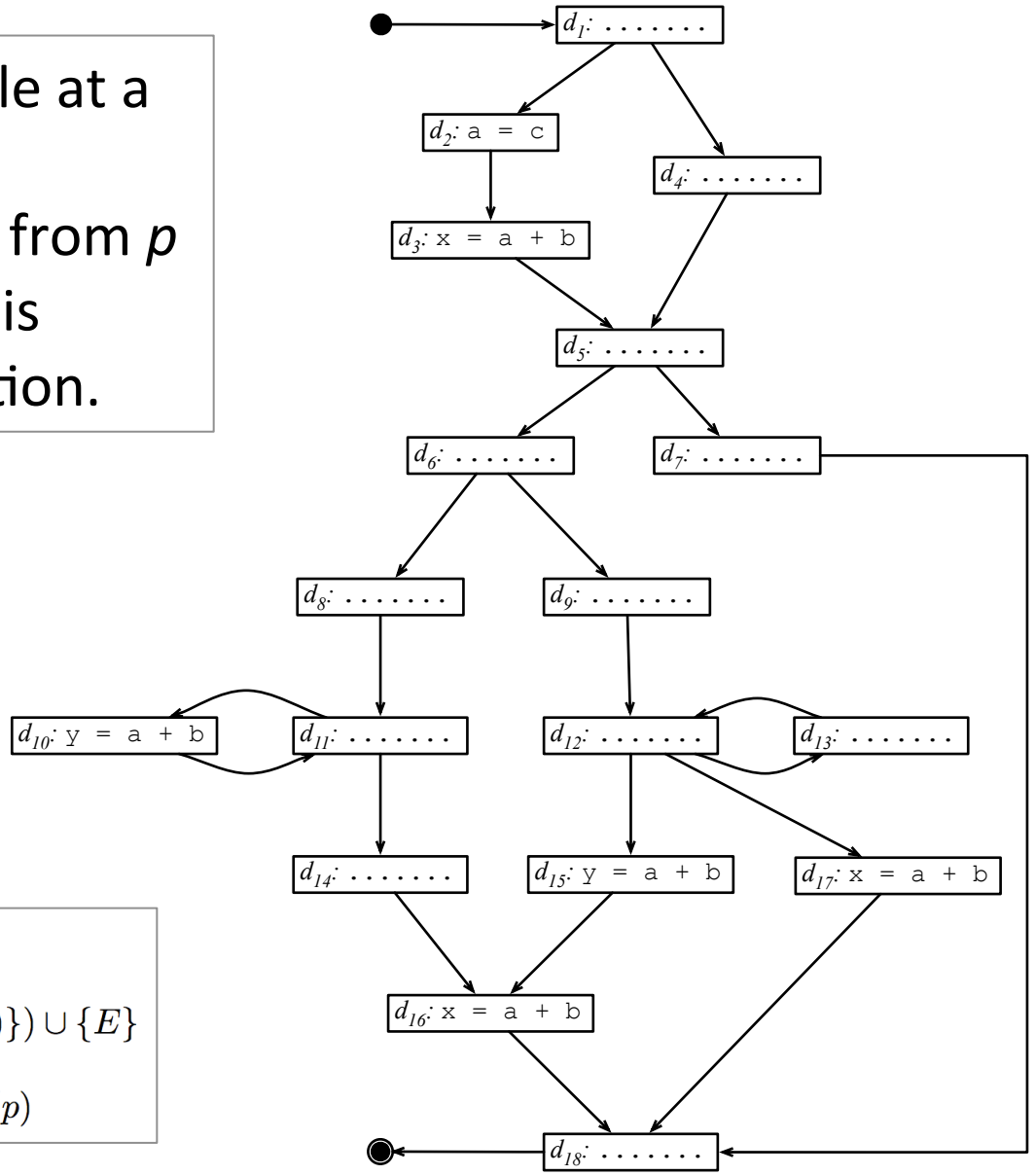
An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined until its computation.

What is the IN set of anticipable expressions in the example?

$p : v = E$

$$IN(p) = (OUT(p) \setminus \{Expr(v)\}) \cup \{E\}$$

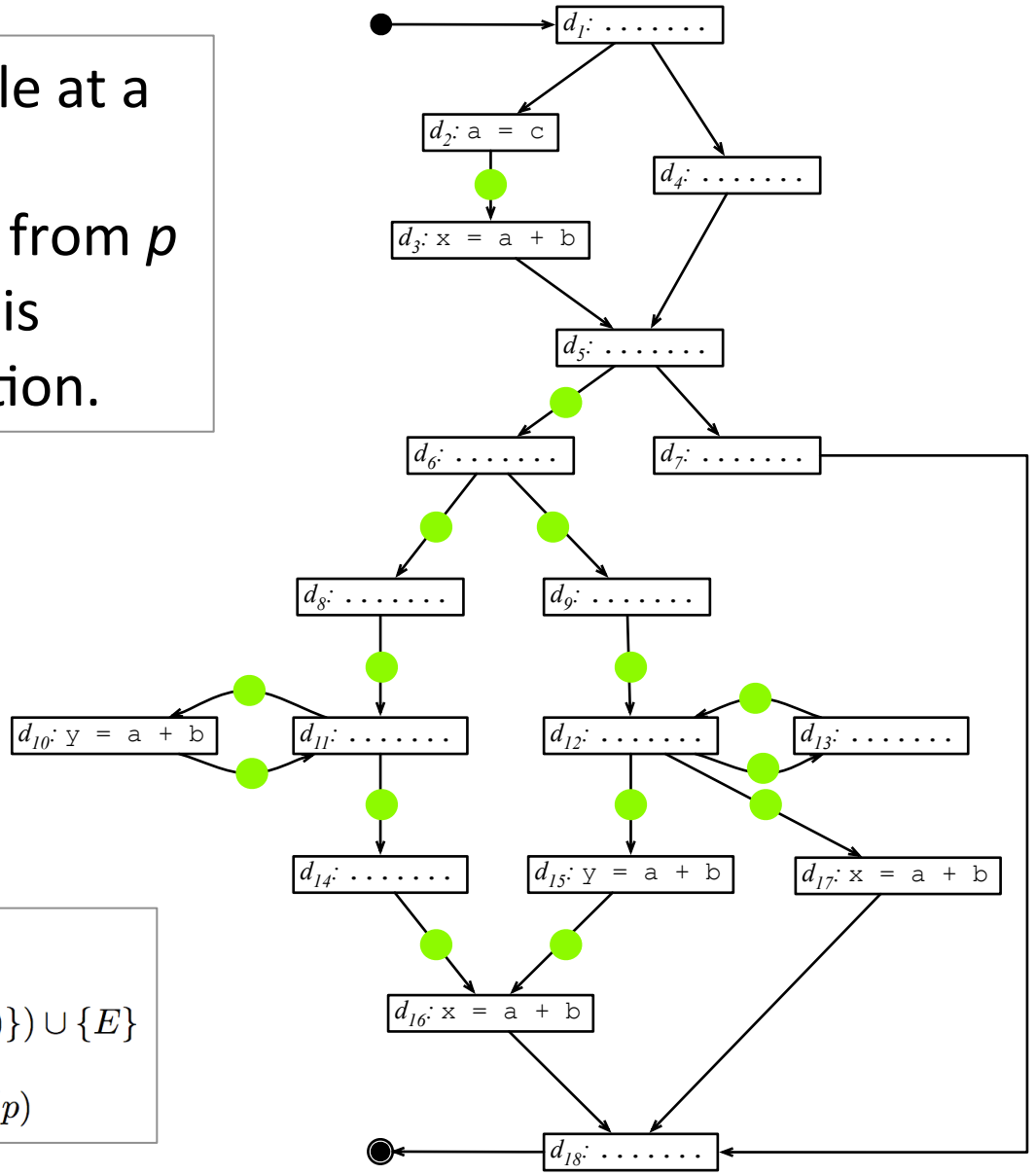
$$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$$



Anticipable Expressions

An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined until its computation.

What is the IN set of anticipable expressions in the example?



$p : v = E$

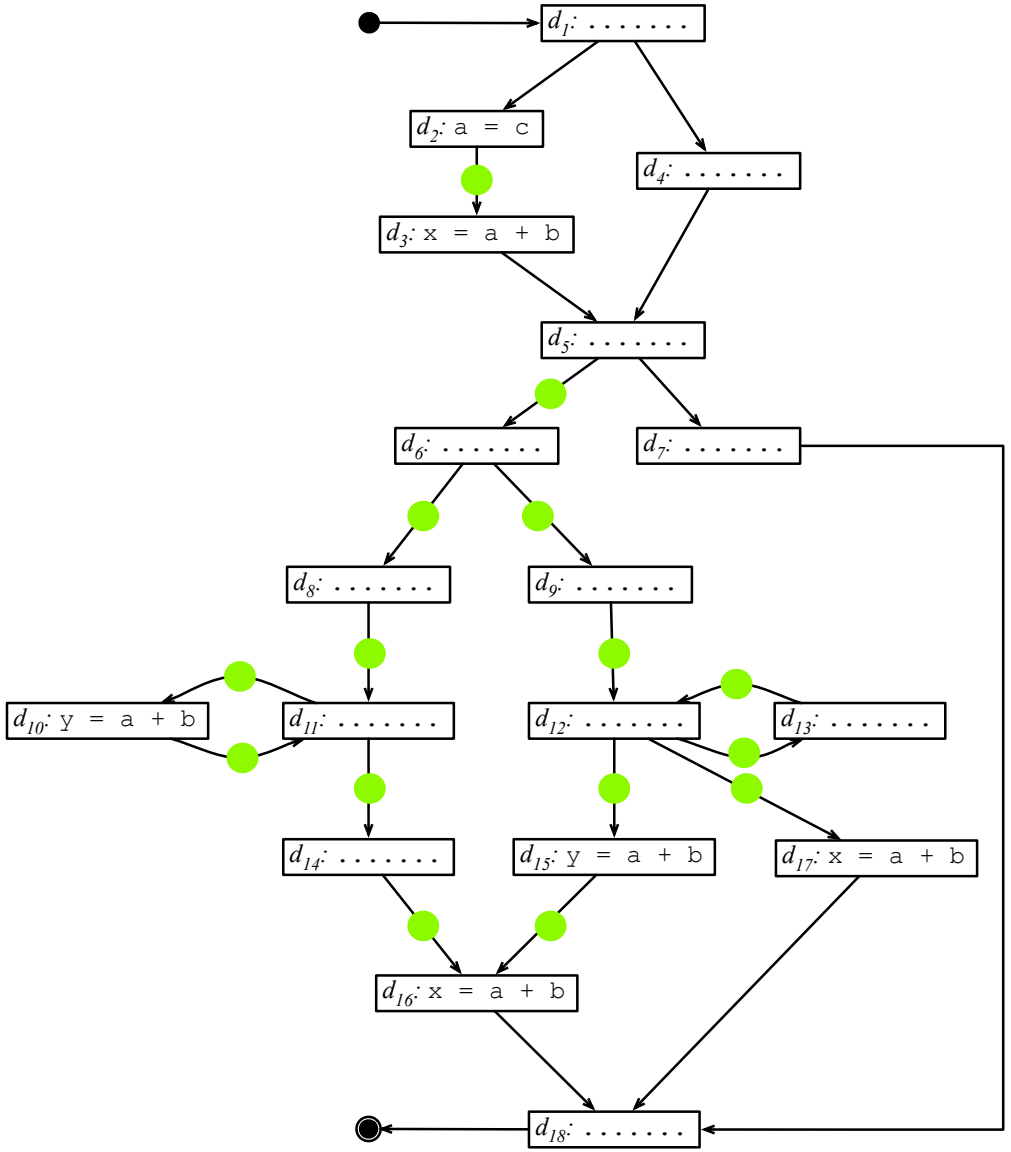
$$IN(p) = (OUT(p) \setminus \{Expr(v)\}) \cup \{E\}$$

$$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$$

$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

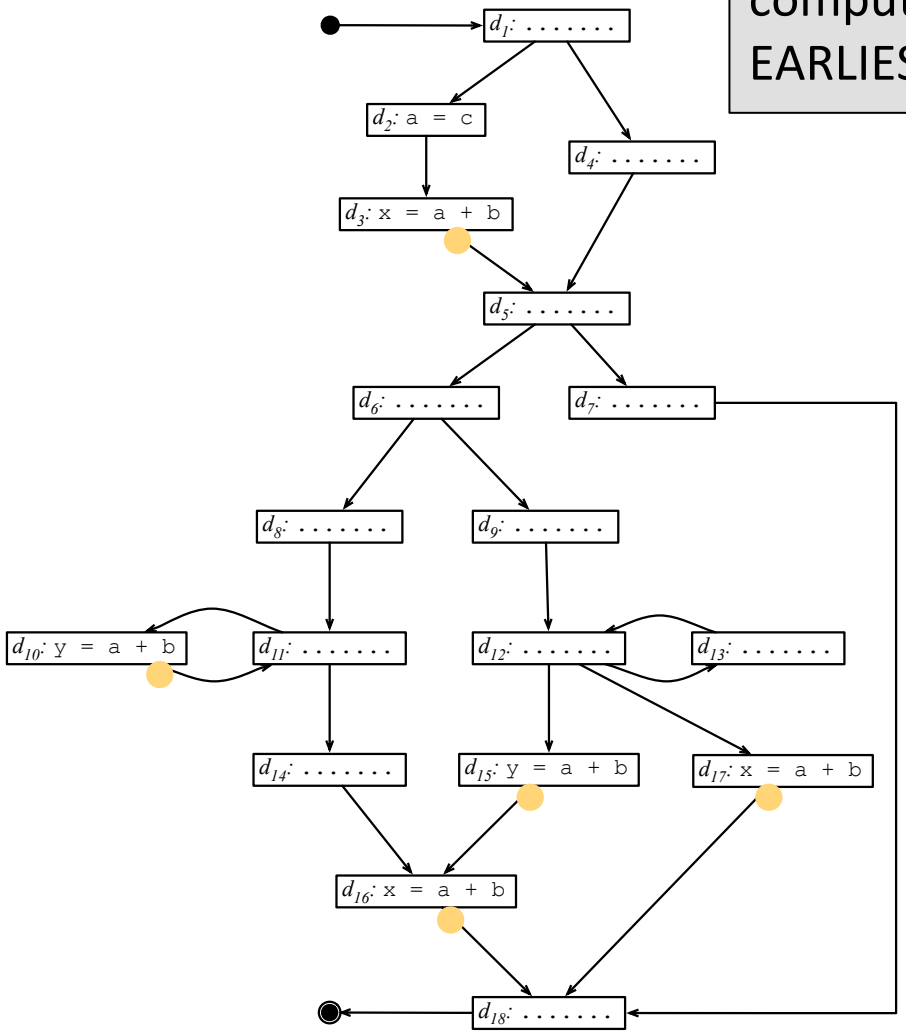
What is KILL(i) $\cup \overline{OUT_{ANTICIPABLE}(i)}$ in our running example?

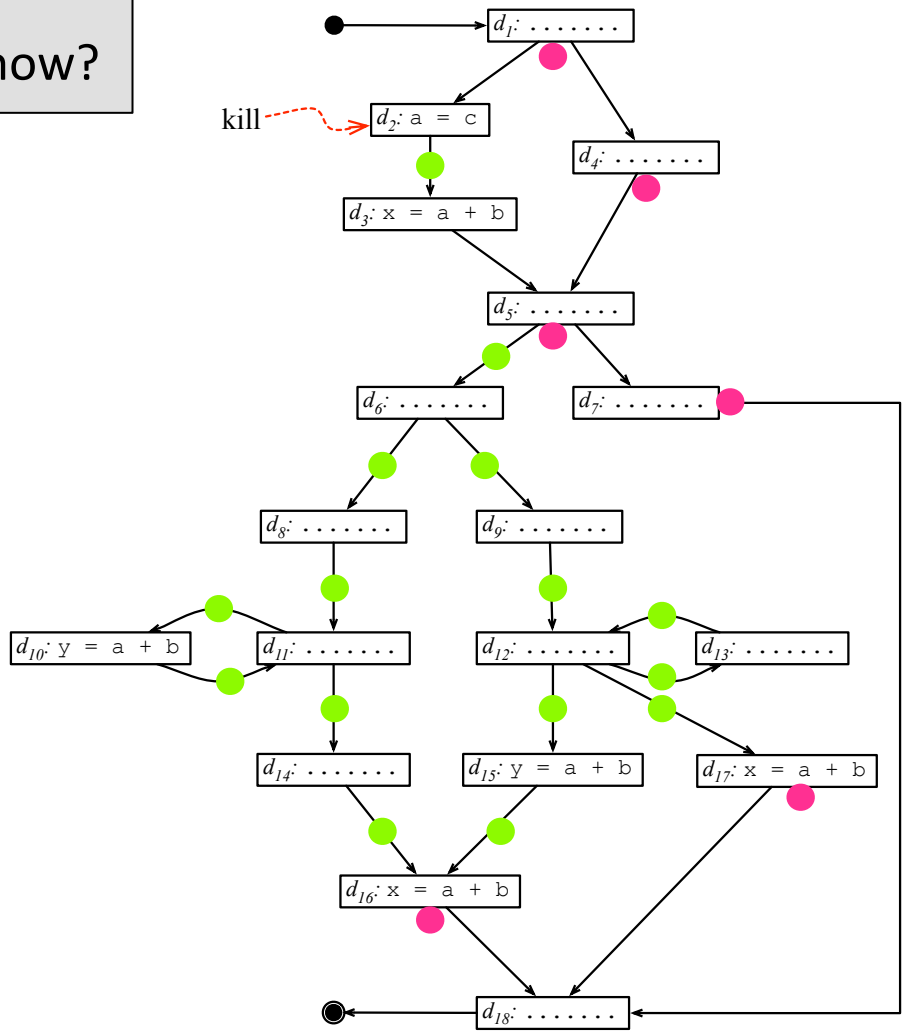
This figure shows the IN sets of anticipability analysis.



$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

Can you compute EARLIEST now?

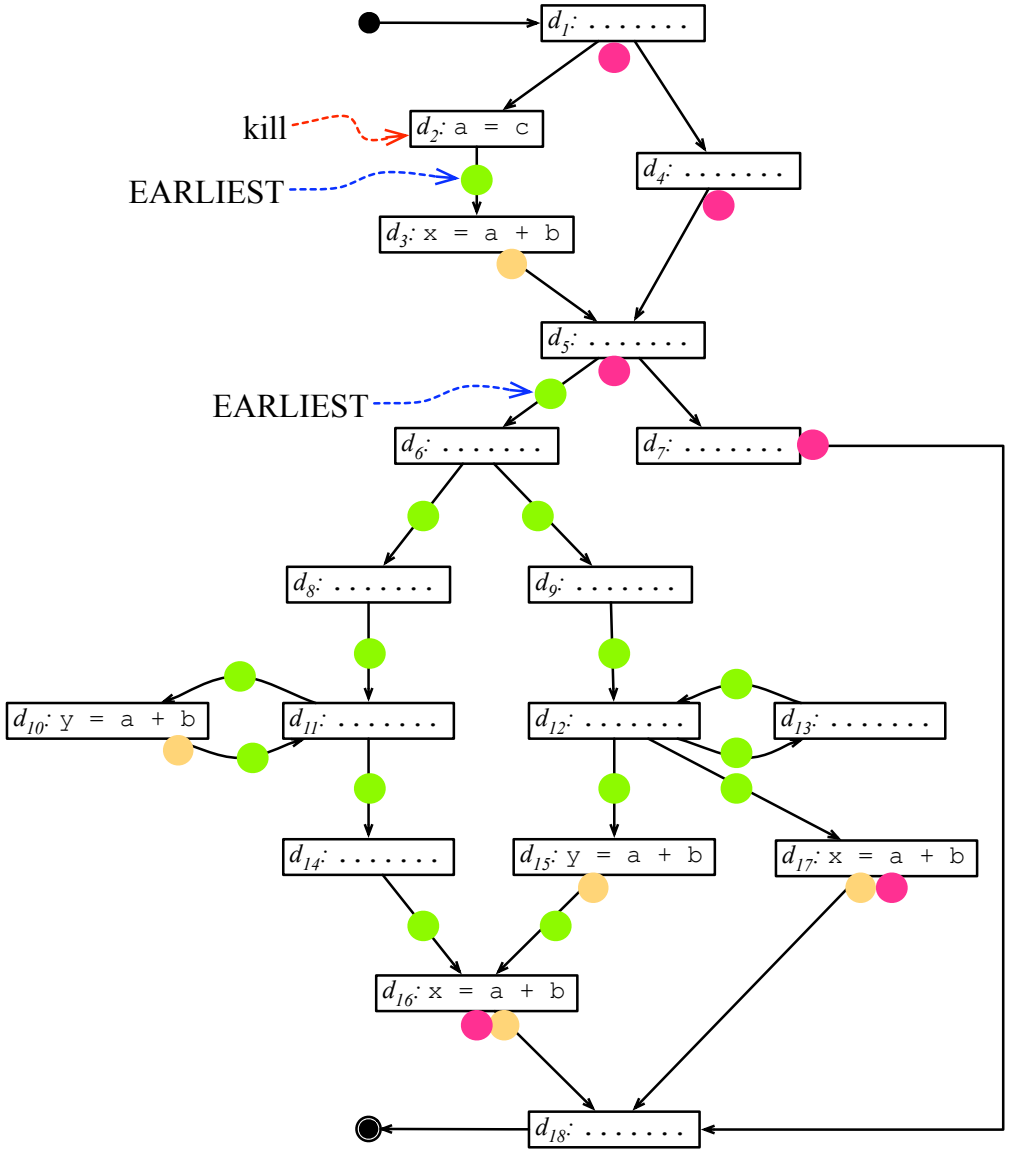




$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

We have two EARLIEST edges in this CFG.

- Anticipable at IN(i)
- Not anticipable at OUT(i)
- Available at OUT(i)

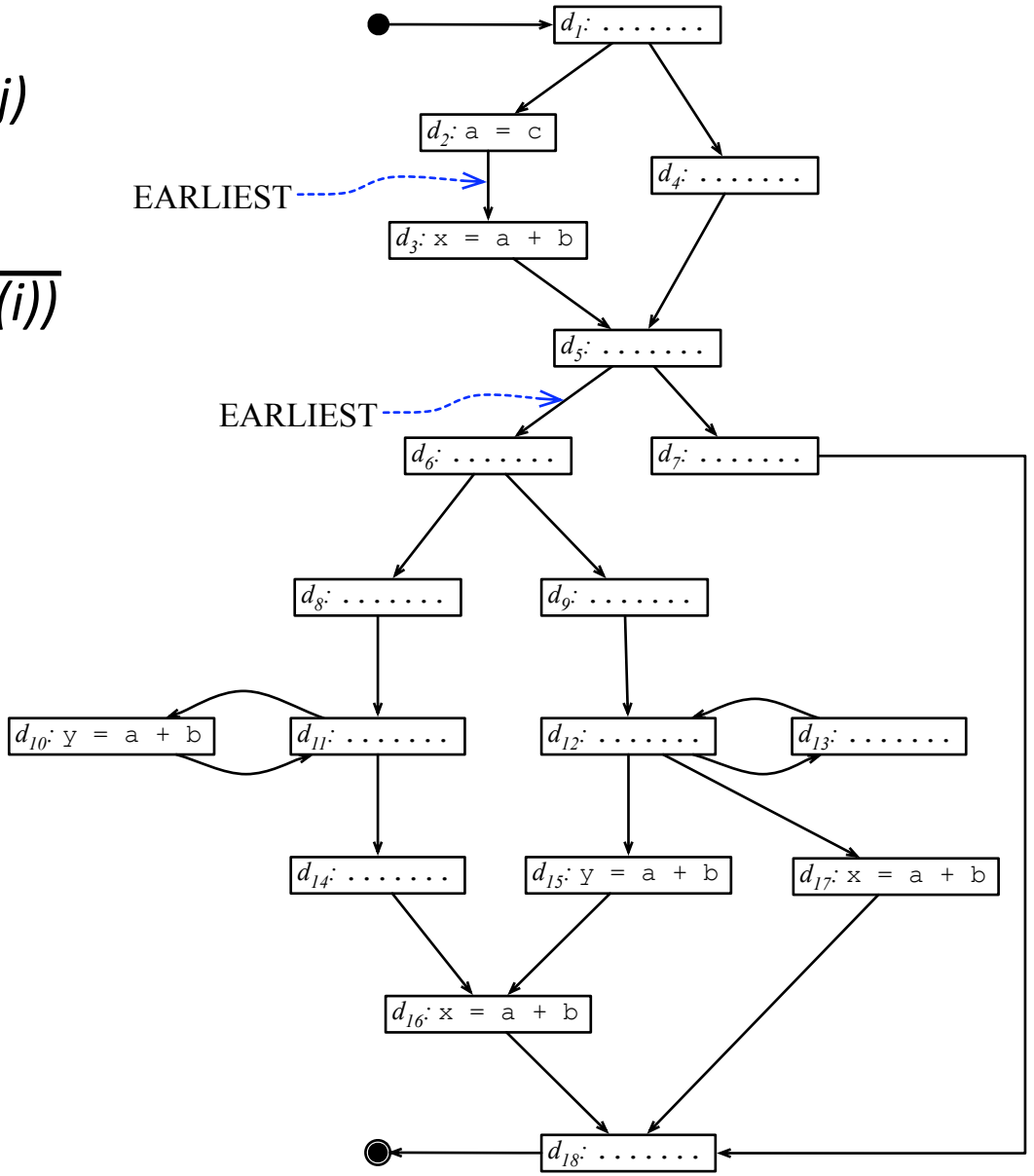


Latest

$$IN_{LATER}(j) = \cap_{i \in pred(j)} LATER(i, j)$$

$$LATER(i, j) = EARLIEST(i, j) \cup \overline{(IN_{LATER}(i) \cap \overline{EXPR(i)})}$$

The goal now is to compute the latest IN sets, and the latest edges. Can you do it?



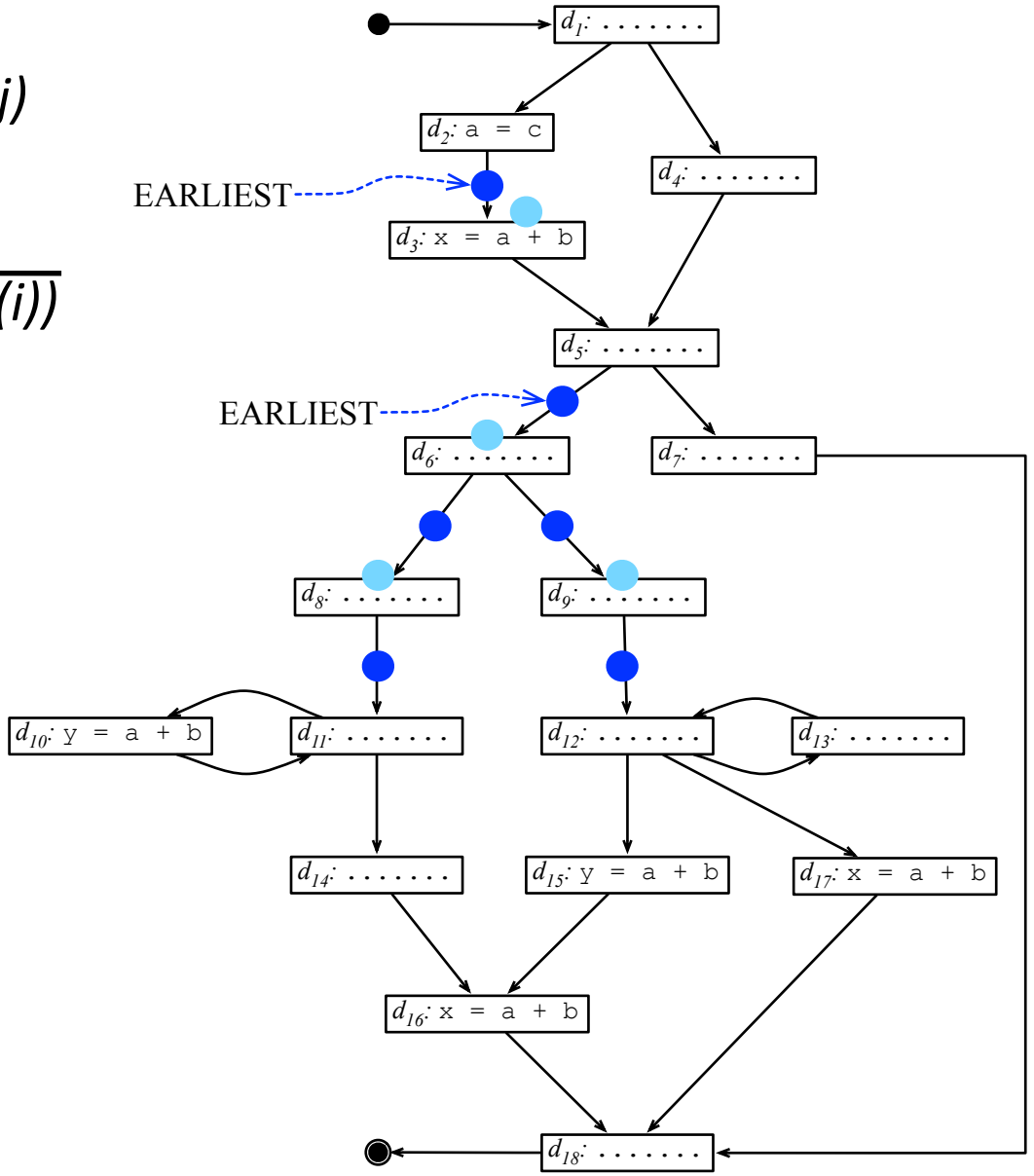
Latest

$$IN_{LATER}(j) = \cap_{i \in pred(j)} LATER(i, j)$$

$$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$$

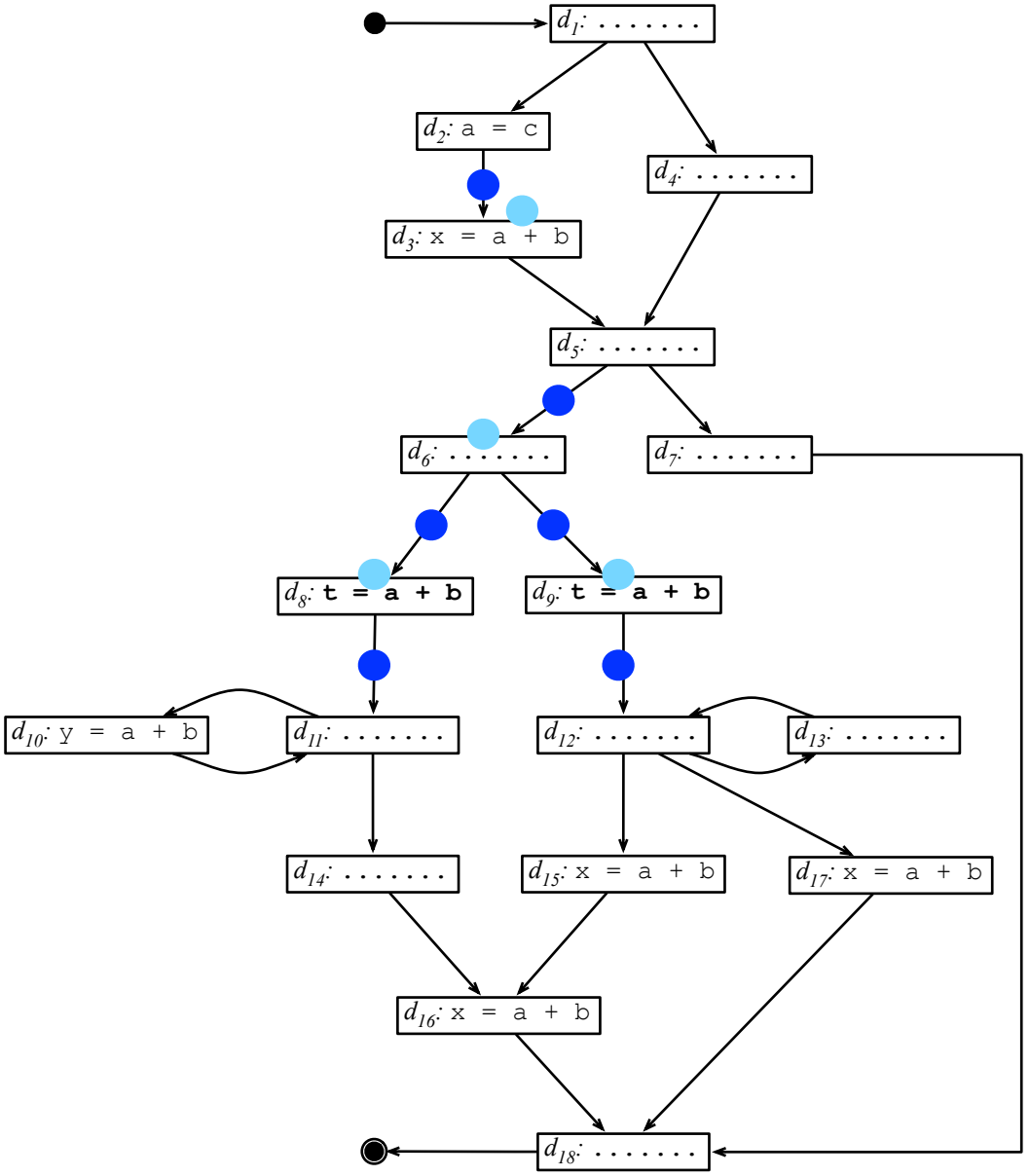
The LATEST edges are marked with the dark blue circles.

The LATEST IN sets are marked with the light blue circles.



$$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$$

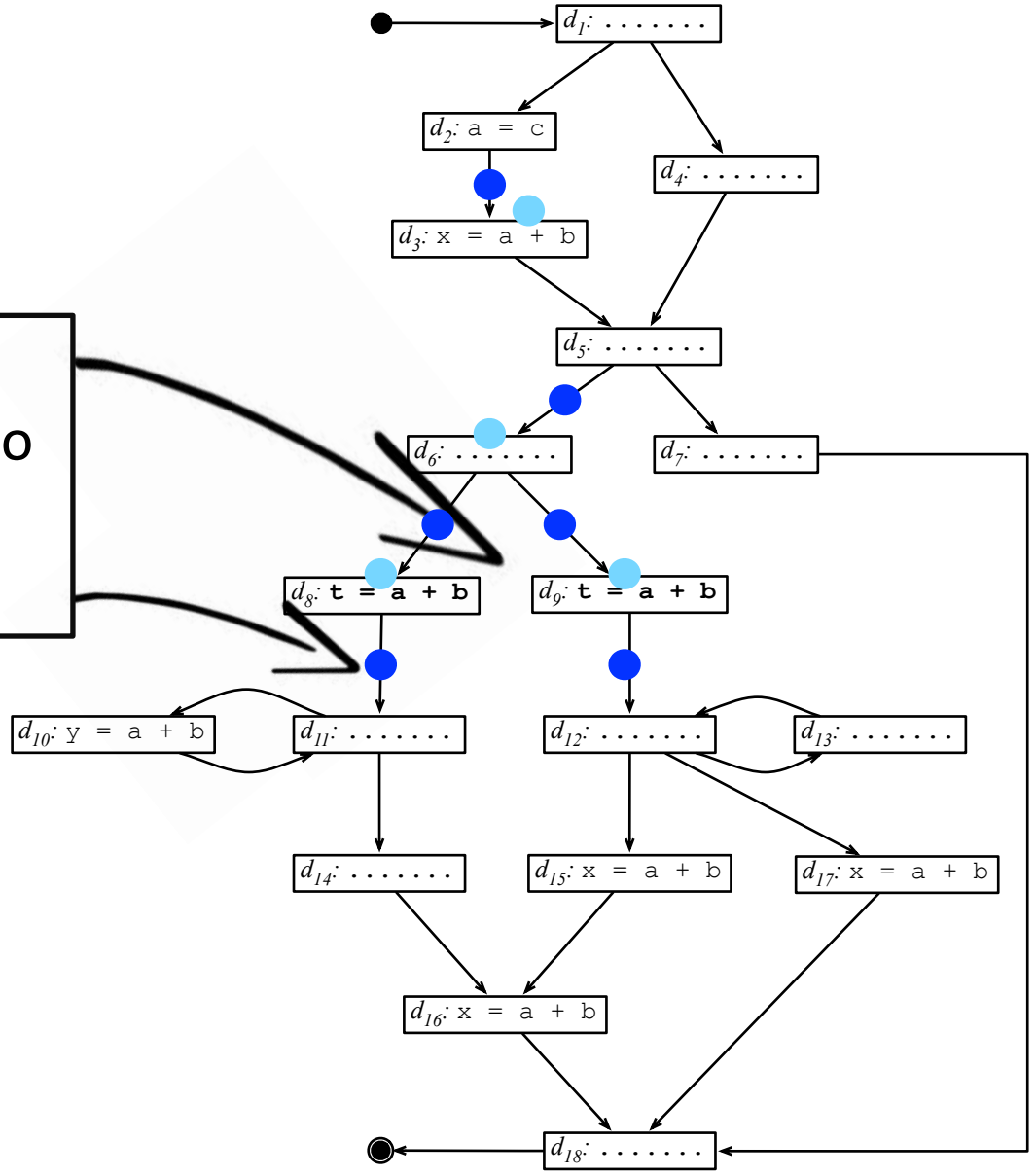
We must now find the sites where we can insert new computations of a + b. Can you compute INSERT(i, j)?



$$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$$

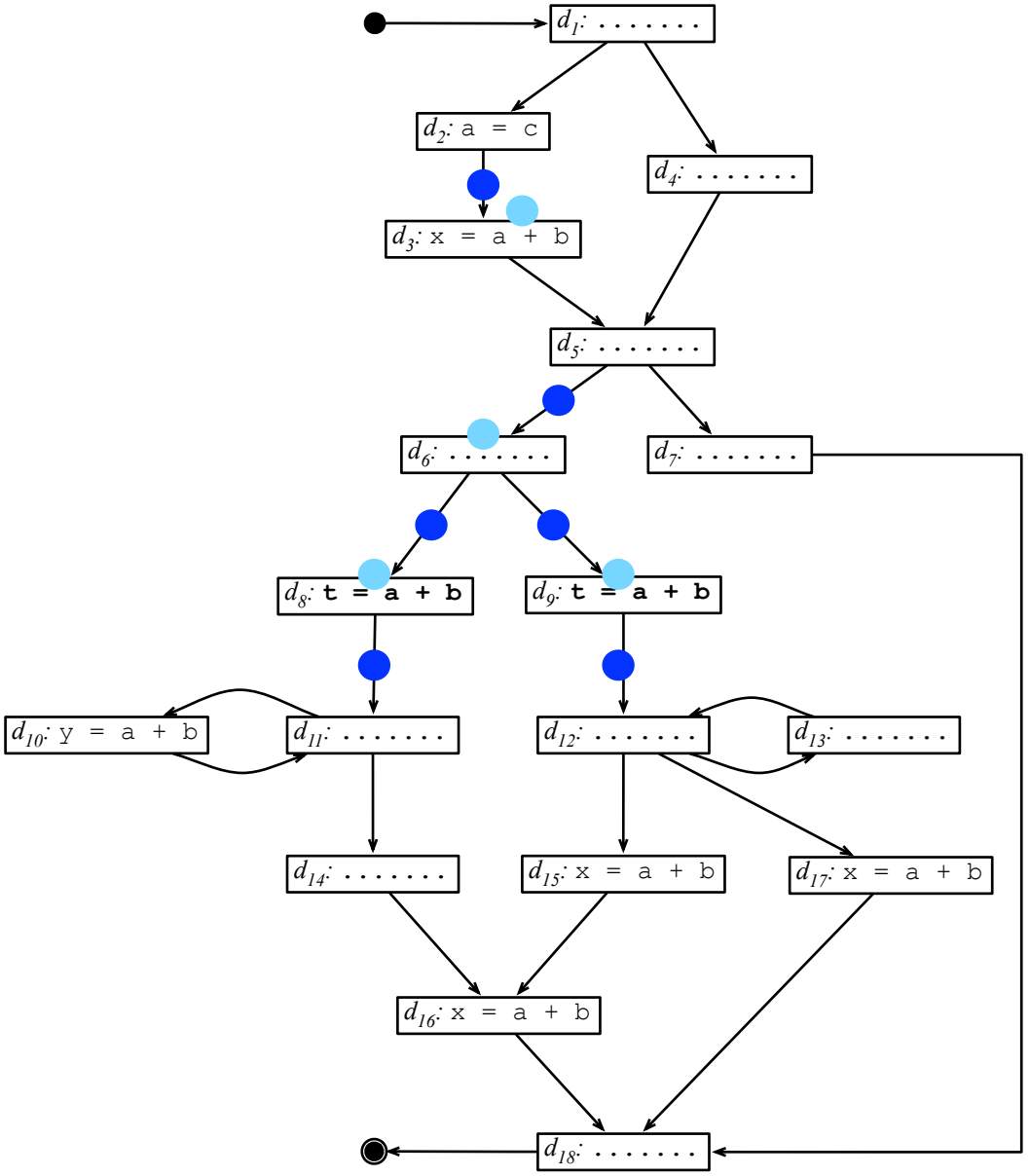
In this example, we only have two sites to insert new computations.

Do you remember why we insert the computation of a + b at d_8, instead of d_{11}?



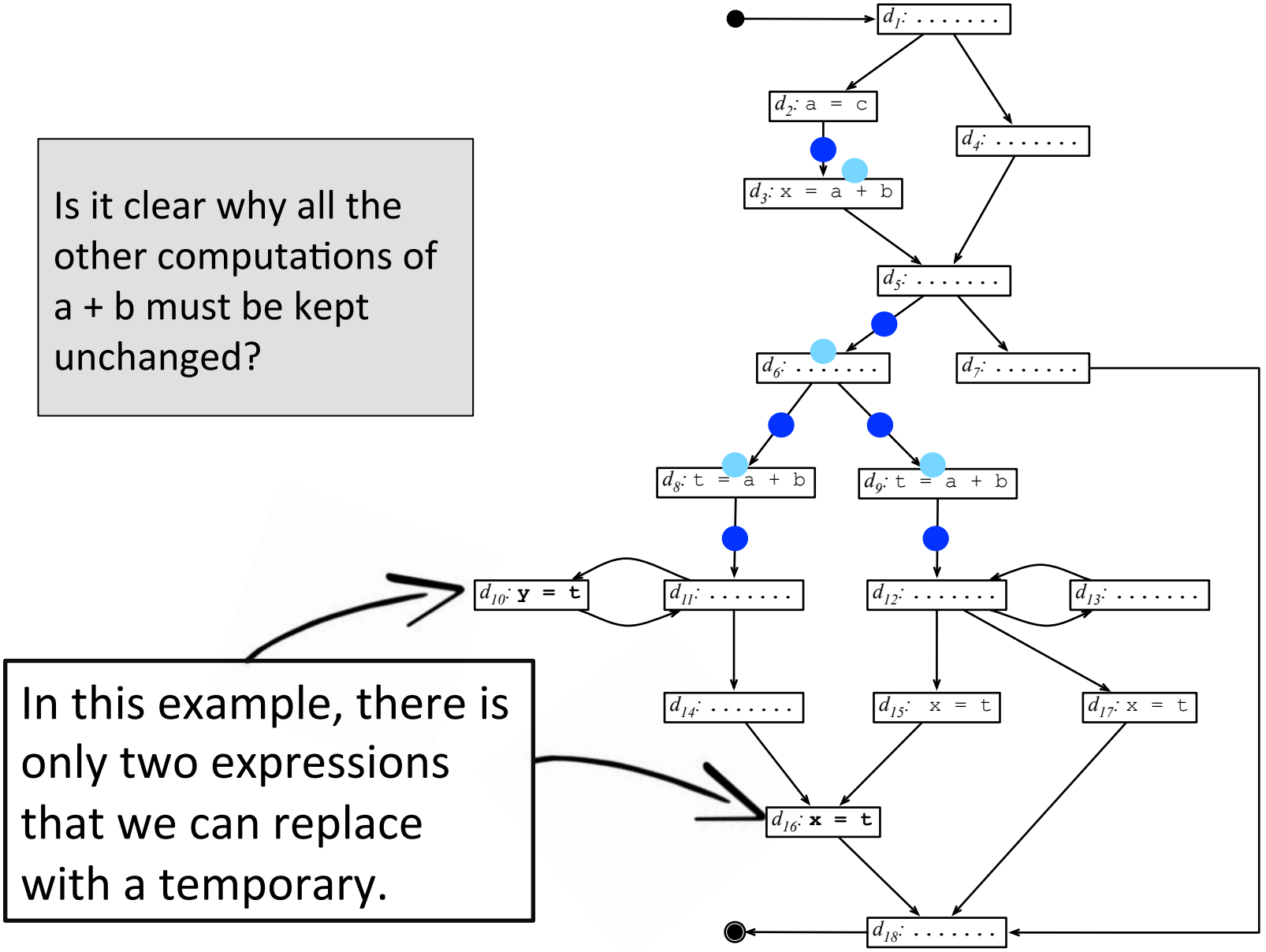
$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

We must now delete redundant computations that exist at program points p. Can you determine DELETE(p)?



$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

Is it clear why all the other computations of a + b must be kept unchanged?



In this example, there is only two expressions that we can replace with a temporary.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

A Bit of History

- Partial Redundancy Elimination is a classical example of code optimization.
- The idea of Lazy Code Motion was invented by Knoop et al.
- The equations in this presentation were taken from Drechler and Stadel, who simplified Knoop's work.

- Knoop, J., Ruthing, O. and Steffen, B. "Lazy Code Motion", ACM Conference on Programming Language Design and Implementation (1992), pp. 224-234
- Drechsler, K. and Stadel, M. "A Variation of Knoop, Ruthing, and Steffen's Lazy Code Motion", ACM SIGPLAN Notices, 28:5 (1993)