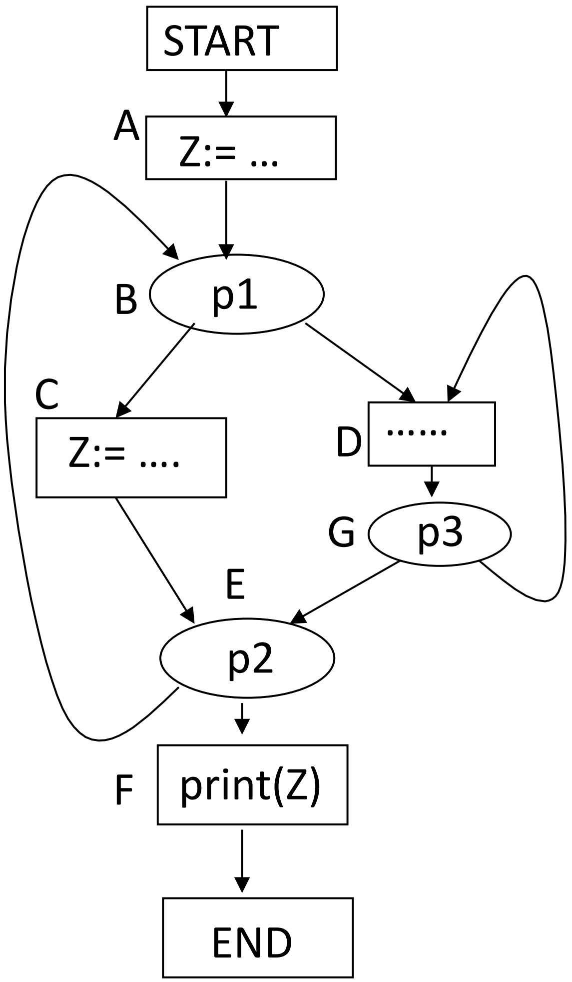
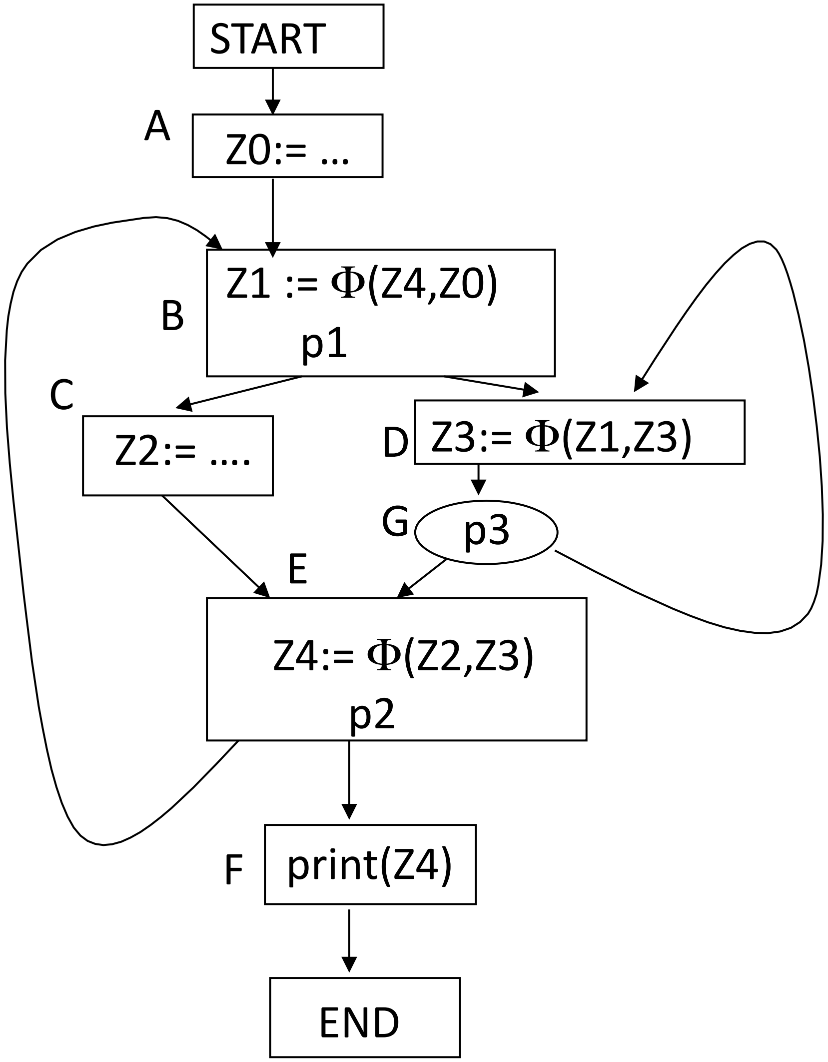
Static Single Assignment (SSA) Form

SSA form

- Static single assignment form
  - Intermediate representation of program in which every use of a variable is reached by exactly one definition
  - Most programs do not satisfy this condition
    - (eg) see program on next slide: use of Z in node F is reached by definitions in nodes A and C
  - Requires inserting dummy assignments called Φ-functions at merge points in the CFG to “merge” multiple definitions
  - Simple algorithm (see transformed example on next slide):
    - Insert Φ-functions for all variables at all merge points in the CFG
    - Solve Reaching Definitions
    - Rename each real and dummy assignment of a variable uniquely

SSA example


START
A
Z:= …
B
p1
C
Z:= ….
D
……
G
p3
E
p2
F
print(Z)
END


START
A
Z0:= …
Z1 := Φ(Z4,Z0)
B
p1
C
Z2:= ….
D
Z3:= Φ(Z1,Z3)
G
p3
E
Z4:= Φ(Z2,Z3)
p2
F
print(Z4)
END

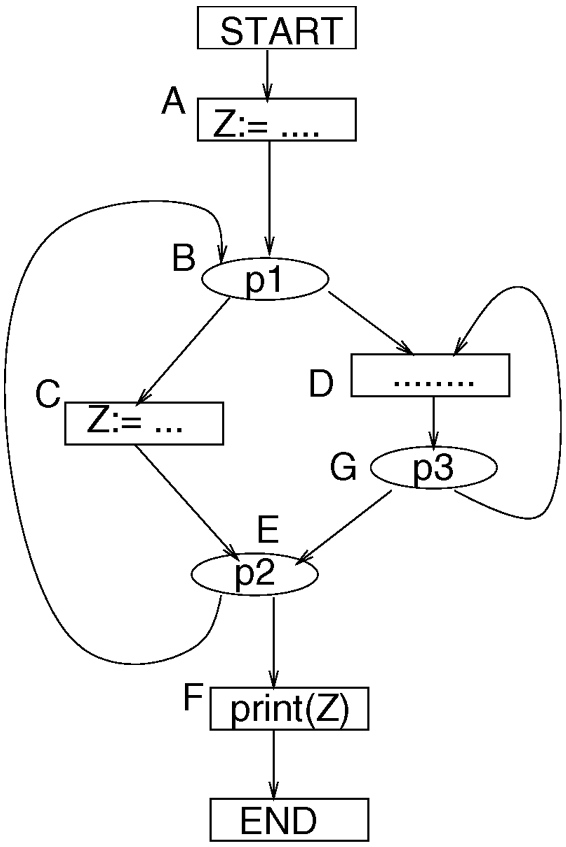
Minimal SSA form

- In previous example, dummy assignment Z3 is not really needed since there is no actual assignment to Z in nodes D and G of the original program

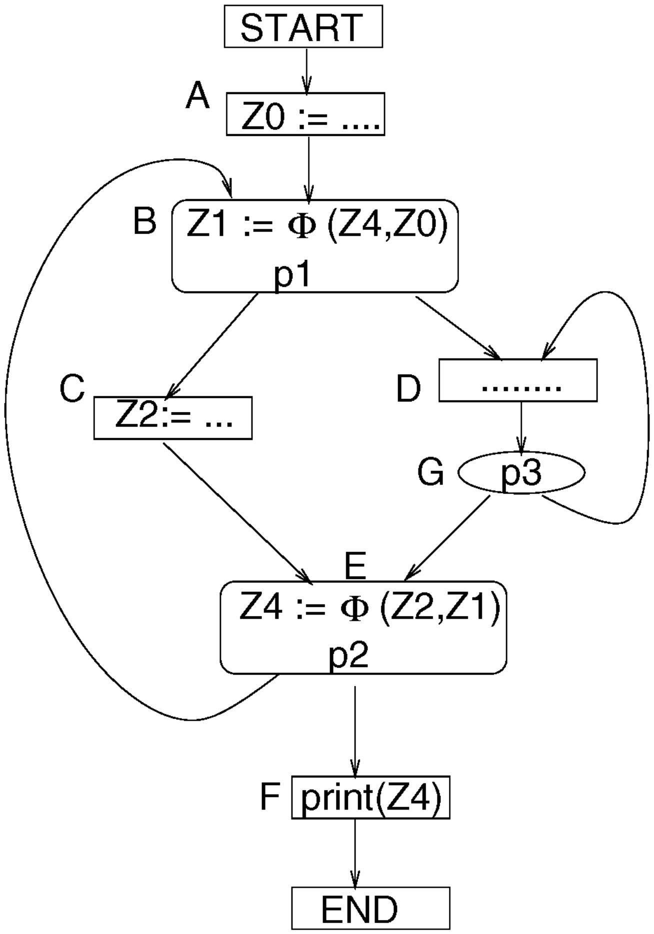
- Minimal SSA form
  - SSA form of program that does not contain such "unnecessary" dummy assignments
  - See example on next slide

- Question: how do we construct minimal SSA form directly?
  - Place ϕ-functions
  - Perform renaming

Minimal-SSA form Example



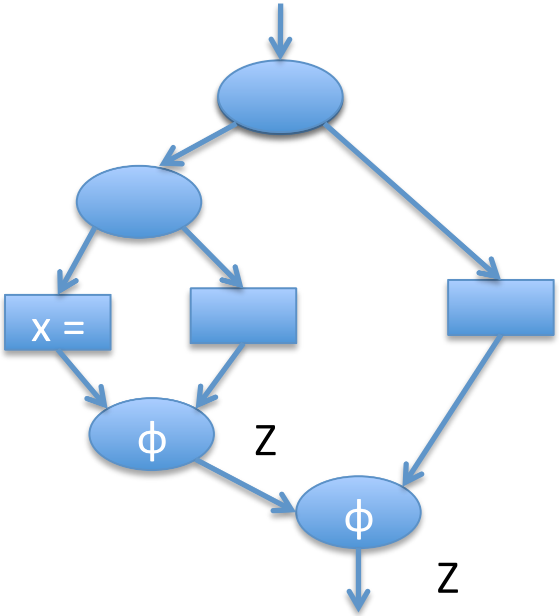
(a) Original Control Flow Graph



(b) Control Flow Graph with Φ-functions

Intuition for Φ-function Placement

- Compute Merge relation M: V → P(V)
- If node N contains an assignment to a variable x, then node Z is in M(N) if:
  1. There is a non-null path P1 := N $\rightarrow^{+}$ Z
     - The value computed at X reaches Z
  2. There is a non-null path P2 := START $\rightarrow^{+}$ Z
  3. P1 and P2 are disjoint except for Z



- If S ⊆ V where there are assignments to variable x, then place ϕ functions for x in nodes $\bigcup_{N \in S} M(N)$

Dominance frontier

- Dominance frontier of node *w*
  - Node *u* is in dominance frontier of node *w* if *w*
    - dominates a CFG predecessor *v* of *u*, but
    - does not strictly dominate *u*
- Dominance frontier = control dependence in reverse graph!

Example from previous slide

	A	B	C	D	E	F	G
A							
B		x					
C					x		
D				x			
E		x					
F							
G					x		

Iterated dominance frontier

- Irreflexive transitive closure of dominance frontier relation
- Related notion: iterated control dependence in reverse graph

- Where to place Φ-functions for a variable Z
  - Let Assignments = {START} U {nodes with assignments to Z in original CFG}
  - Find set I = iterated dominance frontier of nodes in Assignments
  - Place Φ-functions in nodes of set I

- For example
  - Assignments = {START,A,C}
  - DF(Assignments) = {E}
  - DF(DF(Assignments)) = {B}
  - DF(DF(DF(Assignments))) = {B}
  - So I = {E,B}
  - This is where we place Φ-functions, which is correct

Variable Renaming

- Use in a non-ϕ statement:
  - Use immediately dominating definition of V (+ ϕ nodes inserted for V)
- Use in a ϕ operand:
  - Use definition that immediately dominates incoming CFG edge (not ϕ)

Computing SSA form

- Cytron et al algorithm
  - compute DF relation (see slides on computing control-dependence relation)
  - find irreflexive transitive closure of DF relation for set of assignments for each variable
- Computing full DF relation
  - Cytron et al algorithm takes O(|V| +|DF|) time
  - |DF| can be quadratic in size of CFG
- Faster algorithms
  - O(|V|+|E|) time per variable: see Bilardi and Pingali

Using SSA for Optimization

Constant Propagation as an Example

(i)

```
...
x := 1;
y := x + 2;
if (x>z) then y:= 5; fi
... y ...
```

⇒

```
...
x := 1;
y := 3;
if (1>z) then y:= 5; fi
... y ...
```

Constant propagation may simplify control flow as well

(ii)

```
...
x := 1;
y := x + 2;
if (y>x) then y:= 5; fi
... y ...
```

⇒

```
...
x := 1;
y := 3;     ← dead code
if (true) then y:= 5; fi
... 5 ...
```

Overview of algorithm

- Build CFG of program
  - makes control flow explicit
- Perform “symbolic evaluation” to determine constants
- Replace constant-valued variables uses by their values and simplify expressions and control-flow

Step 1: Build the CFG

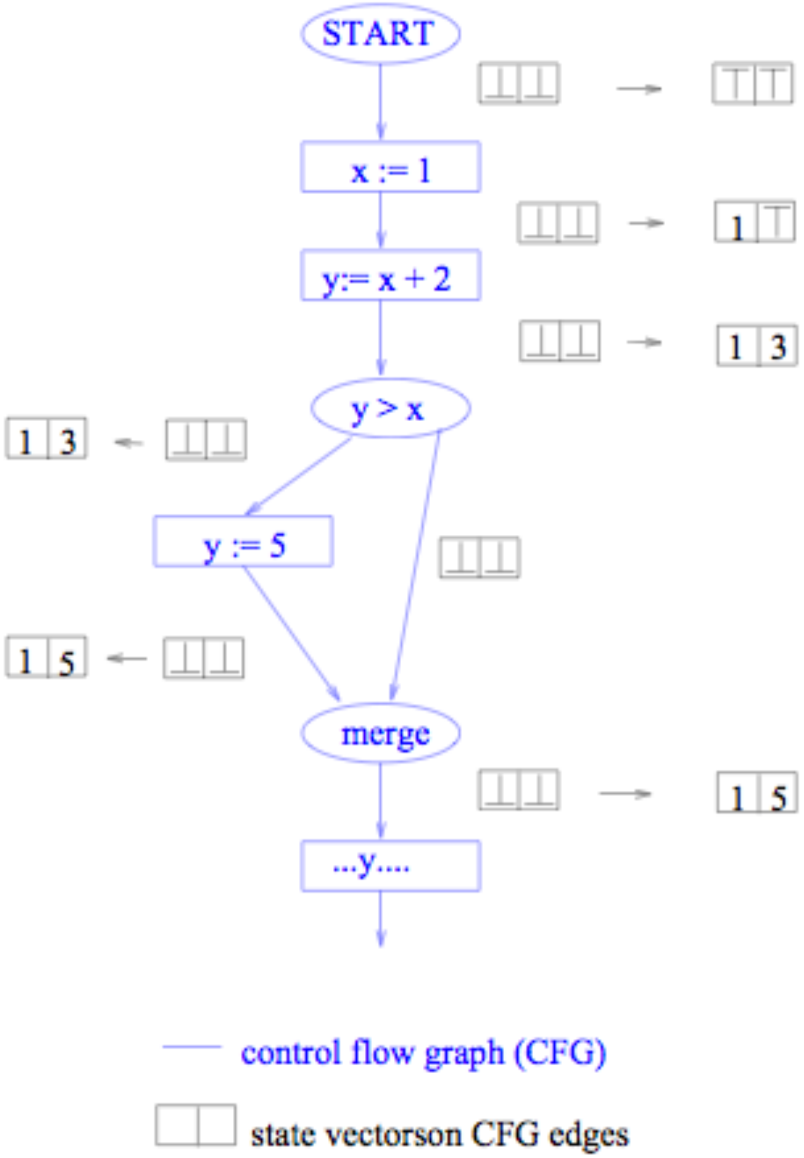
...

x := 1;

y := x + 2;

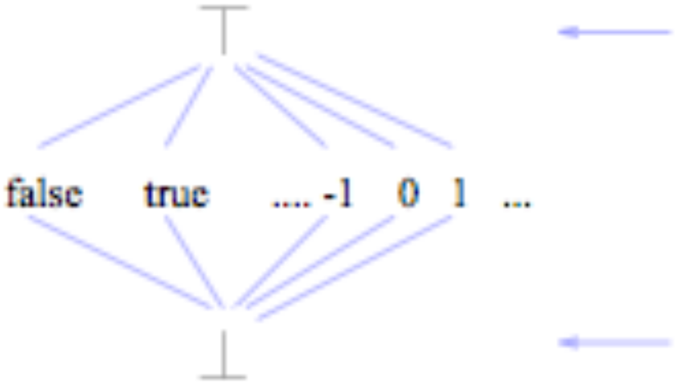
if (y>x***) then*** y:= 5; ***fi***

... y ...



Step 2: Symbolic Evaluation Over CFG

- Propagate values from following lattice



Definitely not constant/Can't determine to be a constant

join(T,0) = T
join(0,-1) = T

meet(0,-1) = ⊥
meet(T,1) = 1

Yet to be determined

- Two operators
  - Join(a,b): lowest value above both a and b (also written as a ∪ b)
  - Meet(a,b): highest value below both a and b (also written as a ∩ b)

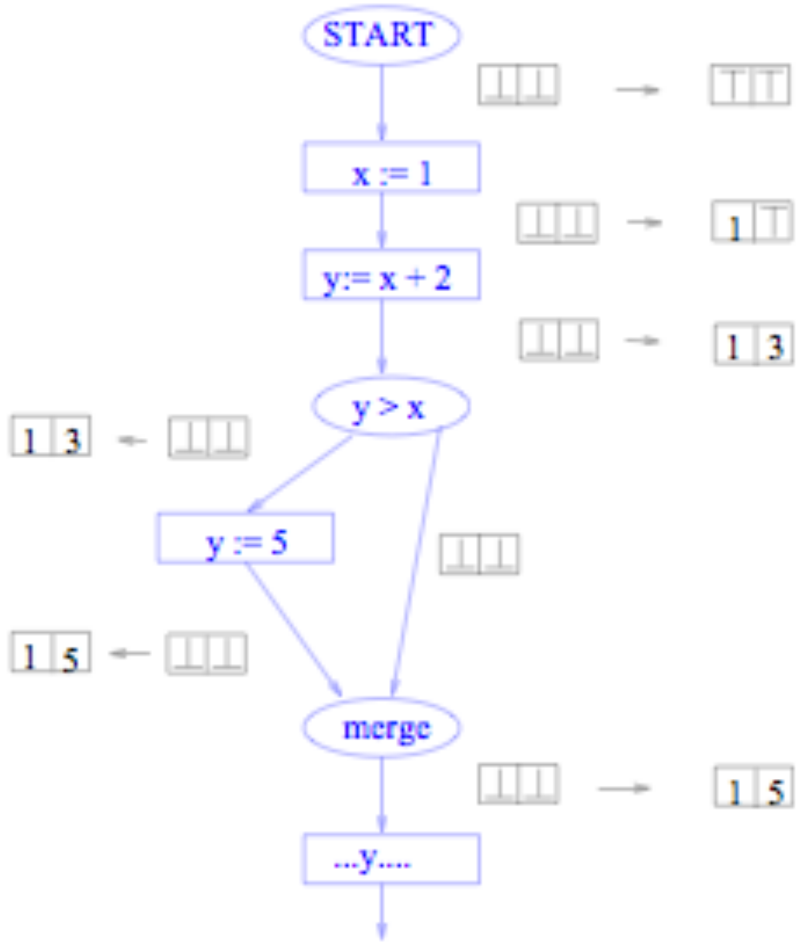
- Symbolic interpretation of expressions
  - EVAL(e, Vin): if any argument of e is T (or ⊥) in Vin, return T (or ⊥ respectively); otherwise, evaluate e normally and return the value

Dataflow Algorithm

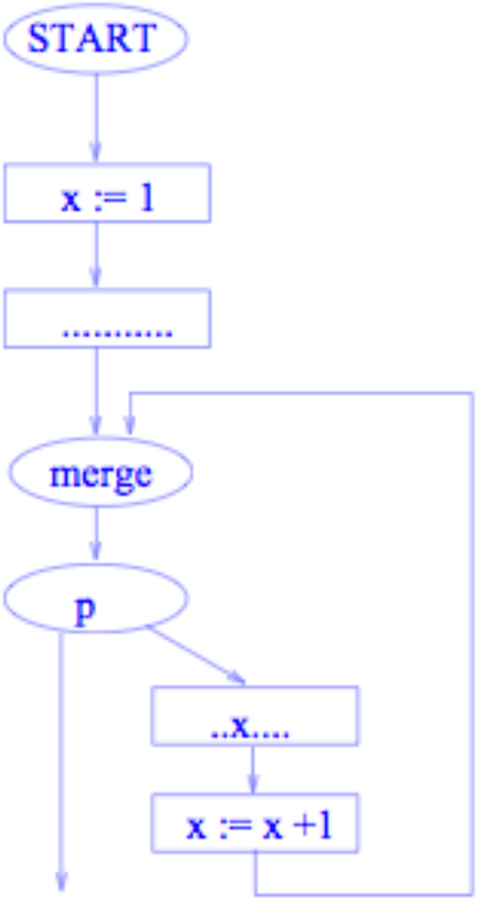
1. Associate one state vector with each edge of CFG
2. Set each entry of state vector on edge out of start to T, and place this edge in worklist

```
3.  while (worklist not empty) {
      Edge ed := worklist.getRandom();
      Vin := state-vector[ed]
      // Symbolically evaluate target node of the edge using state vectors on inputs
      //  and propagate result state vector to output edge of node
      if (target[ed] is "x:= e") {
        Propagate Vin[EVAL[e,Vin)/x] to output edge;
      } else if (target[ed] is "switch(p)") {
        if (EVAL(p, Vin) is T)
          Propagate Vin to all outputs of switch;
        else if (EVAL(p, Vin) is true)
          Propagate Vin to true side of switch;
        else
          Propagate Vin to false side of switch;
      } else // target node is merge
        Propagate join of state vectors on all inputs to output
      }
      If this changes output state vector, enqueue output edge on worklist
    }
```

Applying Algorithm on Running Example



Subtleties of Algorithm



First time through loop, use of *x* in loop is determined to be constant *1*. Next time though loop, it reaches final value T.

Algorithm Complexity

- Height of lattice := 2 → each state vector can change value *2*V* times
- So while loop in algorithm is executed at most *2*E*V* times
- Cost of each iteration: O(V)
- Overall algorithm takes $O(EV^2)$ time

Optimizing Constant Propagation

- Iterative procedure is just a method to solve lattice equations

- Optimize by exploiting sparsity in the dataflow equations
  - Usually, a dataflow equation involves only a small number of dataflow variables

Optimizing Constant Propagation

- Current algorithm uses the CFG to propagate state vectors
- Propagating information for all variables *in lock-step* forces a lot of *useless copying* of information from one vector to another
  - e.g. a variable defined at the top of the procedure and used only at the bottom

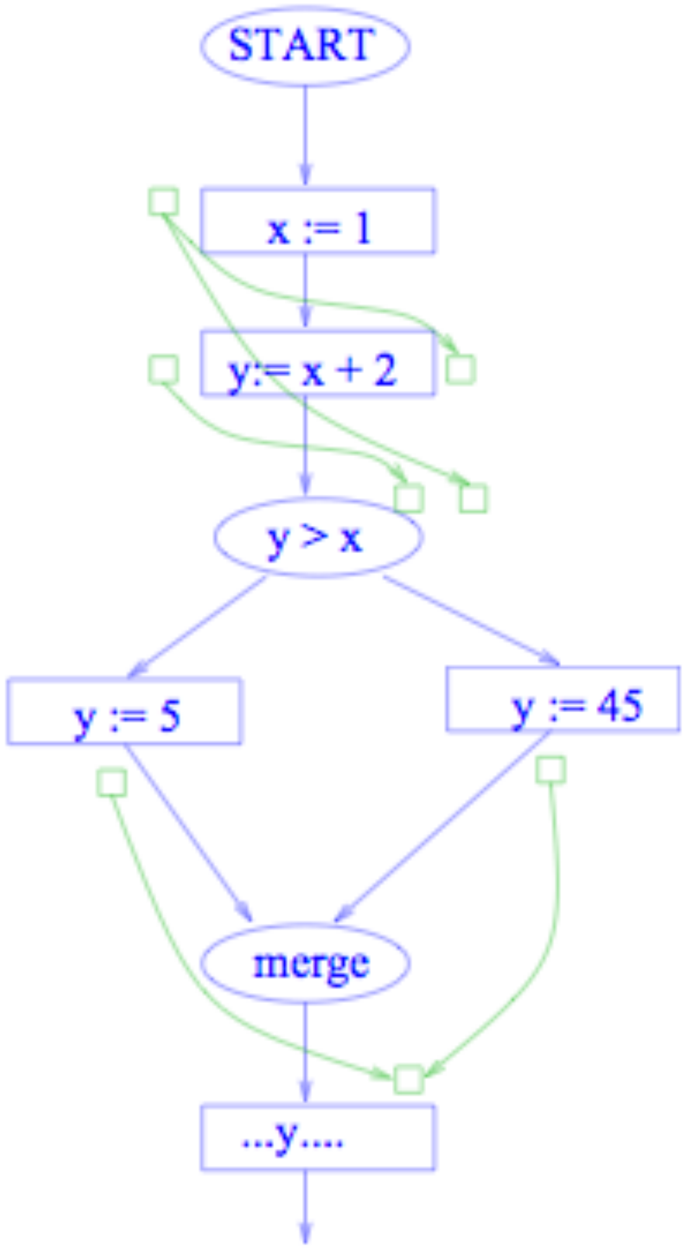
- Solution:
  - Do constant propagation for each variable separately
  - Propagate information *directly from definitions to uses*, skipping over irrelevant portions of control flow graph

Constant Propagation Using Def-Use Chains

1. Associate cell with each lhs and rhs occurrence of all variables, initialize to $\bot$
2. Propagate T along each def-use edge out of START, and enqueue target statements of def-use edges onto worklist
3. Enqueue all definitions with constant RHS onto worklist

```
4.  while (worklist not empty) {
      Def d := worklist.getNext();
      cell[LHS[d]] := Evaluate(RHS[d]) // using cell[Var], ∀ var in RHS[d]
      if (cell[LHS[d]] changes) {
        Propagate cell[LHS[d]] value along def-use chains to each use stmt
        //(take join of cell[LHS[d]] and cell value at use)
        if (cell[use] changes && use is definition)
          worklist.add(use)
      }
    }
```

Example



— control flow graph (CFG)

→ def-use edges

□ cell for value at definition/use

Analysis of Use-Def Based Constant Propagation

- Complexity: O(*sizeof(*def-use chains*)*)
  - This can be as large as O(N^2V), where N is # CFG-Nodes
  - With SSA this is reduced to O(EV)

- Problem with algorithm: **Loss of accuracy**
  - Propagation along def-use chains cannot determine directly that *y := 45* is dead code, so last use of *y* is not marked constant
  - We compute def-use chains before doing constant propagation, so we don't recognize dead code

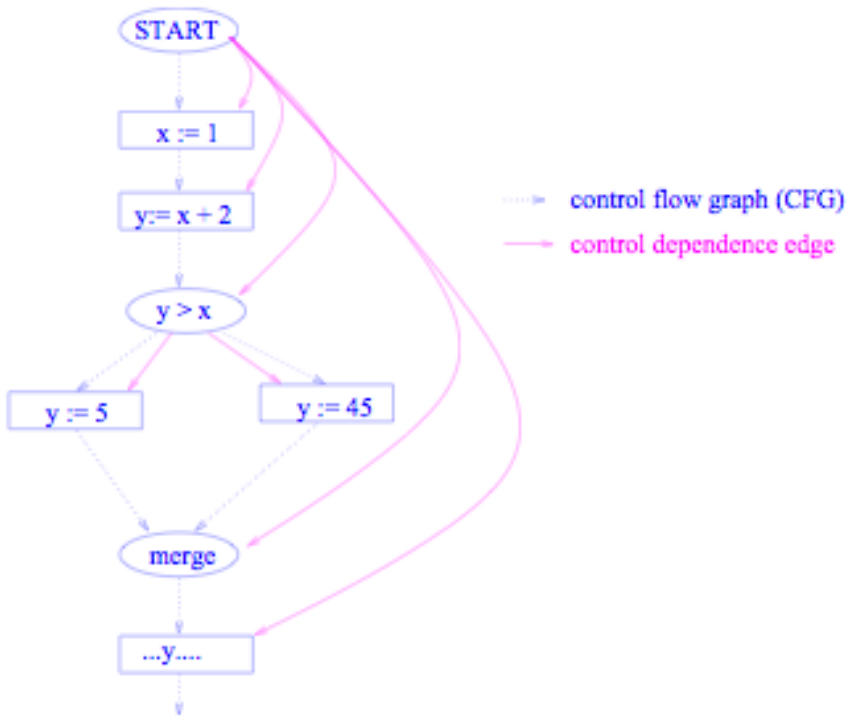
- Possible solution: Repeated cycles of reaching definitions computation, constant propagation and dead code elimination

- Is there a better way?

- Key idea:
  - Find unreachable statements during constant propagation
  - Do not propagate values out of unreachable definitions

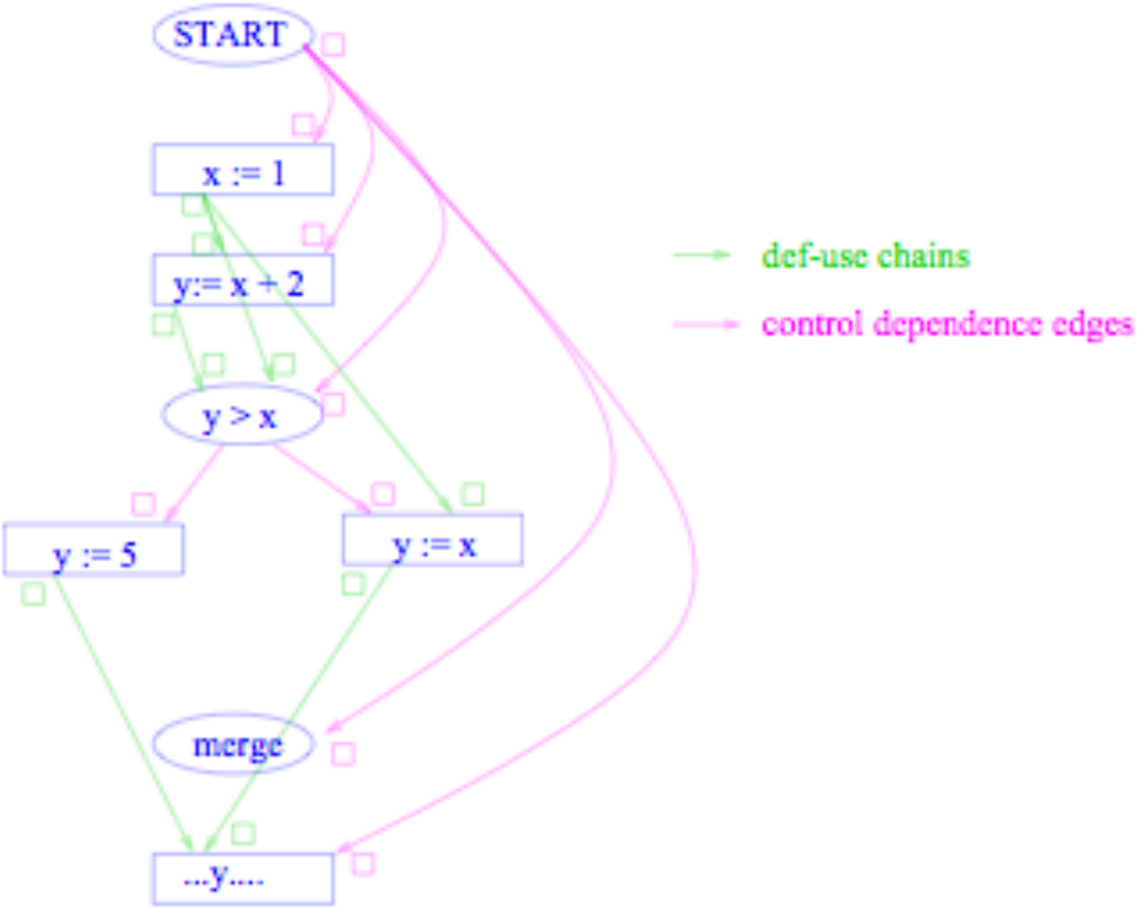
High Level View of Potential Solution

- Use Control Dependence and Def-Use chains
- Control Dependence:
  - Node *n* is control dependent on predicate *p* if *p* determines whether *n* is executed
- Convention: assume START is a predicate, so unconditionally executed statements are control dependent on START
- CDG: Control Dependence Graph



High Level Idea

Propagate "liveness" along control dependence edges while propagating constants along Def-Use chains



Revised Algorithm

1. Associate cell with each lhs and rhs occurrence of all variables and with each statement, initialize to $\bot$

2. Propagate T along each Def-Use edge and control dependence edge out of START. If value in any target cell changes, enqueue target statement onto worklist

3.
```
while (worklist not empty) {
  Stmt d := worklist.getNext();
  if (CDEP-cell[d] is T) {
    switch (type of d) {
      case(definition): {
        cell[LHS[d]] := Evaluate(RHS[d]) // using cell[Var], ∀ var in RHS[d]
        if (cell[LHS[d]] changes) {
          Propagate cell[LHS[d]] value along def-use chains to each use stmt
          //(take join of cell[LHS[d]] and cell value at use)
          if (cell[use] changes) // if cell value at use changes
            worklist.add(use)
        }
      }
      case(switch): {
        Evaluate predicate and propagate along appropriate CDEP edges out of predicate
        if (cell value at target changes)
          worklist.add(target)
      }
    }
  }
}
```

Observations

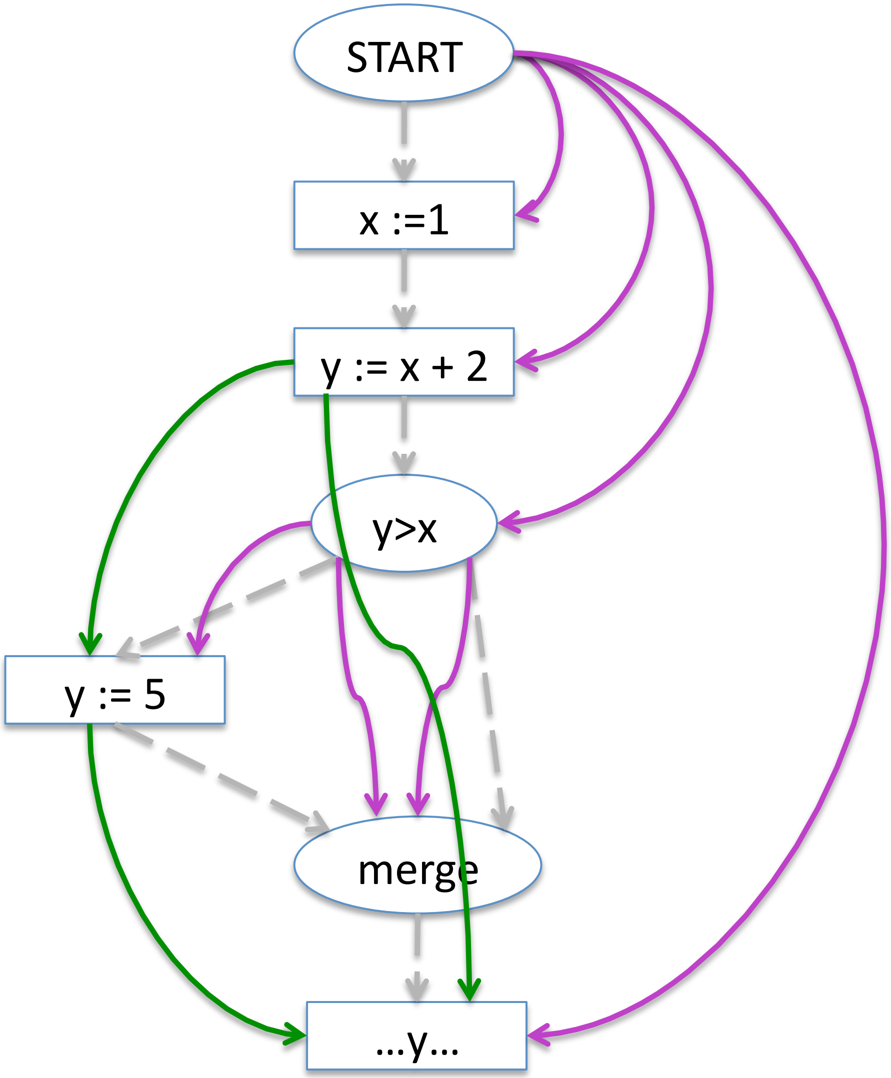
- We do not propagate information out of dead (unreachable) statements

- Precision is still not as good as CFG algorithm
  - We still propagate information out of statements that are executed but are irrelevant to output

- Need algorithm to compute control dependences in general graph

- Size of CDG: O(EN) (can be reduced)

Problematic Case



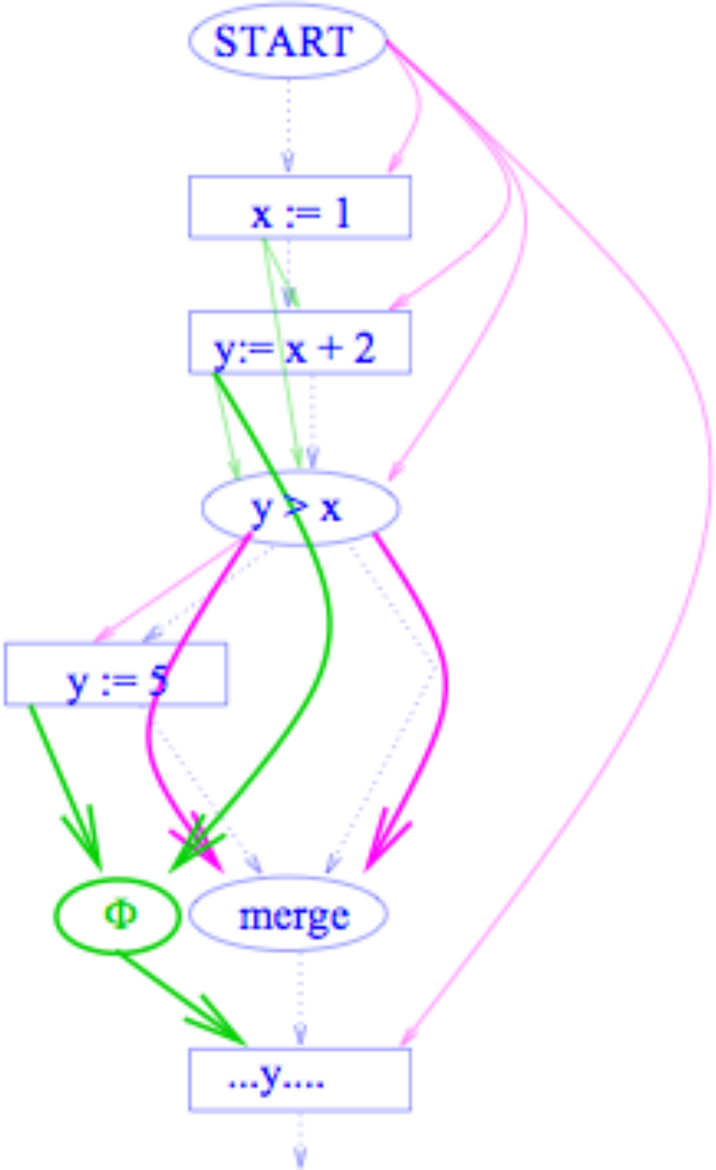
Solutions

- Require that a variable assigned on one side of a conditional be assigned on both sides of conditional (by inserting dummy assignments of form *x:= x*). Programmers don't want to do this

- Make compiler insert dummy assignments. Hard to figure out in presence of unstructured control flow

- Use SSA form: ensure that every use is reached by exactly one definition by inserting ϕ-functions at merges to combine reaching definitions

SSA Algorithm for Constant Propagation



- ϕ-function combines different reaching definitions at a merge into a single one at output of merge
- ϕ-function is like a pseudo-assignment
- Control dependence at merge: compute for each side of the merge separately

- Constant propagation:
  - Similar to previous algorithm, but at merge, propagate join of inputs only from live sides of merge

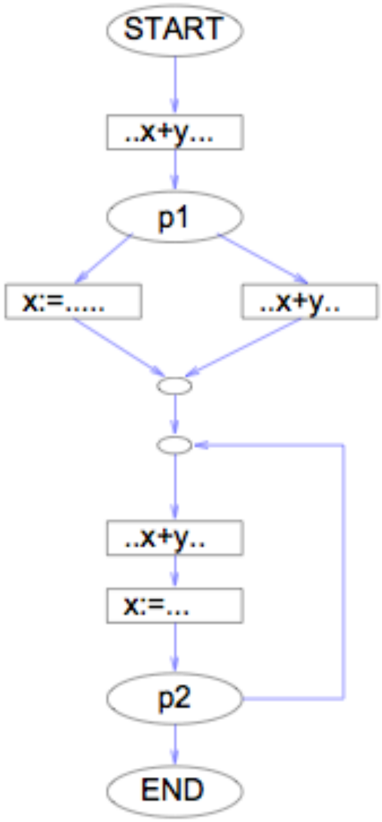
- Minimal SSA permits Def-Use chains to bypass a merge if same definition reaches all sides of merge

Sparse Dataflow Evaluator Graphs

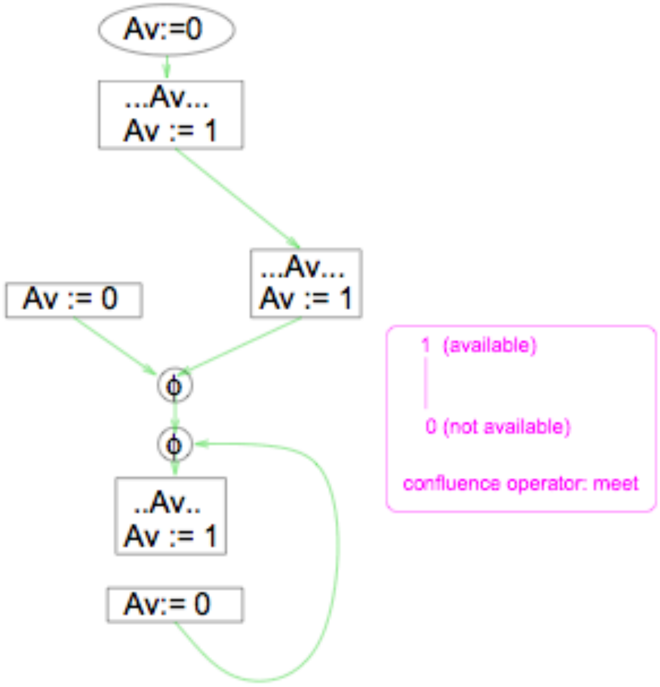
- Same idea can be applied to other dataflow problems
  - Perform dataflow for each sub-problem separately (e.g. for each expression separately in available expressions problem)
  - Build a sparse graph in which only statements that modify or use dataflow information for sub-problem are present and solve that

- Sparse dataflow evaluator graph can be built in O(|E|) time per problem (Pingali & Bilardi PLDI'96)

Sparse Dataflow Evaluator Graphs



Control Flow Graph



Sparse Dataflow Evaluator Graph
for availability of x+y

When is SSA form useful?

- For many dataflow problems, SSA form enables sparse dataflow analysis that
  - yields the same precision as bit-vector CFG-based dataflow analysis
  - but is asymptotically faster since it permits the exploitation of sparsity

- SSA has two distinct features
  - factored def-use chains (more compact than base def-use)
  - renaming
  - you do not have to perform renaming to get advantage of SSA for many dataflow problems

- The bit-vector approach allows an implicit form of parallelism to be exploited

- When a problem is not formulated using the bit-vector approach, SSA is preferable
  - Constant propagation
  - Useful in pointer analysis
  - Value numbering