Lecture 3

Local Optimizations, Intro to SSA

I. Basic blocks & Flow graphs

II. Abstraction 1: DAG

III. Abstraction 2: Value numbering

IV. Intro to SSA

ALSU 8.4-8.5, 6.2.4

I. Basic Blocks & Flow Graphs

Basic block = a sequence of 3-address statements

- only the first statement can be reached from outside the block (no branches into middle of block)
- all the statements are executed consecutively if the first one is (no branches out or halts except perhaps at end of block)

- **We require basic blocks to be *maximal***, i.e., they cannot be made larger without violating the conditions

Flow graph

- **Nodes: basic blocks**
- **Edges: B_i -> B_j, iff B_j can follow B_i immediately in *some* execution**
  - Either first instruction of B_j is target of a goto at end of B_i
  - Or, B_j physically follows B_i, which does not end in an unconditional goto.

Partitioning into Basic Blocks

Identify the leader of each basic block

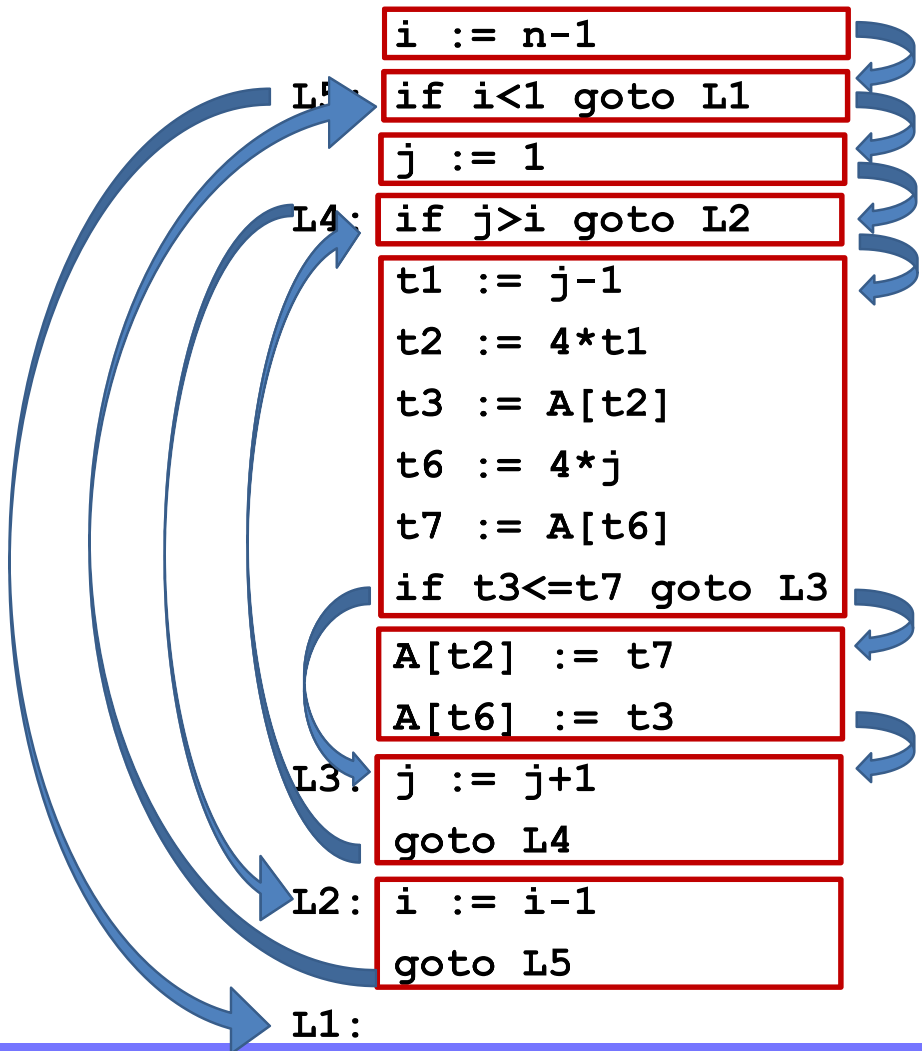
- **First instruction**
- **Any target of a jump**
- **Any instruction immediately following a jump**

Basic block starts at leader & ends at instruction immediately before a leader (or the last instruction)

```
    i  := n-1
L5: if i<1 goto L1
    j  := 1
L4: if j>i goto L2
    t1  := j-1
    t2  := 4*t1
    t3  := A[t2]
    t6  := 4*j
    t7  := A[t6]
    if t3<=t7 goto L3
    A[t2]  := t7
    A[t6]  := t3
L3: j  := j+1
    goto L4
L2: i  := i-1
    goto L5
L1:
```

ALSU 8.4

Flow Graph


i := n-1
L5: if i<1 goto L1
j := 1
L4: if j>i goto L2
t1 := j-1
t2 := 4*t1
t3 := A[t2]
t6 := 4*j
t7 := A[t6]
if t3<=t7 goto L3
A[t2] := t7
A[t6] := t3
L3: j := j+1
goto L4
L2: i := i-1
goto L5
L1:

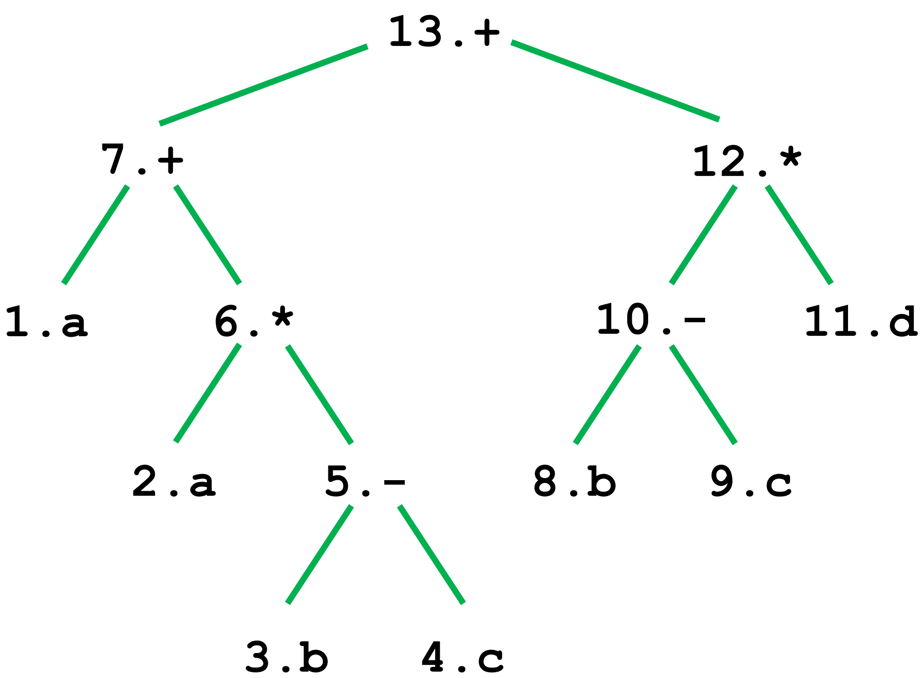
II. Local Optimizations (within basic block)

- **Common subexpression elimination**
  - array expressions
  - field access in records
  - access to parameters

Graph Abstractions

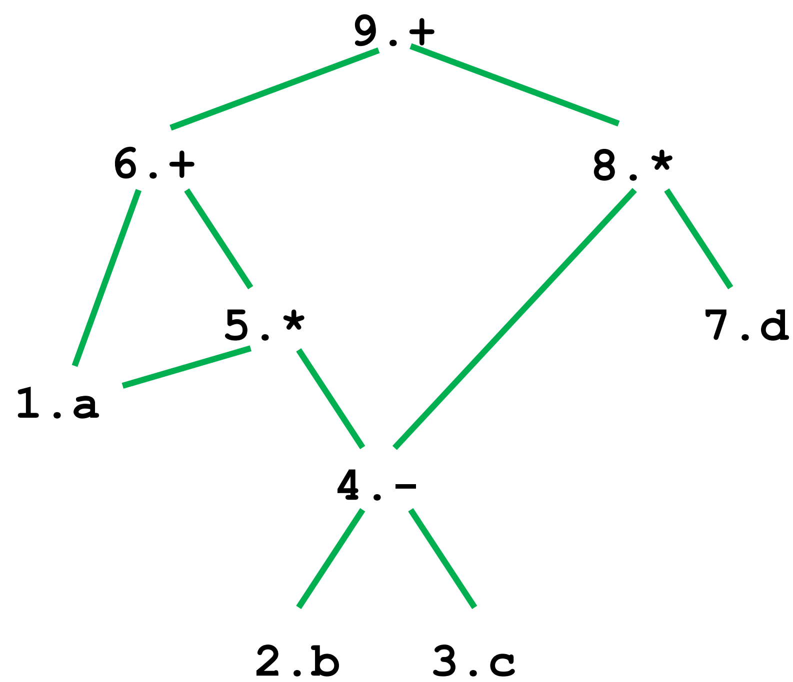
Example 1:

- **grammar (for bottom-up parsing): E -> E + T | E – T | T, T -> T*F | F, F -> (E) | id**
- **expression:** **`a+a*(b-c)+(b-c)*d`**



Parse tree

ALSU 4.5



Expression DAG

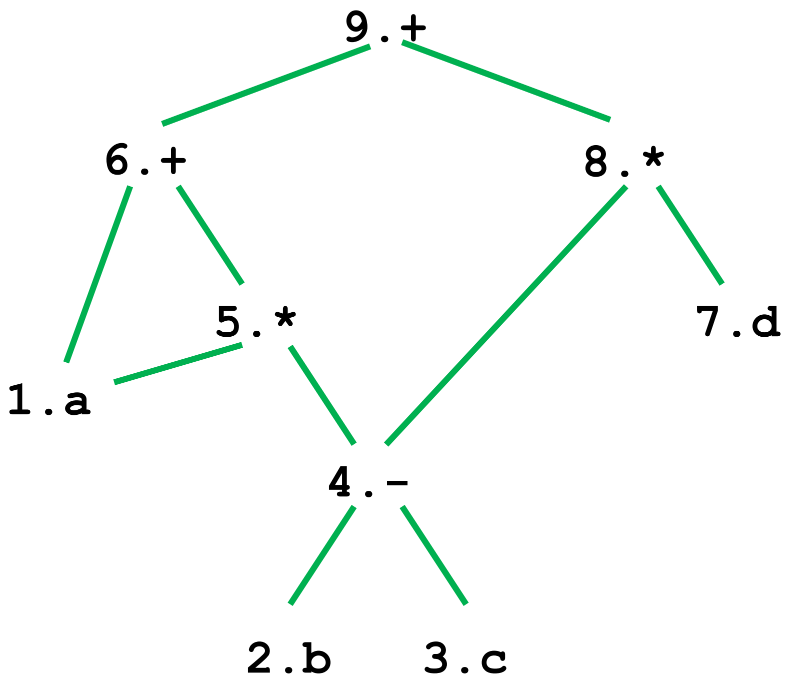
ALSU 6.1.1

Graph Abstractions

Expression: `a+a*(b-c)+(b-c)*d`

Optimized code:

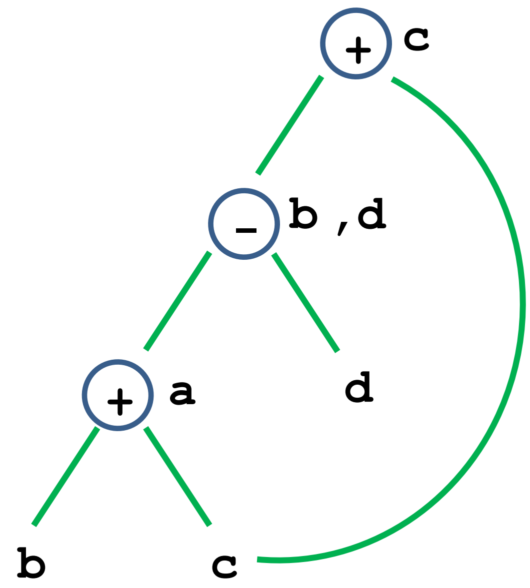
```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```



How well do DAGs hold up across statements?

Example 2:

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



Is this optimized code correct?

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

Depends on whether b is live on exit from the block

Critique of DAGs

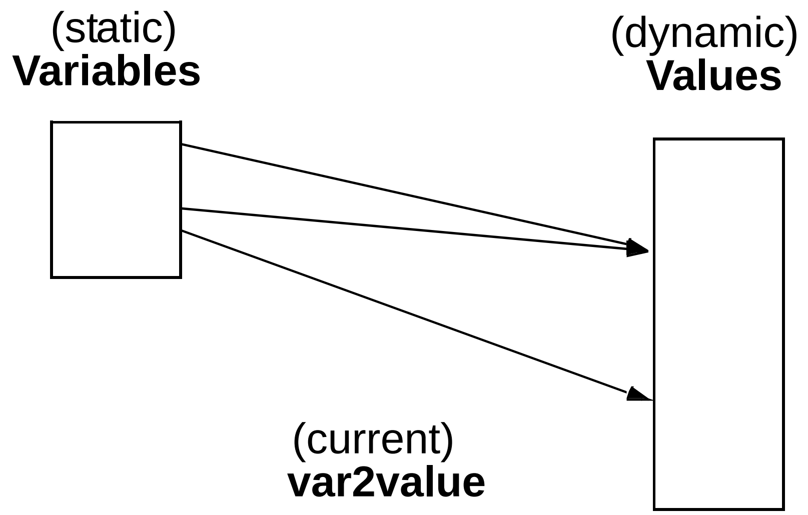
- **Cause of problems**
  - Assignment statements
  - Value of variable depends on TIME

- **How to fix problem?**
  - build graph in order of execution
  - attach variable name to latest value

- **Final graph created is not very interesting**
  - Key: variable->value mapping across time
  - loses appeal of abstraction

III. Value Number: Another Abstraction

- John Cocke & Jack Schwartz in unpublished book: “Programming Languages and their Compilers”, (1970)
- More explicit with respect to VALUES, and TIME



- **each value has its own “number”**
  - **common subexpression means same value number**
- var2value: current map of variable to value
  - used to determine the value number of current expression

r1 + r2 => var2value(r1)+var2value(r2)

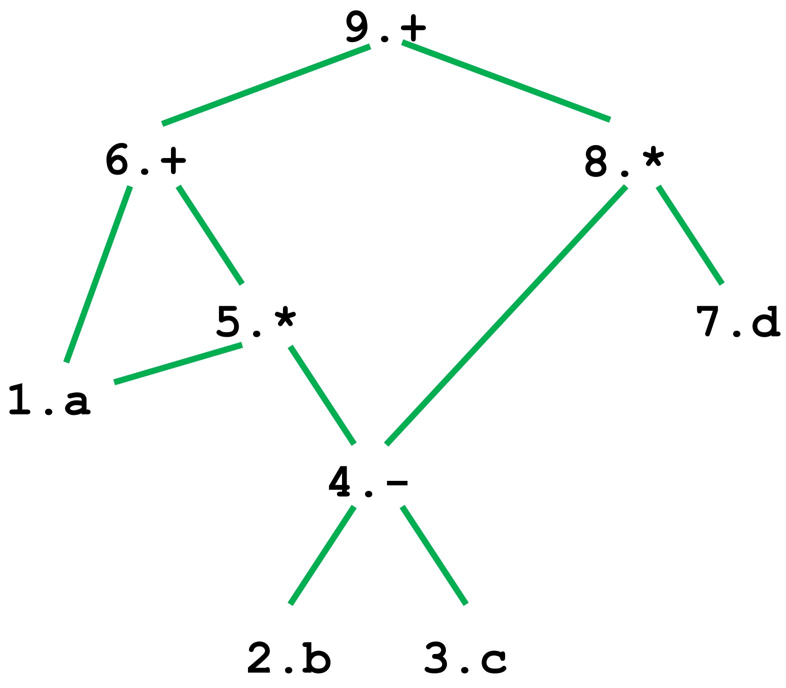
ALSU 6.1.2

Value Numbering: Expression Example

Expression: `a+a*(b-c)+(b-c)*d`

Optimized code:

```
t4 = b - c
t5 = a * t4
t6 = a + t5
t8 = t4 * d
t9 = t6 + t8
```



Value Numbering Algorithm

```
Data structure:
     VALUES = Table of
        expression     /* [OP, valnum1, valnum2] */
        var            /* name of variable currently holding expr */

For each instruction (dst = src1 OP src2) in execution order

  valnum1=var2value(src1); valnum2=var2value(src2)

  IF [OP, valnum1, valnum2] is in VALUES
     v = the index of expression
     Replace instruction with: dst = VALUES[v].var

  ELSE
     Add
        expression = [OP, valnum1, valnum2]
        var        = tv
     to VALUES
     v = index of new entry; tv is new temporary for v
     Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                               dst = tv

  set_var2value (dst, v)
```

More Details

- **What are the initial values of the variables?**
  - values at beginning of the basic block

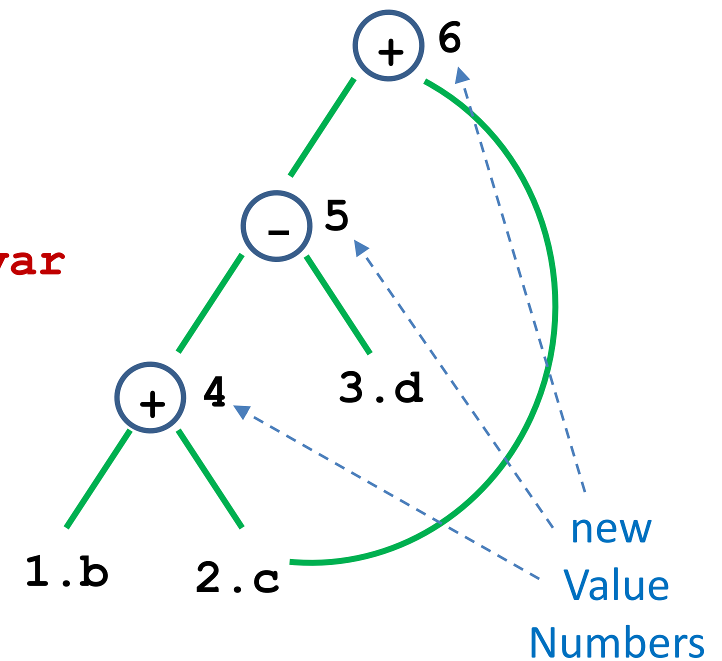
- **Possible implementations:**
  - Initialization: create “initial values” for all variables
  - Or dynamically create them as they are used

- **Implementation of VALUES and var2value: hash tables**

Value Numbering: Basic Block Example

```
a = b+c        t4 = b + c  // tv = VALUES[valnum1].var
                                   OP VALUES[valnum2].var
               a = t4      // dst = tv
b = a-d        t5 = t4 - d
               b = t5
c = b+c        t6 = t5 + c
               c = t6
d = a-d        d = t5      // dst =
                              VALUES[v].var
```

VALUES[5] = ([−,4,3], t5)



Q: Assigning to a temporary and then copying to the destination increases the number of instructions—so why do it?

A: If dst is overwritten later, would lose opportunity to eliminate common subexpression since no variable would hold the result

Question

- **How do you extend value numbering to constant folding?**

```
a = 1
b = 2
c = a+b
```

Answer: Can add a field to the VALUES table indicating when an expression is a constant and what its value is

DAGs vs. Value Numbering

- **Comparisons of two abstractions**
  - DAGs
  - Value numbering

- **Value numbering**
  - VALUE: distinguish between variables and VALUES
  - TIME
    - Interpretation of instructions in order of execution
    - Keep dynamic state information

IV. Intro to SSA

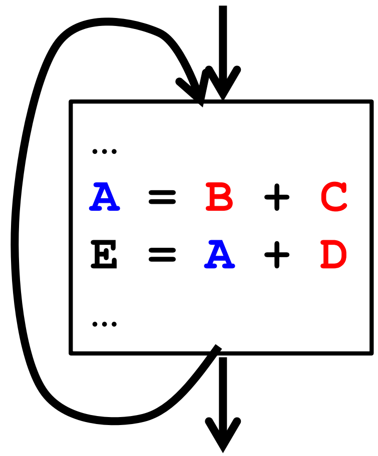
Global Optimizations: look beyond the basic block

- **Global versions of local optimizations**
  - global common subexpression elimination
  - global constant propagation
  - dead code elimination
- **Loop optimizations**
  - reduce code to be executed in each iteration
  - code motion
  - induction variable elimination
- **Other control structures**
  - Code hoisting: eliminates copies of identical code on parallel paths in a flow graph to reduce code size.

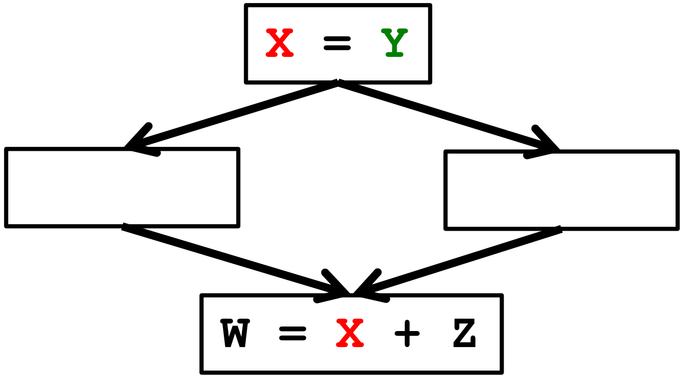
We will cover these optimizations in later lectures

Recurring Optimization Theme: Where Is a Variable Defined or Used?

- Example: Loop-Invariant Code Motion
  - Are **B**, **C**, and **D** only defined outside the loop?
  - Other definitions of **A** inside the loop?
  - Uses of **A** inside the loop?

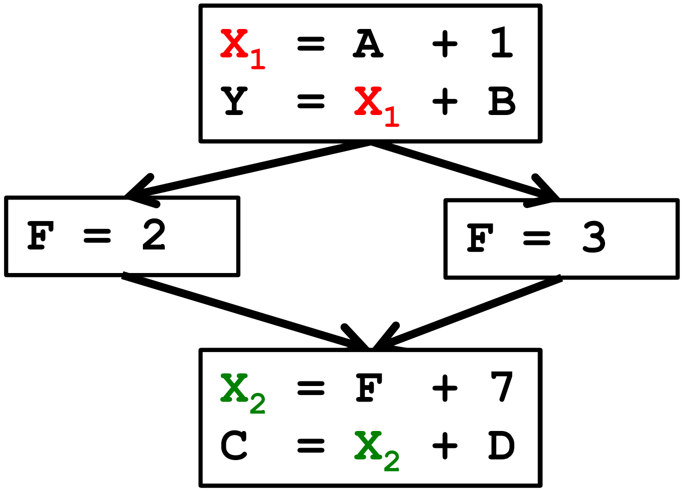


- Example: Copy Propagation
  - For a given use of **X**:
    - Are all reaching definitions of **X**:
      - copies from same variable: e.g., **X = Y**
    - Where **Y** is not redefined since that copy?
  - If so, substitute use of **X** with use of **Y**



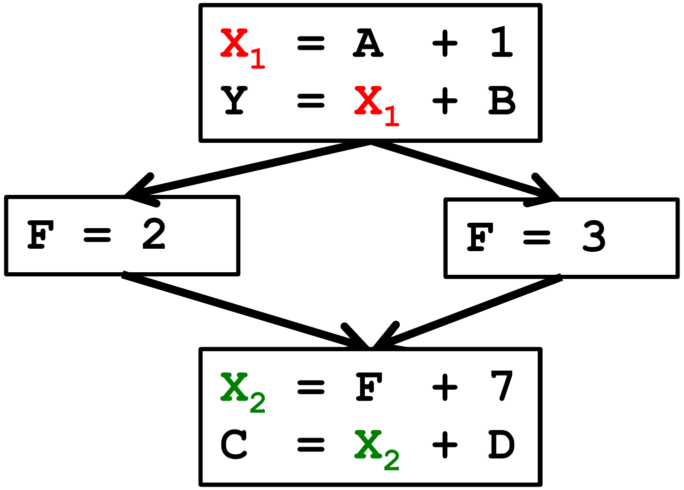
- It would be nice if we could *traverse directly* between related uses and def's
  - this would enable a form of *sparse* code analysis (skip over "don't care" cases)

Appearances of Same Variable Name May Be Unrelated



- The values in reused storage locations may be provably independent
  - in which case the compiler can optimize them as separate values
- Compiler could use renaming to make these different versions more explicit

Definition-Use and Use-Definition Chains



- Definition-Use (DU) Chains:
  - for a given definition of a variable X, what are all of its uses?
- Use-Definition (UD) Chains:
  - for a given use of a variable X, what are all of the reaching definitions of X?

Unfortunately DU and UD Chains Can Be Expensive

```
foo(int i, int j) {
   …
   switch (i) {
   case 0: x=3;break;
   case 1: x=1; break;
   case 2: x=6; break;
   case 3: x=7; break;
   default: x = 11;
   }
   switch (j) {
   case 0: y=x+7; break;
   case 1: y=x+4; break;
   case 2: y=x-2; break;
   case 3: y=x+1; break;
   default: y=x+9;
   }
   …
```

In general,

N defs

M uses

⇒ O(NM) space and time

One solution: limit each variable to ONE definition site

Unfortunately DU and UD Chains Can Be Expensive

```
foo(int i, int j) {
   …
   switch (i) {
   case 0: x=3; break;
   case 1: x=1; break;
   case 2: x=6;
   case 3: x=7;
   default: x = 11;
   }
   x1 is one of the above x's
   switch (j) {
   case 0: y=x1+7;
   case 1: y=x1+4;
   case 2: y=x1-2;
   case 3: y=x1+1;
   default: y=x1+9;
   }
   …
```

One solution: limit each variable to ONE definition site

Static Single Assignment (SSA)

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block (reminiscent of Value Numbering):
  - Visit each instruction in program order:
    - LHS: assign to a *fresh version* of the variable
    - RHS: use the *most recent version* of each variable

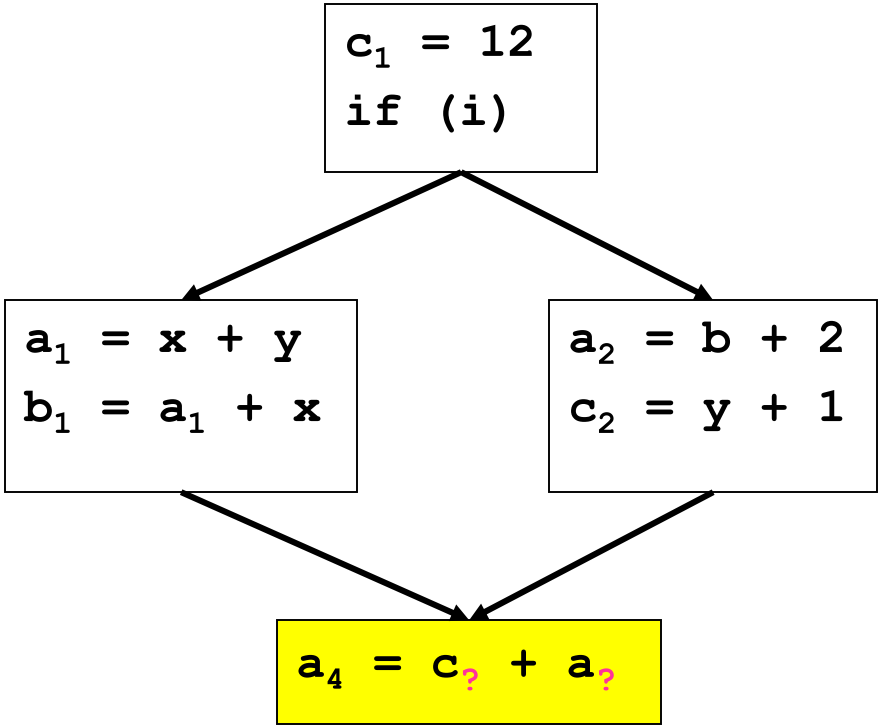
```
a = x + y
b = a + x
a = b + 2
c = y + 1
a = c + a
```

⇒

$$
\begin{aligned}
a_1 &= x + y \\
b_1 &= a_1 + x \\
a_2 &= b_1 + 2 \\
c_1 &= y + 1 \\
a_3 &= c_1 + a_2
\end{aligned}
$$

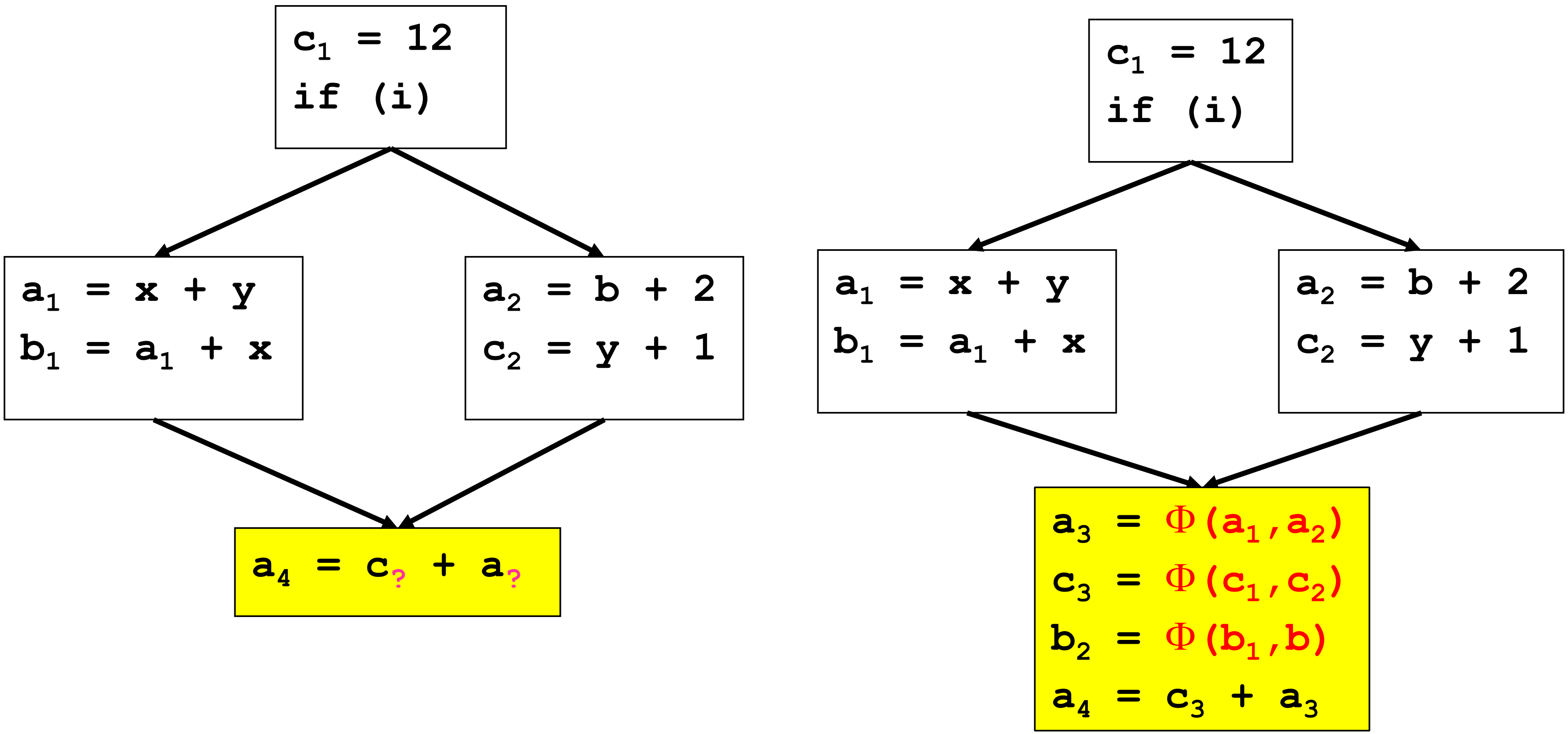
What about Joins in the CFG?

```
c = 12
if (i) {
   a = x + y
   b = a + x
} else {
   a = b + 2
   c = y + 1
}
a = c + a
```



→ Use a notational convention (fiction): a Φ function

Merging at Joins: the Φ function



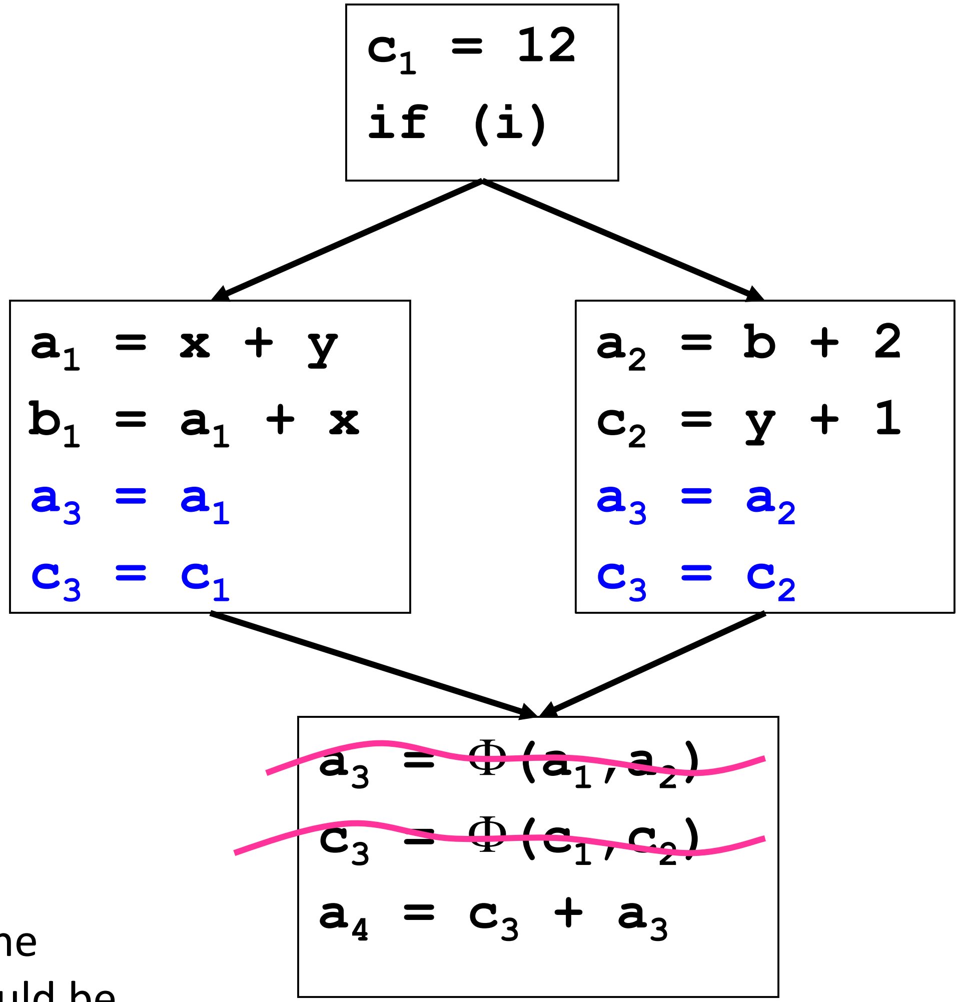
The Φ function

- Φ merges multiple definitions along multiple control paths into a single definition.
- At a basic block with *p* predecessors, there are *p* arguments to the Φ function.

$$x_{new} = \Phi(x_1, x_2, x_3, \ldots, x_p)$$

- How do we choose which x_i to use?
  - We don't really care!
- How do we emit code for this?

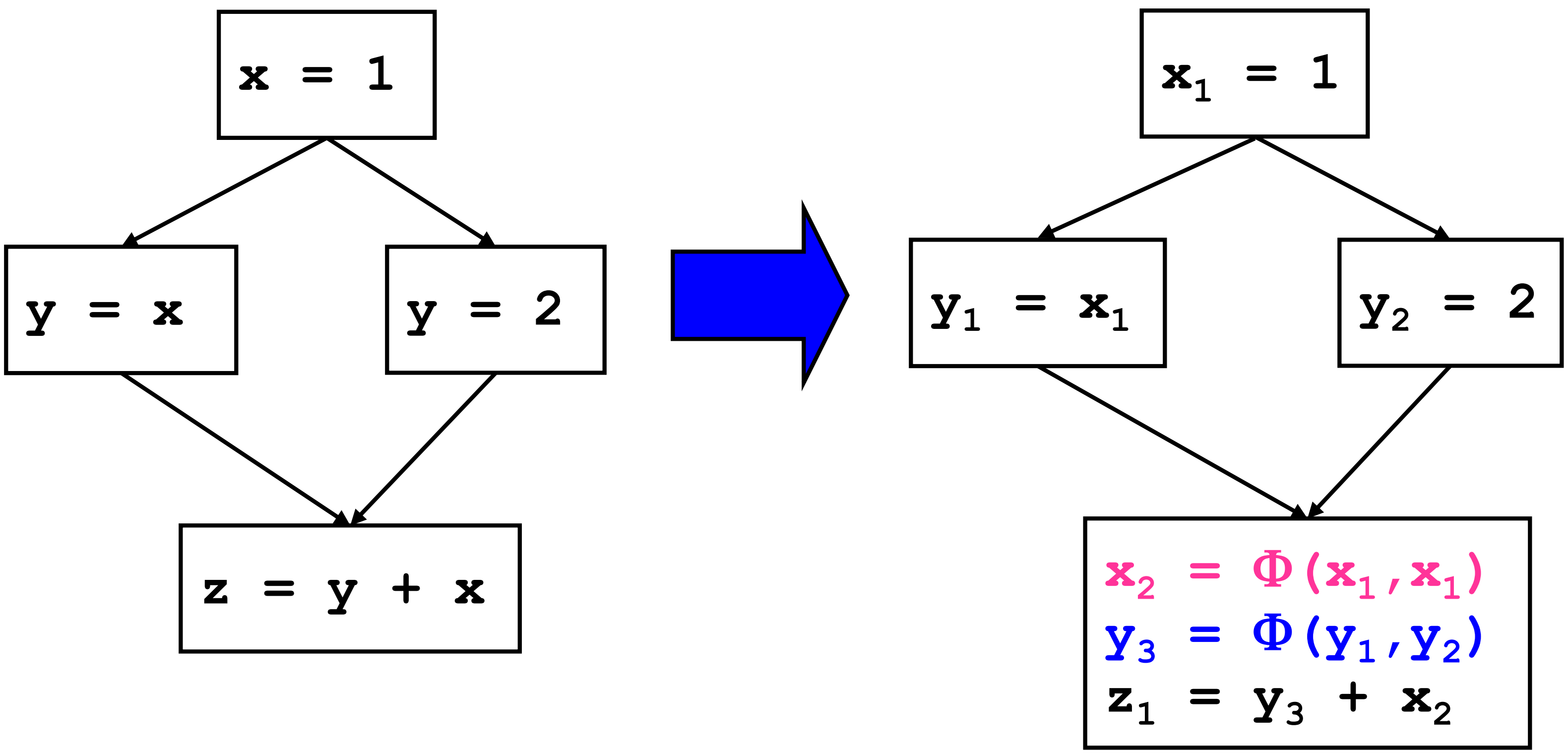
“Implementing” Φ



Never really done
this way, but could be

Trivial SSA

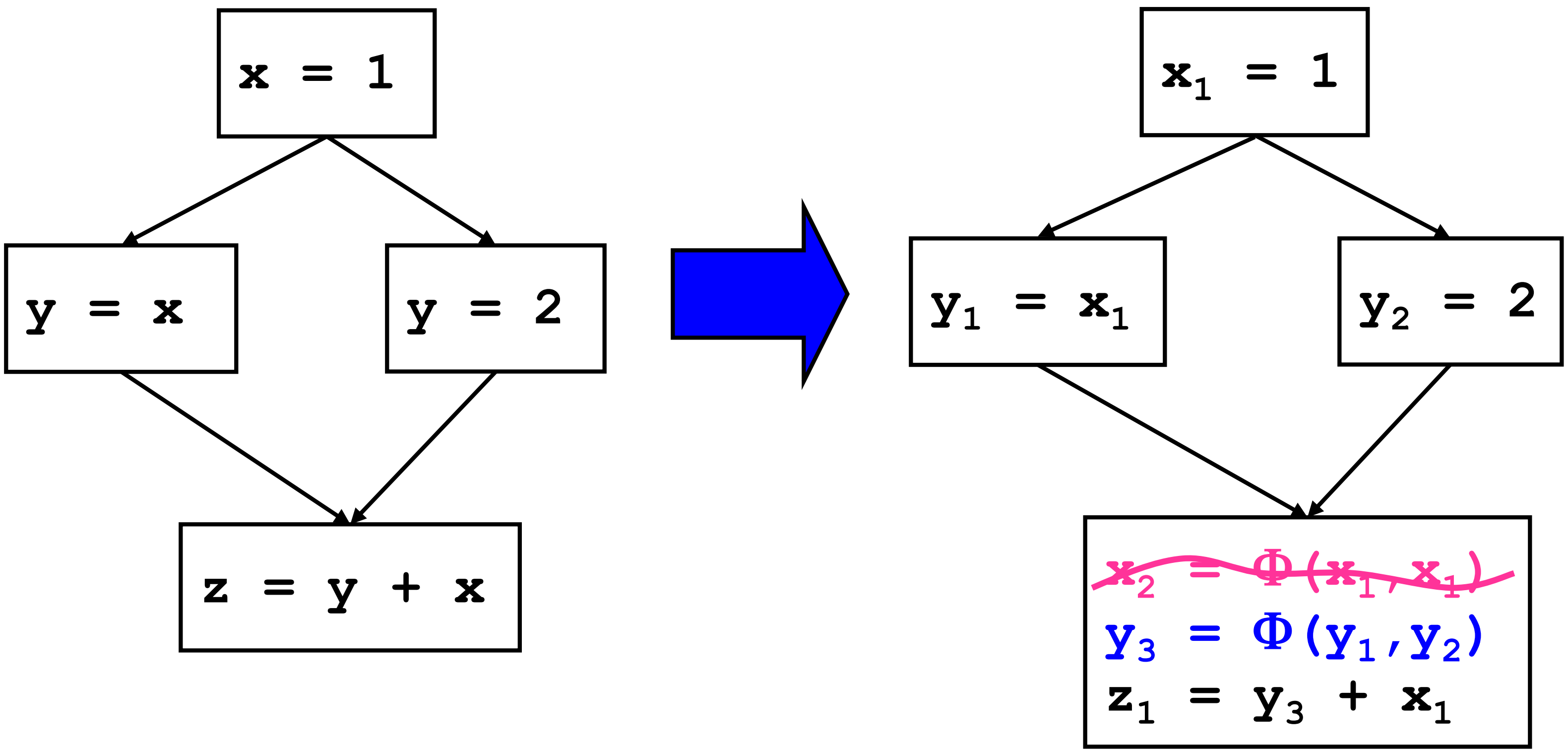
- Each assignment generates a fresh variable
- At each join point insert Φ functions for all live variables



In general, too many Φ functions inserted

Minimal SSA

- Each assignment generates a fresh variable
- At each join point insert Φ functions for all live variables with multiple outstanding defs



Today's Class

I. Basic blocks & Flow graphs

II. Abstraction 1: DAG

III. Abstraction 2: Value numbering

IV. Intro to SSA

Wednesday's Class

- LLVM Compiler: Further Details
  - Play around a bit with LLVM before class