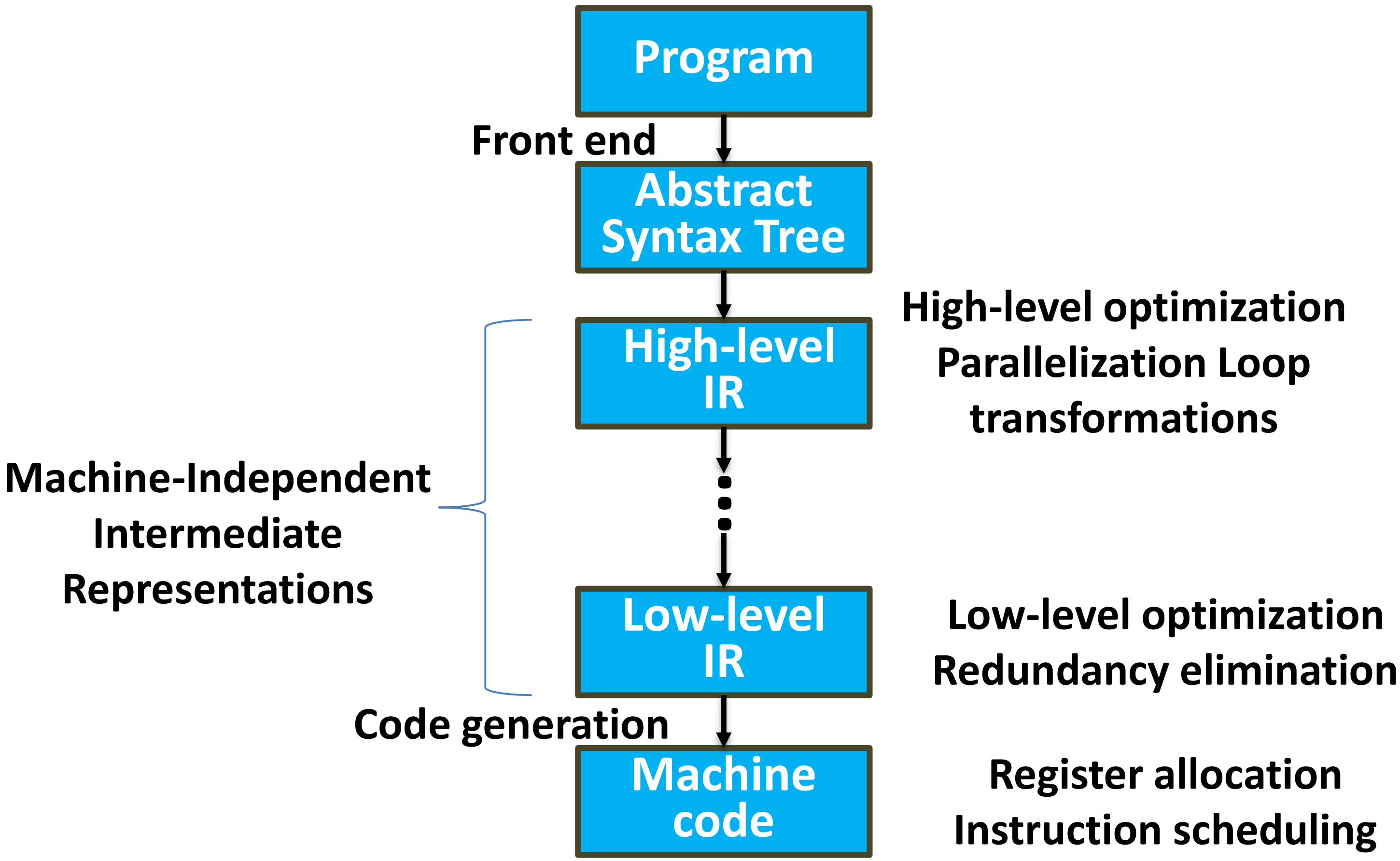
Data flow analysis

ZheMin Yang

Compiler Organization



Flow Graph

- Basic block = a maximal sequence of consecutive instructions s.t.
  - flow of control only enters at the beginning
  - flow of control can only leave at the end (no halting or branching except perhaps at end of block)
- Control Flow Graphs
  - Nodes: basic blocks
  - Edges
    - **Bi → Bj, iff Bj can follow Bi immediately in execution**

What is Data Flow Analysis?

- Data flow analysis:
  - Flow-sensitive: sensitive to the control flow in a function
  - Intra-procedural analysis

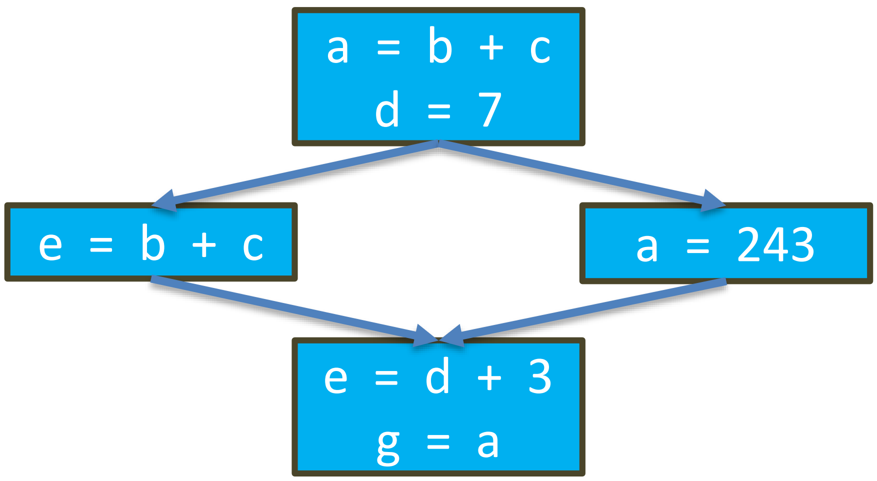
Definitions

- Flow-sensitive
  - Sensitive to the order of statements in a program
- Path-sensitive
  - Sensitive to the path(branch) taken in the execution
- Context-sensitive
  - Sensitive to the calling context
- Object-sensitive
- Field-sensitive

- Intra-procedural analysis
- Inter-procedural analysis

What is Data Flow Analysis?

- Examples of optimizations:
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination



Value of x?
Which "definition" defines x?
Is the definition still meaningful (live)?

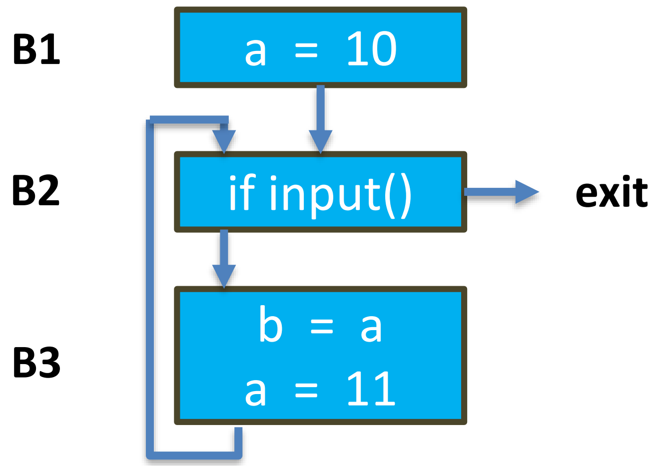
Static Program vs. Dynamic Execution

- **Statically**: Finite program

- **Dynamically**: Can have infinitely many possible execution paths

Static Program vs. Dynamic Execution

- Data flow analysis abstraction:
  - For each point in the program:
  - combines information of all the instances of the same program point.
- Example of a data flow question:
  - Which definition defines the value used in statement "b = a"?

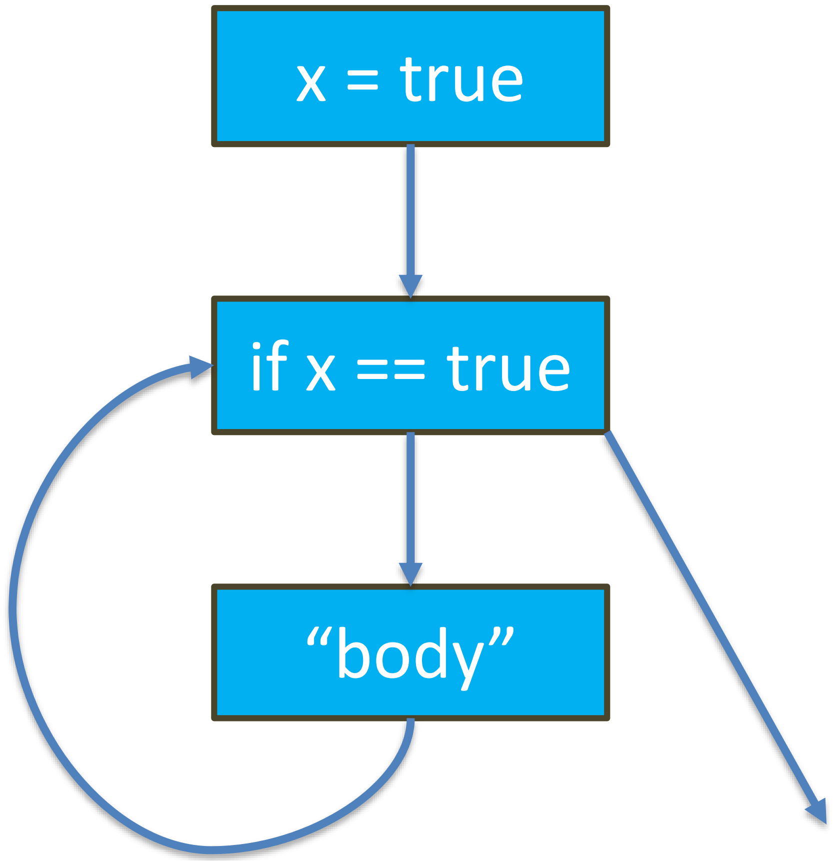


An Obvious Theorem

```
boolean x = true;
while (x) {
        …// no change to x
}
```

- Doesn't terminate.
- Proof: only assignment to x is at top, so x is always true.

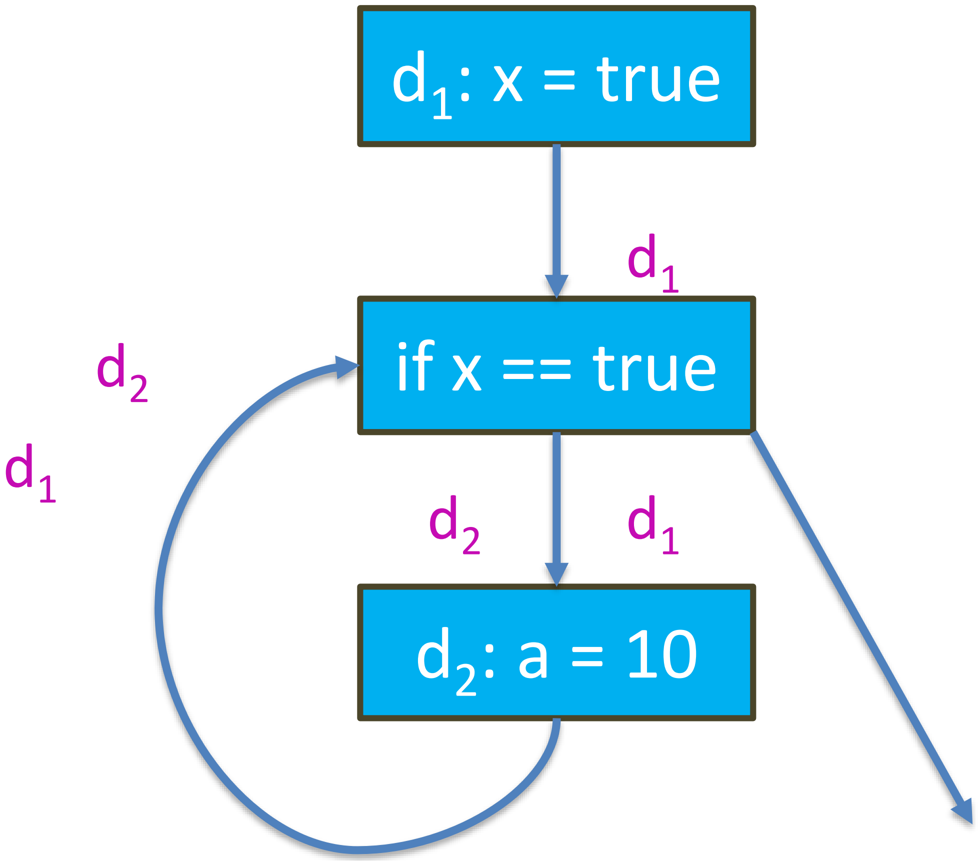
As a Flow Graph



Formulation: Reaching Definitions

- Each place some variable x is assigned is a definition.
- Ask: for this use of x, where could x last have been defined.
- In our example: only at x=true.

Example: Reaching Definitions



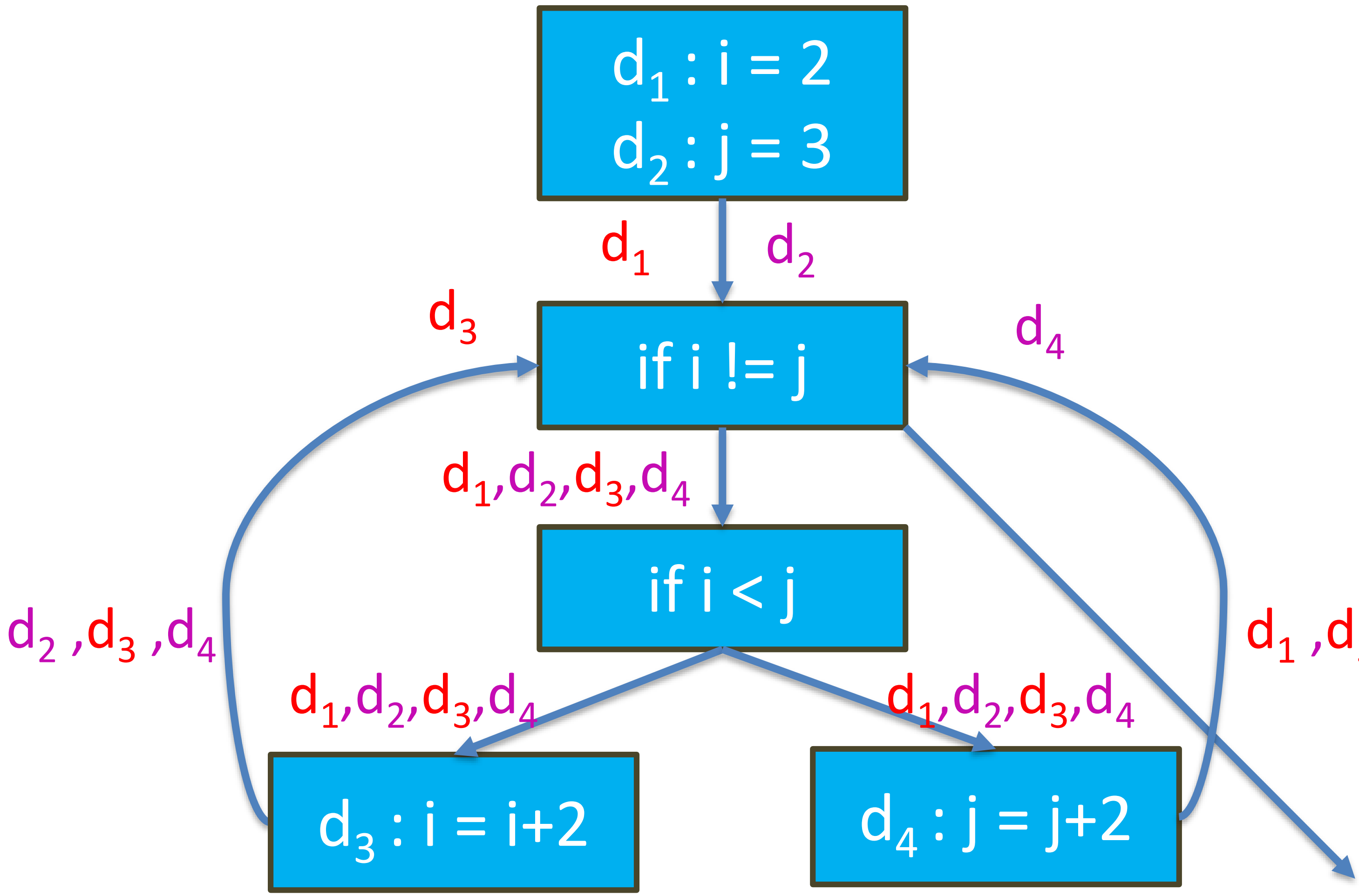
Clincher

- Since at x == true, d1 is the only definition of x that reaches, it must be that x is true at the point.
- The conditional is not really a conditional and **can be replace by a branch**.

Not Always That Easy

```
int i = 2; int j = 3;
while ( i != j ) {
        if (i < j) i += 2;
        else j += 2
}
```

The Flow Graph

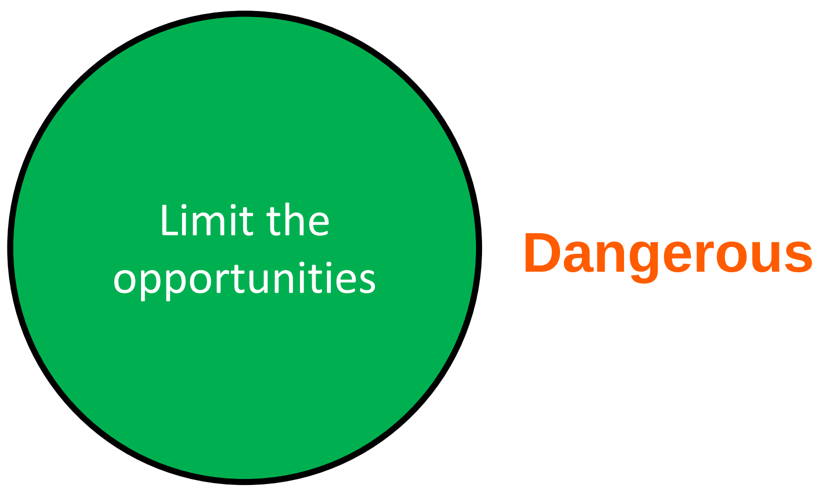


DFA Is Sufficient Only

- In this example, i can be defined in two places, and j in two places.
- No obvious way to discover that i!=j is always true.
- But OK, because reaching definitions is sufficient to catch most opportunities for ***constant folding*** (replacement of a variable by its only possible value).

Be Conservative!

- (Code optimization only)
- It s OK to discover a subset of the opportunities to make some code-improving transformation.
- It s not OK to think you have an opportunity that you don't really have.

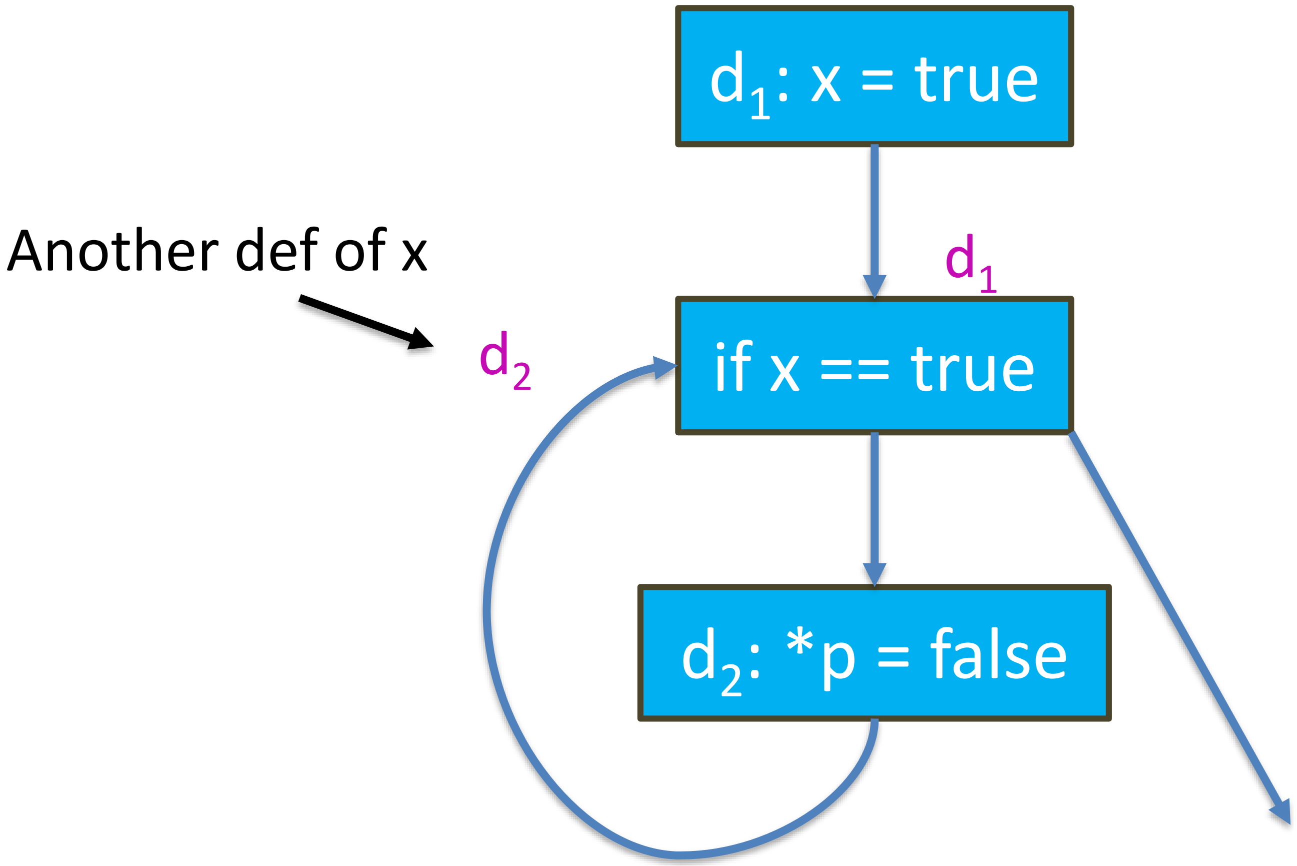


Example: Be Conservative

```
Boolean x = true;
While (x) {
        … *p = false; …
}
```

- Is it possible that p points to x?

As a Flow Graph



Possible Resolution

- Just as data-flow analysis of "reaching definitions" can tell what definitions of x might reach a point, another DFA can eliminate cases where p definitely does not point to x.
- Example: the only definition of p is p = &y and there is no possibility that y is an alias of x.

Reaching Definitions Formalized

- A definition d of a variable x is said to *reach* a point p in a flow graph if:
  - 1. Every path from the entry of the flow graph to p has d on the path, and
  - 2. After the last occurrence of d there is no possibility that x is redefined.

*Usually it's a **MAY** problem, not a **MUST** problem*

Data-Flow Equations --- (1)

- A basic block can *generate* a definition.
- A basic block can either
  - 1. *Kill* a definition of x if it surely redefines x.
  - 2. Transmit a definition if it may not redefine the same variable(s) as that definition.

Data-Flow Equations --- (2)

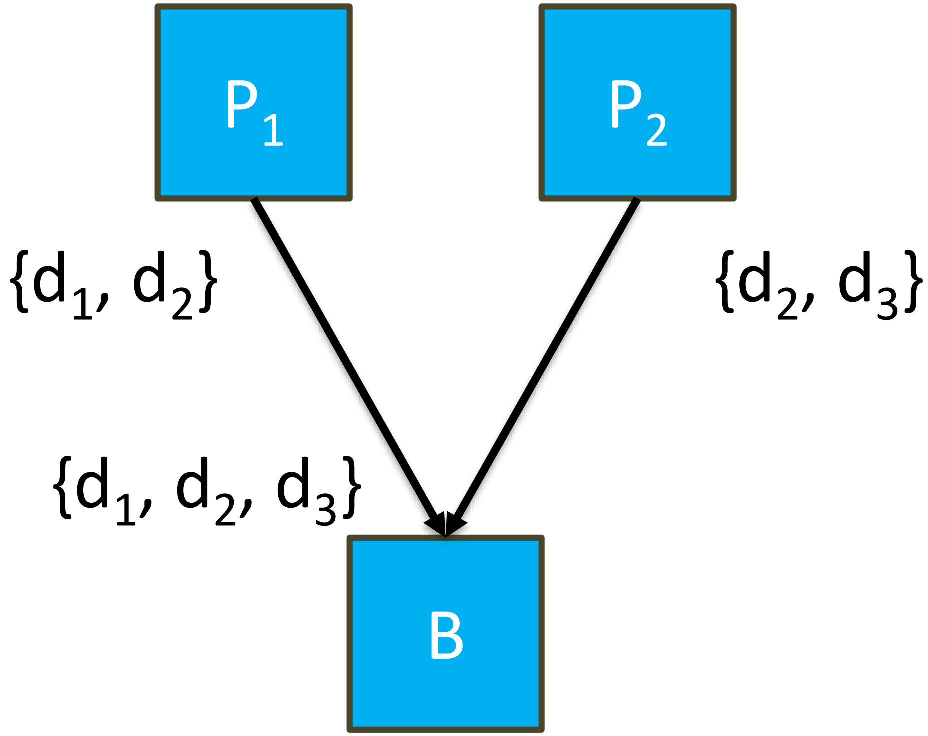
- Variables:
  - 1. IN(B) = set of definitions reaching the beginning of block B.
  - 2. OUT(B) = set of definitions reaching the end of B.

Data-Flow Equations --- (3)

- Two kinds of equations:
  - 1. Confluence equations : IN(B) in terms of outs of predecessors of B.
  - 2. Transfer equations : OUT(B) in terms of IN(B) and what goes on in block B.

Confluence Equations

$IN(B) = \cup_{\text{predecessors } P \text{ of } B} OUT(P)$



Transfer Equations

- *Generate* a definition in the block if its variable is not definitely rewritten later in the basic block.
- *Kill* a definition if its variable is definitely rewritten in the block.
- An internal definition may be both killed and generated.

Example: Gen and Kill

IN = {$d_2(x)$, $d_3(y)$, $d_3(z)$, $d_5(y)$, $d_6(y)$, $d_7(z)$}

Kill includes {$d_1(y)$, $d_2(x)$, $d_3(y)$, $d_5(y)$, $d_6(y)$,. }

Gen = {$d_2(x)$, $d_3(x)$, $d_3(z)$, … , $d_4(y)$}

d_1: y = 3
d_2: x = y + z
d_3: *p = 10
d_4: y = 5

OUT = {$d_2(x)$, $d_3(x)$, $d_3(z)$,…, $d_4(y)$, $d_7(z)$}

Transfer Function for a Block

- For any block B:

$$OUT(B) = (IN(B) - Kill(B)) \cup Gen(B)$$

Iterative Solution to Equations

- For an n-block flow graph, there are 2n equations in 2n unknowns.
- Alas, the solution is not unique.
  - Standard theory assumes a field of constants; sets are not a field.
- Use iterative solution to get the least fixed point.
  - Identifies any def that might reach a point.

Iterative Solution --- (2)

IN(entry) = $\emptyset$;

For each block B do OUT(B) = $\emptyset$;

While (changes occur) do

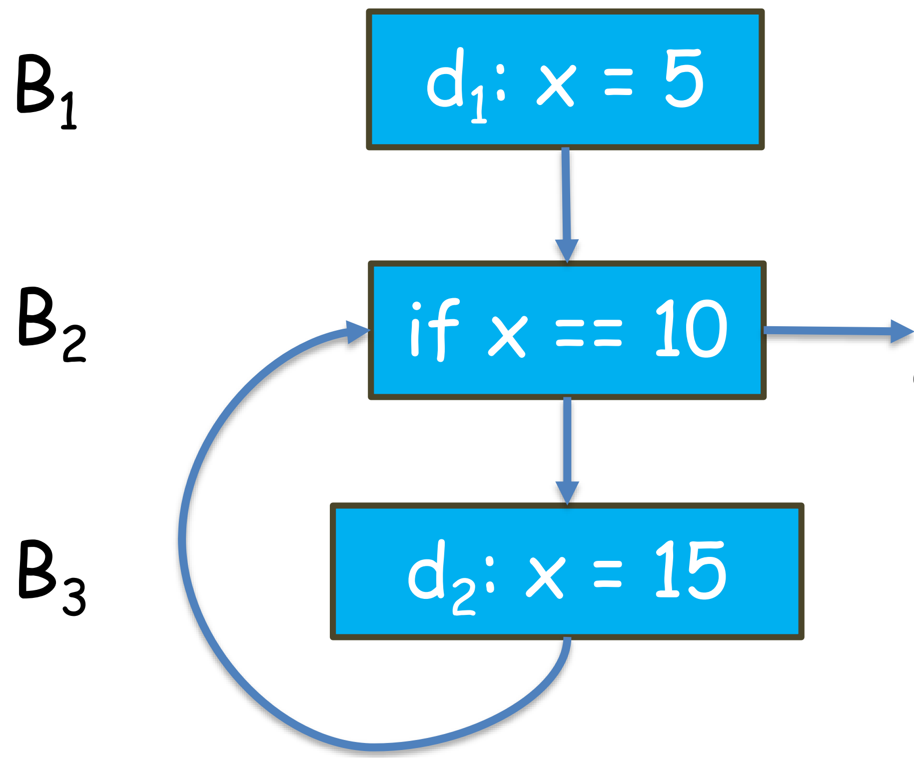
  for each block B do {

    IN(B) = $\cup_{\text{predecessors } P \text{ of } B}$ OUT(P);

    OUT(B) = (IN(B) - Kill(B)) $\cup$ Gen(B);

  }

Example: Reaching Definitions



$IN(B_1) = \{\}$

$OUT(B_1) = \{d_1\}$

$IN(B_2) = \{d_1, d_2\}$

$OUT(B_2) = \{d_1, d_2\}$

$IN(B_3) = \{d_1, d_2\}$

$OUT(B_3) = \{d_2\}$

Aside: Notice the Conservatism

- Not only the most conservative assumption about when a def is killed or gen'd.
- Also the conservative assumption that any path in the flow graph can actually be taken.
- Fine, as long as the optimization is triggered by limitations on the set of RD's, not by the assumption that a def does not reach.

Another Data-Flow Problem: Available Expressions

- An expression $x + y$ is *available* at a point if no matter what path has been taken to that point from the entry, $x + y$ has been evaluated, and neither x nor y have even possibly been redefined.
- Useful for global common- subexpression elimination.

Avoid Reevaluation/Recalculation

Equations for AE

- The equations for AE are essentially the same as for RD, with one exception.
- Confluence of paths involves **intersection** of sets of expressions rather than union of sets of definitions.

Gen(B) and Kill(B)

- An expression x + y is *generated* if it is computed in B, and afterwards there is no possibility that either x or y is redefined.
- An expression x + y is *killed* if it is not generated in B and either x or y is possibly redefined.

Example

Kills x + y,
w * x, etc.

x = x + y
z = a + b

Kills z – w,
x + z, etc.

Generates
a + b

Transfer Equations

- Transfer is the same idea:

$$OUT(B) = (IN(B) - Kill(B)) \cup Gen(B)$$

Confluence Equations

- Confluence involves intersection, because an expression is available coming into a block if and only if it is available coming out of **each predecessor**.

$$IN(B) = \cap_{\text{predecessors P of B}} OUT(P)$$

Iterative Solution

```
IN(entry) = ∅;
For each block B do OUT(B) = ALL;
While (changes occur ) do
   for each block B do {
```

$IN(B) = \bigcap_{\text{predecessors P of B}} OUT(P);$

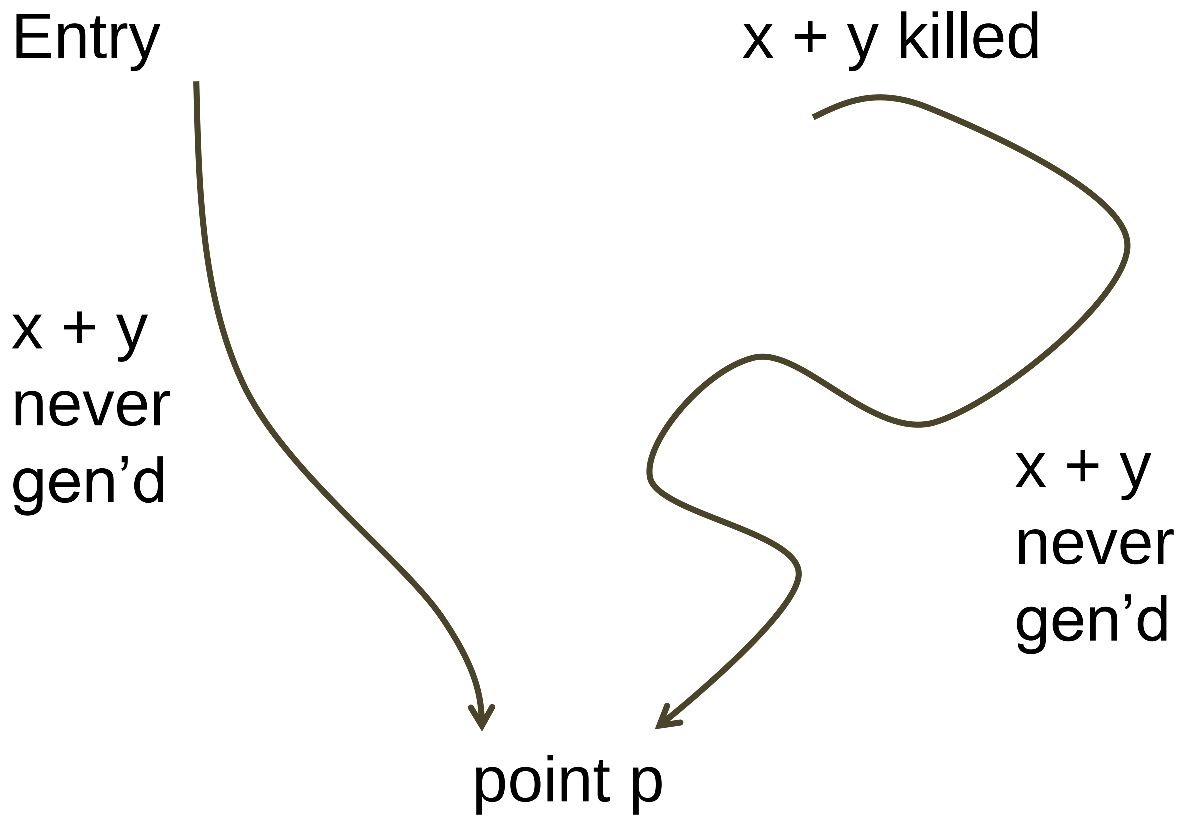
$OUT(B) = (IN(B) – Kill(B)) \cup Gen(B);$

```
   }
```

Why It Works

- An expression x + y is **unavailable** at point p iff there is a path from the entry to p that either:
  - 1. Never evaluates x + y, or
  - 2. Kills x + y after its **last** evaluation.
- IN(entry) = ∅ takes care of (1).
- OUT(B) = ALL, plus intersection during iteration handles (2).

Example



Subtle Point

- It is conservative to assume an expression isn't available, even if it is.
- But we don't have to be "insanely conservative."
  - If after considering all paths, and assuming x + y killed by any possibility of redefinition, we still can t find a path explaining its unavailability, then x + y is available.

Live Variable Analysis

- Variable x is *live* at a point p if on **some** path from p, x is used before it is redefined.
- Useful in code generation: if x is not live on exit from a block, there is no need to copy x from a register to memory.

Equations for Live Variables

- LV is essentially a "backwards“ version of RD.
- In place of Gen(B): Use(B) = set of variables x possibly used in B prior to any certain definition of x.
- In place of Kill(B): Def(B) = set of variables x certainly defined before any possible use of x.

Transfer Equations

- Transfer equations give IN's in terms of OUT's:

$$IN(B) = (OUT(B) - Def(B)) \cup Use(B);$$

Confluence Equations

- •Confluence involves union over successors, so a variable is in OUT(B) if it is live on entry to any of B's successors.

$$OUT(B) = \cup_{\text{successors } S \text{ of } B} IN(S);$$

Iterative Solution

```
OUT(exit) = ∅;
for each block B do IN(B) = ∅;
While (changes occur) do
   for each block B do {
```

$OUT(B) = \bigcup_{\text{successors } S \text{ of } B} IN(S);$

$IN(B) = (OUT(B) - Def(B)) \cup Use(B);$

```
   }
```