Dataflow Analysis

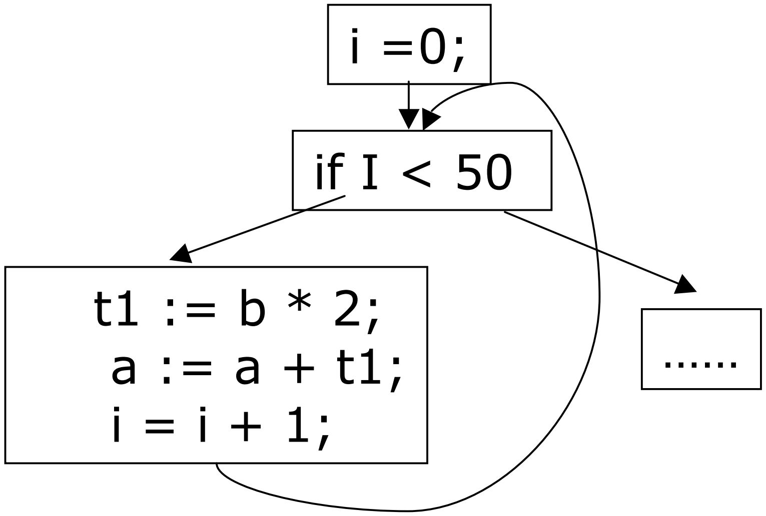
Iterative Data-flow Analysis and Static-Single-Assignment

Optimization And Analysis

- Improving efficiency of generated code
  - Correctness: optimized code must preserve meaning of the original program
  - Profitability: optimized code must improve code quality
- Program analysis
  - Ensure safety and profitability of optimizations
  - Compile-time reasoning of runtime program behavior
    - Undecidable in general due to unknown program input
    - Conservative approximation of program runtime behavior
    - May miss opportunities, but ensure all optimizations are safe
- Data-flow analysis
  - Reason about flow of values between statements
  - Can be used for program optimization or understanding

Control-Flow Graph

- Graphical representation of runtime control-flow paths
  - Nodes of graph: basic blocks (straight-line computations)
  - Edges of graph: flows of control
- Useful for collecting information about computation
  - Detect loops, remove redundant computations, register allocation, instruction scheduling…
- Alternative CFG: Each node contains a single statement



i =0;

if I < 50

t1 := b * 2;
a := a + t1;
i = i + 1;

……

Building Control-Flow Graphs Identifying Basic Blocks

- Input: a sequence of three-address statements
- Output: a list of basic blocks
- Method:
  - Determine each statement that starts a new basic block, including
    - The first statement of the input sequence
    - Any statement that is the target of a goto statement
    - Any statement that immediately follows a goto statement
  - Each basic block consists of
    - A starting statement S0 (leader of the basic block)
    - All statements following S0 up to but not including the next starting statement (or the end of input)

```
……
     i := 0
s0: if i < 50 goto s1
     goto s2
s1: t1 := b * 2
      a := a + t1
      goto s0
S2: …
```

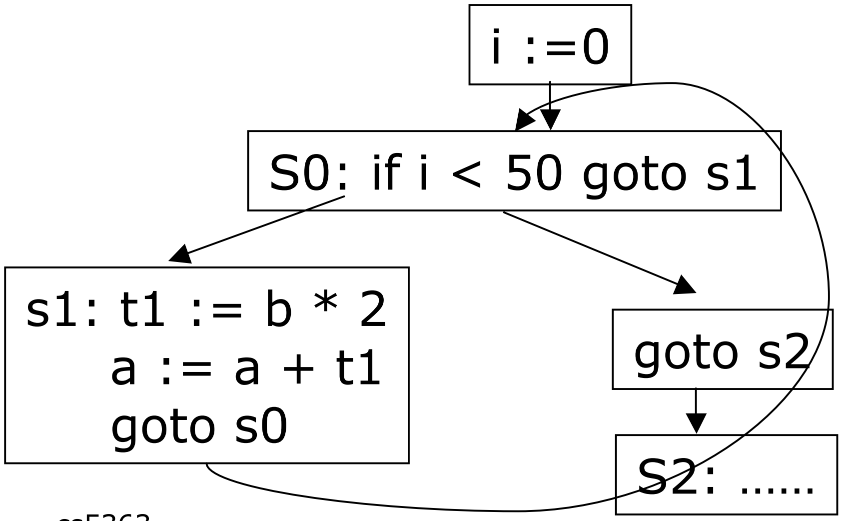
Starting statements:

i := 0
S0,
goto S2
S1,
S2

Building Control-Flow Graphs

- Identify all the basic blocks
  - Create a flow graph node for each basic block
- For each basic block B1
  - If B1 ends with a jump to a statement that starts basic block B2, create an edge from B1 to B2
  - If B1 does not end with an unconditional jump, create an edge from B1 to the basic block that immediately follows B1 in the original evaluation order

```
......
    i := 0
s0: if i < 50 goto s1
    goto s2
s1: t1 := b * 2
     a := a + t1
     goto s0
S2: …
```



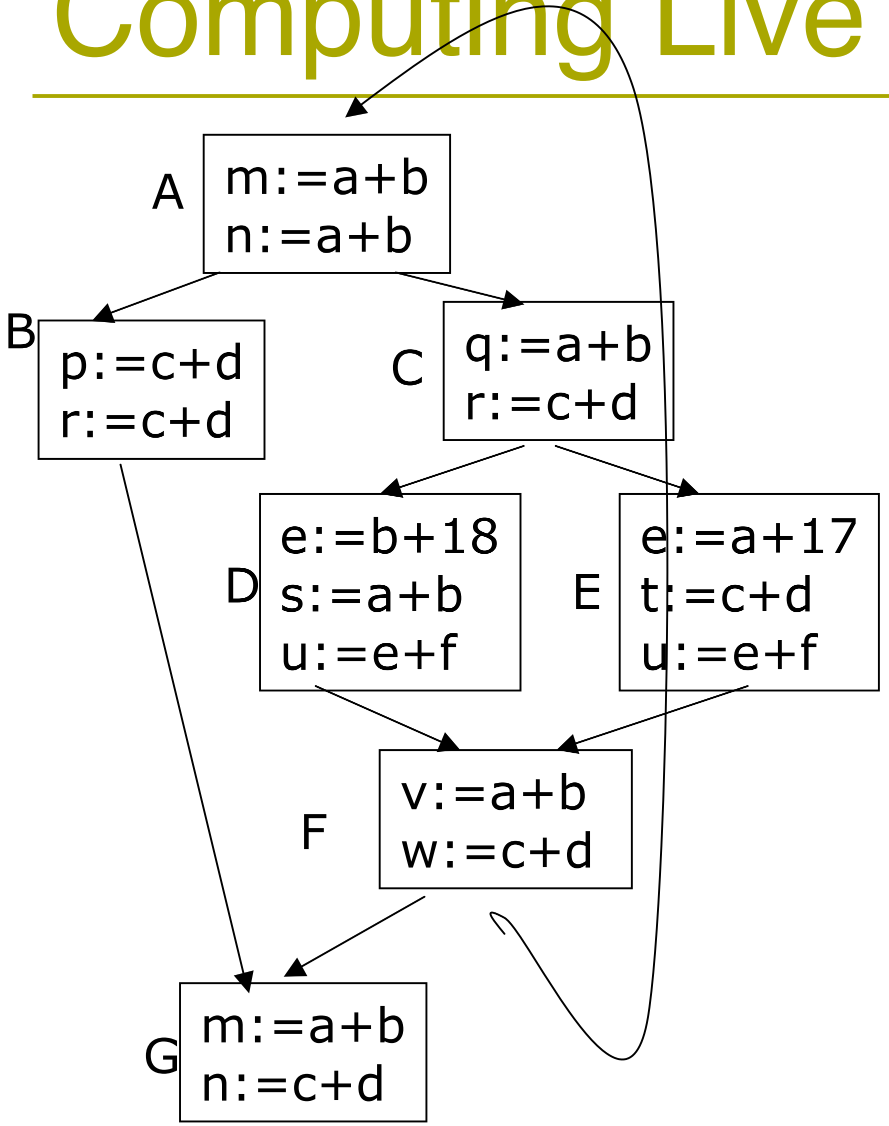
Exercise:
Building Control-flow Graph

```
……
     i = 0; z = x
    while (i < 100) {
     i = i + 1;
     if (y < x) z=y;
     A[i]=i;
}
….
```

Live Variable Analysis

- A data-flow analysis problem
  - A variable v is live at CFG point p iff there is a path from p to a use of v along which v is not redefined
  - At any CFG point p, what variables are alive?
- Live variable analysis can be used in
  - Global register allocation
    - Dead variables no longer need to be in registers
  - SSA (static single assignment) construction
    - Dead variables don’t need ∅-functions at CFG merge points
  - Useless-store elimination
    - Dead variables don’t need to be stored back in memory
  - Uninitialized variable detection
    - No variable should be alive at program entry point

Computing Live Variables



A
m:=a+b
n:=a+b

B
p:=c+d
r:=c+d

C
q:=a+b
r:=c+d

D
e:=b+18
s:=a+b
u:=e+f

E
e:=a+17
t:=c+d
u:=e+f

F
v:=a+b
w:=c+d

G
m:=a+b
n:=c+d

- Domain:
  - All variables inside a function
- Goal: Livein(n) and LiveOut(n)
  - Variables alive at each basic block n
- For each basic block n, compute
  - UEVar(n)

    vars used before defined
  - VarKill(n)

    vars defined (killed by n)
- Formulate flow of data

  $\mathrm{LiveOut}(n)=\bigcup_{m\in succ(n)}\mathrm{LiveIn}(m)$

  $\mathrm{LiveIn}(m)=\mathrm{UEVar}(m) \cup (\mathrm{LiveOut}(m)-\mathrm{VarKill}(m))$

  ==>

  $\mathrm{LiveOut}(n)= \bigcup_{m\in succ(n)} (\mathrm{UEVar}(m) \cup (\mathrm{LiveOut}(m)-\mathrm{VarKill}(m))$

Algorithm: Computing Live Variables

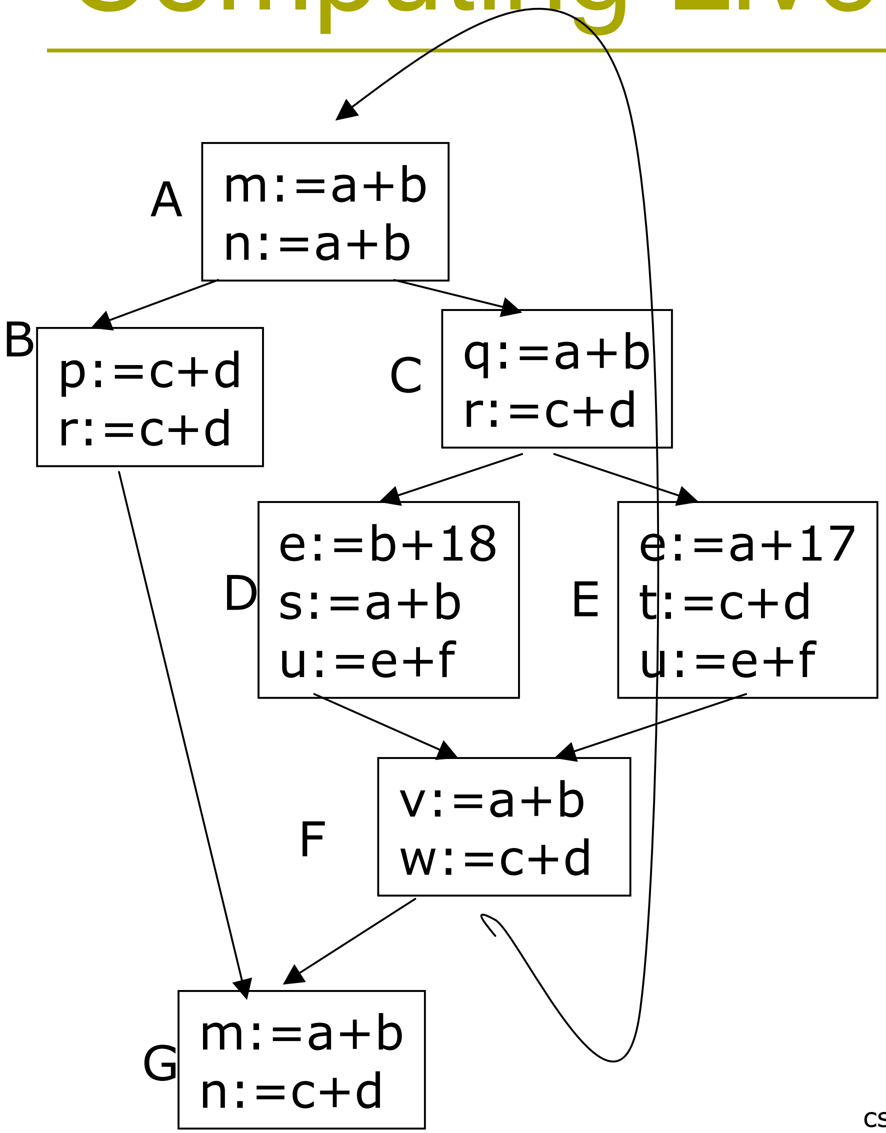
- For each basic block n, let
  - UEVar(n)=variables used before any definition in n
  - VarKill(n)=variables defined (modified) in n (killed by n)

Goal: evaluate names of variables alive on exit from n

- $\text{LiveOut}(n) = \bigcup_{m \in \text{succ}(n)} (\text{UEVar}(m) \cup (\text{LiveOut}(m) - \text{VarKill}(m))$

```
for each basic block bi
    compute UEVar(bi) and VarKill(bi)
    LiveOut(bi) := $\emptyset$
for (changed := true; changed; )
    changed = false
    for each basic block bi
        old = LiveOut(bi)
        LiveOut(bi)= $\bigcup_{m \in succ(bi)}$ (UEVar(m) $\cup$ (LiveOut(m) - VarKill(m))
        if (LiveOut(bi) != old) changed := true
```

Solution
Computing Live Variables



A
m:=a+b
n:=a+b

B
p:=c+d
r:=c+d

C
q:=a+b
r:=c+d

D
e:=b+18
s:=a+b
u:=e+f

E
e:=a+17
t:=c+d
u:=e+f

F
v:=a+b
w:=c+d

G
m:=a+b
n:=c+d

- Domain
  - a,b,c,d,e,f,m,n,p,q,r,s,t,u,v,w

	UE var	Var kill	Live Out	Live Out	Live Out
A	a,b	m,n	∅	a,b,c,d,f	a,b,c,d,f
B	c,d	p,r	∅	a,b,c,d	a,b,c,d
C	a,b, c,d	q,r	∅	a,b,c,d,f	a,b,c,d,f
D	a,b,f	e,s,u	∅	a,b,c,d	a,b,c,d,f
E	a,c, d,f	e,t,u	∅	a,b,c,d	a,b,c,d,f
F	a,b, c,d	v,w	∅	a,b,c,d	a,b,c,d,f
G	a,b, c,d	m,n	∅	∅	∅

Other Data-Flow Problems Reaching Definitions

- Domain of analysis
  - The set of definition points in a procedure
- Reaching definition analysis
  - A definition point d of variable v reaches CFG point p iff
    - There is a path from d to p along which v is not redefined
  - At any CFG point p, what definition points can reach p?
- Reaching definition analysis can be used in
  - Build data-flow graphs: where each operand is defined
  - SSA (static single assignment) construction
    - An IR that explicitly encodes both control and data flow

Reaching Definition Analysis

- For each basic block n, let
  - DEDef(n)= definition points whose variables are not redefined in n
  - DefKill(n)= definitions obscured by redefinition of the same name in n
- Goal: evaluate all definition points that can reach entry of n
  - Reaches_exit(m)= DEDef(m) ∪ (Reaches_entry(m) - DefKill(m))
  - Reaches_entry(n)= $\bigcup_{m \in pred(n)}$ Reaches_exit(m)

Example

```
void fee(int x, int y)
{
        int I = 0;
        int z = x;
        while (I < 100) {
           I = I + 1;
           if (y < x) z = y;
           A[I] = I;
        }
}
```

- Compute the set of reaching definitions at the entry and exit of each basic block through each iteration of the data-flow analysis algorithm

More About Dataflow Analysis

- Sources of imprecision
  - Unreachable control flow edges, array and pointer references, procedural calls
- Other data-flow programs
  - Very busy expression analysis
    - An expression e is very busy at a CFG point p if it is evaluated on every path leaving p, and evaluating e at p yields the same result.
    - At any CFG point p, what expressions are very busy?
  - Constant propagation analysis
    - A variable-value pair (v,c) is valid at a CFG point p if on every path from procedure entry to p, variable v has value c
    - At any CFG point p, what variables have constants?

The Overall Pattern

- Each data-flow analysis takes the form

  Input(n) := ∅ if n is program entry/exit

  := Λ m∈Flow(n) Result(m) otherwise

  Result(n) = ƒn (Input(n))

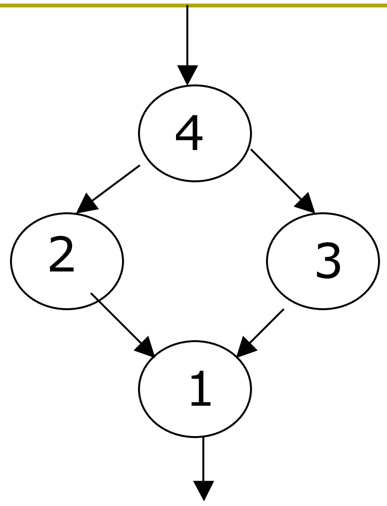
  - Λ is ∩ or ∪ (may vs. must analysis)
    - May analysis: properties satisfied by at least one path (∪)
    - Must analysis: properties satisfied by all paths(∩)
  - Flow(n) is pred(n) or succ(n) (forward vs. backward flow)
    - Forward flow: data flow forward along control-flow edges.
      - Input(n) is data entering n, Result is data exiting n
      - Input(n) is ∅ if n is program entry
    - Backward flow: data flow backward along control-flow edges.
      - Input(n) is data exiting n, Result is data entering n
      - Input(n) is ∅ if n is program exit
  - ƒn is the transfer function associated with each block n

Iterative dataflow algorithm

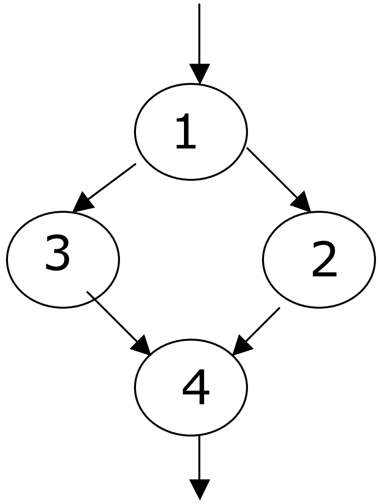
```
for each basic block bi
    compute Gen(bi) and Kill(bi)
    Result(bi) := ∅
for (changed := true; changed; )
    changed = false
    for each basic block bi
        old = Result(bi)
        Result(bi)=
        ∩ or ∪
    [m∈pred(bi) or succ(bi)]
  (Gen(m) ∪ (Result(m)-Kill(m))
        if (Result(bi) != old)
            changed := true
```

- Iterative evaluation of result until a fixed point is reached
    - Always terminate?
        - If the results are bounded and grow monotonically, then yes; Otherwise, no.
        - Fixed-point solution is independent of evaluation order
    - What answer is computed?
        - Unique fixed-point solution
        - Meet-over-all-paths solution
    - How long does it take the algorithm to terminate?
        - Depends on traversing order of basic blocks

Traverse Order Of Basic Blocks



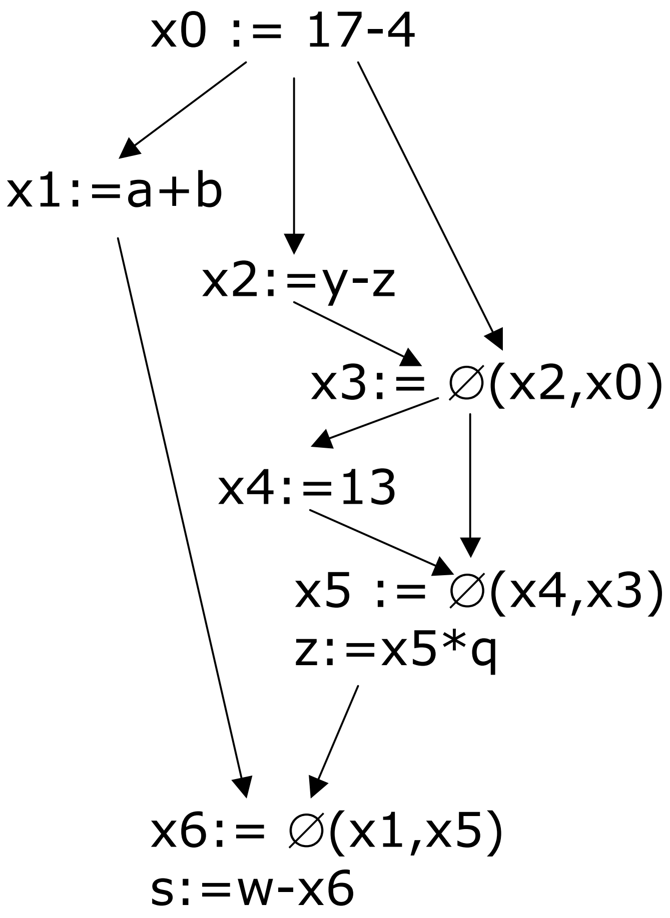
postorder



Reverse postorder

- Facilitate fast convergence to the fixed point
- Postorder traversal
  - Visits as many of a node’s successors as possible before visiting the node
  - Used in backward data-flow analysis
- Reverse postorder traversal
  - Visits as many of a node’s predecessors as possible before visiting the node
  - Used in forward data-flow analysis

Static Single Assignment form



x0 := 17-4

x1:=a+b

x2:=y-z

x3:= ∅(x2,x0)

x4:=13

x5 := ∅(x4,x3)
z:=x5*q

x6:= ∅(x1,x5)
s:=w-x6

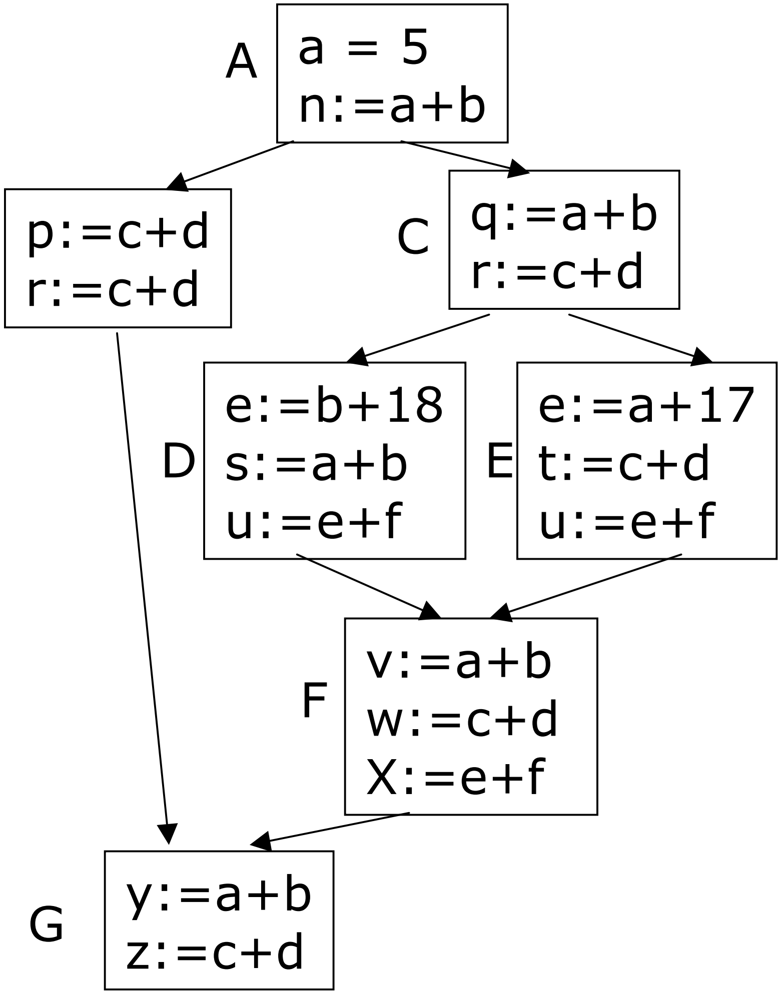
- Data-flow analysis
  - Analyze data flow properties on control flow graph
  - Each analysis needs several passes over CFG
- Static Single Assignment form
  - Encode both control-flow and data-flow in a single IR
    - An intermediate representation
  - Each variable is assigned exactly once
    - Each use of variable has a single definition
- Steps:
  - Rename redefinition of variables
  - Use ∅-functions to merge conflicting definitions when paths meet

Construction Of SSA form

- Naïve algorithm: maximum SSA
  - Many extraneous ∅-functions are inserted
  - Need better algorithm to insert ∅-functions only when needed

```
(1)Insert ∅-functions
    for every basic  block bi that has multiple predecessors
        for each variable y used in bi
            insert ∅-function y = ∅(y,y,…y),
                where each y in ∅ corresponds to a predecessor
(2) Renaming
    Compute reaching definitions on CFG
        Each variable use has only one reachable definition
    Rename all definitions so that each defines a different name
    Rename all uses of variables according to its definition point
```

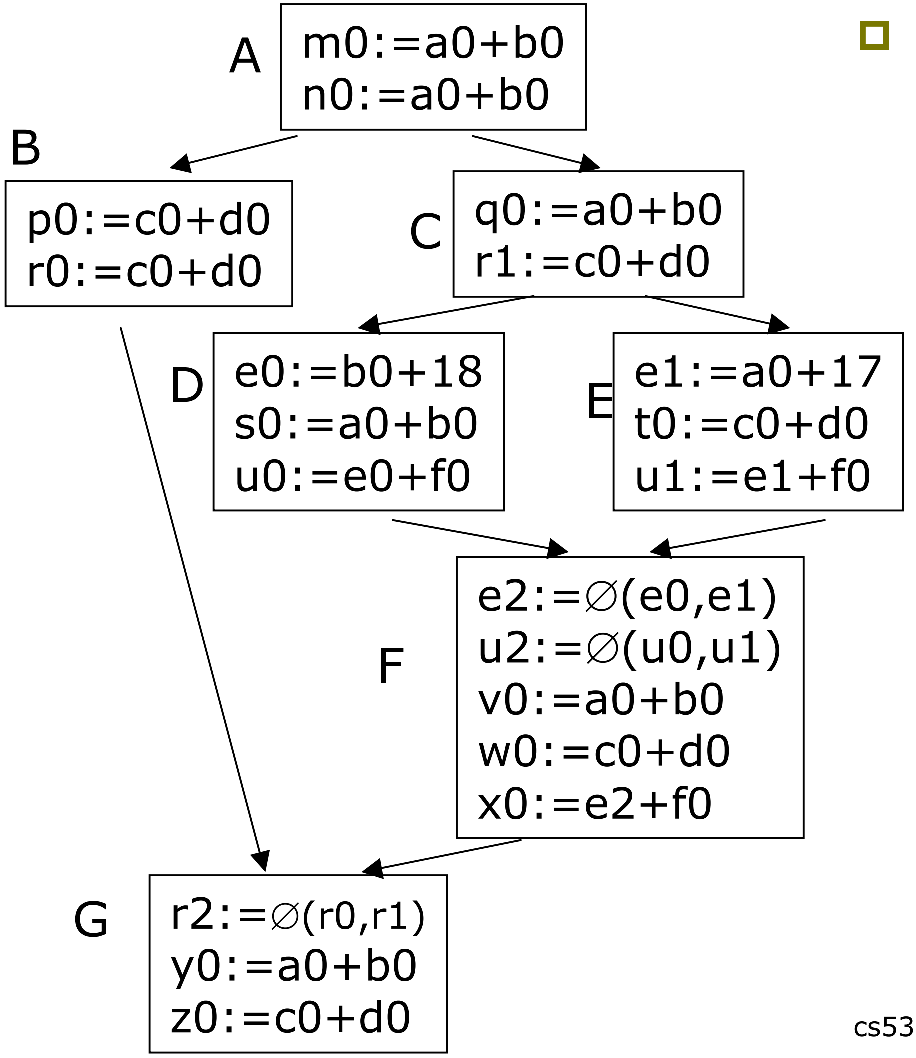
Dominators



- For each basic block y
  - x dominates y ($x \in Dom(y)$) if
    - x appears on all paths from entry to y
  - x strictly dominates y if
    - $x \in Dom(y)$ and $x \neq y$
    - i.e. $x \in Dom(y)-\{y\}$
  - x immediately dominates y if
    - $x \in Dom(y)$ and
      $\forall z \in Dom(y), z \in Dom(x)$
    - Written as x = IDom(y)
- Immediate dominators

  IDom(F)=C
  IDom(G)=A
  IDom(D)=C

Where to insert ∅-functions



A
m0:=a0+b0
n0:=a0+b0

B
p0:=c0+d0
r0:=c0+d0

C
q0:=a0+b0
r1:=c0+d0

D
e0:=b0+18
s0:=a0+b0
u0:=e0+f0

E
e1:=a0+17
t0:=c0+d0
u1:=e1+f0

F
e2:=∅(e0,e1)
u2:=∅(u0,u1)
v0:=a0+b0
w0:=c0+d0
x0:=e2+f0

G
r2:=∅(r0,r1)
y0:=a0+b0
z0:=c0+d0

- For variables defined in basic block n, which joint points in CFG need ∅-functions for them?
  - A definition in n forces a ∅-function just outside the region of CFG that n dominates
  - A ∅-function must be inserted at each dominance frontier of n

$m \in DF(n)$ iff

(1) n dominates a predecessor of m

$\exists\, q \in preds(m)$ s.t. $n \in Dom(q)$

(2) n does not strict dominate m

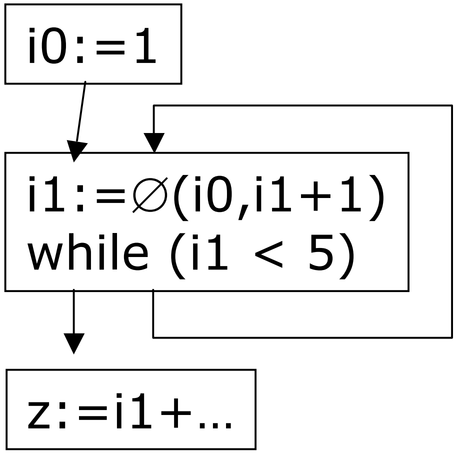
$m \notin Dom(n) - \{n\}$

Example: Constructing SSA

```
void fee(int x, int y)
{
        int I = 0;
        int z = x;
        while (I < 100) {
           I = I + 1;
           if (y < x) z = y;
           A[I] = I;
        }
}
```

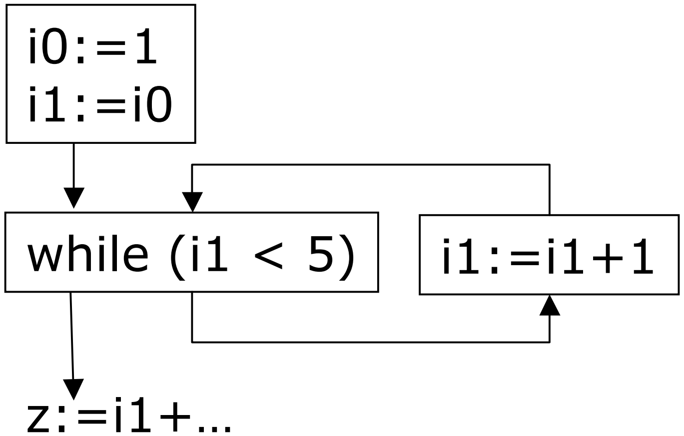
Reconstructing Executable Code

- SSA form is not directly executable on machines
  - Must rewrite ∅-functions into copy instructions
    - Need to split incoming edges of each ∅-function
    - Need to break cycles in ∅-function references
  - Rewriting made complex by SSA transformations
    - All phi functions of the same join point need to be evaluated concurrently


i0:=1

i1:=∅(i0,i1+1)
while (i1 < 5)

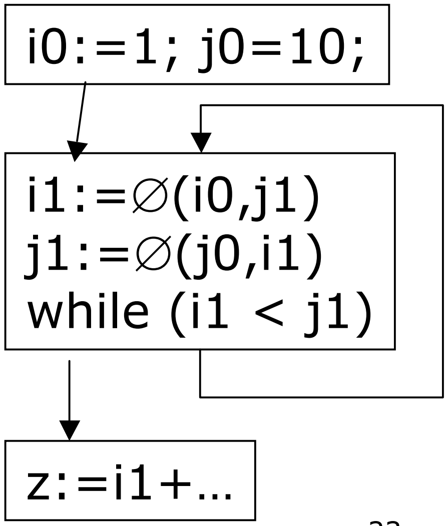
z:=i1+…


i0:=1
i1:=i0

while (i1 < 5)

i1:=i1+1

z:=i1+…


i0:=1; j0=10;

i1:=∅(i0,j1)
j1:=∅(j0,i1)
while (i1 < j1)

z:=i1+…

Appendix:Very Busy Expressions

- Domain of analysis
  - Set of expressions in a procedure
  - An expression e is very busy at a CFG point p if it is evaluated on every path leaving p, and evaluating e at p yields the same result.
  - At any CFG point p, what expressions are very busy?
- If an expression e is very busy at p, we can evaluate e at p and then remove all future evaluation of e.
  - Code hoisting --- reduces code space, but may lengthen live range of variables
- For each basic block n, let
  - UEExpr(n)= expressions used before any operands being redefined in n
  - ExprKill(n)= expressions whose operands are redefined in n

Goal: evaluate very busy expressions on exit from n

- $\text{VeryBusy}(n) = \bigcup_{m \in \text{succ}(n)} (\text{UEExpr}(m) \cap (\text{VeryBusy}(m) - \text{ExprKill}(m))$

Appendix: Constant Propagation

- Domain of analysis
  - Set of variable-value pairs in a procedure
  - A pair (v,c) is valid at a CFG point p if on every path from procedure entry to p, variable v has value c.
  - (v,_): v has undefined value; (v,⊥): v has unknown value;

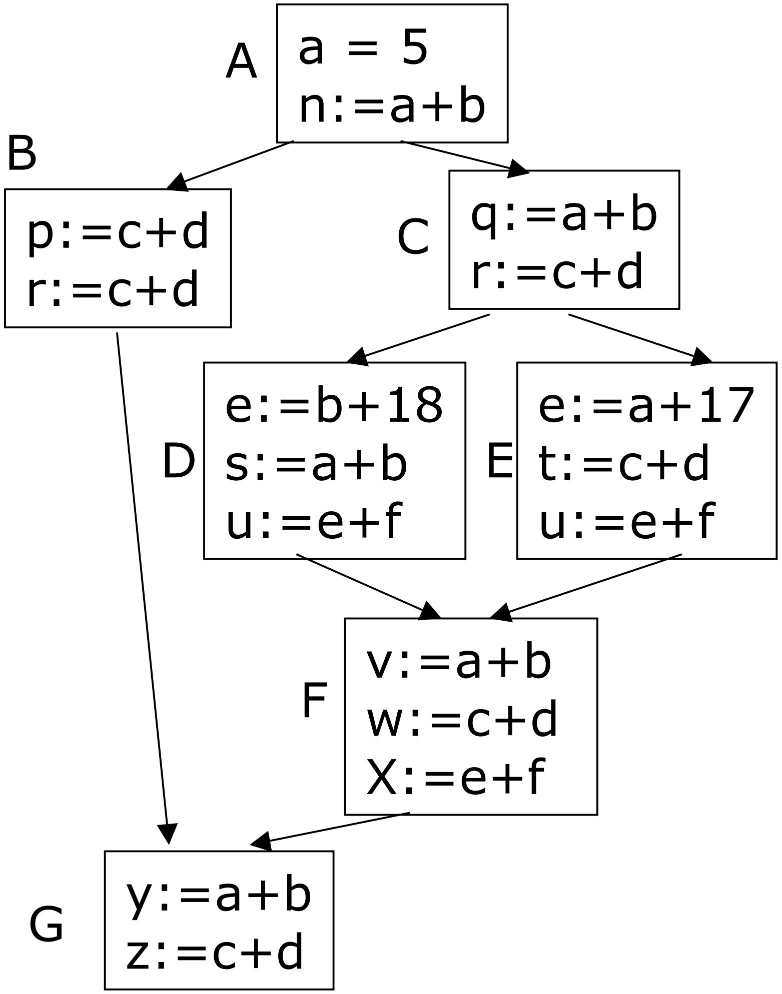
    (v, ci): v has a constant value ci
- If a variable v always has a constant value c at point p, the compiler can replace uses of v at p with c
  - Allows specialization of code based on value cz
- For each basic block n,
  - Evaluate all variable-value pairs valid on entry to n

$$\text{Constants}(n) = \bigwedge_{m \in \text{preds}(n)} \text{Fm}(\text{Constants}(m))$$

where /\ : pair-wise meet of var-val pairs

Fm(Constants(m)): var-val pairs on exit from m

Constant Propagation
Local Sets And Meet-over-all-paths



A
a = 5
n:=a+b

B
p:=c+d
r:=c+d

C
q:=a+b
r:=c+d

D
e:=b+18
s:=a+b
u:=e+f

E
e:=a+17
t:=c+d
u:=e+f

F
v:=a+b
w:=c+d
X:=e+f

G
y:=a+b
z:=c+d

- For each basic block n,

  $$\text{Constants}(n) = \bigwedge_{m \in \text{preds}(n)} \text{Fm}(\text{Constants}(m))$$

  where Fm(Constants(m)) is var-val pairs on exit from m

  $$(v,c1) \wedge (v,c2) = \begin{cases} (v,c1) \text{ if } c1 == c2; \\ (v, \perp) \text{ otherwise} \end{cases}$$

- Compute Fm(input)

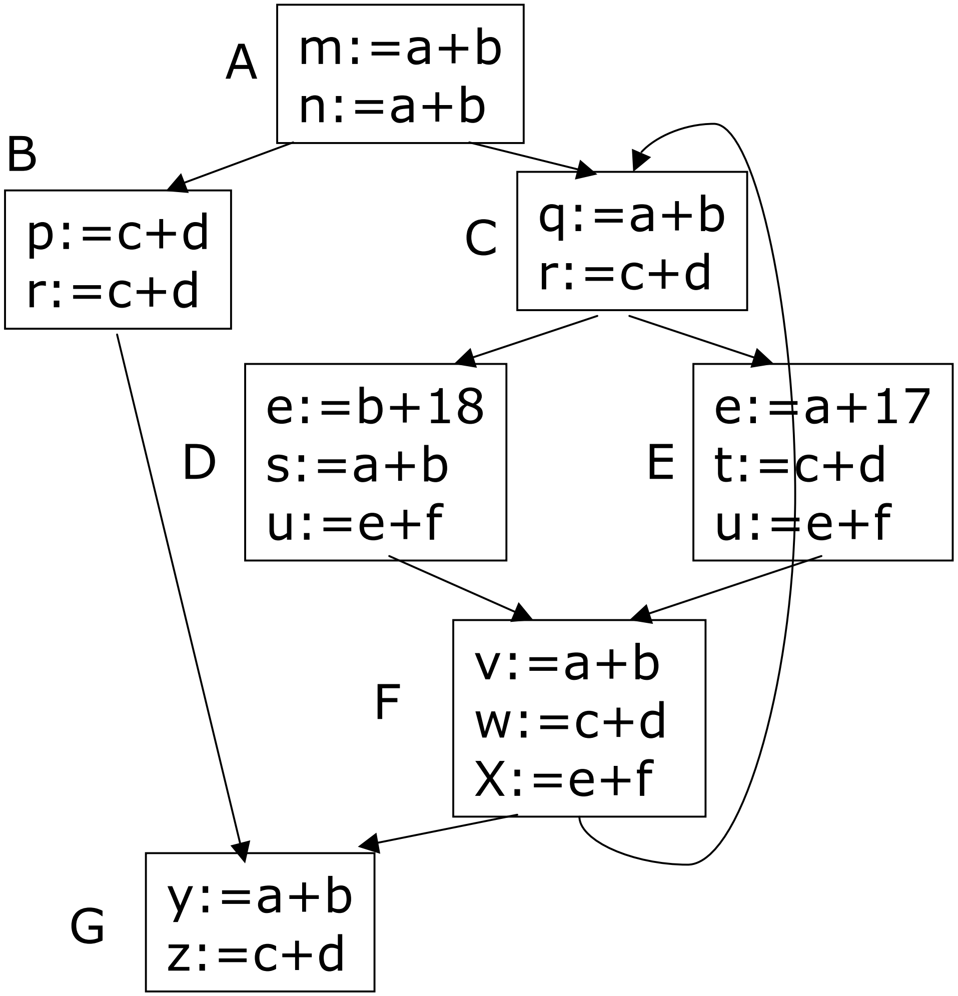
```
Let m = S1, S2, …, Sk
for each i = 1, …, k
   If Si is x := y
      Suppose (x,c1),(y,c2) ∈ input
      input = (input – {(x,c1)})∩{(x,c2)}
   If Si is y op z
      Suppose (x,c1),(y,c2),(z,c3) ∈ input
      input = (input – {(x,c1)})∩{(x,c2 op c3)}
```

$$\text{c2 op c3} = \begin{cases} \text{Constant if c2,c3 are constants} \\ \perp \text{ otherwise} \end{cases}$$

More On Constant Propagation

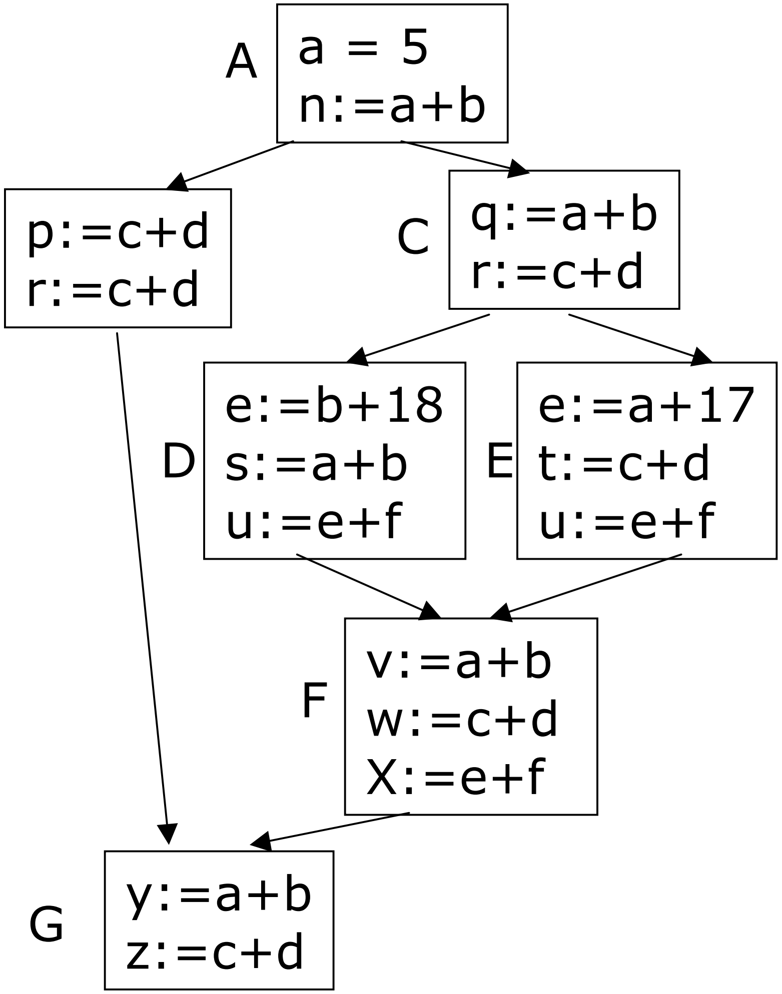
- Termination of constant propagation
  - Iterative data-flow algorithms are guaranteed to terminate if the result sets are bounded and grow monotonically.
  - Constant propagation does not have a bounded result set --- the set of all constant values is infinite
  - However, each variable-value pair can be updated at most twice. So constant propagation is guaranteed to terminate
- Using constant propagation to specialize code
  - Constant folding: evaluate integer expressions at compile time instead of runtime
  - Eliminate unreachable code: if a conditional test is always false, the entire branch can be removed
  - Enable more precision in other program analysis. E.g., knowing the bounds of loops can eliminate superfluous reordering constraints.

Appendix: Computing Dominators



- Domain of analysis
  - Set of basic blocks in a procedure
  - A basic block x dominates basic block y in CFG if x appears on all paths from entry to y
  - At any CFG node y, what basic blocks dominate y?
- For each basic block n
  - Dom(n)= {n} ∪ (∩ Dom(m)) m∈preds(n)
  - IDom(n) = the block in Dom(n) with smallest RPO sequence number
    - Each basic block n has a single IDom(n)
    - Can use IDom relation to build a dominator tree

Computing Dominance Frontiers



A
a = 5
n:=a+b

p:=c+d
r:=c+d

C
q:=a+b
r:=c+d

D
e:=b+18
s:=a+b
u:=e+f

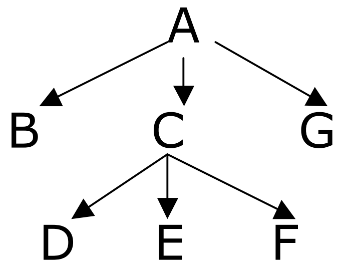
E
e:=a+17
t:=c+d
u:=e+f

F
v:=a+b
w:=c+d
X:=e+f

G
y:=a+b
z:=c+d

```
for each CFG node n
   DF(n) = ∅
for each CFG node n
   if n has multiple predecessors
       for each predecessor p of n
           runner := p
           while runner ≠ IDom(n)
               DF(runner) :=
                         DF(runner) ∪ {n}
               runner := IDom(runner)
```

Dominance tree:



Inserting ∅-Functions (skip)

Finding global names:

```
Globals:= ∅
for each variable x
   Blocks(x) = ∅
for each block bi: S1,S2,…,Sk
   VarKill := ∅
   for j = 1 to k
      let Sj be x := y op z
      if y ∉ VarKill then
          Globals := Globals ∪ {y}
      if z ∉ VarKill then
          Globals := Globals ∪ {z}
      VarKill := VarKill ∪ {x}
      Blocks(x) := Blocks(x) ∪ {b}
```

Inserting ∅-functions:

```
for each name x ∈ Globals
    WorkList := Blocks(x)
    for each block b ∈ WorkList
        for each block d in DF(b)
            insert a ∅-function for x in d
            WorkList:=WorkList ∪ {d}
```

Renaming After ∅-Insertion(skip)

Main

```
for each name x ∈ Globals
    counter[x] := 0
    stack[x] := 0
Rename (n0)
```

Create new name:

```
NewName(x)
   i := counter[x]
   counter[x] := counter[x] + 1
   push xi onto stack[x]
   return xi
```

Recursive renaming:

```
Rename(bi)
   for each “x:=∅(…)” in bi
       rename x as NewName(x)
   for each operation “x:=y op z” in bi
       rewrite y as top(stack[y])
       rewrite z as top(stack[z])
       rewrite x as NewName(x)
   for each m ∈ succ(bi)
       fill in ∅-function parameters in m
   for each n such that bi = IDom(n)
       Rename(n)
   for each operation “x:=y op z” in bi
       and each “x:=∅(…)” in bi
       pop(stack[x])
```