CS614: Advanced Compilers

Optimizations based on SSA

Manas Thakur
CSE, IIT Bombay

Spring 2025

In the last class

- Sparse constant propagation based on SSA form.
    - Simple sparse constant propagation (SSCP), faster than simple CP
    - Conditional sparse constant propagation (SCCP), faster than conditional CP
    - Precision better than flow-insensitive CP.

Optimization 2: Global Value Numbering (GVN)

Common sub-expression elimination

- **Idea:** If a program computes the same value multiple times, reuse the value.

```
a = b + c;
c = b + c;
d = b + c;
```

→

```
t = b + c;
a = t;
c = t;
d = b + c;
```

- How about the following code?

```
x = a + b;
y = a;
z = y + b;
```

- We need something more powerful that exact expression matching.

Value Numbering

- Each *non-trivial* (non-copy) computation is given a number, called its value number.
- Two expressions using the same operators, and operands with the same value numbers, must be equivalent.

```
x = a + b;
y = a;
z = y + b;
```

➡

```
v1 = a
v2 = b
x = v1 + v2
y = v1
z = v1 + v2
```

➡

```
v1 = a
v2 = b
x = v1 + v2
// replace y with v1
//   and z with x
```

- Common “value” elimination!
- Usually performed by *hashing* the expressions based on initial value numbers.

Extending value numbering beyond *basic blocks*

- There may be common expressions across different basic blocks.

- How to reconcile values produced on different control-flow paths?

- **Problem:** A simple assignment `x = y` does *not* imply that all references to `x` can be replaced by `y` after the assignment.

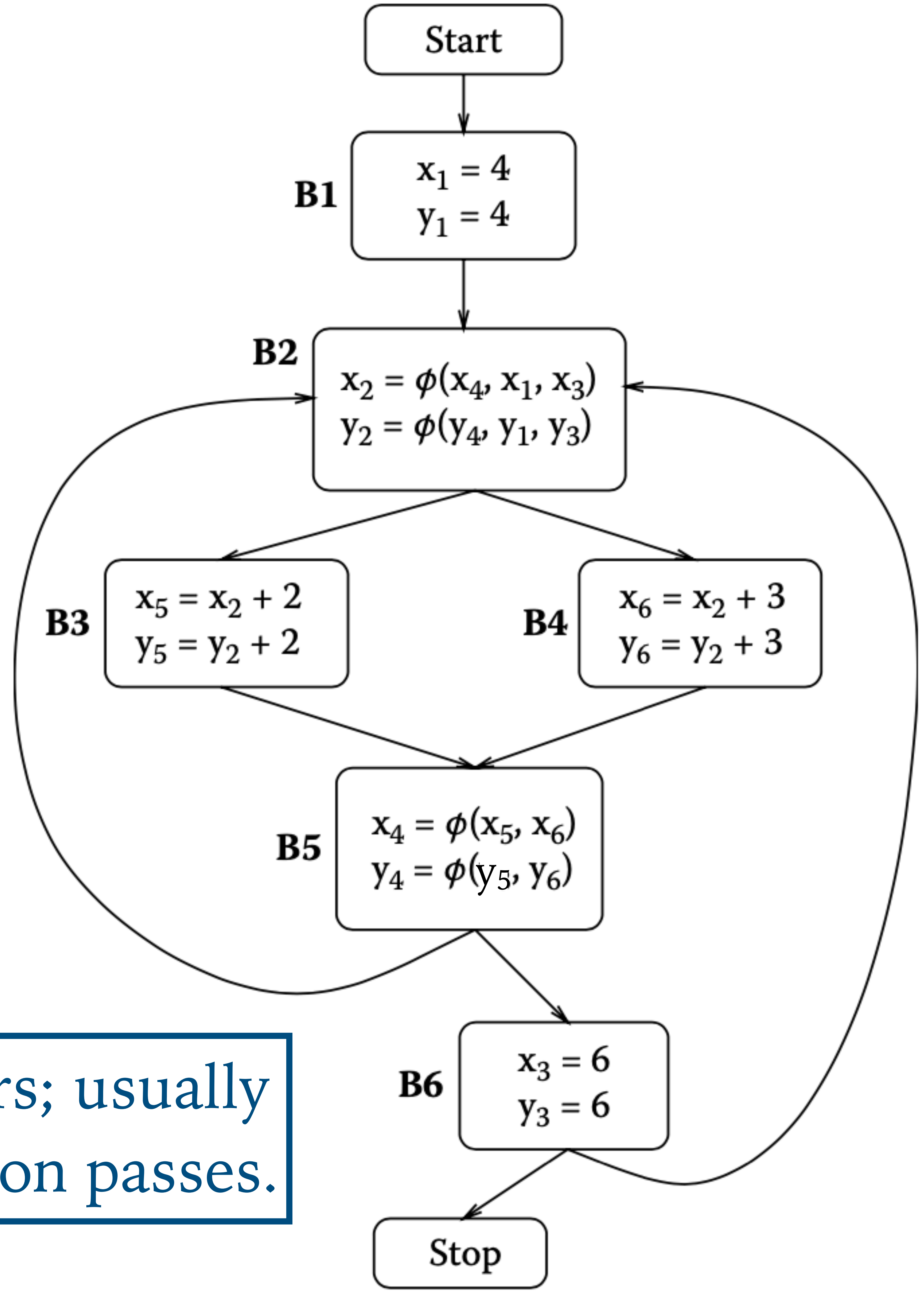
- Do we have a technique that already ensures the above property?
  - Convert the program to SSA form!

Global Value Numbering (GVN) over SSA

- In SSA form, an assignment is an assertion of *equivalence* throughout the analysis scope.
- **Step 1:** Partition all SSA variables by the *form* of the expression assigned to them.
  - That is:
    - `v1 = 2` and `w1 = a2 + 1` are always inequivalent.
    - `v3 = a1 + b2` and `w1 = d1 + e2` may *possibly* be equivalent.
- **Step 2:** If two expressions a_i op b_j and c_k op d_l are in the same partition, and a_i != c_k or b_j != d_l, then split the expressions to two different partitions.
- **Step 3:** Continue splitting until no more splits are possible.
- Expressions still in the same partition are equivalent and can be given the same value numbers!

Example: GVN

```
    x = 4
    y = 4
L1: if c goto L3
    x = x + 2
    y = y + 2
    goto L3
L2: x = x + 3
    y = y + 3
L3: if d goto L1
    x = 6
    y = 6
    if e goto L1
```



Initial partitions:

```
P1 = {x1,y1}
P2 = {x2,y2}
P3 = {x5,y5,x6,y6}
P4 = {x4,y4}
P5 = {x3,y3}
```

Final partitions:

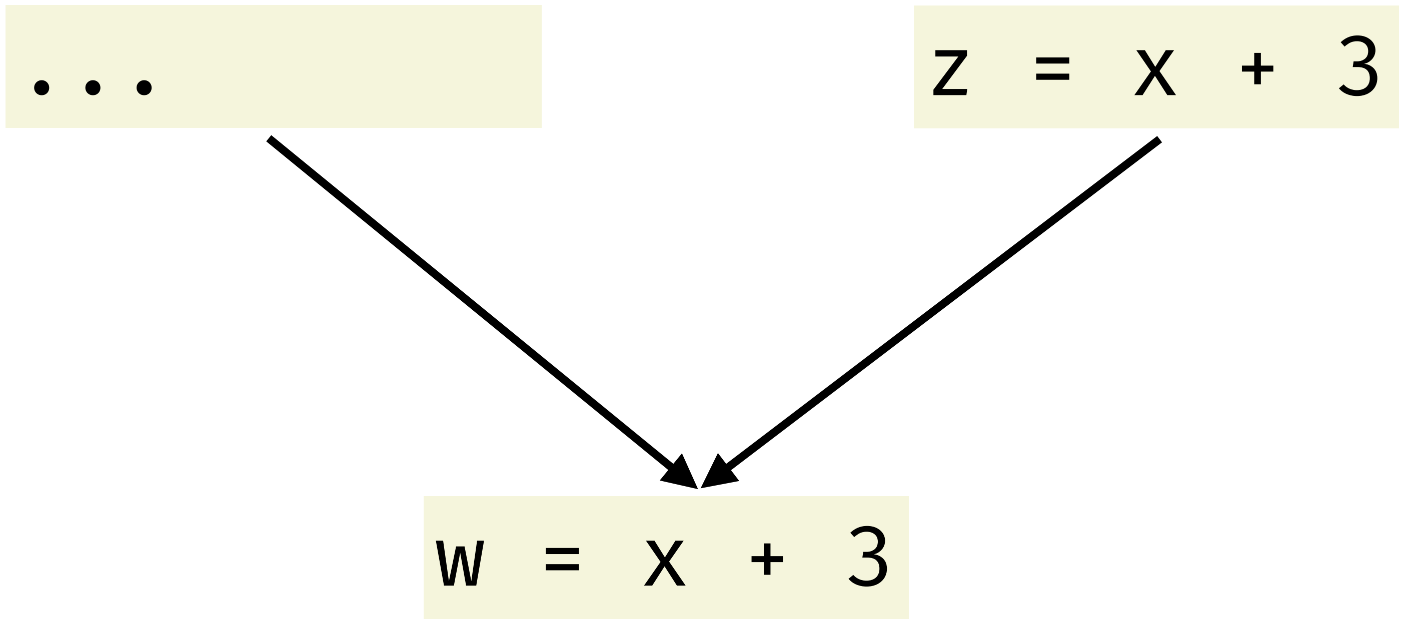
```
Q1 = {x1,y1}
Q2 = {x2,y2}
Q3 = {x5,y5}
Q4 = {x6,y6}
Q5 = {x4,y4}
Q6 = {x3,y3}
```

GVN is very common in compilers; usually applied after every few optimization passes.

Optimization 3: Partial Redundancy Elimination (PRE)

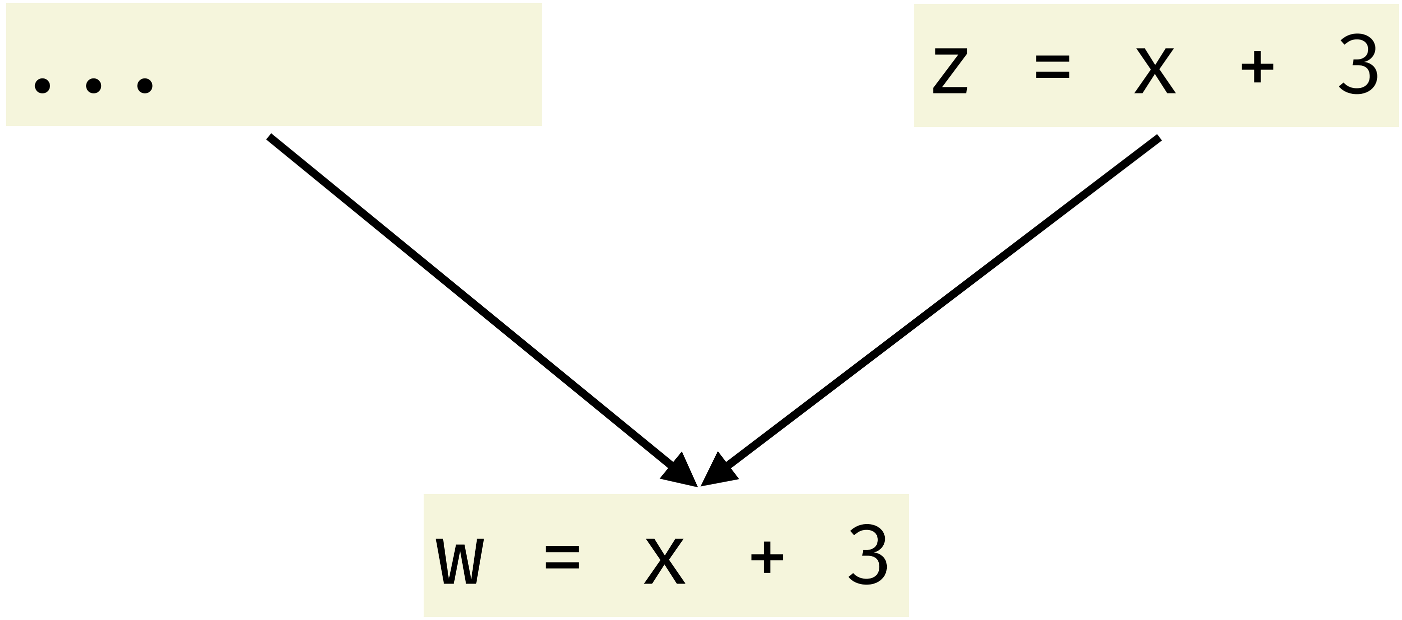
Partial Redundancy Elimination

- Is there something redundant in this program?

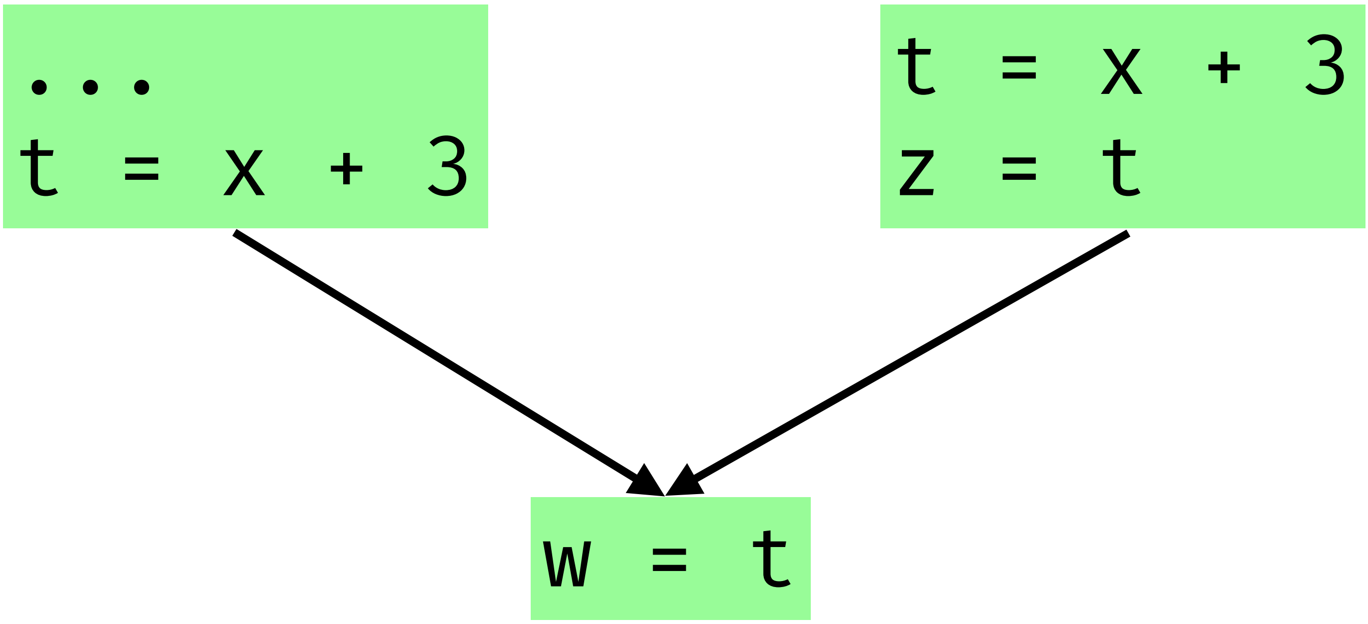


- `x + 3` is computed twice if the right branch is taken.
- What if the left branch is taken?
- If an expression is redundant only in *some* paths, it is called *partially redundant*.

Partial Redundancy Elimination



- ➤ We can *add* a computation to one basic block:



- ➤ And get rid of a redundancy that used to manifest sometimes (*partially*) by making it *fully* redundant!

PRE: Considerations

- We need to determine:
  - which expressions are *partially available*
  - which expressions are used in future (*anticipated*)
  - where to hoist the redundant computation (*possible placement* and *insertion*)
  - which existing computations to *remove*

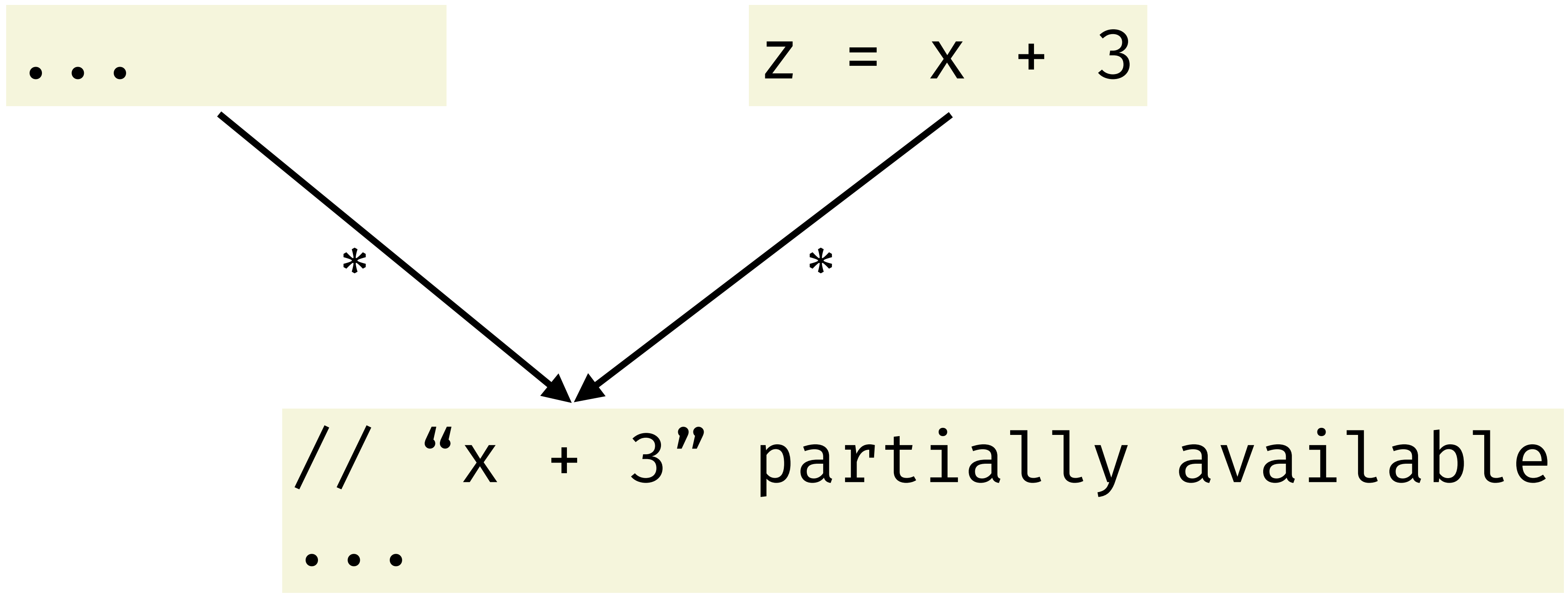
- Let's start with available expressions (recall common-subexpression elimination):

```
AvIn[n] = ∀p∈pred[n] ∩ AvOut[p]
AvOut[n] = Gen[n] ∪ (AvIn[n] – Kill[n])
```

Partially Available Expressions

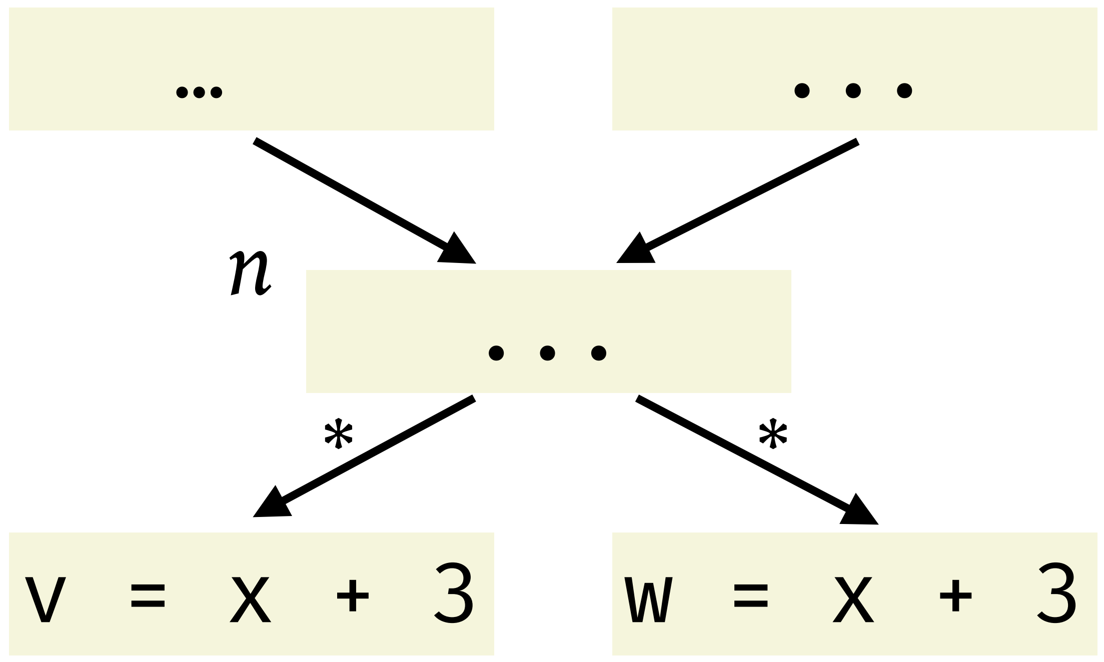
- Similar to available expressions except that an expression must be computed (and not killed) along *some* (instead of *all*) paths:

```
PavIn[n] = ∀p∈pred[n] ∪ PavOut[p]
PavOut[n] = Gen[n] ∪ (PavIn[n] - Kill[n])
```



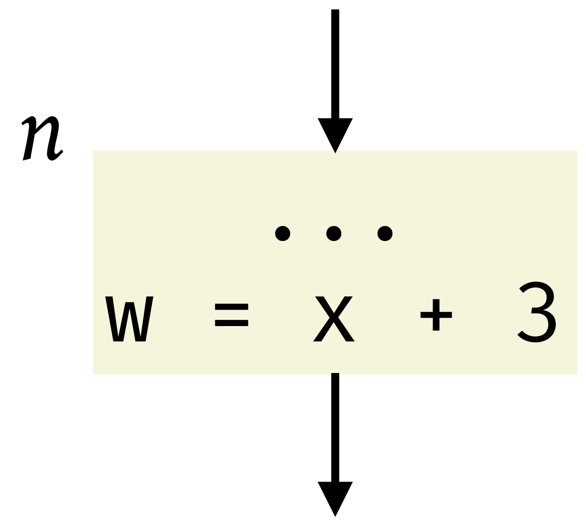
Anticipated Expressions

- An expression is *anticipated* at node n if it is computed (with the same values of operands) in each path from n to *exit*.



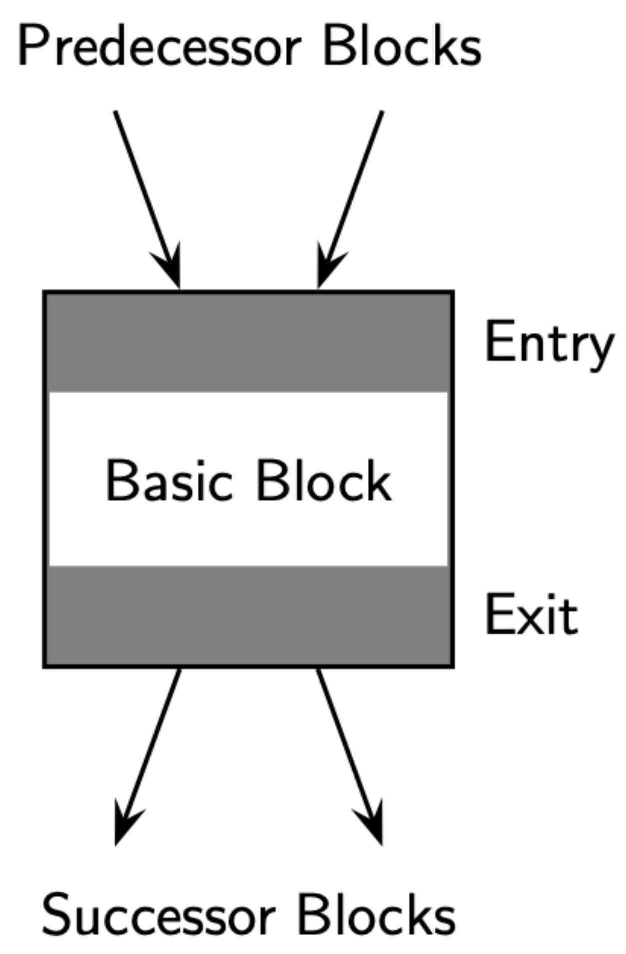
- An expression is *anticipated locally* (also called *upwards exposed*) at node n if it is computed at n without prior modification of its operands (given by `AntLoc[n]`).

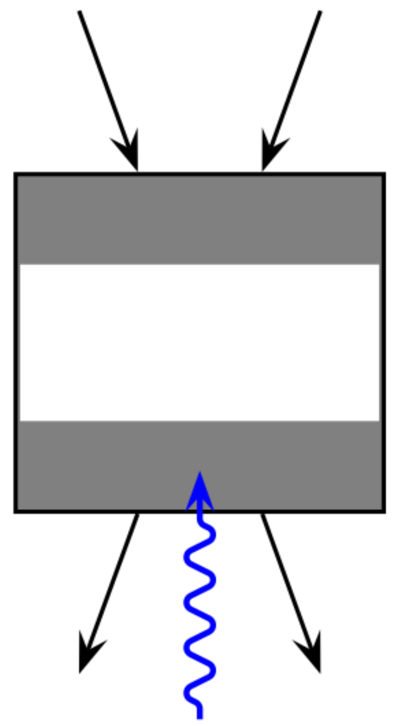
```
AntOut[n] = ∀s∈succ[n] ∩ AntIn[s]
AntIn[n] = AntLoc[n] ∪ (AntOut[n] - Kill[n])
```



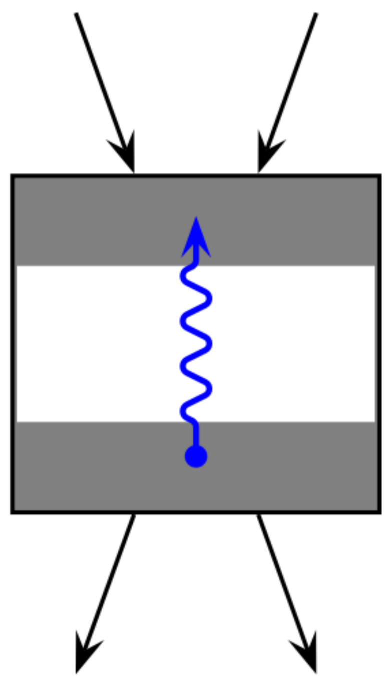
Partial Redundancies and Hoisting

- An expression is *partially redundant* at node n if it is partially available at n **and** anticipated at n.
- A key part of *partial-redundancy elimination* is to decide where to *hoist* computations of an expression for converting its partial redundancy to full redundancy (which may then get eliminated later).
- Can an expression be hoisted to?

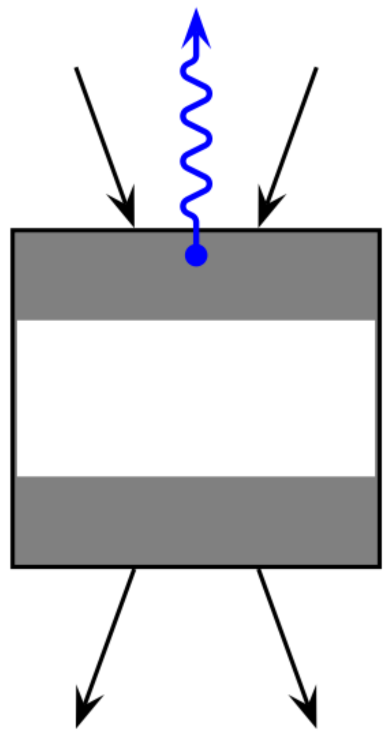




exit of a block

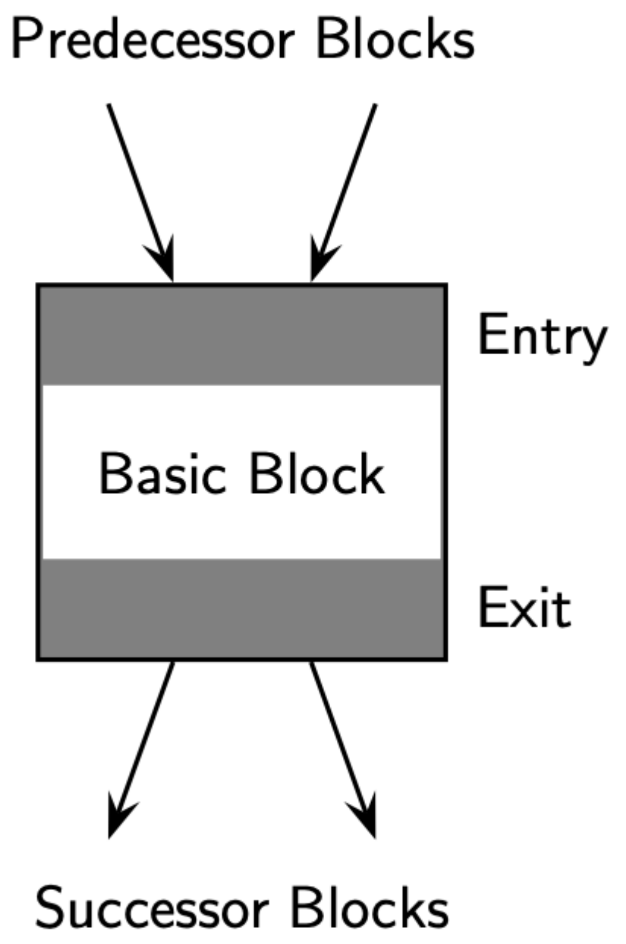


entry of a block



above a block

Inserting computations to introduce full redundancy



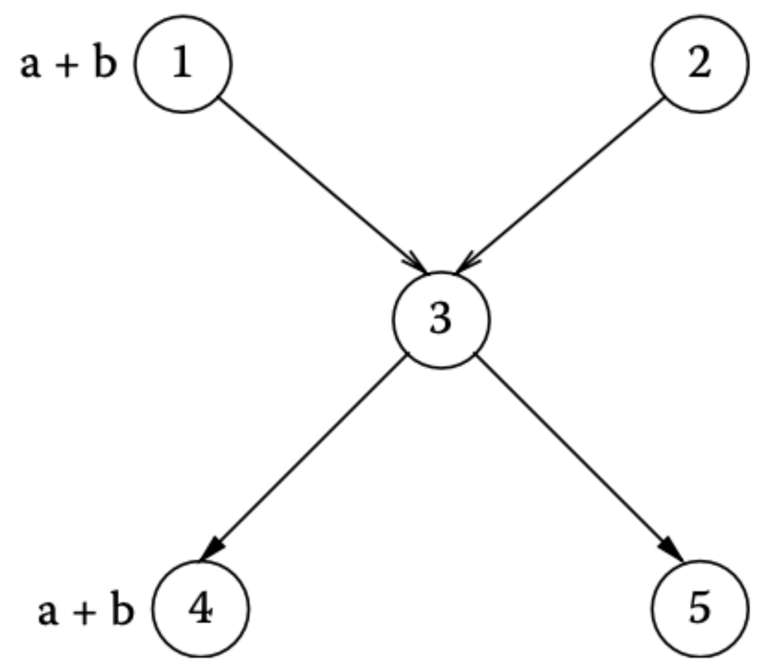
- Let's define a set of expressions that can be *possibly placed* (or hoisted) at a node *n*.
- The out-set is simple:

```
PPOut[n] = ∀s∈succ[n] ∩ PPIn[s]
```

- The in-set is tricky, and generates our longest dataflow equation :-)

Inserting computations to introduce full redundancy

- Among the partially available expressions at *n*, we can place those at its entry that
  - are either anticipated locally, or can be placed at its exit (`PpOut`) and don't get killed; and
  - for every predecessor *p*, are either available at *p*'s exit or can be placed at *p*'s exit.
    - When can the latter not be the case?




```
PpIn[n] = PavIn[n] ∩ (AntLoc[n] ∪ (PpOut[n] - Kill[n]))
                  ∩ ∀p∈pred[n] (AvOut[p] ∪ PpOut[p])
```

Where do we insert new computations then?!

- We don't want to insert an expression at node *n* if it can be placed at a predecessor of *n*.
- Insert an expression *e* at the exit of a node *n* if
  - Exit of *n* is a possible placement point for *e*;
  - *e* is not already available at *n*; and
  - moving *e* further up does not work because either e cannot be placed at *n*'s entry or because *n* kills *e*.


```
Insert[n] = PpOut[n] ∩ !AvOut[n] ∩ (!PpIn[n] ∪ Kill[n])
```

Finally, removing existing computations

- We have identified partial redundancies and added expressions to convert them to full redundancies.
- Now we can remove full redundancies!
- From a node *n*, we can remove the computation of expressions that are anticipated locally and can be placed at *n*'s beginning:

```
Remove[n] = AntLoc[n] ∩ PpIn[n]
```