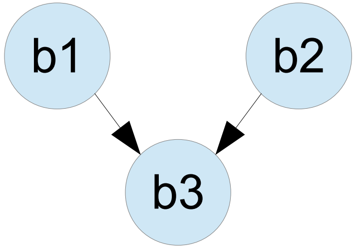
SSA form and uses

- earlier we introduced single static assignment form
- LHS of each assignment gets a unique name
- simplifies later analysis/optimizations
- relatively easy within a basic block
- problems arise when a name referenced within a block can come from two or more different predecessor blocks

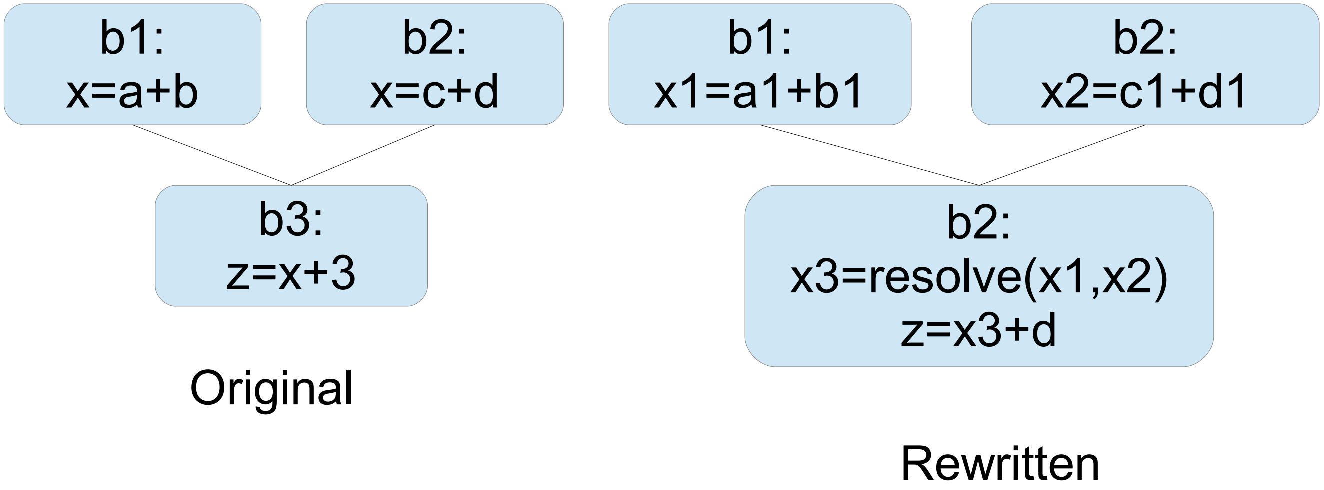


suppose b1 and b2 each define an x, and b3 uses x

how do we relabel x within b3?

Introducing a resolve function

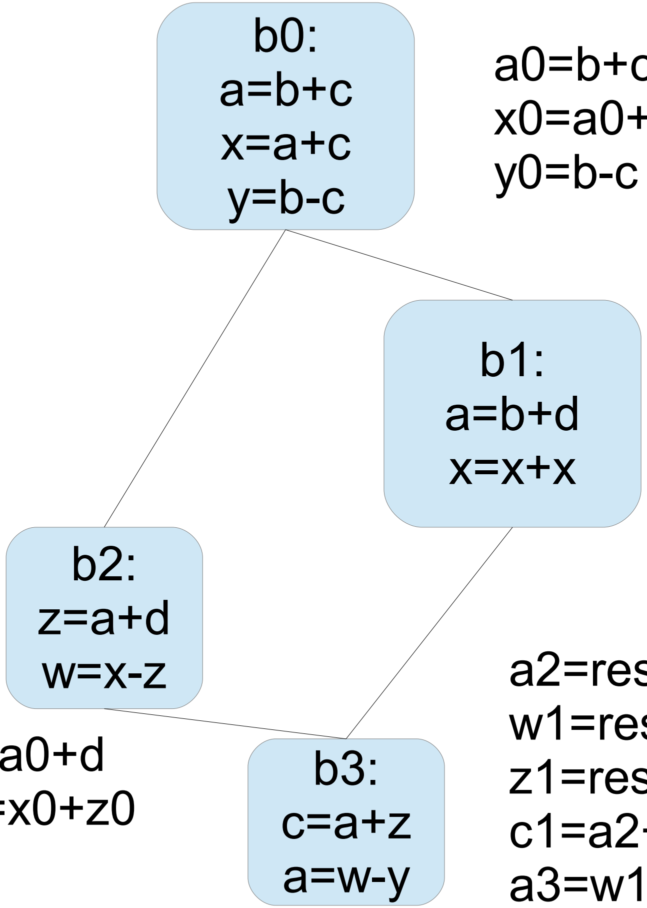
- will introduce a resolve function at the start of basic blocks, takes the conflicting names as parameters and assigns “correct” value to newly named variable



When do we need resolve

- a name whose value is set (defined) differently on multiple incoming paths does need resolve
- a name coming from a single source doesn't need resolve
- sometimes a named value can flow through multiple paths unchanged before it reaches b: may not need resolve
- sometimes a named value isn't used after a certain point, should not need resolve in any further blocks
- we want to be able to identify and place resolve functions only where they are actually needed

Renaming/resolve example



a0=b+c
x0=a0+c
y0=b-c

a1=b+d
x1=x0+x0

z0=a0+d
w0=x0+z0

a2=resolve(a0,a1)
w1=resolve(w0,w)
z1=resolve(z0,z)
c1=a2+z1
a3=w1-y0

b,c,d must be defined prior to b0 or are used uninitialized in b0, b1, or b2

w* and z* must be defined prior to b0 or they may be used uninitialized in b3

(note that y0 passes through from b0 to b3 unchanged)

Resolve and reality

- there isn't generally anything like a resolve function in our target languages, so later we'll have to implement a way to translate resolves to the target language
- efficient calculation of the minimal set of resolve functions is a difficult problem, we'll often settle for a reasonable approximation (i.e. some unnecessary resolves, but hopefully not many)

Where do definitions reach?

- given statement “x = y OP z”, some questions:
  - which subsequent blocks change the value of x?
  - which subsequent blocks use x before it is changed?
  - what is the last block using x on any given path?
- we'll use dominating nodes/sets/trees, and dominance frontiers in algorithms to try and form answers
- recall: in a CFG, dominating nodes are those that lie on *every* path from the graph entry point to n

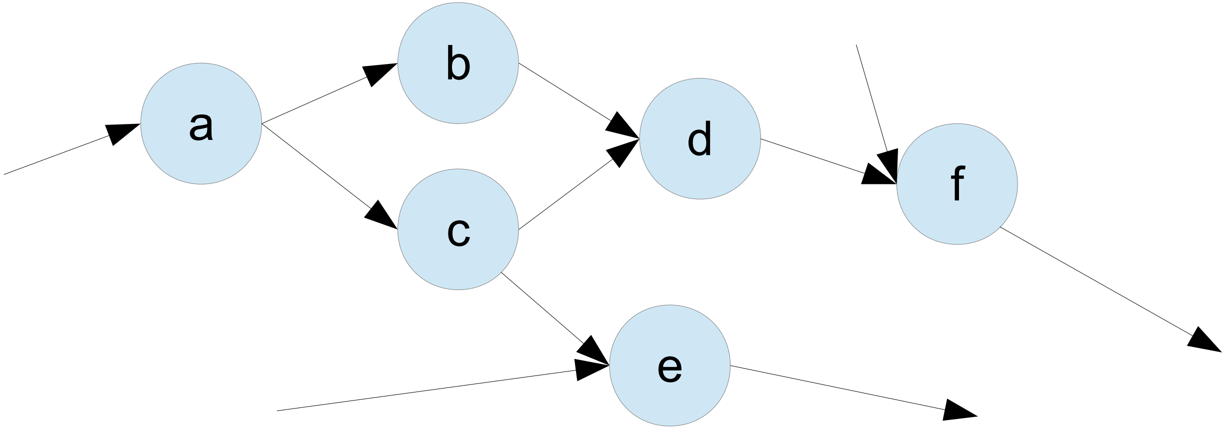
Relevance of dominating nodes

- I tend to use block and node interchangably, each node in the CFG representing exactly one basic block
- Suppose block b dominates some set of other blocks, denoted DomBy(b)
  - if n is in DomBy(b), then b lies on every path to n
  - any definition made in b is thus always available in n unless some block between b and n overwrites it
  - b doesn't need to request any resolve functions be added to n (if some block between b and n changes one of b's definitions then *that* block could request a resolve function be added to n)

Dominance frontiers

- given a block b, and set DomBy(b) of nodes it dominates:
  - let DF(b) be the dominance frontier of b
  - y is in DF(b) if x is in DomBy(b), but y is not, and edge(x,y) is in the CFG
  - this means y is the first node along a path leaving DomBy(b)
- given a block b and any block y that is in DF(b)
  - any definition, d, in b *can* reach y if not overwritten first
  - but since b is not on all the paths to y, an alternative definition of d might reach y
  - thus b should request a resolve function be added to y

Example: Dom(a) and DF(a)



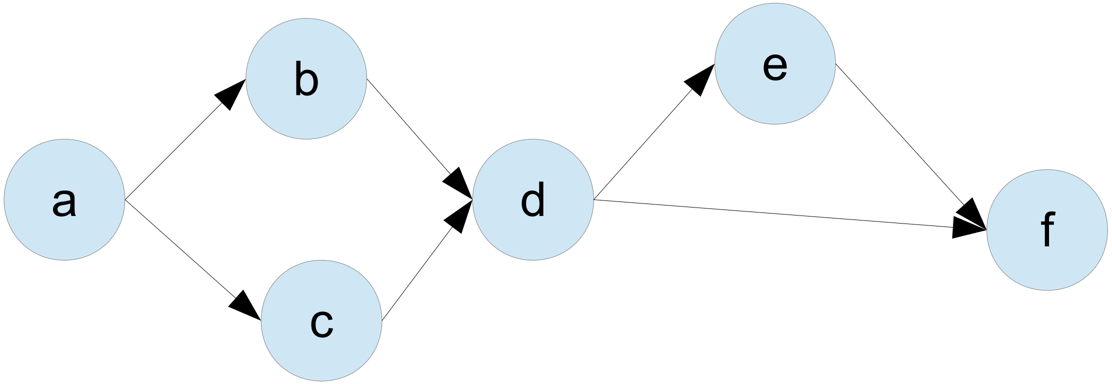
- assuming nodes a-f are part of some larger CFG
- DomBy(a) = { b, c, d }
- DF(a) = { e, f }
- a's definitions may require resolves in blocks e and f

Finding DF for a node

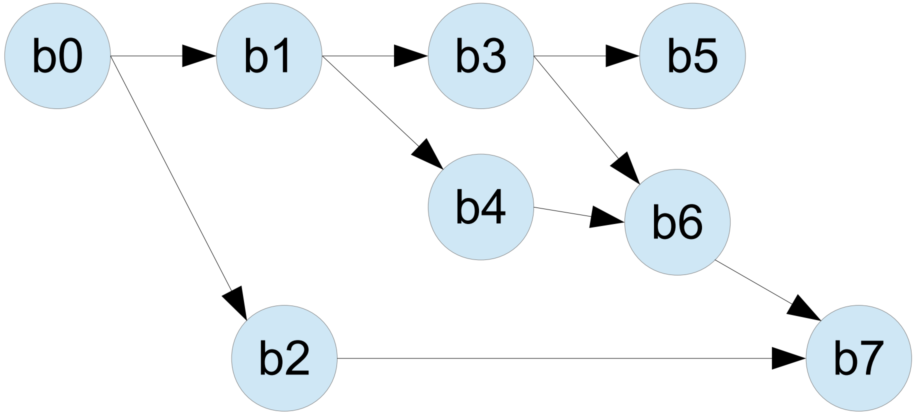
- for block b, the places it may need to request resolve functions can be identified from DF(b)
- to find the nodes in DF(b), explore the tree of nodes in DomBy(b) and identify nodes on paths leaving the tree
- definition: node b is node n's immediate dominator iff
  - n is in DomBy(b), and
  - there is no node, x, such that n is in DomBy(x) and x is in DomBy(b)
- we'll simplify the CFG to represent just the immediate dominators, this will be our dominator tree

Immediate dominator: example

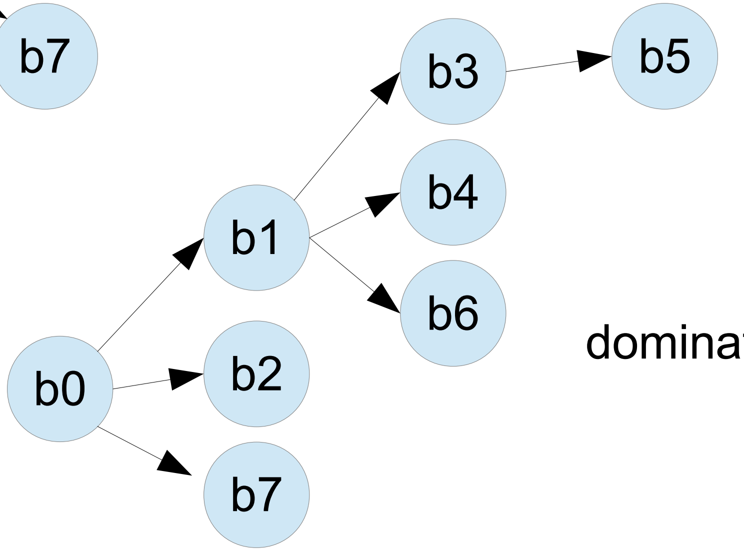
- in the graph below, a is the immediate dominator for b,c,d
- d is the immediate dominator for e, f



Example: dominating tree



original CFG



dominating tree

Dominating trees

- for every node in the dom tree
  - n's parent is its immediate dominator
  - every node dominating n is in the path from the entry point to n
- next steps
  - we'll use the dom tree to compute DF(b)
  - then we'll use that to place the resolve functions
  - then we'll perform any necessary renaming
  - then we'll look at replacing the resolve functions with actual target language code

Computing DF given the tree

```
for every node, n:
   initialize DF(n) to { }
for every node, n:
   if n has multiple predecessors then
      for each predecessor, p, of n:
         i = p
         while i is not the immediate dominator of p
            add n to DF(i)
            i = it's immediate dominator
```

Example (from slide 12)

- Consider node b7 from our earlier example
  - it's immediate dominator in the tree was b0
  - it's predecessors in the CFG were b2 and b6

```
for predecessor b2
   i = b2
   i != b0, so add b7 to DF(b2)
   set i to it's immed dom, which is b0, so stops
for predecessor b6
   i = b6
   i != b0, so add b7 to DF(b6)
   set i to its immed dom, which is b1
   i != b0, so add b7 to DF(b1)
   set i to its immed dom, which is b0, so stops
```

Placing resolve functions

- if n is in DF(b), then any of b's definitions might need a resolve function in n
- they don't need to be there if the definition is never actually used in/after n
- compiler can compose:
  - a list of which names are used across blocks, i.e. generated in one and used elsewhere
  - a list of which blocks define which names

Names and definitions algorithm

```
Names = { }
Defs is a set of sets: which blocks define which names
for each block, b
   varkill = { }
   for each operation (in sequence) x = y OP z
       if y isn't in varkill then add y to Names
       if z isn't in varkill then add z to Names
       add x to varkill
       if x not previously defined, set Defs(x) = {b}
       otherwise add b to Defs(x)
```

Adding the resolve functions

```
for each name, v, in Names
    processing = Defs(v)
    for each block b in processing
        for each target block, t, in DF(b)
            *if t doesn't have a resolve function for v
                add a resolve function for v to t
                add t to processing

* we could prune this step further by adding liveout
  tests to see if b's v is live in d
```

Renaming in SSA

- typically the renaming is done by keeping the source code variable name as a base, and adding an integer index
- a counter is kept for each name, and incremented each time a new name is needed for that base name
- each base name is given a stack
  - push new names as they are generated
  - pop names on exit from the block that generated them

Translating resolves into code

- the resolve functions are simply an abstraction within SSA
- they need to be replaced with the code that copies the right value to the right variable prior to block entry
  - identify the relevant incoming edge in the CFG
  - insert a copy statement from the source block's variable name to the destination block's variable name
- e.g. suppose b_m produced x_i, b_n produced x_j, and the target block had x_k = resolve(x_i, x_j)
  - the edge from b_m would insert code $x_k=x_i$
  - the edge from b_n would insert code $x_k=x_j$