CSE211: Compiler Design

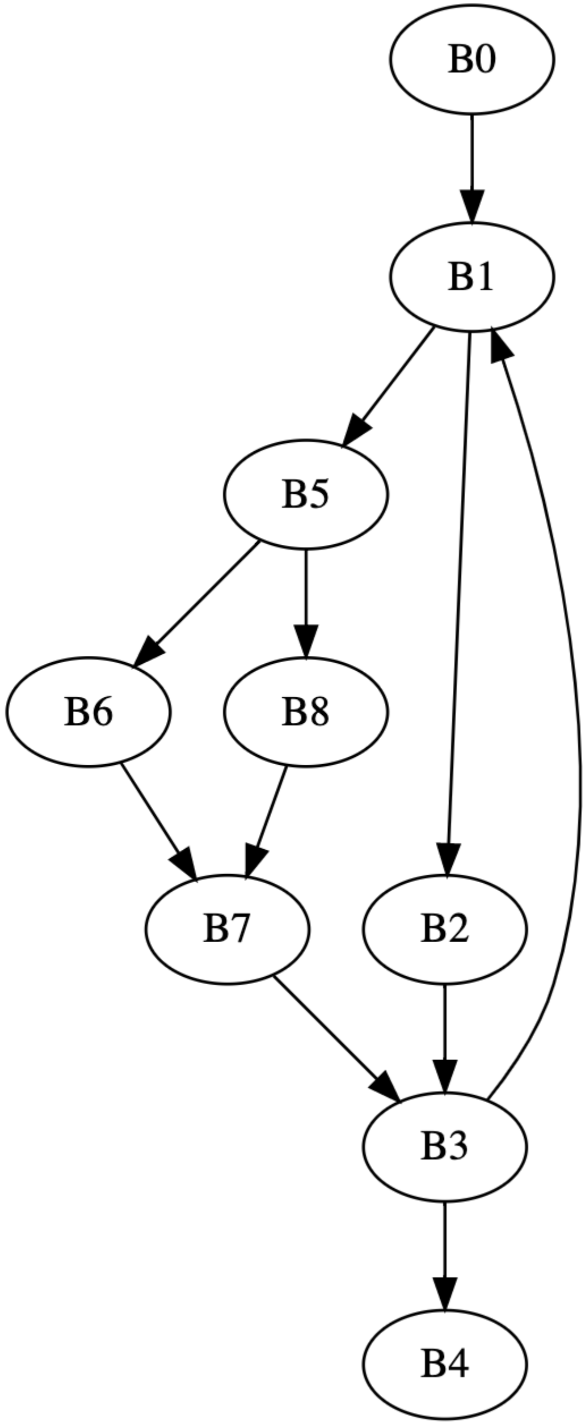
Oct. 22, 2020

- **Topic**: Local value numbering continued and data flow analysis

- **Questions**:

  Questions/comments about homework 1?

  What are some difficult programs for local value numbering?



Announcements

- Homework 2 released! Have a look but don't panic
  - Remember, due dates pushed back 1 week
  - Part 1 should be possible after today's lecture
  - The theory for Part 2 is in lecture. We will go over code next lecture.

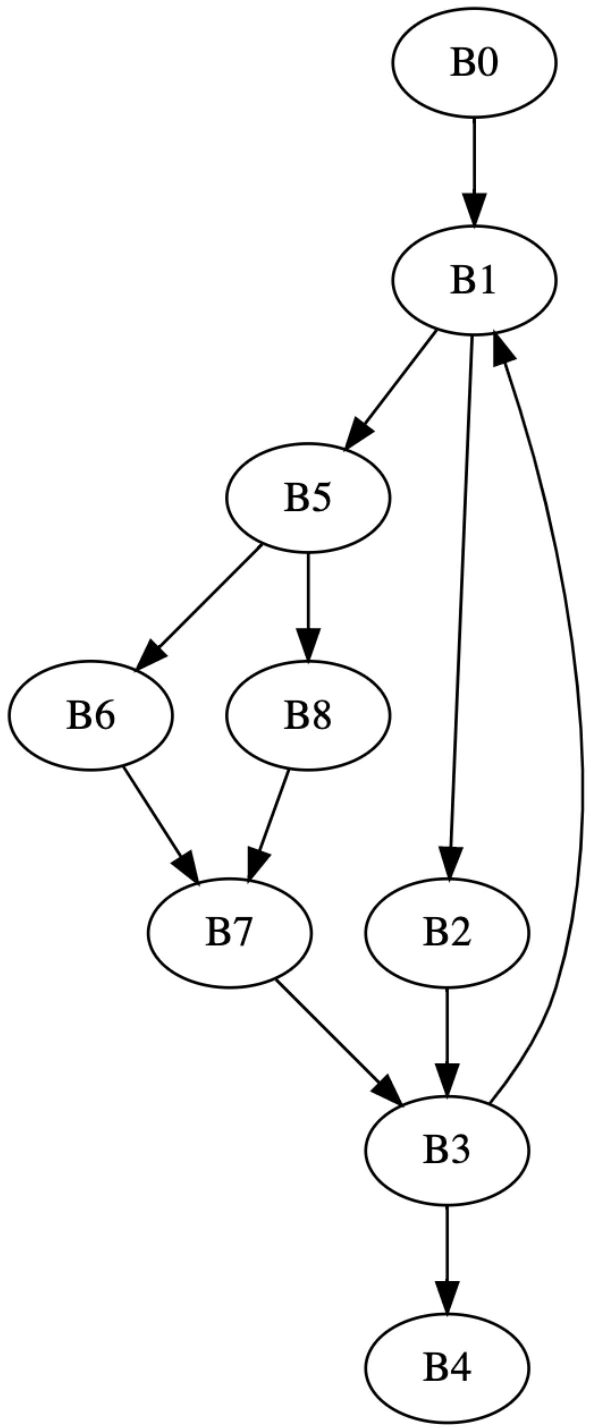
CSE211: Compiler Design

Oct. 22, 2020

- **Topic**: Local value numbering continued and data flow analysis
- **Questions**:

  Questions/comments about homework 1?

  What are some difficult programs for local value numbering?



Local Value Numbering

- Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

→

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
}
```

Local Value Numbering

- Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

→

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
        “b0 + c1” : “a2”,
}
```

Local Value Numbering

- Algorithm: Now that variables are numbered
- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

```
→  a2 = b0 + c1;
   b4 = a2 - d3;
   c5 = b4 + c1;
   d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
}
```

Local Value Numbering

- Algorithm: Now that variables are numbered
- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

```
→  a2 = b0 + c1;
   b4 = a2 - d3;
   c5 = b4 + c1;
   d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
}
```

Local Value Numbering

- Algorithm: Now that variables are numbered
- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;   ←
d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
}
```

Local Value Numbering

- Algorithm: Now that variables are numbered
- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

```
   a2 = b0 + c1;
   b4 = a2 - d3;
→  c5 = b4 + c1;
   d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
}
```

mismatch due to numberings!

Local Value Numbering

- Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
      "b4 + c1" : "c5",
}
```

Local Value Numbering

- Algorithm: Now that variables are numbered
- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.
- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
      "b4 + c1" : "c5",
}
```

Local Value Numbering

- Algorithm: Now that variables are numbered

- Iterate sequentially through instructions. Keep a hash table of the rhs (numbered variables and operation) mapped to their lhs.

- At each step, check to see if the rhs has already been computed.

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = b4;
```

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
      "b4 + c1" : "c5",
}
```

match!

Adding Commutativity

- Certain operators are commutative (e.g. ADD and MULTIPLY)
- In this case, the analysis should consider a deterministic order of operands.
- You can use variable numbers or lexigraphical order

Local Value Numbering

- Algorithm optimization: for commutative operations, re-order operands into a deterministic order

→

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
}
```

Local Value Numbering

- Algorithm optimization: for commutative operations, re-order operands into a deterministic order

cannot re-order because - is not commutative

→

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
       "c1 - b0" : "a2",
}
```

Local Value Numbering

- Algorithm optimization: for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
      "c1 - b0" : "a2",
}
```

Local Value Numbering

- Algorithm optimization: for commutative operations, re-order operands into a deterministic order

re-ordered because a2 < d3 lexigraphically

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
      "c1 - b0" : "a2",
      "a2 * d3" : "f4",
}
```

Local Value Numbering

- Algorithm optimization: for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
      "c1 - b0" : "a2",
      "a2 * d3" : "f4",
}
```

Local Value Numbering

- Algorithm optimization: for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
      "c1 - b0" : "a2",
      "a2 * d3" : "f4",
      "b0 - c1" : "c5",
}
```

Local Value Numbering

- Algorithm optimization: for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = a2 * d3;
```

```
H = {
      "c1 - b0" : "a2",
      "a2 * d3" : "f4",
      "b0 - c1" : "c5",
}
```

Local Value Numbering

- Algorithm optimization: for commutative operations, re-order operands into a deterministic order

```
a2 = c1 - b0;
f4 = d3 * a2;
c5 = b0 - c1;
d6 = f4;
```

```
H = {
      “c1 - b0” : “a2”,
      ”a2 * d3” : “f4”,
      ”b0 - c1” : “c5”,
}
```

Local Value Numbering w/out adding registers

- We’ve assumed we have access to an unlimited number of virtual registers.
- In some cases we may not be able to add virtual registers
  - If an expensive register allocation pass has already occurred.
- We need to give back a program such that variables without numbers is still valid.

Local Value Numbering w/out adding registers

- Example:

```
a = x + y;
a = z;
b = x + y;
```

numbering

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
```

local value numbering with unlimited virtual registers

```
a3 = x1 + y2;
a5 = z4;
b6 = a3;
```

```
a = x + y;
a = z;
b = a;
```

if we drop the numbers, the optimization is invalid.

Local Value Numbering w/out adding registers

- Solutions?

```
a = x + y;
a = z;
b = x + y;
```

numbering →

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

We cannot optimize the first line, but we can optimize the second

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
}
```

```
H = {
}
```

→

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
Current_val = {
                   "a" : 3,
}

H = {
       "x1 + y2" : "a3",
}
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
                   "a" : 3,
}
```

```
H = {
       "x1 + y2" : "a3",
}
```

```
a3 = x1 + y2;
a5 = z4;        <-- current
b6 = x1 + y2;
c7 = x1 + y2;
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
Current_val = {
                  "a" : 5,
}

H = {
      "x1 + y2" : "a3",
}
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;   ←
c7 = x1 + y2;
```

```
Current_val = {
                    "a" : 5,
}


H = {
        "x1 + y2" : "a3",
}
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
Current_val = {
                  "a" : 5,
}

H = {
      "x1 + y2" : "a3",
}
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;   ←
c7 = x1 + y2;
```

```
Current_val = {
                  "a" : 5,
                  "b" : 6
}

H = {
       "x1 + y2" : "b6",
}
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
Current_val = {
                 "a" : 5,
                 "b" : 6
}

H = {
       "x1 + y2" : "b6",
}
```

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = x1 + y2;
```

```
Current_val = {
                  "a" : 5,
                  "b" : 6
}

H = {
       "x1 + y2" : "b6",
}
```

Local Value Numbering w/out adding registers

- Keep another hash table to keep the current variable number

```
a3 = x1 + y2;
a5 = z4;
b6 = x1 + y2;
c7 = b6;
```

```
Current_val = {
                  "a" : 5,
                  "b" : 6
}

H = {
       "x1 + y2" : "b6",
}
```

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

```
Current_val = {
}
```

```
a = x + y;
b = x + y;
a = z;
c = x + y;
```

```
H = {
}
```

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

```
Current_val = {
}
```

```
H = {
}
```

→

```
a3 = x1 + y2;
b4 = x1 + y2;
a6 = z5;
c7 = x1 + y2;
```

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

```
Current_val = {
                "a" : 6,
                "b" : 4
}

H = {
    "x1 + y2" : "a3"
}
```

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

```
Current_val = {
                "a" : 6,
                "b" : 4
}
```

```
H = {
    "x1 + y2" : "a3"
}
```

but we could have replaced it with b4!

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

```
Current_val = {
                  "a" : 3,
}
```

rewind to this point →

```
a3 = x1 + y2;
b4 = x1 + y2;
a6 = z5;
c7 = x1 + y2;
```

```
H = {
   "x1 + y2" : "a3"
}
```

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

```
Current_val = {
                "a" : 3,
                "b" : 4
}


H = {
    "x1 + y2" : ["a3", "b4"],
}
```

hash a list of possible values

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

```
Current_val = {
                "a" : 6,
                "b" : 4
}
```

fast forward again →

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = x1 + y2;
```

```
H = {
    "x1 + y2" : ["a3", "b4"],
}
```

Local Value Numbering w/out adding registers

- Final heuristic: keep sets of possible values

fast forward again →

```
a3 = x1 + y2;
b4 = a3;
a6 = z5;
c7 = b4;
```

```
Current_val = {
                  "a" : 6,
                  "b" : 4
}


H = {
   "x1 + y2" : ["a3", "b4"],
}
```

Local Value Numbering Pitfalls

- Consider a 3 address code that allows memory accesses

```
a[i] = x[j] + y[k];
b[i] = x[j] + y[k];
```

is this transformation allowed?
No!

only if the compiler can prove that `a` does not alias `x` and `y`

```
a[i] = x[j] + y[k];
b[i] = a[i];
```

In the worst case, every time a memory location is updated, the compiler must update the value for all pointers.

Local Value Numbering Pitfalls

- How to number:
  - Number each pointer/index pair

```
(a[i],3) = (x[j],1) + (y[k],2);
b[i] = x[j] + y[k];
```

Local Value Numbering Pitfalls

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that are not proven not to alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

Local Value Numbering Pitfalls

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that are not proven not to alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

compiler analysis:

can we trace `a,x,y` to

```
a = malloc(…);
x = malloc(…);
y = malloc(…);
```

// `a,x,y` are never overwritten

Local Value Numbering Pitfalls

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that are not proven not to alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

in this case we do not have to update the number

compiler analysis:

can we trace `a,x,y` to

```
a = malloc(…);
x = malloc(…);
y = malloc(…);
```

// `a,x,y` are never overwritten

Local Value Numbering Pitfalls

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that are not proven not to alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Local Value Numbering Pitfalls

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that are not proven not to alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

in this case we do not have to update the number

`restrict a`

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Local Value Numbering Pitfalls

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that are not proven not to alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (a[i],3);
```

Optimizing over wider regions

- Local value numbering operated over just one basic block.
- We want optimizations that operate over several basic blocks (a region), or across an entire procedure (global)
- For this, we need Control Flow Graphs and Flow Analysis

Control Flow Graphs

A graph where:

- nodes are basic blocks
- edges mean that it is possible for one block to branch to another

```
start:
r0 = ...;
r1 = ...;
br r0, if, else;

if:
r2 = ...;
br end_if;

else:
r3 = ...;

end_if:
r4 = ...;
```

Control Flow Graphs

A graph where:

- nodes are basic blocks
- edges mean that it is possible for one block to branch to another

```
start:
r0 = ...;
r1 = ...;
br r0, if, else;
```

```
if:
r2 = ...;
br end_if;
```

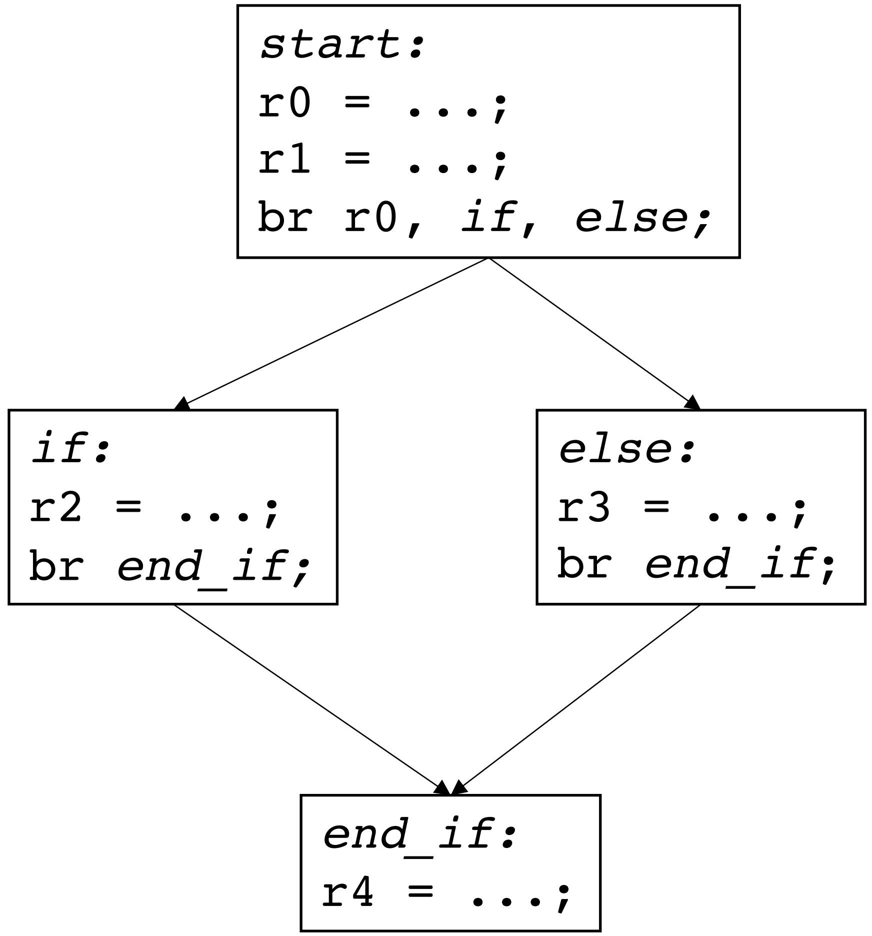
```
else:
r3 = ...;
br end_if;
```

```
end_if:
r4 = ...;
```

Control Flow Graphs

A graph where:

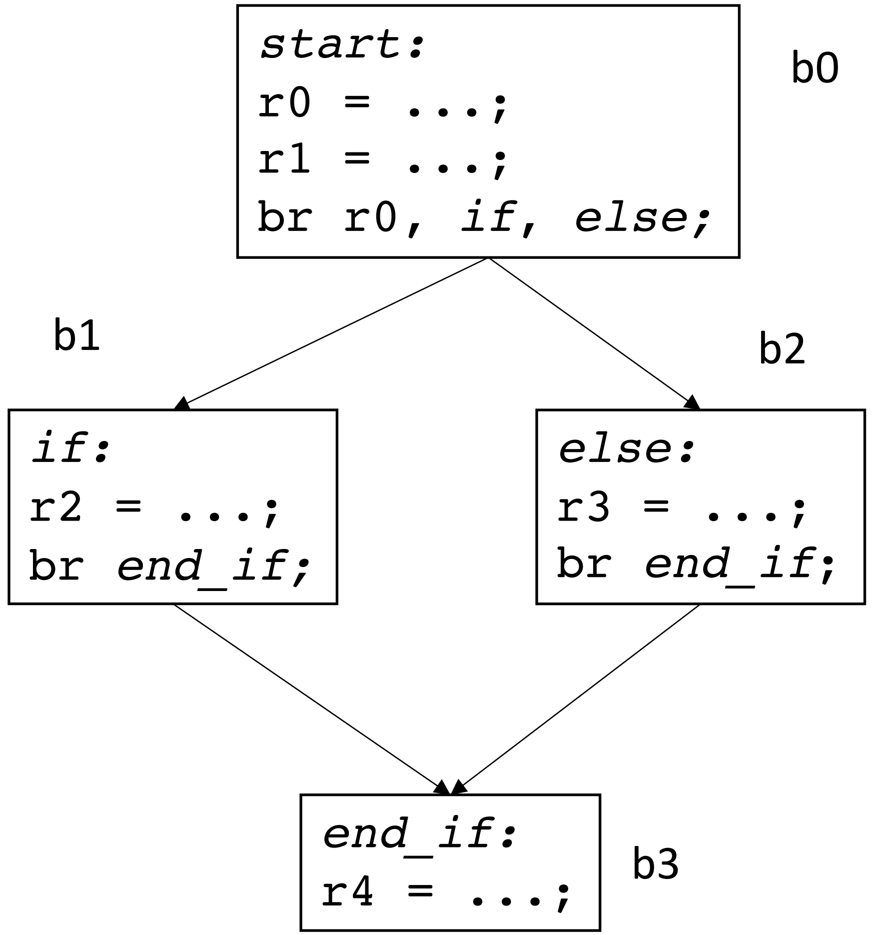
- nodes are basic blocks
- edges mean that it is possible for one block to branch to another



Control Flow Graphs

Simple analysis:

What property did we rely on for local value numbering?



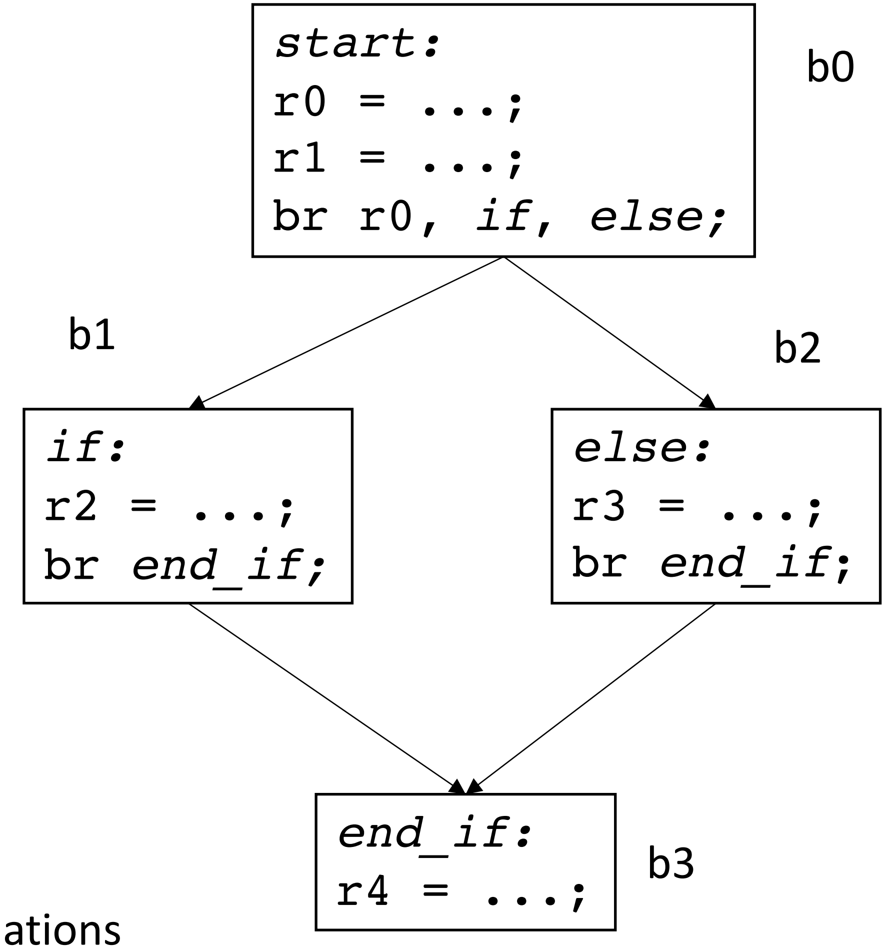
Control Flow Graphs

Simple analysis:

What property did we rely on for local value numbering?

we say that a node b_x “dominates” another node b_y iff:

every path from the start to b_y goes through b_x



are there any non-trivial domination relations in this graph?

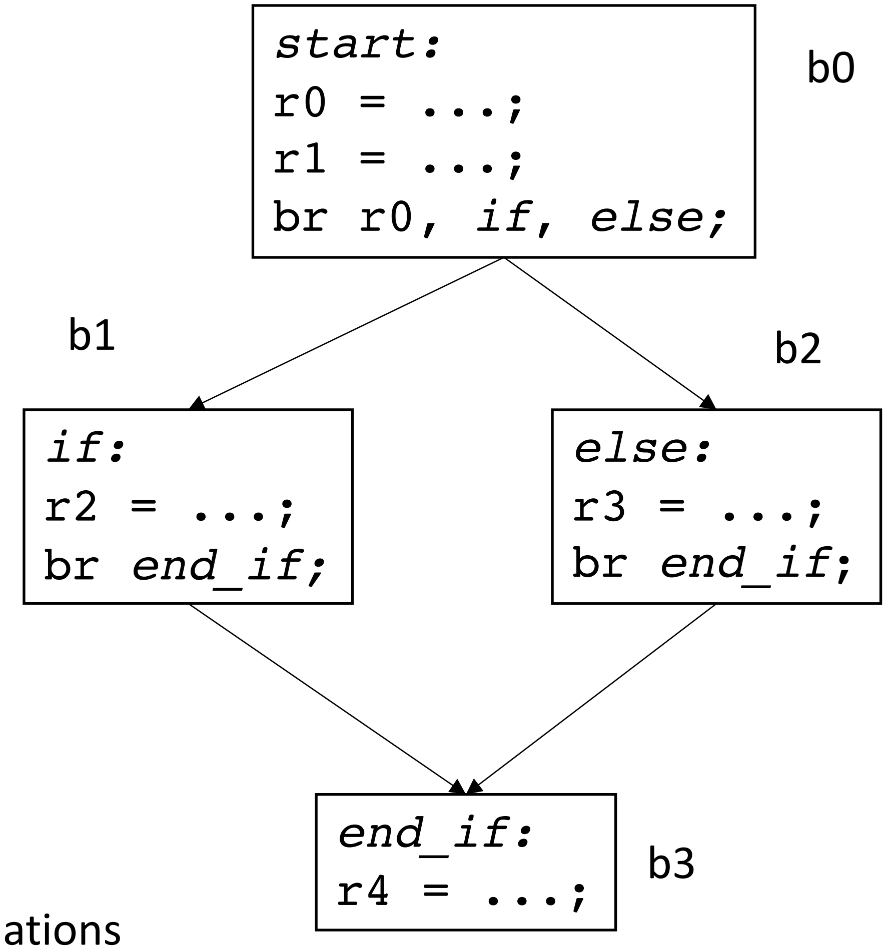
Control Flow Graphs

Simple analysis:

What property did we rely on for local value numbering?

we say that a node b_x "dominates" another node b_y iff:

every path from the start to b_y goes through b_x



are there any non-trivial domination relations in this graph?

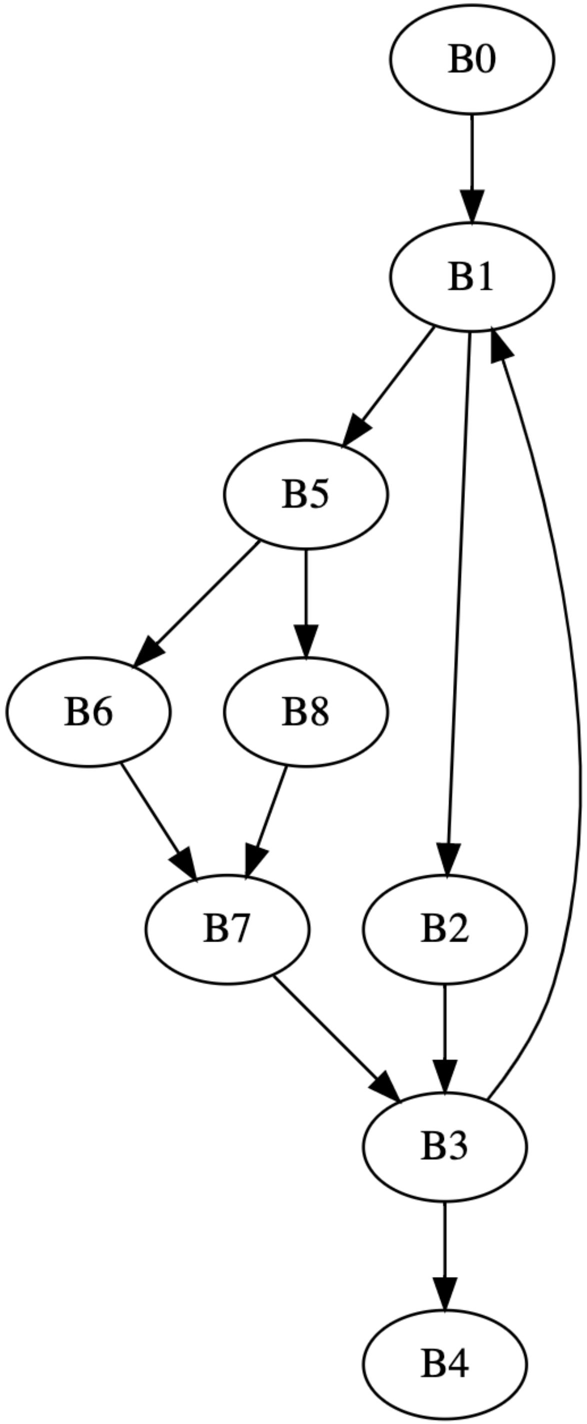
b0 dominates b3

Other examples

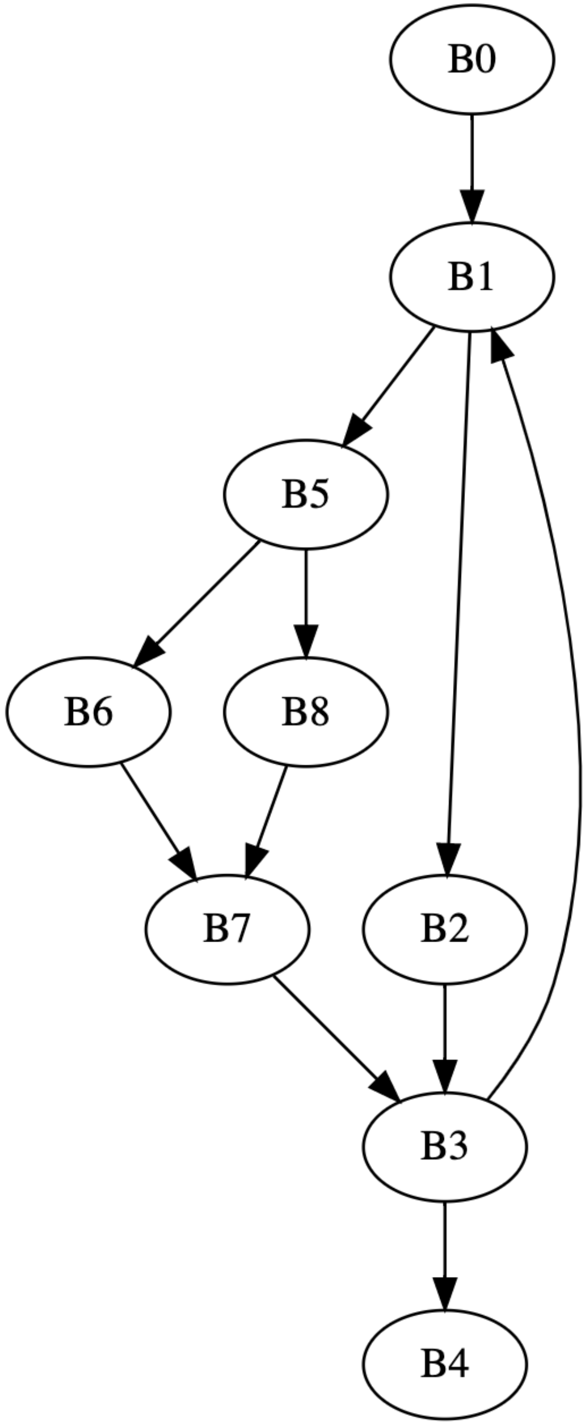
- The PyCFG tool draws CFGs for simple python code:
  - Single statement basic blocks

Dominance

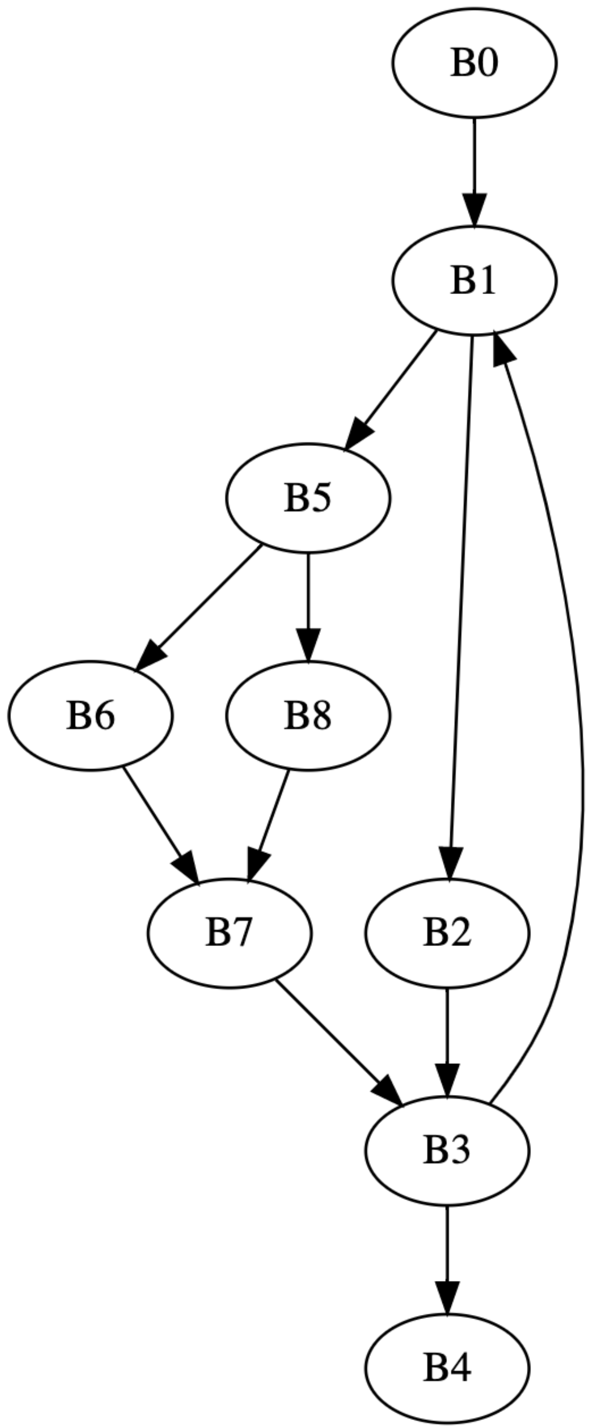
- Given a CFG, determine for each node b_x, the set of nodes that dominate b_x


B0
B1
B5
B6
B8
B7
B2
B3
B4

Node	Dominators
B0	B0
B1	B0, B1
B2	B0, B1, B2
B3	
B4	
B5	
B6	B0,B1,B5,B6
B7	
B8	

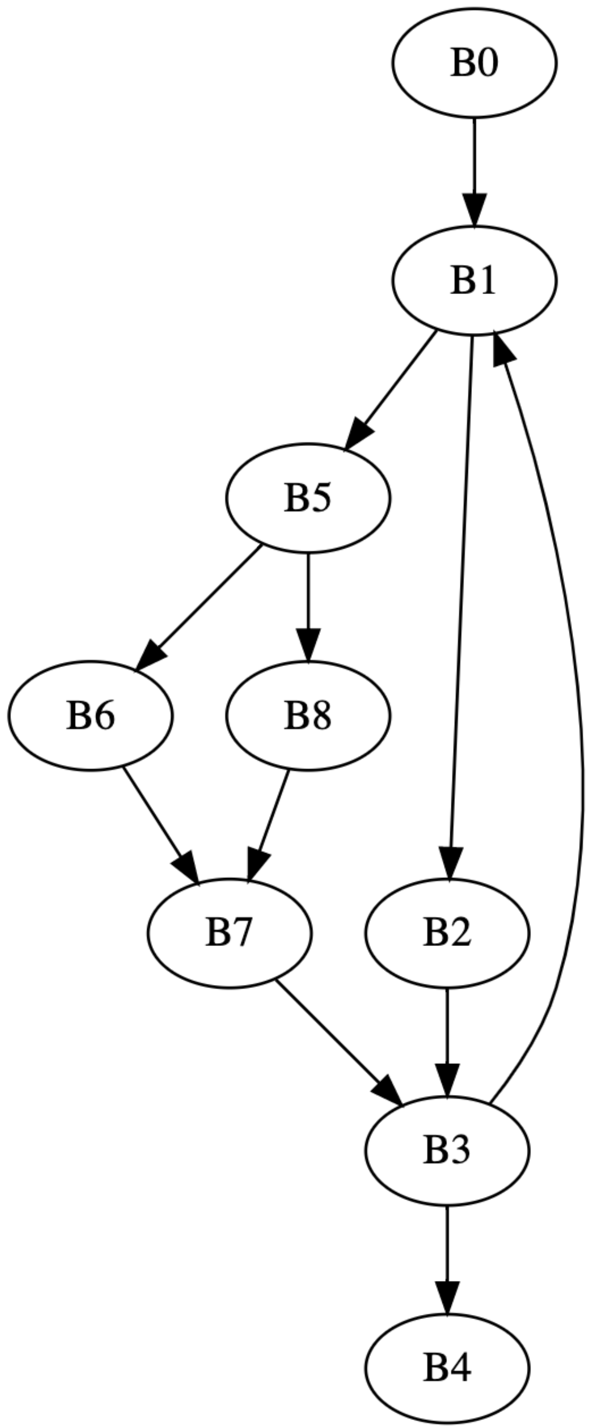


Node	Dominators
B0	B0
B1	B0, B1
B2	B0, B1, B2
B3	B0, B1, B3
B4	B0, B1, B3, B4
B5	B0, B1, B5
B6	B0, B1, B5, B6
B7	B0, B1, B5, B7
B8	B0, B1, B5, B8



Node	Dominators
B0	B0
B1	B0, B1
B2	B0, B1, B2
B3	B0, B1, B3
B4	B0, B1, B3, B4
B5	B0, B1, B5
B6	B0, B1, B5, B6
B7	B0, B1, B5, B7
B8	B0, B1, B5, B8

Can treat this sequence as region,
i.e. and perform local value numbering over it



Computing Dominance

- Iterative fixed point algorithm

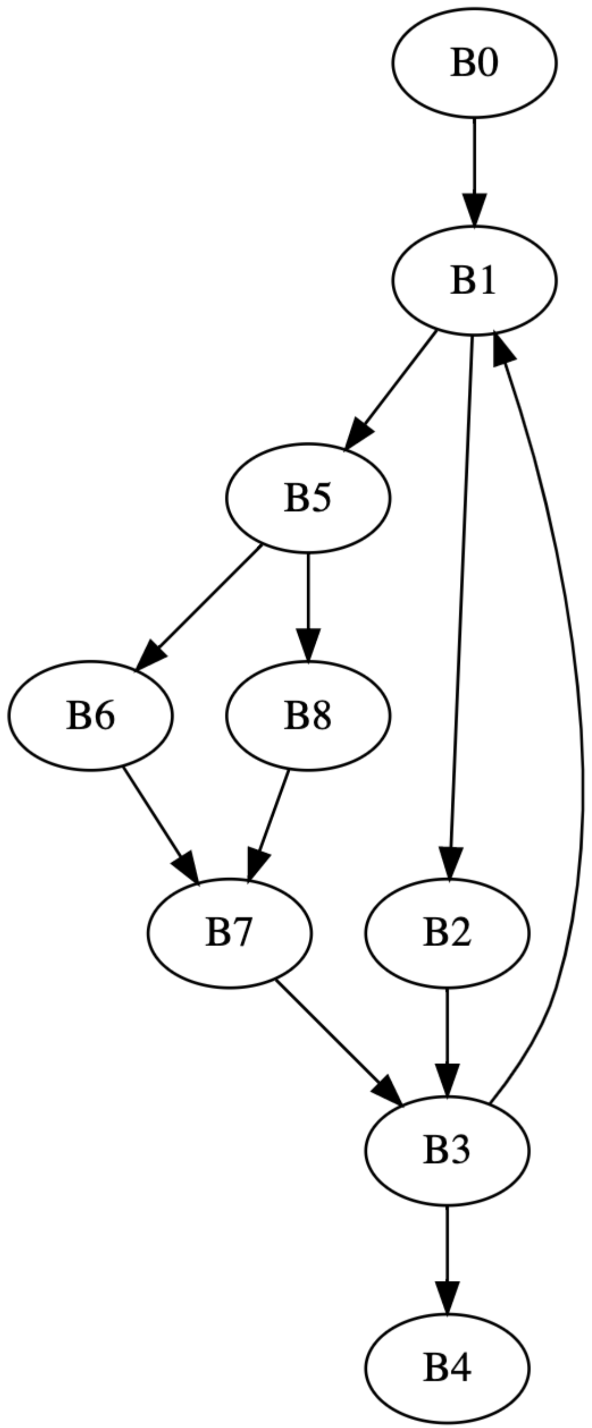
- Initial state, all nodes start with all other nodes are dominators:
  - *Dom(n) = N*
  - *Dom(start) = {start}*

iteratively compute:

$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in preds}(n)} Dom(m) \right)$$

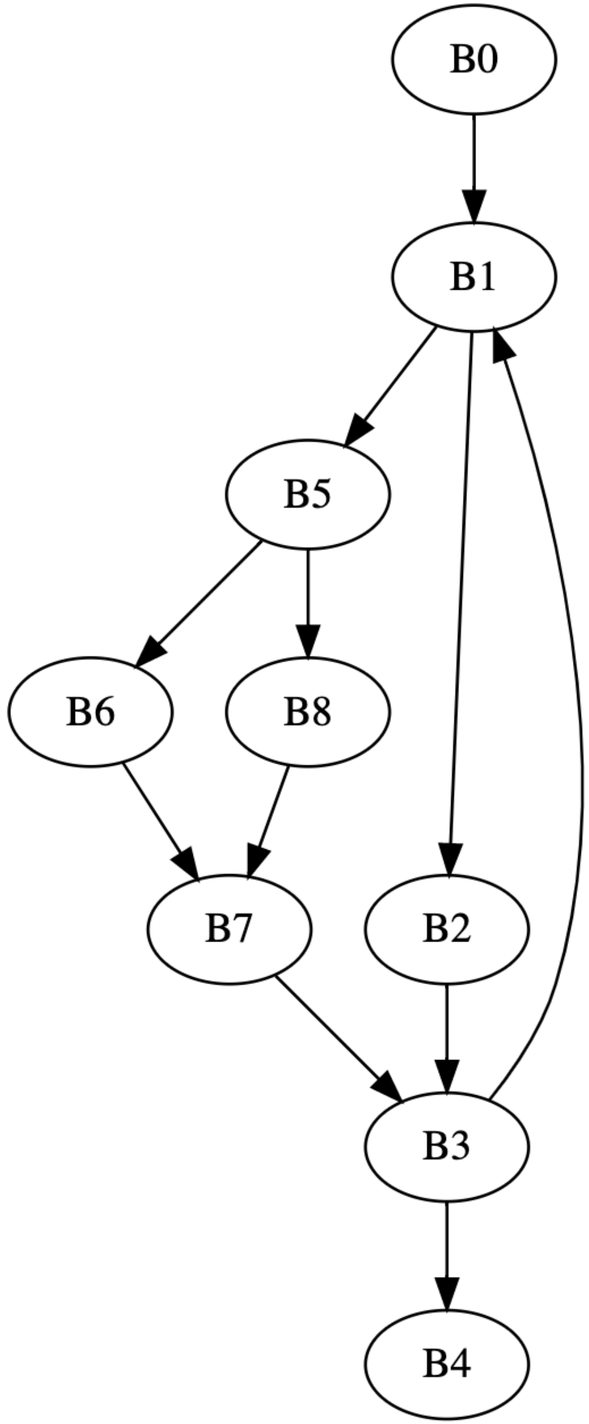
initial conditions

Node	Initial
B0	B0
B1	*N*
B2	*N*
B3	*N*
B4	*N*
B5	*N*
B6	*N*
B7	*N*
B8	*N*



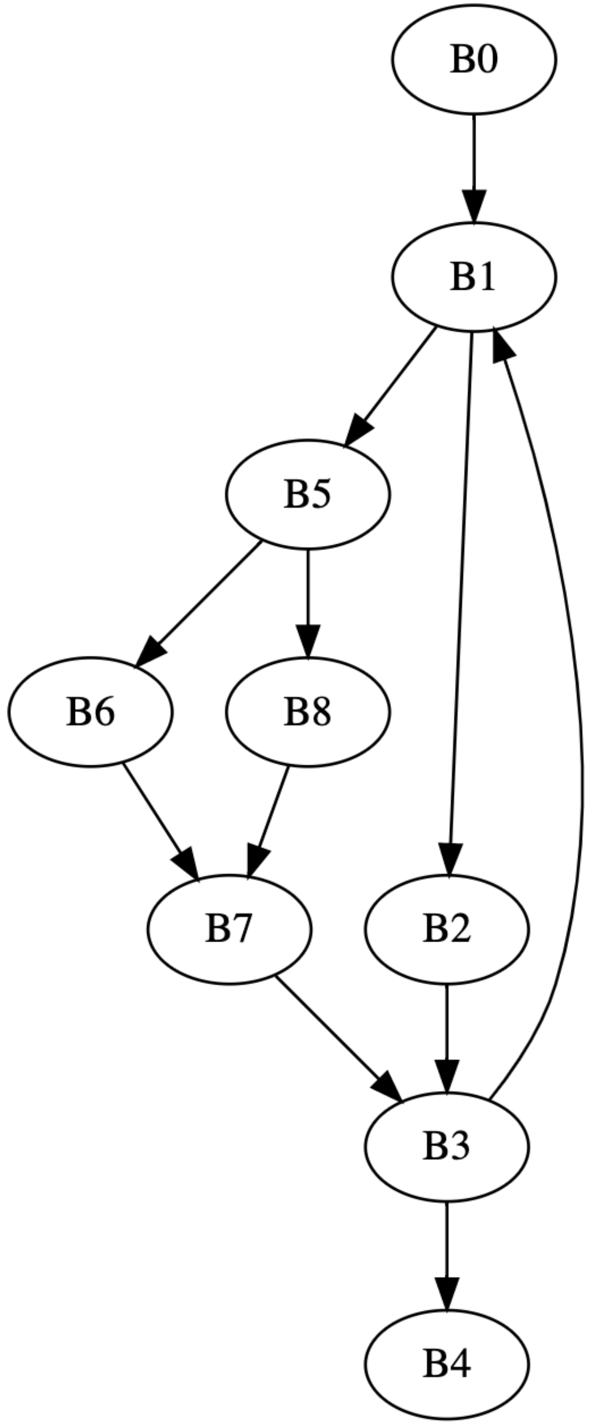
$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in } preds(n)} Dom(m) \right)$$

Node	Initial	I1
B0	B0	...
B1	*N*	B0, B1
B2	*N*	B0, B1, B2
B3	*N*	B0, B1, B2, B3
B4	*N*	
B5	*N*	
B6	*N*	
B7	*N*	
B8	*N*	



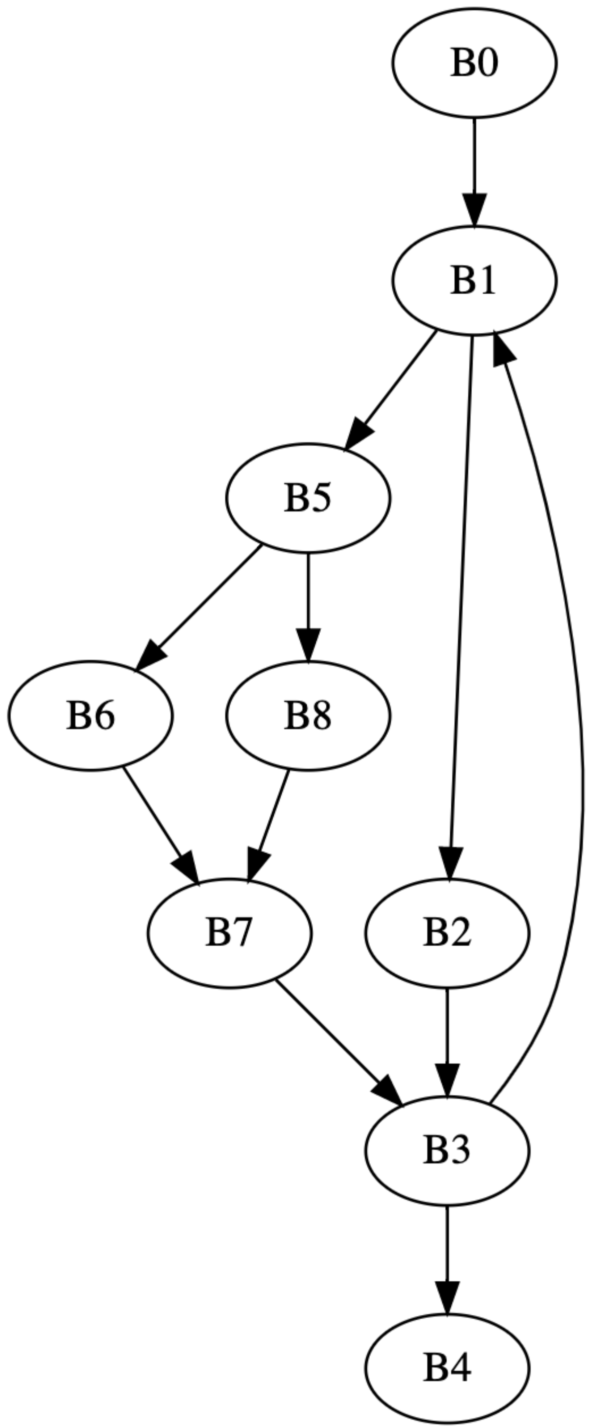
$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in } preds(n)} Dom(m) \right)$$

Node	Initial	I1
B0	B0	B0
B1	*N*	B0,B1
B2	*N*	B0,B1,B2
B3	*N*	B0,B1,B2,B3
B4	*N*	B0,B1,B2,B3,B4
B5	*N*	B0,B1,B5
B6	*N*	B0,B1,B5,B6
B7	*N*	B0,B1,B5,B6,B7
B8	*N*	B0,B1,B5,B8



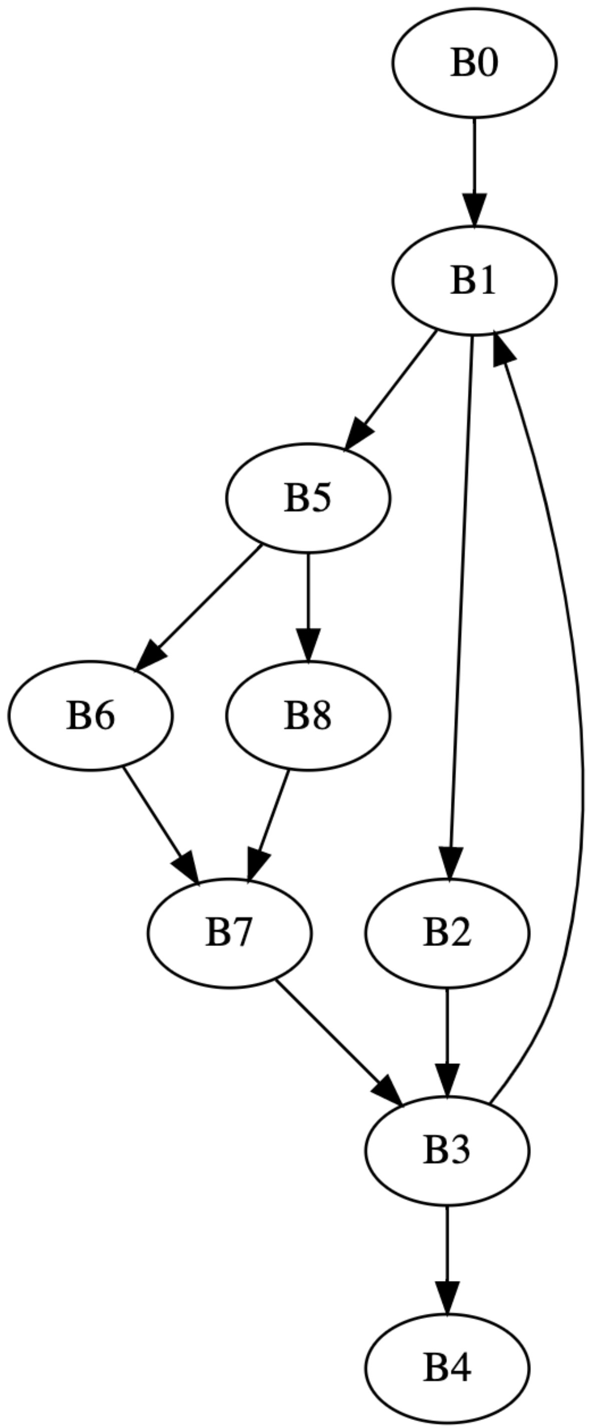
$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in preds}(n)} Dom(m) \right)$$

Node	Initial	I1	I2
B0	B0	B0	...
B1	*N*	B0,B1	...
B2	*N*	B0,B1,B2	
B3	*N*	B0,B1,B2,B3	
B4	*N*	B0,B1,B2,B3,B4	
B5	*N*	B0,B1,B5	
B6	*N*	B0,B1,B5,B6	
B7	*N*	B0,B1,B5,B6,B7	B0, B1, B5
B8	*N*	B0,B1,B5,B8	



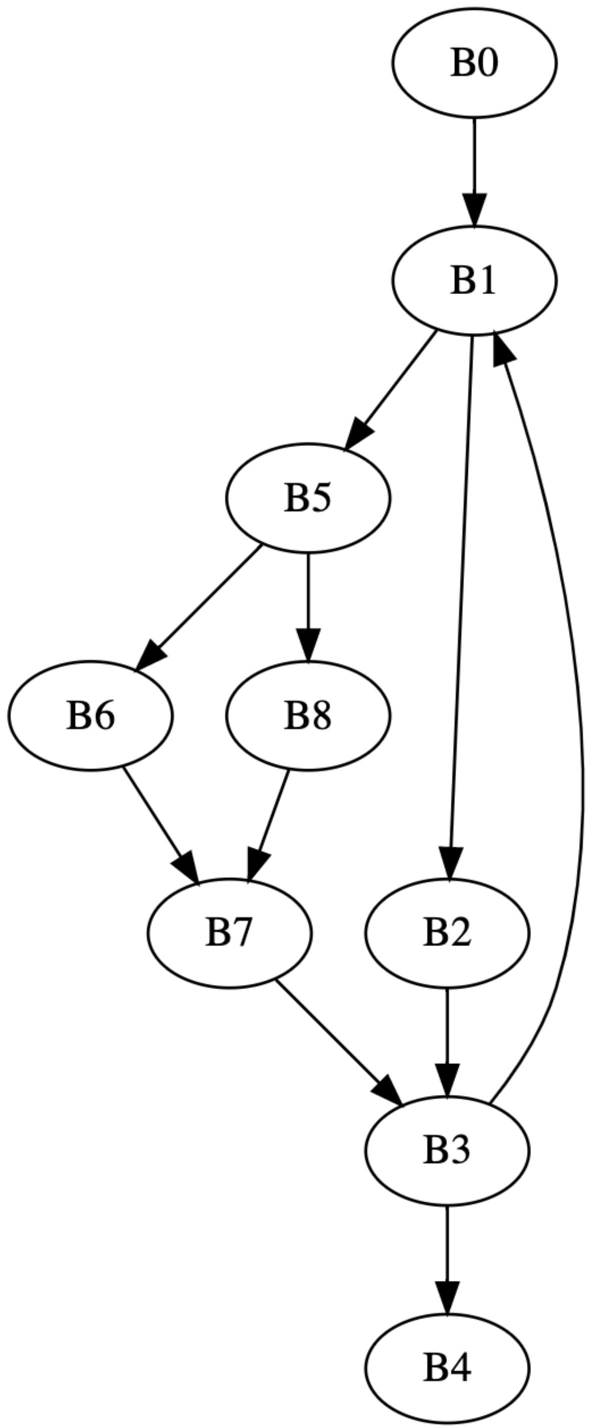
$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in } preds(n)} Dom(m) \right)$$

Node	Initial	I1	I2
B0	B0	B0	...
B1	*N*	B0,B1	...
B2	*N*	B0,B1,B2	...
B3	*N*	B0,B1,B2,B3	B0,B1,B3
B4	*N*	B0,B1,B2,B3,B4	B0,B1,B3,B4
B5	*N*	B0,B1,B5	...
B6	*N*	B0,B1,B5,B6	...
B7	*N*	B0,B1,B5,B6,B7	B0,B1,B5,B7
B8	*N*	B0,B1,B5,B8	...



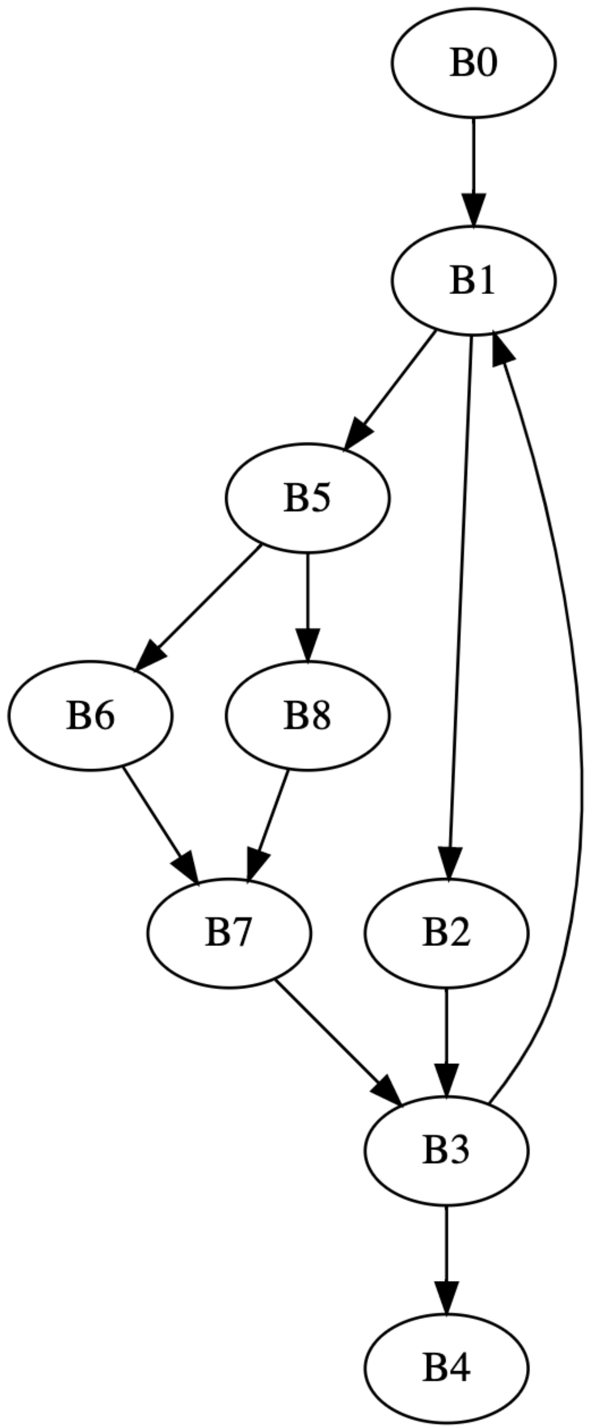
$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in } preds(n)} Dom(m) \right)$$

Node	Initial	I1	I2	I3
B0	B0	B0	…	
B1	*N*	B0,B1	…	
B2	*N*	B0,B1,B2	…	
B3	*N*	B0,B1,B2,B3	B0,B1,B3	
B4	*N*	B0,B1,B2,B3,B4	B0,B1,B3,B4	
B5	*N*	B0,B1,B5	…	
B6	*N*	B0,B1,B5,B6	…	
B7	*N*	B0,B1,B5,B6,B7	B0,B1,B5,B7	
B8	*N*	B0,B1,B5,B8	…	



$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in } preds(n)} Dom(m) \right)$$

Node	Initial	I1	I2	I3
B0	B0	B0	...	...
B1	*N*	B0,B1	...	...
B2	*N*	B0,B1,B2	...	...
B3	*N*	B0,B1,B2,B3	B0,B1,B3	...
B4	*N*	B0,B1,B2,B3,B4	B0,B1,B3,B4	...
B5	*N*	B0,B1,B5	...	...
B6	*N*	B0,B1,B5,B6	...	...
B7	*N*	B0,B1,B5,B6,B7	B0,B1,B5,B7	...
B8	*N*	B0,B1,B5,B8	...	...



No change so algorithm terminates!

Next week

- Flow analysis examples continued:
  - node traversal order for faster convergence
  - Live variable analysis

- Generalized framework for flow analysis

- SSA form