# Dataflow analysis: intro/iterative

- control flow analysis produces control flow graph (CFG)
- dataflow analysis uses CFG to identify optimization opportunities
- SSA as an intermediate representation, gives good results without needing overly cumbersome data structures
- value numbering (local and superlocal) applied to tree-like subsets of the CFG gives good way to find redundant expressions, simplify expressions, apply constant folding, etc
- deeper analysis needed to find things like uninitialized variables, since we need to account for cycles, reconvergent paths, etc

# Subroutine and program level

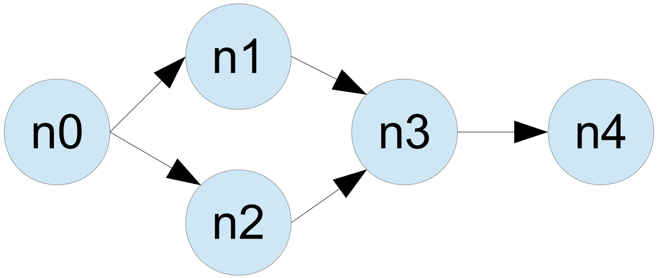
- next stages of dataflow analysis take place at the subroutine and whole-program levels
- effective analysis is muddled by things such as
  - references to values from other compilation units
  - ambiguous value references (e.g. pointers, variable array indices)
  - pass by reference parameters
- first, will consider iterative dataflow analysis

# Iterative analysis: subroutine level

- earlier we looked at calculating liveout(b) by repeatedly recalculating it for each of the individual blocks in a subroutine, stopping when no further changes occurred
- we'll apply very similar techniques across a variety of analysis metrics, and look at how those metrics can then be applied
- later will also apply similar techniques at whole-program level

# Dominators in CFG

- given the CFG for a subroutine, a collection of nodes and edges with a unique entry node, n0
- the dominating set for a node, n, is the set containing n and those nodes that lie on *every* path from n0 through n
- dominating set for n4 below is { n0, n3, n4 }



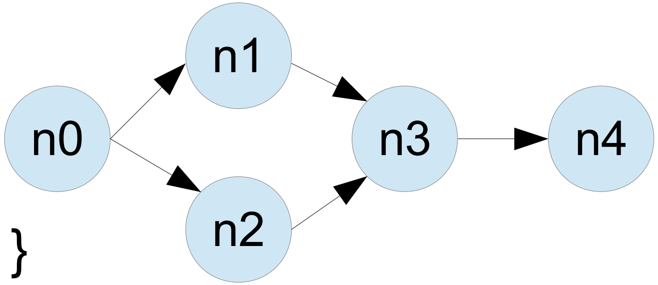
# Algorithm: dominating sets

- given k blocks, n0,...,nk-1

```
for i = 0..k-1: Dom(ni) = { ni }
changed = true
while (changed)
   changed = false
   for i = 1 to k-1
       temp = { ni } U Preds(ni)
       if temp != Dom(ni) then
          changed = true
          Dom(ni) = temp
```

Preds(n) is the intersection of Dom(nj) across all the predecessor nodes of n

- example: Dom(ni) sets on each pass



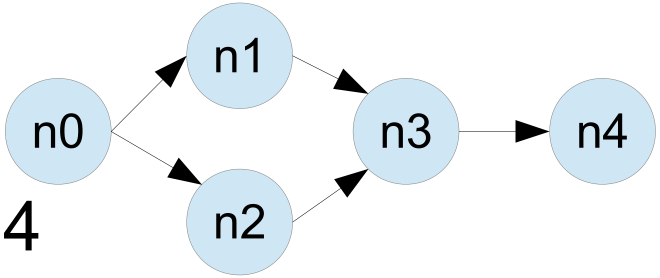
  1) { n0 } { n1 } { n2 } { n3 } {n4 }
  2) { n0 } { n1, n0 } { n2, n0 }, { n3 } { n4, n3 }
  3) { n0 } { n1, n0 } { n2, n0 }, { n3, n0 } { n4, n3 }
  4) { n0 } { n1, n0 } { n2, n0 }, { n3, n0 } { n4, n3, n0 }
  5) same as step 4
- no set can get bigger than k, guaranteed to terminate
- evaluating in an order other than the arbitrary sequence 0..k-1 might give more efficient calculation ...

# Reverse post-order traversal (RPO)

- post-order traversal processes children of a node first, then the node
- RPO takes post-order traversal sequence then simply reverses it
- computing order(n), assuming k nodes and a global var visitNum
- visitNum initially 0, order(n) initially -1 for each node n

```
rpo(node n)
    for each child, c, of n
        if (order(c) == -1) postorder(c)
    order(n) := (k-1) - visitNum
    visitNum++
```

# RPO advantage



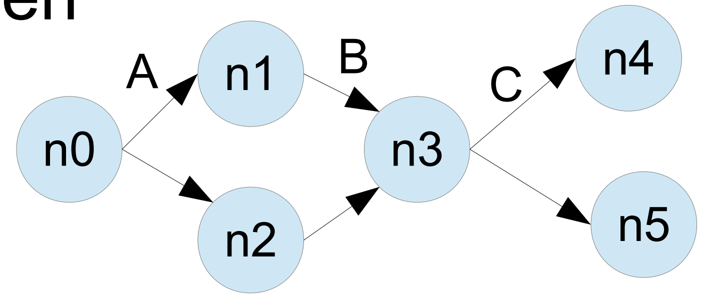
- post-order traversal process order
  - n4, n3, n1, n2, n0
- RPO reverses, giving n0, n2, n1, n3, n4
- note that each node's predecessors are processed before the node itself
- for algorithms like the Dom calculator that is exactly what we're hoping for

# Dom vs liveout

- Dom(n) looks for the nodes that appear on *every* path *into* n
- liveout(n) looks for the values that appear on *any* path leading *leaving* n
- we can actually tweak liveout to make use of RPO for an efficient node-processing order:
  - first, reverse the direction of each edge in the CFG
  - then use RPO on the resulting graph

# Iterative analysis limitations

- both the Dom and the liveout algorithms assume every path is possible
- the actual logic constraints in the code might preclude some paths
- suppose A is taken only if some condition x is true, and C is taken only if condition x is false, then path ABC can never happen



# Iterative limitations cont.

- Ambiguity seriously limits effectiveness
  - using/setting a value in an array (using a non-constant index) forces all array elements to be treated as used/set
  - using/setting a value through a pointer forces all possible targets of the pointer to be treated as used/set (this is even worse if pointer arithmetic is permitted)
- The pointer aspect in particular may cause the compiler to avoid putting values in registers if those values may be the target of a pointer

# Expressions/available-in

- similar to liveout, the expressions whose results are available for use at any point p
- expression e is available at point p iff
  - on every path from the subroutine entry to p, e has been evaluated and none of its subexpressions are altered before p
- AvailableIn(n): the set of expressions available in n
- DEexpr(n): downward exposed expressions of n:
  - evaluated in n, subexpressions not subsequently altered in n
- exprkill(n): set of expressions killed in n (i.e. by n altering a subexpression used by e)

# Definitions reaching n

- also similar to liveout, identifying set of variable (including temp variable) definitions that reach a point in the CFG
- assignment of value to a variable is a definition, recorded as a pair: the variable name and instruction number
- Reaches(n): set of definitions that reach n
- DEdef(n): the downward exposed definitions of n (definitions in n that aren't subsequently killed in n)
- defkill(n): the definitions killed by n (n alters the variable through a new definition)

# Expanding to whole-program

- compiler has to make worst-case assumptions about which values are altered by each subroutine
- assume anything the subroutine may alter it does alter
  - includes global variables, pass by ref, pointer accessibility
- maymodify(f) the set of names whose values f may alter
  - computed using the names locally modified in f together with the maymodify(g) for every function g that f calls
- again, iterative computation, repeating until no change