CS577 Modern Language Processors

Spring 2008

Lecture 4

GENERATING BETTER CODE

What does a conventional compiler do to improve quality of generated code?

- Eliminate redundant computation
- Move computations so they are executed less often
- Inline small functions (a double win)
- Choose data layouts wisely
- Allocate register resources wisely
- Schedule instructions wisely

Hope to get modest constant-time improvement (x2 is excellent).

Actual improvement depends on source language and target architecture.

Some Important Classic Optimizations

- Constant propagation
- Constant folding
- Strength reduction
- Dead code elimination
- Common subexpression elimination
- Invariant hoisting
- Inlining

Most of these are best done on malleable register-based IR.

Some key ideas: basic blocks, control-flow graphs, static single assignment, dominators, dataflow analysis.

© 2006-2008 Andrew Tolmach

Redundancy Elimination

Consider this example:

```
static void foo(int x,int y, int u, int v) {
  int w = (x + y) + (u - v);
  u = x + y;
  x = u - v;
}
```

Simple Redundancy Elimination

Corresponding bytecode and possible compiled output (using symbolic names):

```
load x
load y
add                 add rx,ry,r0
load u
load v
sub                 sub ru,rv,r1
add
store w             add r0,r1,rw
load x
load y
add
store u             add rx,ry,ru
load u
load v
sub
store x             sub ru,rv,rx
```

Where are the redundant computations?

INTERMEDIATE REPRESENTATION (IR) FOR OPTIMIZATION

Very common to use a somewhat abstract, register-based **3-address code**.

Assume an infinite number of temporary registers (also that local variables and arguments are already in registers to start with).

Instruction set:

`a <- b bop c`	Binary operation (for any binary operator bop)
`a <- b`	Move
`a <- M[b]`	Memory fetch (from address b)
`M[a] <- b`	Memory store (to address a)
`L:`	Label
`goto L`	Unconditional branch
`if a relop b goto L`	Conditional branch (for any relational operator relop)
`a <- f(a1,...,an)`	Function call (where f is fixed or computed)

How does the "level" of this language compare to JVM bytecodes?

Local Value Numbering

A simple approach to common-subexpression elimination that works on straight-line code.

- Process each 3-addr instruction in order.

- Maintain a mapping from identifiers (x) and binop expressions (left,op,right) to value numbers.

- Whenever an entry exists already, rewrite the instruction to use it.

(Alternatively, could build DAG showing relationships between entries.)

First, our example in the new IR:

```
g <- x + y
h <- u - v
w <- g + h
u <- x + y
x <- u - v
```

Local Value Numbering (2)

```
Initial code   Final code   Mapping entries
g <- x + y     g <- x + y   x -> 1              1:x
                            y -> 2              2:y
                            (1,+,2) -> 3
                            g -> 3              3:g
h <- u - v     h <- u - v   u -> 4              4:u
                            v -> 5              5:v
                            (4,-,5) -> 6
                            h -> 6              6:h
w <- g + h     w <- g + h   (3,+,6) -> 7
                            w -> 7              7:w
u <- x + y     u <- g       u -> 3
x <- u - v     x <- u - v   (3,-,5) -> 8
                            x -> 8              8:x
```

Now do copy propagation to get rid of `u <- g`.

Issues with Naming

- If there are (re-)assignments, a name is not the same thing as a value!

- Value numbering successfully distinguishes between different values with the same name (e.g., `u` in the example).

- But can still lose access to a value if its name gets overwritten.

Modified Example:

```
Initial code     Final code     Mapping entries
z <- x + y       z <- x + y     x -> 1                1:x
                                y -> 2                2:y
                                (1,+,2) -> 3
                                z -> 3                3:z
h <- u - v       h <- u - v     u -> 4                4:u
                                v -> 5                5:v
                                (4,-,5) -> 6
                                h -> 6                6:h
z <- z + h       z <- z + h     (3,+,6) -> 7
                                z -> 7                7:z  3:??
u <- x + y       u <- ??
```

Using Unique Names

Idea: rename variables so that every assignment gets a unique name.

```
Initial code     Renamed code      Final code        Mapping entries

z <- x + y       z0 <- x0 + y0     z0 <- x0 + y0     x0 -> 1              1:x0
                                                     y0 -> 2              2:y0
                                                     (1,+,2) -> 3
                                                     z0 -> 3              3:z0
h <- u - v       h0 <- u0 - v0     h0 <- u0 - v0     u0 -> 4              4:u0
                                                     v0 -> 5              5:v0
                                                     (4,-,5) -> 6
                                                     h0 -> 6              6:h0
z <- z + h       z1 <- z0 + h0     z1 <- z0 + h0     (3,+,6) -> 7
                                                     z1 -> 7              7:z1
u <- x + y       u1 <- x0 + y0     u1 <- z0          u1 -> 3
```

What about control flow joins?

How can we generate unique names when control can reach uses in multiple ways?

```
Initial code                   Renamed code

     if a > 0 goto L1              if a0 > 0 goto L1
     b = x + y                     b0 = x0 + y0
     goto L2                       goto L2
L1:  b = x - y                 L1: b1 = x0 - y0
L2:  c = a + b                 L2: c0 = a0 + ??
```

Will consider soon in context of SSA form.

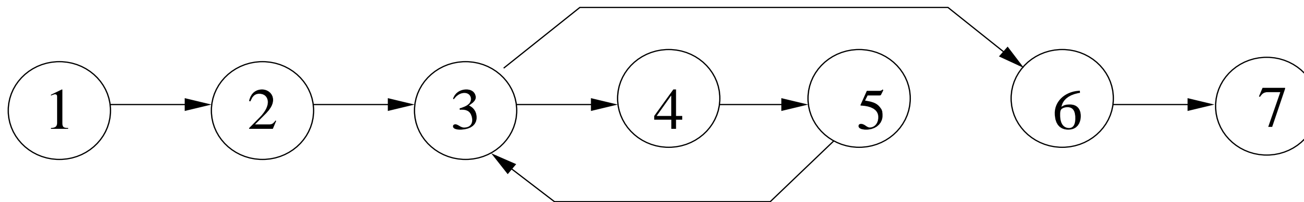
Simple Control Flow Graphs (CFG's)

- One node per instruction (and perhaps for procedure entry and exit)

- Edge from A to B if control **might** flow directly from A to B.

Example (Appel, "Modern Compiler Implementation," Ex. 17.3)

```
1             a <- 5
2             c <- 1
3    L1:      if c > a goto L2
4             c <- c + c
5             goto L1
6    L2:      a <- c - a
7             c <- 0
```

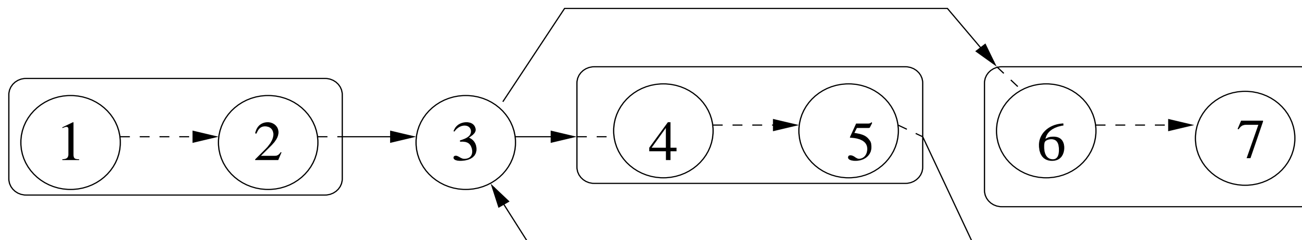


Basic Blocks

Often useful to factor a program into its **basic blocks**, which are sequences of consecutive instructions in which control always enters at the top and exits from the bottom.

```
1               a <- 5
2               c <- 1
3       L1:     if c > a goto L2
4               c <- c + c
5               goto L1
6       L2:     a <- c - a
7               c <- 0
```

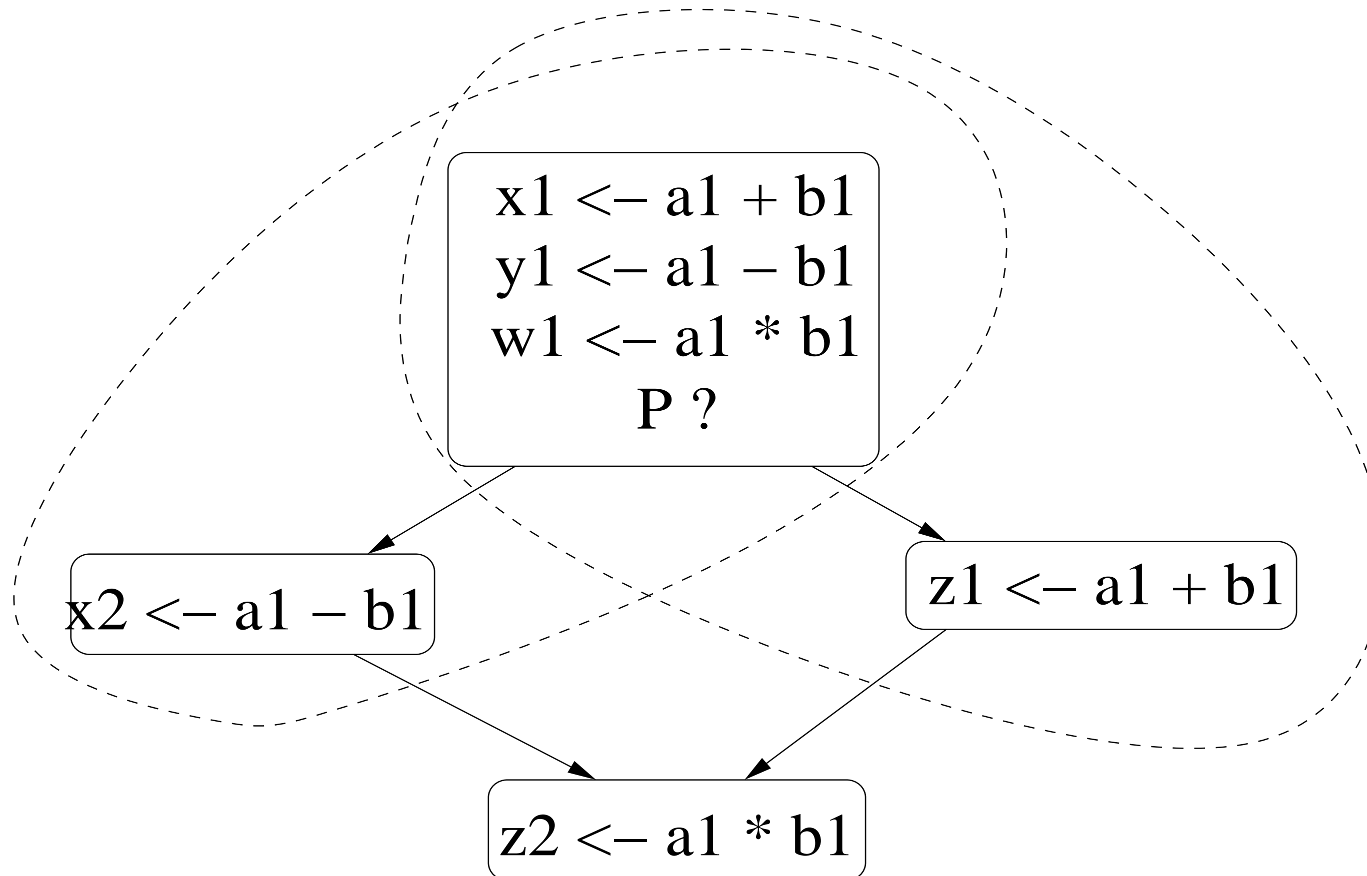
Often use basic blocks as nodes in CFG.



Superlocal Value Numbering

Do analysis over paths in **extended** basic blocks.

(An EBB has one entry, but can have multiple exits. It forms a subtree of the CFG; all the blocks in the EBB except perhaps the root have a unique predecessor inside the EBB).



DOMINATORS

To define dominators, assume that CFG has a distinguished start node S, and has no disconnected subgraphs (nodes unreachable from S).

Then we say node d **dominates** node n if **all** paths from S to n include d.

(In particular, every node dominates itself.)

Fact: d dominates n iff $d = n$ or d dominates all predecessors of n.

So can define the set $D(n)$ of nodes that dominate n as follows:

- $D(S) = \{S\}$
- $D(n) = \{n\} \cup (\bigcap_{p \in pred(n)} D(p))$

where $pred(n) =$ set of predecessors of n in CFG.

DOMINATOR TREE

The **immediate dominator** of n, $idom(n)$, is defined thus:

- $idom(n)$ dominates n
- $idom(n)$ is not n
- $idom(n)$ does not dominate any other dominator of n (except n itself)

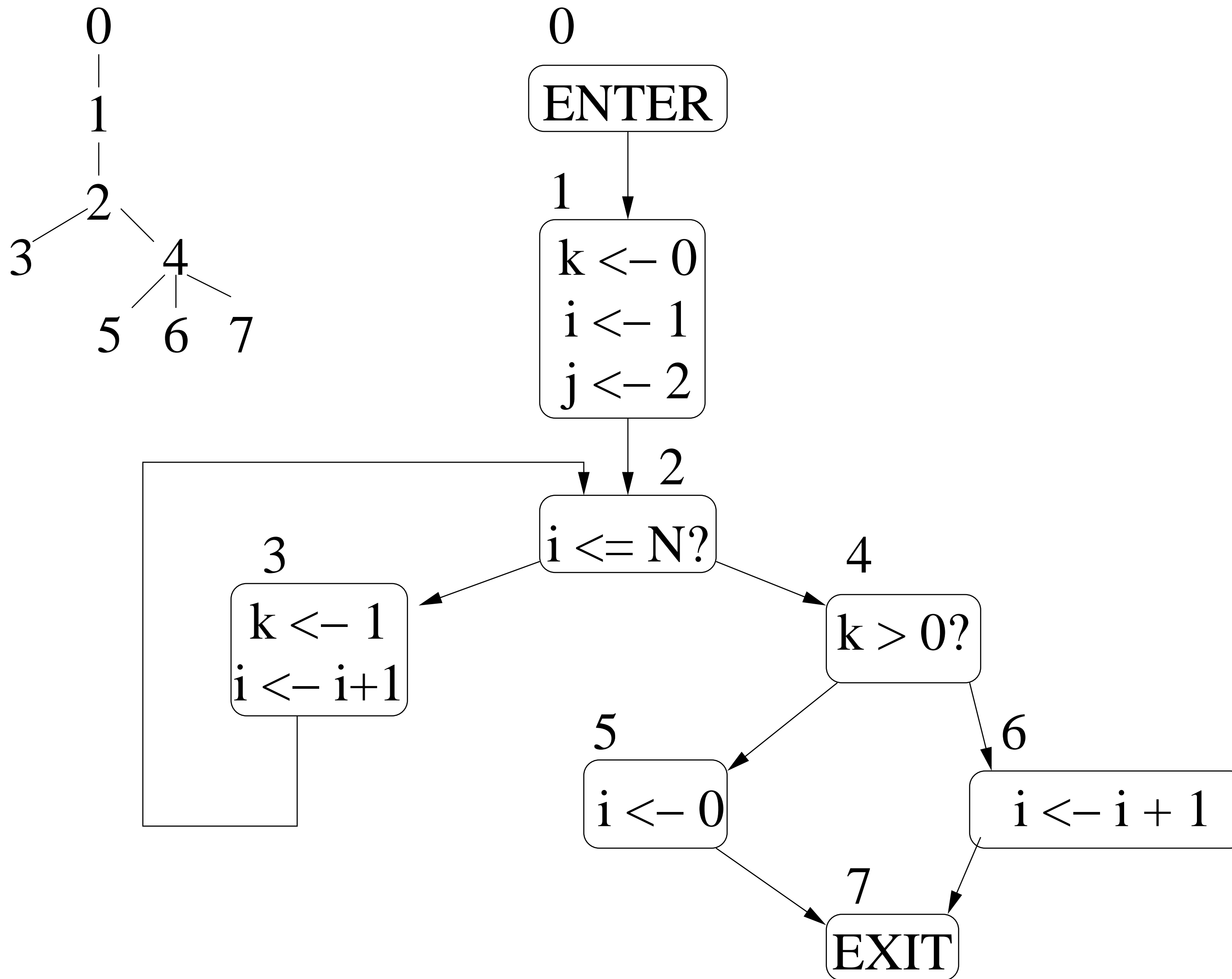
Fact: every node (except S) has a unique immediate dominator

Hence the immediate dominator relation defined a tree, called the **dominator tree**, whose nodes are the nodes of the CFG, where the parent of a node is its immediate dominator.

Have $D(n) = \{n\} \cup (\text{ancestors of } n \text{ in dominator tree})$

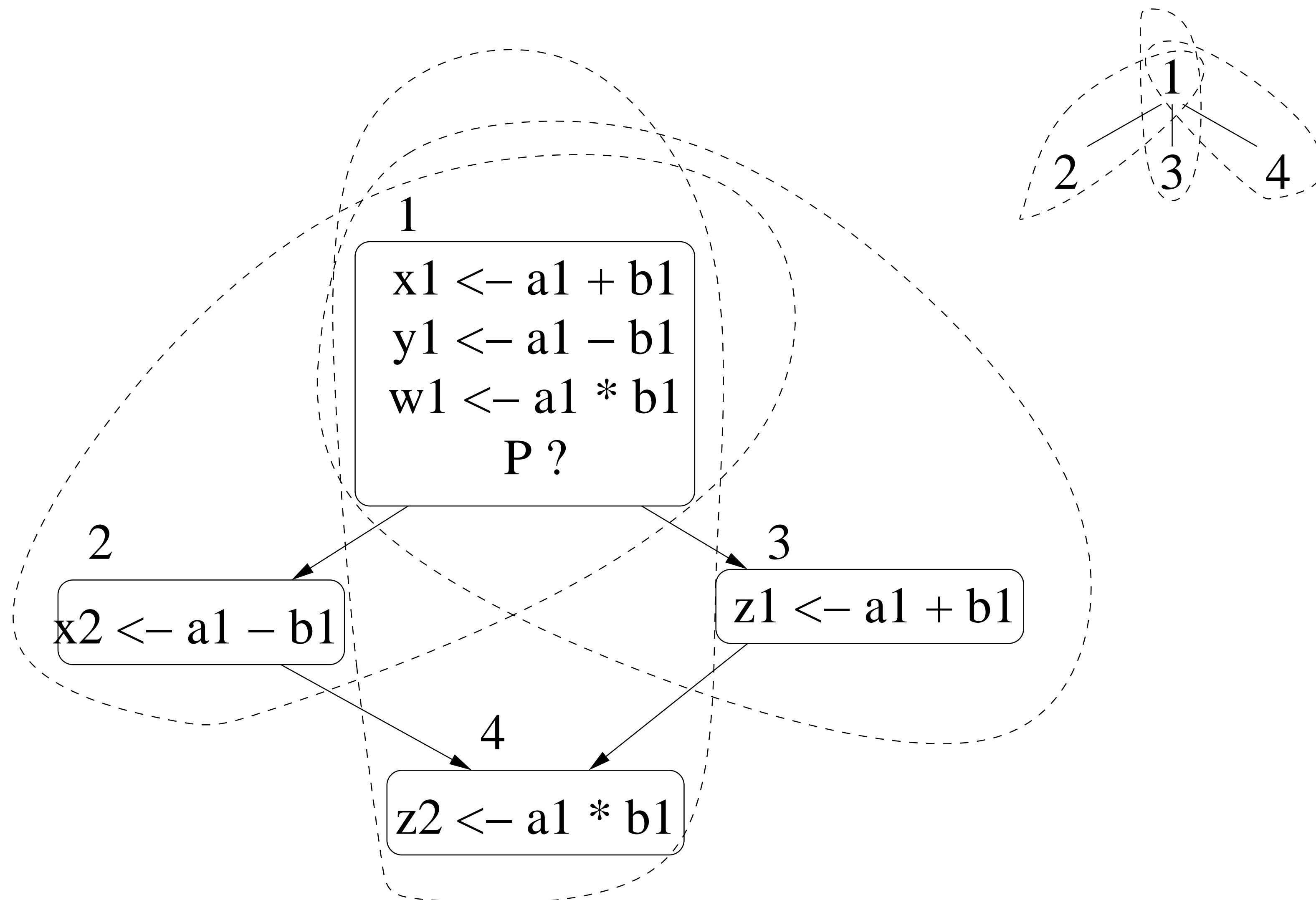
(Nontrivial) Fact: The dominator tree of a CFG can be computed in almost-linear time.

Dominator Tree Example



DOMINATOR-BASED VALUE NUMBERING

Do analysis over paths in dominator tree.



Static Single Assignment (SSA) Form

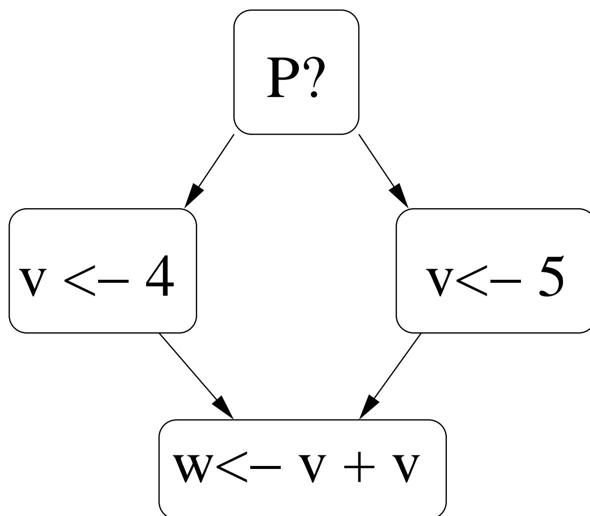
- Every variable has just one (static) definition (though defining instruction may be executed many times)

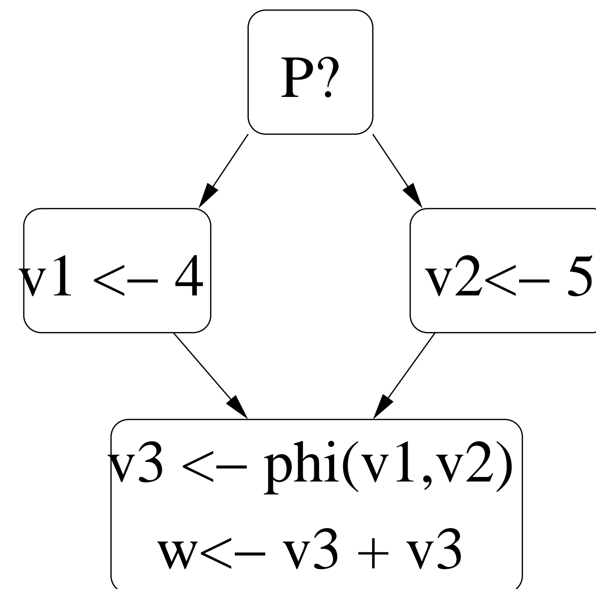
- For straightline code, this is just what we did for value numbering:

```
Original code              SSA Code
v <- 4                     v1 <- 4
w <- v + 5                 w1 <- v1 + 5
v <- 6                     v2 <- 6
w <- v + 7                 w2 <- v2 + 6
```

- For general flow, must introduce ϕ-nodes ("phi"-nodes). These are fictitious operations, (usually) not intended to have execution significance. To interpret them, must view code as CFG, with the in-edges to each node having a well-defined order.

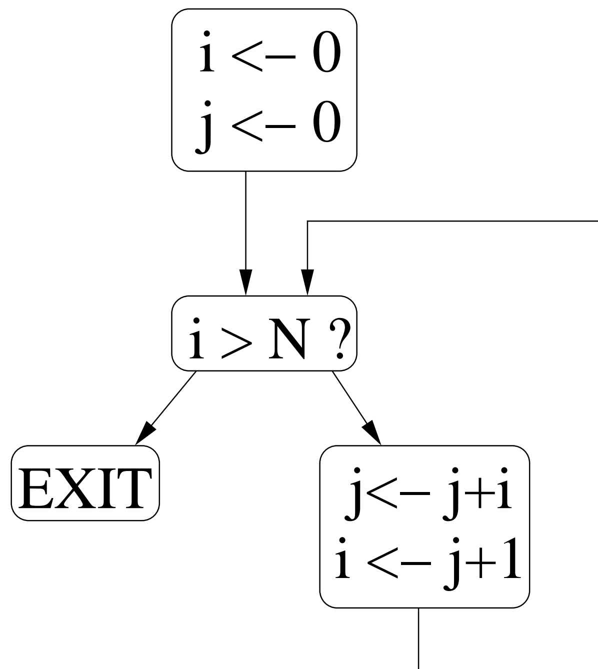
SSA Example 1



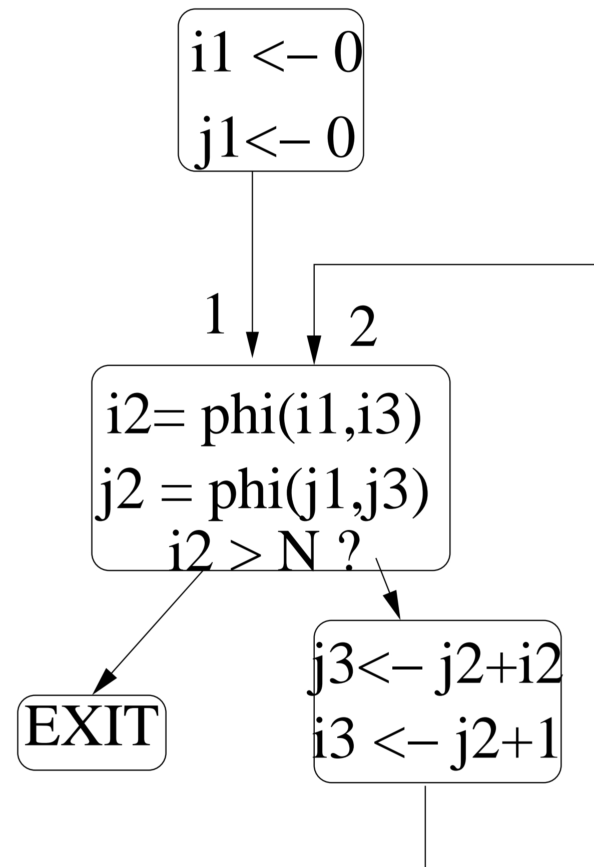


SSA EXAMPLE 2

Original CFG



SSA CFG



COMPUTING SSA

Where should we put ϕ assignments, and for which variables?

Answer: there are many options, so long as single-assignment property is obeyed and each use of a variable in the original program has a corresponding uniquely-defined SSA variable.

Simplistic approach: put ϕ assignments in every join node, for every variable in scope. Much too expensive!

Suffices to put a ϕ assignment for `x` in join nodes that are not **dominated** by a single definition of `x`. (More later.)

AVAILABLE EXPRESSIONS

Even with dominator-based VN, we cannot find redundant expressions computed on **different** paths.

An alternative approach is to compute **available expressions**.

For SSA graphs, an expression is **available** at node n if it is computed at least once on **every** path from S to n.

If an expression is available at a node where it is being recomputed, it is possible to replace the recomputation by a variable representing the result of the previous computation.

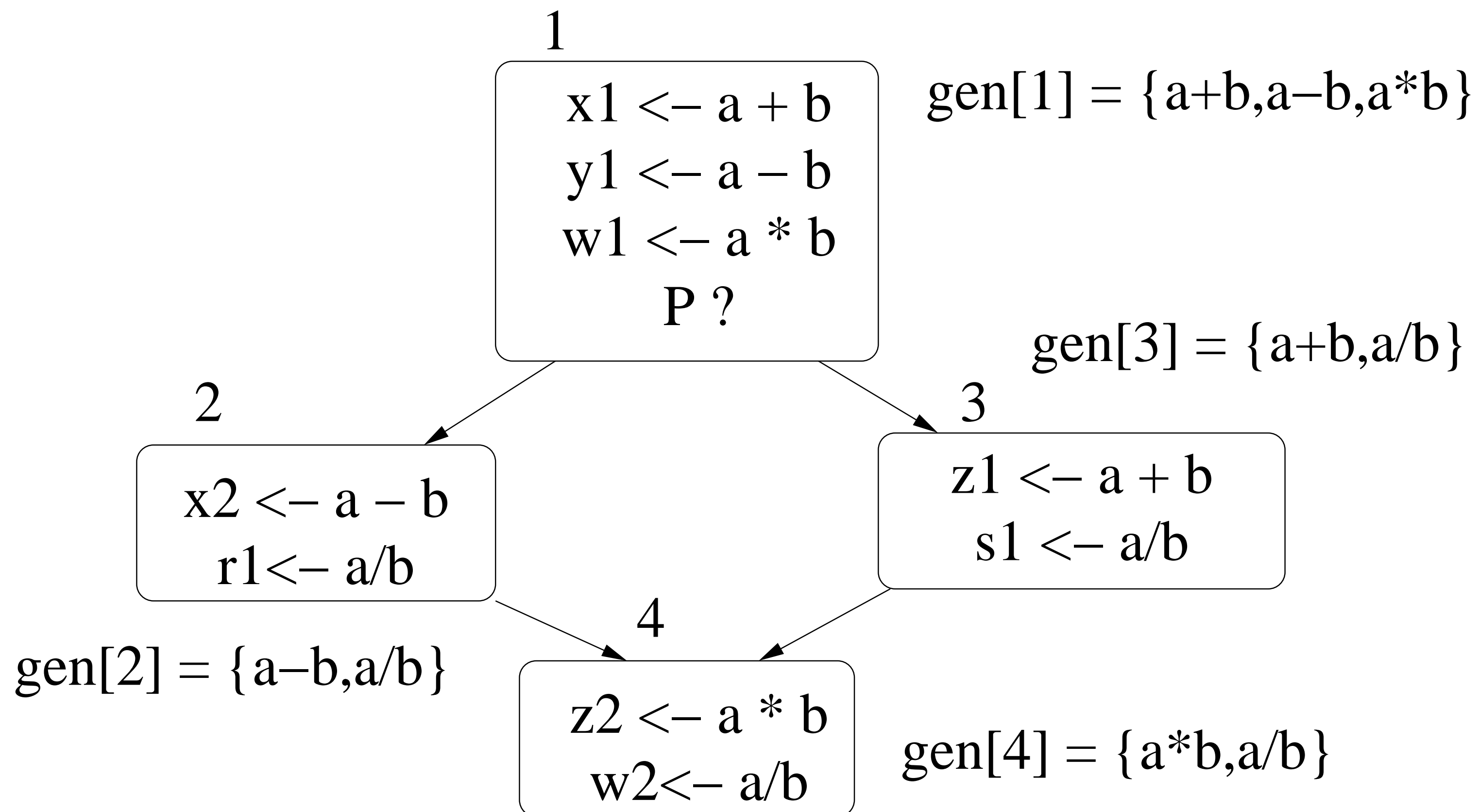
This is a classic **data flow analysis** problem, specified by the following equations:

$gen(\texttt{t <- b bop c}) = \{\texttt{b bop c}\}$
$gen(\textit{other instruction}) = \emptyset$

$in(n) = \bigcap_{p \in pred(n)} out(p)$
$out(n) = in(n) \cup gen(n)$

Here we want $in(n)$, the set of expressions available on entry to n.

Available Expressions Example



Solutions

Here's the (unique) solution to the data flow equations.

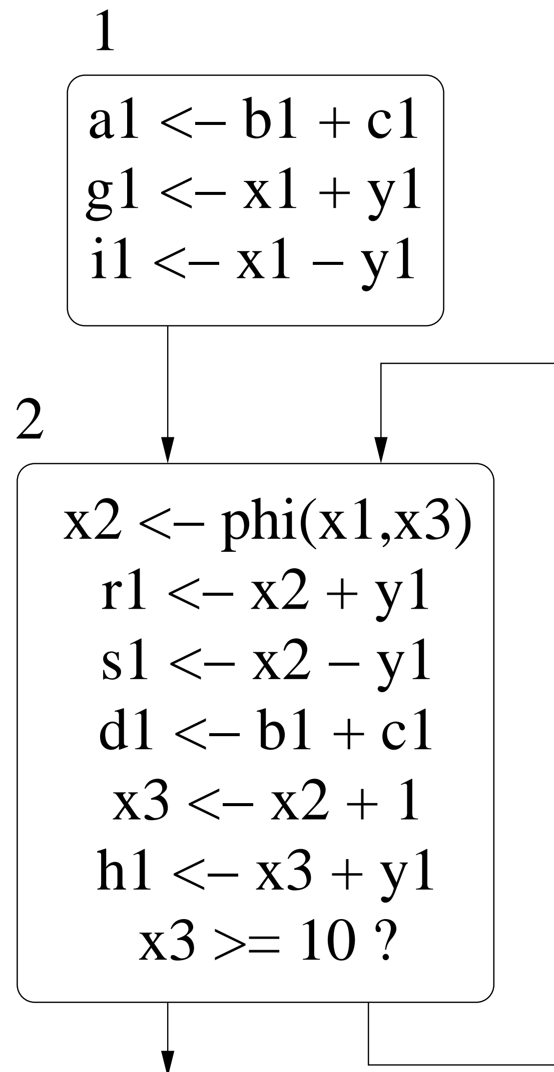
```
in[1] = {}                      out[1] = {a+b,a-b,a*b}
in[2] = {a+b,a-b,a*b}           out[2] = {a+b,a-b,a*b,a/b}
in[3] = {a+b,a-b,a*b}           out[3] = {a+b,a-b,a*b,a/b}
in[4] = {a+b,a-b,a*b,a/b}       out[4] = {a+b,a-b,a*b,a/b}
```

So nothing needs to be recomputed in nodes 2, 3, or 4.

Another Available Expressions Example

Original code:

```
    a <- b + c
    g <- x + y
    i <- x - y
L:  r <- x + y
    s <- x - y
    d <- b + c
    x <- x + 1
    h <- x + y
    if x < 10 goto L
```



Solutions

Here's a solution (the maximal one, which is what we want):

```
in[1] = {}                        out[1] = {b1+c1,x1+y1,x1-y1}
in[2] = {b1+c1,x1+y1,x1-y1}       out[2] = {b1+c1,x1+y1,x1-y1,
                                             x2+y1,x2-y1,x2+1,x3+y1}
```

Using this, we can avoid recomputing `b1+c1` in block 2.

Standard available expressions algorithm doesn't let us avoid recomputing `x2+y1`, but perhaps we could be clever and notice that because `x1+y1` and `x3+y1` are available into block 2 on paths 1 and 2, respectively, ϕ`(x1,x3)+y1` is available too..

By the way, here's another solution to the dataflow equations (a less useful one):

```
in[1] = {}              out[1] = {b1+c1,x1+y1,x1-y1}
in[2] = {b1+c1}         out[2] = {x2+y1,x2-y1,b1+c1,x2+1,x3+y1}
```

Note the importance of taking an "optimistic" view of `in[2]`.

SOLVING DATAFLOW EQUATIONS

Completely general method: Iteration to a fixed point.

For Available Expressions problem:

- can precompute `gen` set for each node
- start with the optimistic approximation that all the `in` and `out` sets are **full** (contain all possible expressions),
- on each iteration, recompute `in` and `out` using the most recent approximations we have for them (and `gen`)
- iterate until computed sets don't change

This gives us a **greatest fixed point**, i.e., the **largest** sets that solve the equations. Note that if we started with empty sets, the `in` sets would not contain expressions that remain available after a loop iteration, due to the $\cap$ operation in `in[]`; this would be the **least fixed point** solution.

Computing GFP Solution Example

Let `A` = set of all potentially interesting expressions, namely `{b1+c1,x1+y1,x1-y1,x2+y1,x2-y1,x2+1,x3+y1}`.

n	pred[n]	gen[n]
1	-	{b1+c1,x1+y1,x1-y1}
2	{1,2}	{x2+y1,x2-y1,b1+c1,x2+1,x3+y1}

n	iteration 0		iteration 1		iteration 2	
	in[n]	out[n]	in[n]	out[n]	in[n]	out[n]
1	-	A	-	gen[1]	-	gen[1]
2	A	A	gen[1]	A	gen[1]	A

Will see more interesting examples another time.

SSA GRAPH CONSTRUCTION

Can use dominator information to construct a "minimal" SSA graph.

Note the following dominance properties, which follow from the requirement that each variable is necessarily defined before it is used.

1. If x is used in a non-ϕ statement in block n, then the definition of x dominates n.

2. If x is the ith argument of a ϕ-function in CFG block n, then the definition of x dominates the ith predecessor of n.

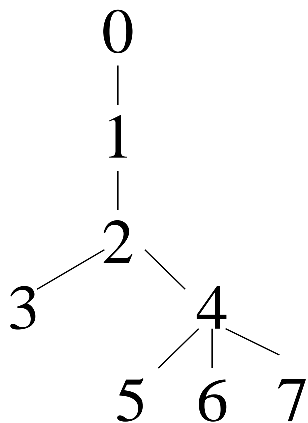
We say x **strictly dominates** w if x dominates w but $x \neq w$.

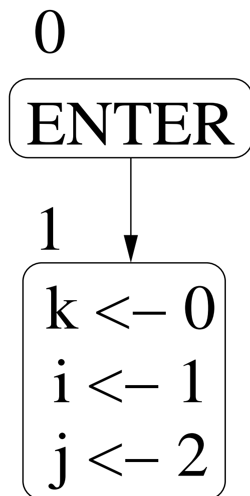
The **dominance frontier** of a definition x, $DF(x)$, is the set of nodes w such that x dominates an (immediate) predecessor of w, but x does not strictly dominate w.

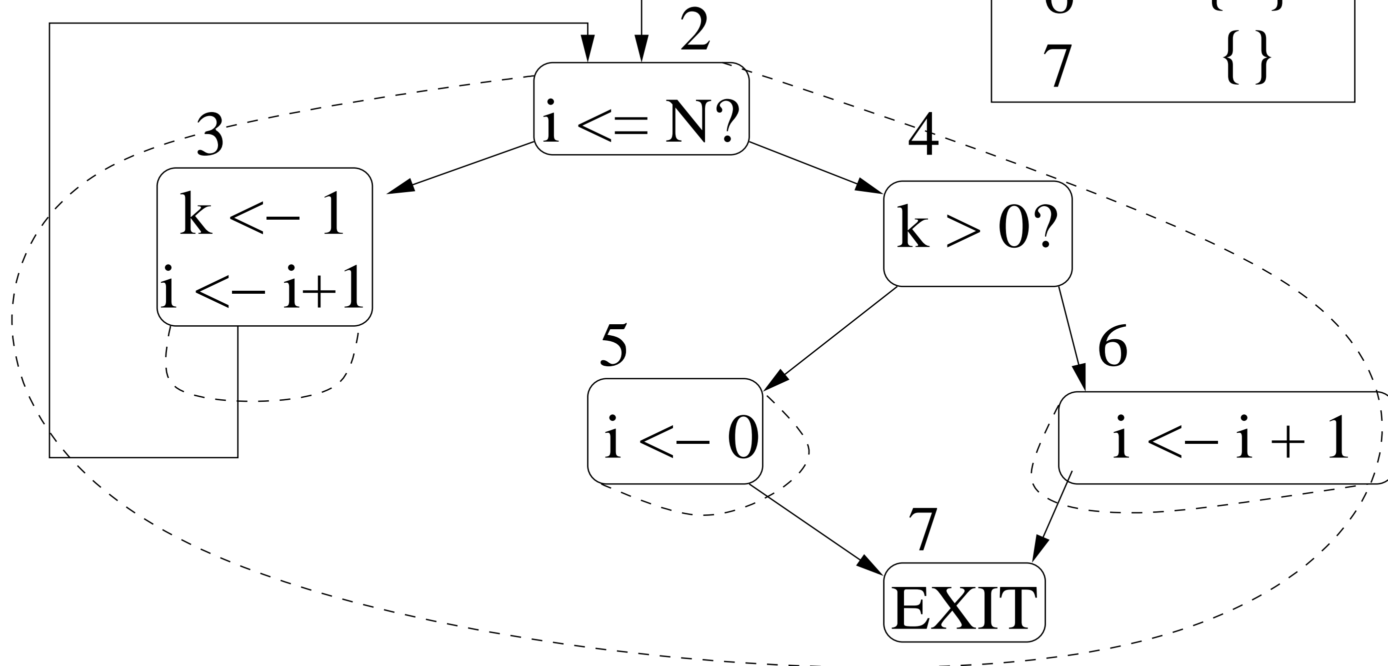
Intuition: Any node in $DF(x)$ is the join point of two disjoint paths from x and from the ENTRY node.

Can easily compute $DF(x)$ from the dominator tree.

Dominance Frontier Example







n	DF(n)
0	{ }
1	{ }
2	{2}
3	{2}
4	{ }
5	{7}
6	{7}
7	{ }

Dominance Frontier Criterion

If node x defines a, any node in $DF(x)$ requires a ϕ-function for a.

- Since such a ϕ-function is itself a definition for a, we must (in general) iterate until there are no more ϕ-functions to place.

In our example, must place ϕ-nodes for `i` and `k` in node 2 and for `i` in node 7.

Example in SSA Form