# Data Flow Analysis

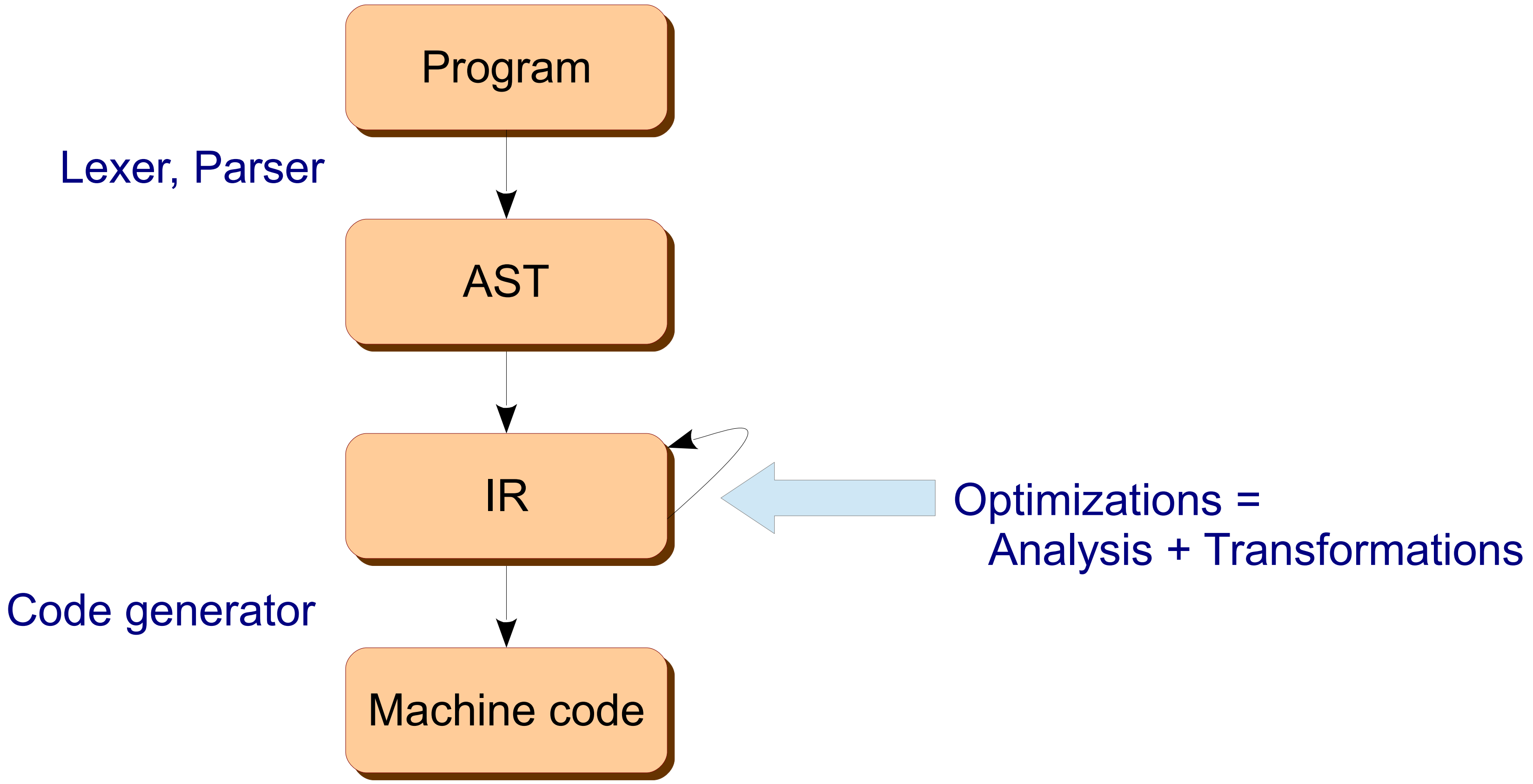
Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2017

# Outline

- What is DFA?
  - Reaching definitions
  - Live variables
- DFA framework
  - Monotonicity
  - Confluence operator
  - MFP/MOP solution
- Analysis dimensions

# Compiler Organization



# Compiler Basics

- Program as Data
- Control-Flow Graph (CFG)
- Basic Blocks
- Optimizations
  - *gcc -O2 prog.c*

```
int main() {
  int x = 1;
  if (x > 0)
      ++x;
  else
      x = 100;
  printf(“%d\n”, x);
}
```

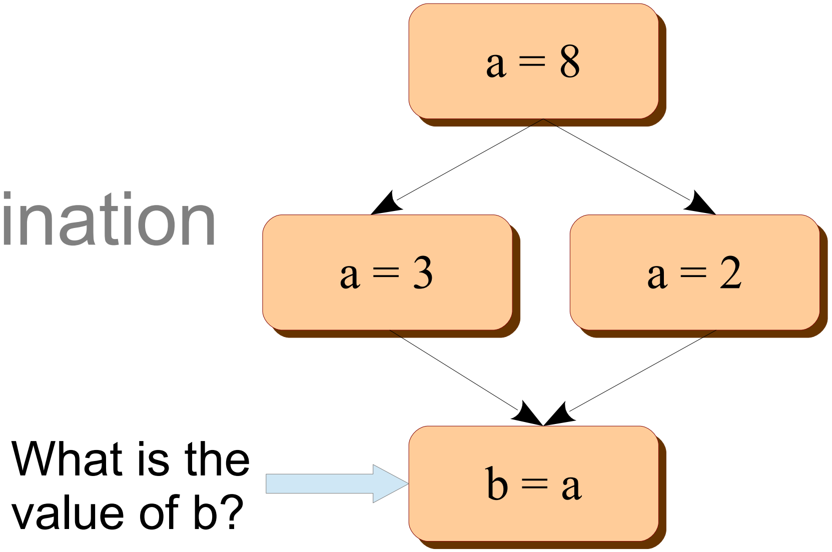
```
int main() {
  int x = 1;
  if (1 > 0)
      ++x;
  else
      x = 100;
  printf(“%d\n”, x);
}
```

```
int main() {
  int x = 1;
  ++x;
  printf(“%d\n”, x);
}
```

```
int main() {
  printf(“%d\n”, 2);
}
```

# Data Flow Analysis

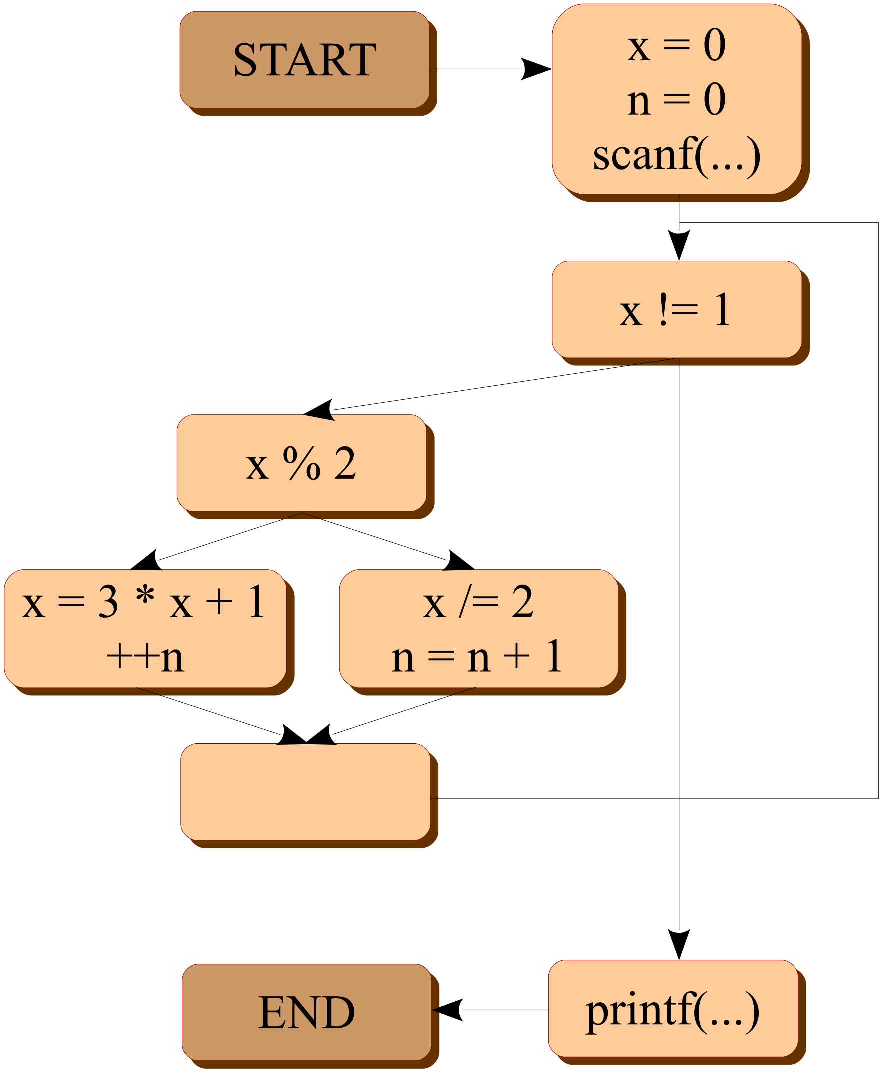
- Flow-sensitive: Considers the control-flow in a function
- Operates on a flow-graph with nodes as basic-blocks and edges as the control-flow
- Examples
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination



# Classwork

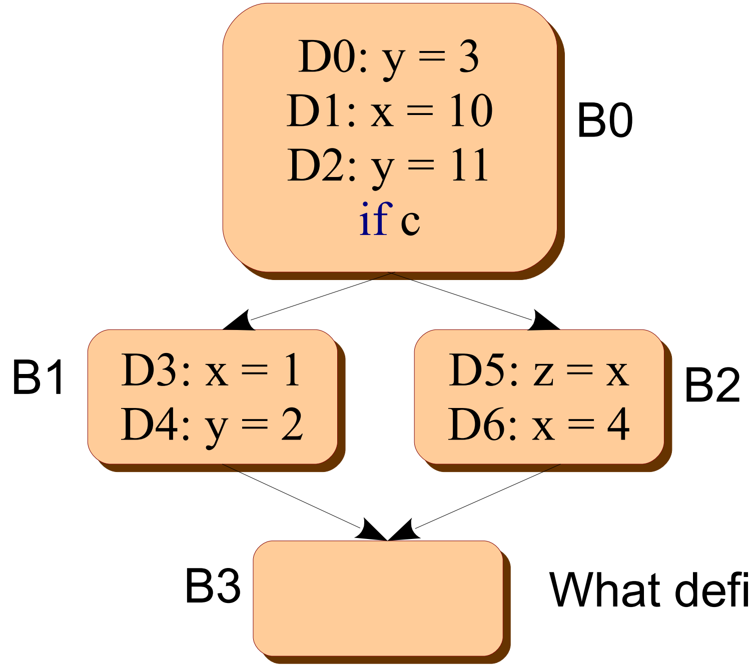
- Draw the CFG for the following program.

```
int main() {
  int x = 0, n = 0;
  scanf("%d", &x);

  while (x != 1) {
    if (x % 2) {
      x = 3 * x + 1;
      ++n;
    } else {
      x /= 2;
      n = n + 1;
    }
  }
  printf("%d\n", n);
}
```



# Reaching Definitions

- Every assignment is a definition
- A definition d reaches a program point p if there exists a path from the point immediately following d to p such that d is not killed along the path.



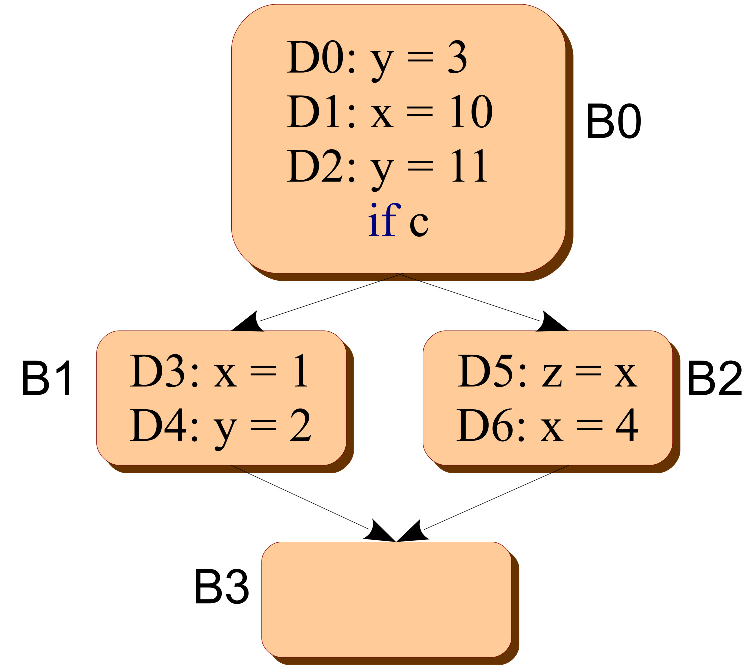

What definitions reach B3?

# DFA Equations

- in(B) = set of data flow facts entering block B
- out(B) = …
- gen(B) = set of data flow facts generated in B
- kill(B) = set of data flow facts from the other blocks killed in B

# DFA for Reaching Definitions

- in(B) = U out(P) where P is a predecessor of B
- out(B) = gen(B) U (in(B) – kill(B))
- Initially, out(B) = { }




gen(B0) = {D1, D2}
gen(B1) = {D3, D4}
gen(B2) = {D5, D6}
gen(B3) = { }

kill(B0) = {D3, D4, D6}
kill(B1) = {D0, D1, D2, D6}
kill(B2) = {D1, D3}
kill(B3) = { }

| | in1 | out1 | in2 | out2 | in3 | out3 |
|---|---|---|---|---|---|---|
| **B0** | {} | {D1, D2} | {} | {D1, D2} | {} | {D1, D2} |
| **B1** | {} | {D3, D4} | {D1, D2} | {D3, D4} | {D1, D2} | {D3, D4} |
| **B2** | {} | {D5, D6} | {D1, D2} | {D2, D5, D6} | {D1, D2} | {D2, D5, D6} |
| **B3** | {} | {} | {D3, D4, D5, D6} | {D3, D4, D5, D6} | {D2, D3, D4, D5, D6} | {D2, D3, D4, D5, D6} |

# Algorithm for Reaching Definitions

**for** each basic block B

compute gen(B) and kill(B)

out(B) = {}

**do** {

**for** each basic block B

in(B) = U out(P) where P \in pred(B)

out(B) = gen(B) U (in(B) - kill(B))

} **while** in(B) changes for any basic block B

Can you do better?
Hint: Worklist

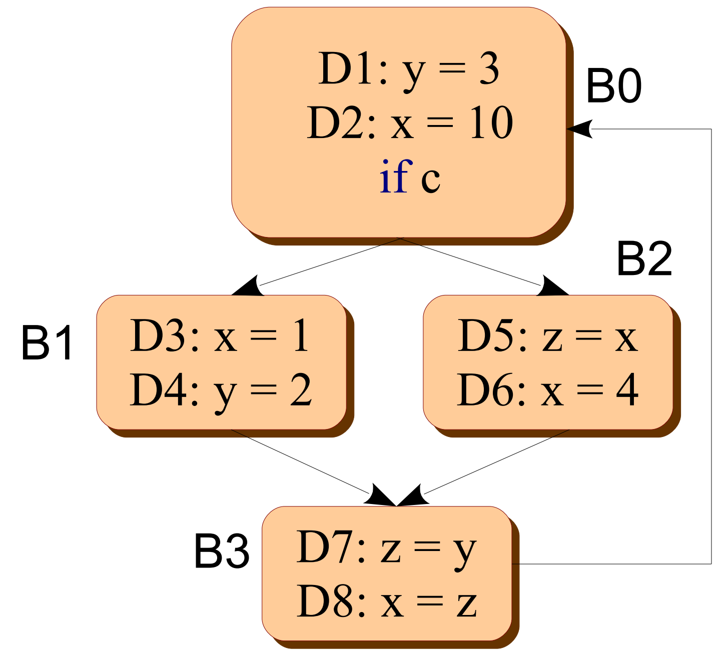
# Classwork

- in(B) = U out(P) where P is a predecessor of B
- out(B) = gen(B) U (in(B) – kill(B))

- Initially, out(B) = { }

gen(B0) = {D1, D2}
gen(B1) = {D3, D4}
gen(B2) = {D5, D6}
gen(B3) = {D7, D8}

kill(B0) = {D3, D4, D6, D8}
kill(B1) = {D1, D2, D6, D8}
kill(B2) = {D2, D3, D7, D8}
kill(B3) = {D2, D3, D5, D6}




| | in1 | out1 | in2 | out2 | in3 | out3 | in4 | out4 |
|---|---|---|---|---|---|---|---|---|
| B0 | {} | {D1, D2} | {D7, D8} | {D1, D2, D7} | {D4, D7, D8} | {D1, D2, D7} | {D1,4,7, 8} | {D1,2,7} |
| B1 | {} | {D3, D4} | {D1, D2} | {D3, D4} | {D1, D2, D7} | {D3, D4, D7} | {D1,2,7} | {D3,4,7} |
| B2 | {} | {D5, D6} | {D1, D2} | {D1, D5, D6} | {D1, D2, D7} | {D1, D5, D6} | {D1,2,7} | {D1,5,6} |
| B3 | {} | {D7, D8} | {D3, D4, D5, D6} | {D4, D7, D8} | {D1, D3, D4, D5, D6} | {D1,4, 7, 8} | {D1,3,4,5,6,7} | {D1,4,7,8} |

# DFA for Reaching Definitions

| | |
|---|---|
| **Domain** | Sets of definitions |
| **Transfer function** | in(B) = U out(P)<br>out(B) = gen(B) U (in(B) - kill(B)) |
| **Direction** | Forward |
| **Meet / confluence operator** | U |
| **Initialization** | out(B) = { } |

# Memory Optimization

- Reuse memory / register wherever possible.
- y is dead at lines 2, 3, 4.
- It is also dead at else block.
- z and y can reuse memory / register.

```
0  int x = 2, y = 3, z = 1;
1  if (x == 2) {
2        y = z;
3        x = 9;
4        y = 7;
5        x = x – y;
6  } else {
7        y = x + z;
8        ++x;
9  }
10 printf("%d", y);
```

# DFA for Live Variables

| | |
|---|---|
| **Domain** | Sets of variables |
| **Transfer function** | in(B) = use(B) U (out(B) - def(B))<br>out(B) = U in(S) where S is a successor of B |
| **Direction** | Backward |
| **Meet / confluence operator** | U |
| **Initialization** | in(B) = { } |

A variable v is live at a program point p if v is used along some path in the flow graph starting at p.
Otherwise, the variable v is dead.

# Classwork

- Write an algorithm for Live Variable Analysis

```
for each basic block B
  compute gen(B) and kill(B)
  out(B) = {}

do {
  for each basic block B
     in(B) = U out(P) where P \in pred(B)
     out(B) = gen(B) U (in(B) - kill(B))
} while in(B) changes for any basic block B
```

Algo for **reaching definitions**

| | |
|---|---|
| **Domain** | Sets of variables |
| **Transfer function** | in(B) = use(B) U (out(B) - def(B))<br>out(B) = U in(S) where S is a successor of B |
| **Direction** | Backward |
| **Meet / confluence operator** | U |
| **Initialization** | in(B) = { } |

Parameters for **live variable analysis**

# Direction and Confluence

| | Forward | Backward |
|---|---|---|
| **U** | Reaching Definitions | Live Variables |
| ∩ | Available Expressions | Very Busy Expressions |

An expression is available at a program point P if the expression is computed along each path to P (from START) without getting invalidated.

An expression is very busy at a program point P if along each path from P (to END) the expression is computed without getting invalidated.

# Data Flow Framework

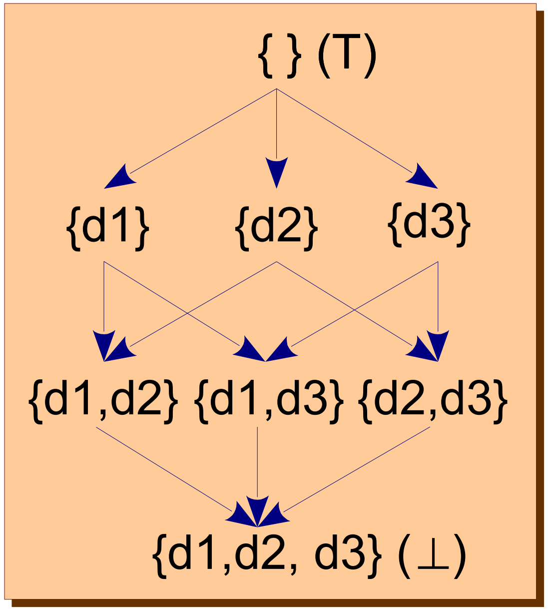
- Point: start or end of a basic block
- Information flow direction: forward / backward
- Transfer functions
- Meet / confluence operator

- One can define a transfer function over a path in the CFG $f_k(f_{k-1}(\ldots f_2(f_1(f_0(\top))\ldots))$ // small k (block)
- $MOP(x) = \sqcap f_K(\top)$ $K \in Paths(x)$ // capital K (path)

*Meet over all paths*
*Path enumeration is expensive*

# Structure in Data Flow Framework

- A semilattice $L$ with a binary meet operator $\sqcap$, such that $a, b, c \in L$
  - Idempotency: $a \sqcap a = a$
  - Commutativity: $a \sqcap b = b \sqcap a$
  - Associativity: $a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$
- $\sqcap$ imposes an order on $L$
  - $a >= b \Leftrightarrow a \sqcap b = b$
- $L$ has a bottom element $\perp$, $a \sqcap \perp = \perp$
- $L$ has a top element T, $a \sqcap T = a$




Reaching Definitions Lattice

# Monotone Framework

- A framework $<L, \sqcap, F>$ is monotone if $F$ is monotonic, i.e.,

$$(\forall f \in F)(\forall x, y \in L),\ x \geq y \Rightarrow f(x) \geq f(y)$$

- If a data-flow framework is monotonic, the convergence (termination) is guaranteed for finite height lattices.

# Distributive Framework

- A framework <L, ⊓,F> is distributive if F is distributive, i.e.,

$$(\forall f \in F)(\forall x, y \in L)\ f(x \sqcap y) \leq f(x) \sqcap f(y)$$

- Maximal fixed point (MFP) solution is obtained with our iterative DFA.
- MFP is unique and order independent.
- The best we can do is MOP (most feasible, but undecidable).
- In general, MFP ≤ MOP ≤ Perfect solution.
- If distributive, MFP = MOP.
- Every distributive function is also monotonic.

# Outline

- What is DFA?
  - Reaching definitions
  - Live variables
- DFA framework
  - Monotonicity
  - Confluence operator
  - MFP/MOP solution
- Analysis dimensions

**How many ancestor names do you need to almost uniquely identify a student in campus?**

# Analysis Dimensions

An analysis's precision and efficiency is guided by various design decisions.

- Flow-sensitivity
- Context-sensitivity
- Path-sensitivity
- Field-sensitivity

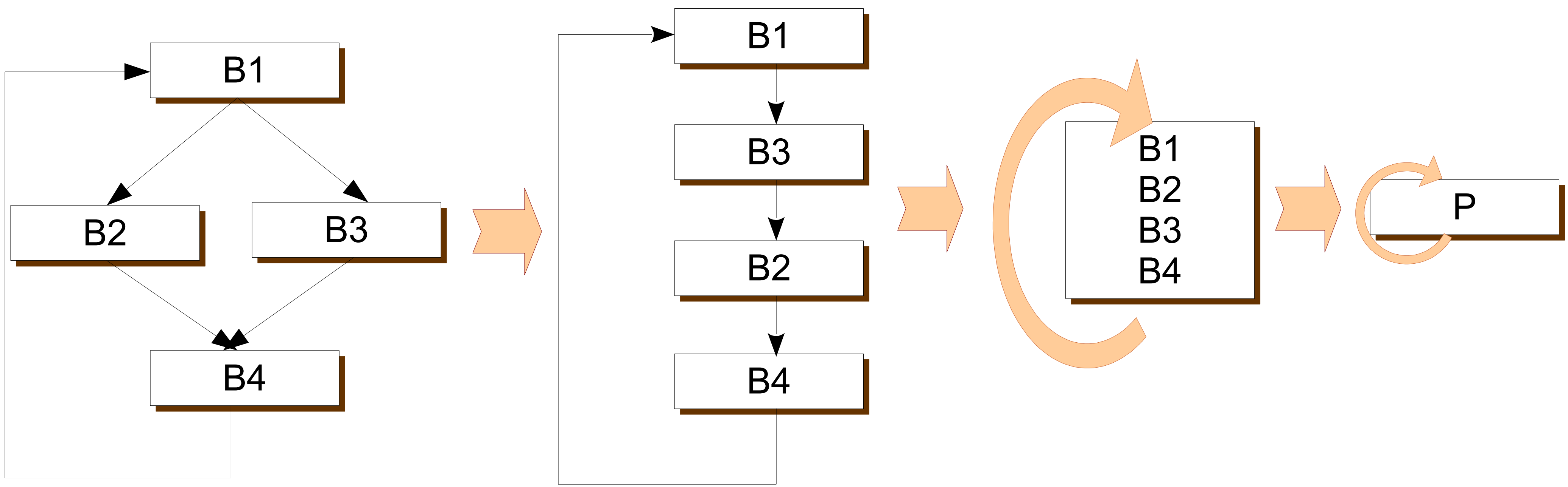

How many hands are required to know the time precisely?

# Flow-sensitivity

```
L0:  a = 0;
L1:  a = 1;
L2:  ...
```

Flow-sensitive solution: *at L1 a is 0, at L2 a is 1*
Flow-insensitive solution: *in the program a is in {0, 1}*

Flow-insensitive analyses ignore the control-flow in the program.



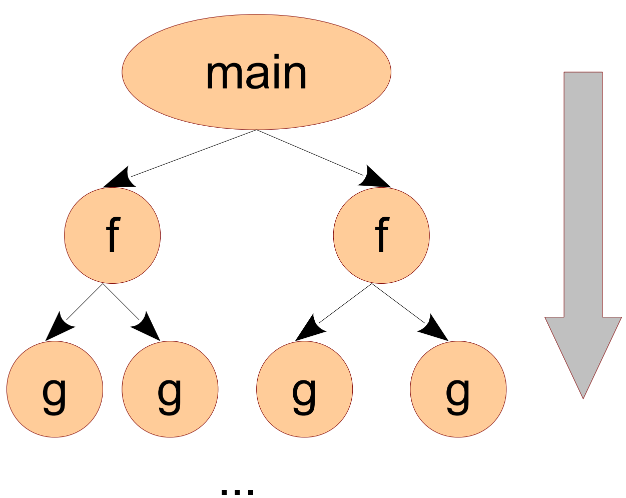
# Context-sensitivity

```
main() {
      L0:  fun(0);
      L1:  fun(1);
}
```

```
fun(int x) {
      y = x;
}
```

Context-sensitive solution:
*y is 0 along L0, y is 1 along L1*

Context-insensitive solution:
*y is in {0, 1} in the program*




Exponential
Number of
contexts

Along main-f1-g1, …
Along main-f1-g2, …
Along main-f2-g1, …
Along main-f2-g2, ...

Exponential time requirement

Exponential storage requirement

# Context-sensitivity

```
main() {
        L0:  fun(0);
        L1:  fun(1);
}
```

```
fun(int x) {
        y = x;
}
```

Context-sensitive solution:
*y is 0 along L0, y is 1 along L1*

Context-insensitive solution:
Inter-procedural → *y is in {0, 1} in the program*
intra-procedural → *y is in {-∞, +∞} in the program*

# Path-sensitivity

```
if (a == 0)
      b = 1;
else
      b = 2;
```

Path-sensitive solution:
*b is 1 when a is 0, b is 2 when a is not 0*

Path-insensitive solution:
*b is in {1, 2} in the program*

```
if (c1)
   while (c2) {
     if (c3)
       …
     else
       for (; c4; )
          ...
   }
else
   ...
```

c1 and c2 and c3, …
c1 and c2 and !c3 and c4, …
c1 and c2 and !c3 and !c4, …
c1 and !c2, …
!c1 ...
...

# Field-sensitivity

```
struct T s;

s.a = 0;
s.b = 1;
```

Field-sensitive solution:
  *s.a is 0, s.b is 1*

Field-insensitive solution:
  *s is in {0, 1}*

Aggregates are collapsed into a single variable.
  e.g., arrays, structures, unions.

This reduces the number of variables tracked during the analysis and reduces precision.

# Classwork

- Find the values of variables in
  - context + flow-sensitive analysis
  - interprocedural context-insensitive but flow-sensitive analysis
  - intraprocedural flow-insensitive analysis

```
int g = 0;
void fun(int n) {
   g = n;
}
void main() {
   int a = 1;
   a = 2;
   fun(a);   // L1
   a = 3;
   fun(a);   // L2
}
```

# Concrete versus Abstract Interpretation

- Concrete: runtime, actual values
- Abstract: approximate, typically compile-time

Maintain one bit for x == 0
Initialized to F (false)

| Concrete Interpretation | Abstract Interpretation |
| --- | --- |
| x is undefined | F |
| x = 0; | x = 0; |
| x is 0 | T |
| ++x; | ++x; |
| x is 1 | F |
| --x; | --x; |
| x is 0 | ? |

# A Note on Choosing Abstraction

Maintain one bit for x == 0
Initialized to F (false)

```
F
x = 0;
T
++x;
F
--x;
?
```

Maintain two bits for value of x
Initialized to 00

```
00
x = 0;
00
++x;
01
--x;
00
```

Maintain one bit for x == 0
Another bit for x < 2
Initialized to 00

```
00
x = 0;
11
++x;
01
--x;
11
```

If type information available, then {01} --x {11} possible.
Otherwise, {01} --x {00}

# Abstraction Storage

- Saturating counters
- Number of values stored faithfully with log(n) bits – (n-2)
- Additional information may help increase the range, e.g., type information as unsigned.

# Classwork

Abstractly interpret the program where abstraction is maintained as two bits for: x == 1 and y == 1.

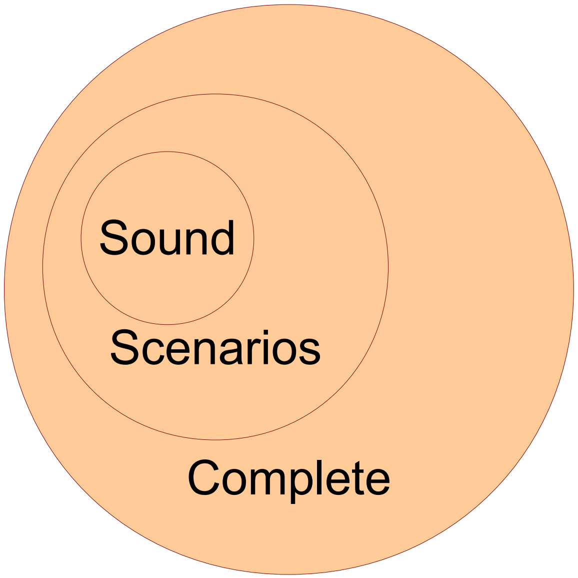
```
x = 1;
y = x;

if (x != y)
    y++;
else
    --x;

x++;
if (x == y)
    ++x;
else
   --y;
```

# Conservative Analysis

- Being safe versus being precise
  - Relation with lattice
  - Initializations and confluence
  - Constructive versus destructive operators
- Safety versus liveness property
  - Absence of bugs versus presence of a bug

# Soundness and Precision

- Analyses enable optimizations.
- An optimization is sound if it maintains the functionality of the original code.
- A program may be optimized in certain scenarios.
- An analysis is sound if it leads to sound optimization.
  - The analysis does not enable optimization outside the above set of scenarios.
- An analysis is complete if it does not disable optimization for any possible scenario.



# On Soundness

- Usually, multiple optimizations expect same information-theoretic behavior from analyses.
  - If more information means analysis A1 is less precise according to optimization O1, often optimization O2 also sees A1 that way.
  - This allows us to argue about analysis soundness without talking about optimizations.
- But this is not always true.
  - Soundness depends upon optimization enabling.
  - And two opposite optimizations may see the information from the same analysis in opposing ways.

# Optimization-specific Soundness

- Consider O1 that changes *p to x if p points to only x.
- Consider O2 that makes p volatile if p points to multiple variables at different program points.
- Analysis A computes points-to information $p \rightarrow \{x, y\}$
  - If A computes more information $p \rightarrow \{x, y, z\}$, O1 is suppressed but O2 is enabled.
  - If A computes less information $p \rightarrow \{x\}$, O1 is enabled and O2 is suppressed.
  - Thus, conservative for one is precise for another.
  - And sound for one is unsound for another.

# Optimization-specific Soundness

- Consider O1 that converts multiplication by 2 to a left-bit-shift operation (x * 2 to x << 1).
- Consider O2 that uses a special circuit (fast operation) when there is a sum of reciprocals of powers of 2 ($1 + \frac{1}{2} + \frac{1}{4} + \ldots$)
- Analysis A is used to compute values of arithmetic expressions.
  - Converting 1.98 to 2 enables O1, disables O2.
  - Converting 1.98 to 1.96875 enables O2, disables O1.
  - Precise for one is imprecise for another.
  - Sound for one is unsound for another.

# Acknowledgements

Course notes from

- Katheryn McKinley
- Monica Lam
- Y. N. Srikant
- Uday Khedker