Tentative syllabus

Mid-term

Exam
• The need for Intermediate Representations
  – Three-Address Code
• Lowering AST in While to IL
• Operational semantics
• Equivalence
• Correctness of lowering
Introduction to optimizations

Formalisms for program analysis
- Basic blocks
- Control flow graphs

Dataflow analyses and optimizations
- Live variables $\rightarrow$ dead code elimination
- In recitation:
  - Available expressions $\rightarrow$ common sub-expression elimination $+$ copy propagation
INTRODUCTION TO OPTIMIZATIONS
Optimization points

- User
  profile program
  change algorithm

- Compiler
  apply IR optimizations

- Compiler
  register allocation
  instruction selection
  peephole transformations

- source code
- Front end
- IR
- Code generator
- target code

today and next week
Overview of IR optimization

• Formalisms and Terminology
  – Control-flow graphs
  – Basic blocks

• Local optimizations *(won’t cover this year)*
  – Optimizing small pieces of a function

• Global optimizations
  – Optimizing functions as a whole

• The dataflow framework
  – Defining and implementing a wide class of optimizations
Semantics-preserving optimizations

• An optimization is **semantics-preserving** if it does not alter the semantics of the original program

• Examples:
  – Eliminating unnecessary statements
  – Computing values that are known statically at compile-time instead of runtime
  – Evaluating constant expressions outside of a loop instead of inside

• Non-examples:
  – Reordering side-effecting computations
  – Replacing bubble sort with quicksort (why?)

• The optimizations we will consider in this class are all semantics-preserving

• How can we find opportunities for optimizations?
Program analysis

• In order to optimize a program, the compiler has to be able to reason about the properties of that program
• An analysis is called sound if it never asserts an incorrect fact about a program
• All the analyses we will discuss in this class are sound
  – (Why?)
if (y < 5)
    x := 137;
else
    x := 42;

Print(x);

“At this point in the program, x holds some integer value”
if (y < 5)  
    x := 137;
else  
    x := 42;
Print(x);

“At this point in the program, x is either 137 or 42”
if (y < 5) 
    x := 137;
else 
    x := 42;
Print(x);

“At this point in the program, x is 137”
if (y < 5)
    x := 137;
else
    x := 42;

Print(x);
CONTROL FLOW GRAPHS
main:  t0 := Call ReadInteger()
a := t0
t1 := Call ReadInteger()
b := t1
L0:   t2 := 0
t3 := b == t2
t4 := 0
t5 := t3 = t4
IfZ t5 Goto L1
c := a
a := b
t6 := c % a
b := t6
Goto L0
L1:   Call PrintInt()
main: t0 := Call ReadInteger()
a := t0
t1 := Call ReadInteger()
b := t1
L0: t2 := 0
  t3 := b == t2
  t4 := 0
  t5 := t3 = t4
  IfZ t5 Goto L1
  c := a
  a := b
  t6 := c % a
  b := t6
  Goto L0
L1: Call PrintInt()
main:
  t0 := Call ReadInteger()
  a := t0
  t1 := Call ReadInteger()
  b := t1

L0:
  t2 := 0
  t3 := b == t2
  t4 := 0
  t5 := t3 = t4
  IfZ t5 Goto L1
  c := a
  a := b
  t6 := c % a
  b := t6
  Goto L0

L1:  Call PrintInt()
Basic blocks

• A basic block is a maximal sequence of IR instructions where
  – There is exactly one label where control enters the sequence, which must be at the start of the sequence
  – There is exactly one label where control leaves the sequence, which must be at the end of the sequence

• Informally: a sequence of instructions that always execute as a group
Control-flow graphs

• A control-flow graph (CFG) is a graph of the basic blocks in a function
  – From here on CFG stands for “control-flow graph” and not “context free grammar”

• Each edge from one basic block to another indicates that control can flow from the end of the first block to the start of the second block

• Dedicated nodes for the start and end of a function

• **Program executions correspond to paths the executions of the program**
Scope of optimizations

• An optimization is **local** if it works on just a single basic block

• An optimization is **global** if it works on an entire control-flow graph

• An optimization is **interprocedural** if it works across the control-flow graphs of multiple functions
  
  – We won't talk about this in this course
Optimizations and analyses

• Most optimizations are only possible given some analysis of the program's behavior
• In order to implement an optimization, we will talk about the corresponding program analyses

• **Program analysis** = algorithm that processes program and infers facts
  – **Sound facts** = facts that hold for all program executions
  – **Sound analysis** = program analysis that infers only sound facts
DEAD CODE ELIMINATION (DCE)
Definition

• An assignment to a variable $v$ is called **dead** if the value of that assignment is never read anywhere.

• **Dead code elimination** removes dead assignments from IR.

• Determining whether an assignment is dead depends on assignments succeeding it.
Dead code elimination example

```
\text{a := b}
\text{c := a}
\text{d := a + b}
\text{e := d}
\text{d := a}
\text{f := e}
\text{Print(d)}
```

Can we remove this statement?
Live variables

• The analysis corresponding to dead code elimination is called liveness analysis

• A variable is live at a point in a program if later in the program its value will be read before it is written to again
Optimizing via liveness analysis

• Dead code elimination works by computing liveness for each variable, then eliminating assignments to dead variables

• Dead code elimination
  – If $x := R \{v_1, \ldots, v_k\}$
  – And $x \notin \{v_1, \ldots, v_k\}$ and $R$ has no side-effects (not a function call)
  – We can eliminate $x := R$
    (if $R$ has side-effects, we can transform the assignment to just $R$)
LIVE VARIABLE ANALYSIS
Plan

• Define liveness analysis on single statements
• Define liveness analysis on basic blocks
• Define liveness analysis on acyclic CFGs
• Define liveness on arbitrary CFGs
The goal

• Given an IL command $C$ and set of live variables $\mathcal{LV}_{\text{out}}$ after $C$ executes, compute the set of live variables $\mathcal{LV}_{\text{in}}$ before $C$ executes

\[
\mathcal{LV}_{\text{in}} \quad \{ \ldots \} \\
\text{lab: } C \\
\mathcal{LV}_{\text{out}} \quad \{ \ldots \}
\]

• We will call this function the transformer of $C$ and write it as

\[
\mathcal{LV}_{\text{in}} = F[C](\mathcal{LV}_{\text{out}})
\]
DEF/USE

• For an IL statement $C$, we define
  – **DEF** – the variables possibly modified by $C$,
  – **USE** – the variables read by $C$

<table>
<thead>
<tr>
<th>Command Type</th>
<th>DEF</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>skip</strong></td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>$x := R$</td>
<td>${x}$</td>
<td>$\text{Vars}(R)$</td>
</tr>
<tr>
<td><strong>Goto</strong> $l'$</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td><strong>IfZ</strong> $x$ <strong>Goto</strong> $l'$</td>
<td>{}</td>
<td>${x}$</td>
</tr>
<tr>
<td><strong>IfNZ</strong> $x$ <strong>Goto</strong> $l'$</td>
<td>{}</td>
<td>${x}$</td>
</tr>
<tr>
<td><strong>Call</strong> $f(x_1,\ldots,x_n)$</td>
<td>{}</td>
<td>${x_1,\ldots,x_n}$</td>
</tr>
<tr>
<td><strong>Ret</strong> $x$</td>
<td>{}</td>
<td>${x}$</td>
</tr>
</tbody>
</table>
Liveness transformer

• Given an IL command $C$ and set of live variables $LV_{out}$ after $C$ executes, compute the set of live variables $LV_{in}$ before $C$ executes

\[
LV_{in} = (LV_{out} \setminus DEF(C)) \cup USE(C)
\]

lab: $C$

\[
\{ \ldots \}
\]

$x := x + 1$

• We define the transformer of $C$: $LV_{in}$
ANALYZING AND OPTIMIZING BASIC BLOCKS
Analyzing basic blocks

- Let $P = C_1, \ldots, C_n$ be a basic block and $LV_{n+1}$ be the initial set of live variables after $C_n$
- To compute $LV_1, \ldots, LV_n$ simply apply $F[C_i]$ for $i=n$ to $1$
Liveness example

{ b }

a := b;
{ a, b }
c := a;
{ a, b }
d := a + b;
{ a, b, d }
e := d;
{ a, b, e }
d := a;
{ b, d, e }
f := e;
{ b, d }

Foo(b, d)
{ }
Dead code elimination

```plaintext
{ b }
{ a, b }
c := a;
{ a, b }
d := a + b;
{ a, b, d }
e := d;
{ a, b, e }
d := a;
{ b, d, e }
f := e;
{ b, d }
Foo(b, d)
{ } 
```
GLOBAL LIVENESS ANALYSIS
Global analysis technical challenges

• Need to be able to handle multiple predecessors/successors for a basic block

• Need to be able to handle multiple paths through the control-flow graph
  – may need to iterate multiple times to compute final value
  – but the analysis still needs to terminate!

• Need to be able to assign each basic block a reasonable default value for before we've analyzed it
Global analysis technical challenges

• Need to be able to handle multiple predecessors/successors for a basic block
  – Join operator

• Need to be able to handle multiple paths through the control-flow graph
  – may need to iterate multiple times to compute final value
  – but the analysis still needs to terminate!
  – Chaotic iteration (fixed point iteration)

• Need to be able to assign each basic block a reasonable default value for before we've analyzed it
  – Bottom value
Global dead code elimination

• Local dead code elimination needed to know what variables were live on exit from a basic block
• This information can only be computed as part of a global analysis
• How do we extend our liveness analysis to handle a CFG?
GLOBAL LIVENESS ANALYSIS: CFGS WITHOUT LOOPS
CFGs without loops

Start

\[ b := c + d; \]
\[ e := c + d; \]

\[ x := c + d; \]
\[ a := b + c; \]

\[ y := a + b; \]

\[ x := a + b; \]
\[ y := c + d; \]

\[ \text{Foo}(x, y) \]

End
Which variables may be live on some execution path?
Which assignments are redundant?

Start

{x, y} 
End

{x, y} Foo(x,y)

{a, b, c, d} 
{b, c, d} 
x := c + d; 
a := b + c; 
{a, b, c, d} 

{a, c, d} 
b := c + d; 
e := c + d; 
{a, b, c, d} 

{a, b, c, d} 
y := a + b; 
{a, b, c, d} 

{a, b, c, d} 
x := a + b; 
y := c + d; 
{x, y}
CFGs without loops

Start

{b, c, d}
{x := c + d;
a := b + c;
{a, b, c, d}

{a, c, d}
b := c + d;
e := c + d;
{a, b, c, d}

{a, b, c, d}

{a, b, c, d}

{b, c, d}
x := c + d;
a := b + c;
{a, b, c, d}

{a, b, c, d}

{a, b, c, d}

{a, b, c, d}

{x, y}

{x, y}

Foo(x,y)

End
CFGs without loops

\[
\begin{align*}
\text{Start} & \quad \rightarrow \quad b := c + d; \\
& \quad \rightarrow \quad a := b + c; \\
& \quad \rightarrow \quad x := a + b; \\
& \quad \rightarrow \quad y := c + d; \\
& \quad \rightarrow \quad \{x, y\} \\
& \quad \rightarrow \quad \text{Foo}(x,y) \\
& \quad \rightarrow \quad \text{End}
\end{align*}
\]

Dead code elimination can lead to more optimizations
CFGs without loops

\[ \begin{align*}
\text{Start} & \quad \rightarrow \quad b := c + d; \\
a := b + c; & \quad \rightarrow \quad x := a + b; \\
y := c + d; & \quad \rightarrow \quad \{x, y\} \\
\text{Foo}(x, y) & \quad \rightarrow \quad \text{End}
\end{align*} \]
Major changes – part 1

• In a local analysis (analysis of one basic block), each statement has exactly one predecessor
• In a global analysis, each statement may have multiple predecessors
• A global analysis must have some means of combining information from all predecessors of a basic block
Combining values

Start

\{b, c, d\}
\(x := c + d;\)
\(a := b + c;\)
\{a, b, c, d\}

Need to combine currently-computed value with new value

\{c, d\}
\(b := c + d;\)
\(e := c + d;\)
\{b, c, d\}

\{a, b, c, d\}
\(y := a + b;\)
\{a, b, c, d\}

\{a, b, c, d\}
\(x := a + b;\)
\(y := c + d;\)
\{x, y\}

\{x, y\}
\(\text{Foo}(x, y)\)

End
Combining values

{c, d}
b := c + d;
e := c + d;
{a, b, c, d}

{b, c, d}
x := c + d;
a := b + c;
{a, b, c, d}

{a, b, c, d}
y := a + b;
{a, b, c, d}

{a, b, c, d}
x := a + b;
y := c + d;
{x, y}

{x, y}
Foo(x,y)

End
Combining values

Start

\{b, c, d\}
x := c + d;
a := b + c;
\{a, b, c, d\}

\{a, c, d\}
b := c + d;
e := c + d;
\{a, b, c, d\}

\{a, b, c, d\}
x := a + b;
y := c + d;
\{x, y\}

\{x, y\}
Foo(x, y)

End
GLOBAL LIVENESS ANALYSIS: HANDLING LOOPS
Major changes – part 2

• In a local analysis, there is only one possible path through a basic block
• In a global analysis, there may be many paths through a CFG
• May need to recompute values multiple times as more information becomes available
• Need to be careful when doing this not to loop infinitely!
  – (More on that later)
CFGs with loops

Start

a := b + c;
d := a + c;
b := c + d;
c := c + d;
IfZ ...

Ret a

End

c := a + b;
a := a + b;
d := b + c;
a := b + c;
d := a + c;
b := c + d;
c := a + b;
Start
How do we use join here?

\[
\begin{align*}
    a & := b + c; \\
    d & := a + c;
\end{align*}
\]

\[
\begin{align*}
    b & := c + d; \\
    c & := c + d; \\
    \text{IfZ} & \ldots
\end{align*}
\]

\[
\begin{align*}
    c & := a + b;
\end{align*}
\]

\[
\begin{align*}
    a & := a + b; \\
    d & := b + c;
\end{align*}
\]

\[
\{a\} \\
\text{Ret } a
\]

End
Major changes – part 3

- In a local analysis, there is always a well-defined “first” statement to begin processing.
- In a global analysis with loops, every basic block might depend on every other basic block.
- To fix this, we need to assign initial values to all of the blocks in the CFG.
CFGs with loops - initialization

Start

\[
\begin{align*}
{} & b := c + d; \\
{} & c := c + d; \\
\end{align*}
\]

{} 
\[
\begin{align*}
a & := b + c; \\
d & := a + c; \\
\end{align*}
\]

{} 
\[
\begin{align*}
c & := a + b; \\
\end{align*}
\]

{} 
\[
\begin{align*}
a & := a + b; \\
d & := b + c; \\
\end{align*}
\]

{a} 
Ret a

End
CFGs with loops - iteration

Start

{} b := c + d;
{} c := c + d;

{} a := b + c;
d := a + c;

{} c := a + b;

{} a := a + b;
d := b + c;
{a}

{a}
Ret a

End
CFGs with loops - iteration

Start

\{\}
\begin{align*}
a & := b + c; \\
d & := a + c;
\end{align*}

\{\}
\begin{align*}
b & := c + d; \\
c & := c + d;
\end{align*}

\{\}
\begin{align*}
c & := a + b;
\end{align*}

\{a, b, c\}
\begin{align*}
a & := a + b; \\
d & := b + c; \\
\{a\}
\end{align*}

\{a\}
Ret a

End
CFGs with loops - iteration

Start

{} b := c + d; c := c + d;

{} a := b + c;
d := a + c;
{a, b, c}

{} c := a + b;

{a, b, c} a := a + b;
d := b + c;
{a}

{a} Ret a

End
CFGs with loops - iteration

Start

{b, c}
\[ a \leftarrow b + c; \]
\[ d \leftarrow a + c; \]
\{a, b, c\}

\{b, c\}
\[ b \leftarrow c + d; \]
\[ c \leftarrow c + d; \]

\{a, b, c\}
\[ a \leftarrow a + b; \]
\[ d \leftarrow b + c; \]
\{a\}

\{a\}
Ret a

End

\{\}
\[ c \leftarrow a + b; \]
CFGs with loops - iteration

\begin{verbatim}
Start

{b, c}
a := b + c;
d := a + c;
{a, b, c}

{} b := c + d;
c := c + d;
{b, c}

{} c := a + b;

{} a := b + c;
d := a + c;
{a, b, c}

{} a := a + b;
d := b + c;
{a}

{} Ret a
End
\end{verbatim}
CFGs with loops - iteration

Start

{b, c}
a := b + c;
d := a + c;
{a, b, c}

{c, d}
b := c + d;
c := c + d;
{b, c}

{}c := a + b;
{a, b, c}

{a, b, c}
a := a + b;
d := b + c;
{a}

{a}Ret a

End
CFGs with loops - iteration

\textbf{Start}

- $\{a, b\}$
- $\{b, c\}$

- $\{c, d\}$
  - $b := c + d$
  - $c := c + d$
  - $\{b, c\}$

- $\{b, c\}$
  - $a := b + c$
  - $d := a + c$
  - $\{a, b, c\}$

- $\{a, b\}$
  - $c := a + b$
  - $\{a, b, c\}$

- $\{a, b, c\}$
  - $a := a + b$
  - $d := b + c$
  - $\{a\}$

- $\{a\}$
  - Ret a

\textbf{End}
CFGs with loops - iteration

Start

{b, c}
a := b + c;
d := a + c;
{a, b, c}

{c, d}
b := c + d;
c := c + d;
{b, c}

{a, b}
c := a + b;
{a, b, c}

{a, b, c}
a := a + b;
d := b + c;
{a}

{a}
Ret a

End
CFGs with loops - iteration

```
a := a + b;
d := b + c;
c := a + b;
d := a + c;
b := c + d;
c := c + d;
```

Start

```
{a, b}
{b, c}
```

```
b := c + d;
c := c + d;
```

```
{c, d}
{b, c}
```

```
{a, b}
c := a + b;
{a, b, c}
```

```
a := b + c;
d := a + c;
{a, b, c}
```

```
{a, b, c}
{a, b, c}
```

```
{a, b, c}
a := a + b;
d := b + c;
{a, c, d}
```

```
{a, b, c}
{a, c, d}
```

```
{a}
Ret a
```

End
CFGs with loops - iteration

Start

{b, c}
a := b + c;
d := a + c;
{a, b, c}

{c, d}
b := c + d;
c := c + d;
{b, c}

{a, b}
c := a + b;
{a, b, c}

{a, b, c}
a := a + b;
d := b + c;
{a, c, d}

{a}
Ret a

End
CFGs with loops - iteration

Start

{b, c}
a := b + c;
d := a + c;
{a, b, c}

{c, d}
b := c + d;
c := c + d;
{b, c}

{a, b}
c := a + b;
{a, b, c}

{a, b, c}
a := a + b;
d := b + c;
{a, c, d}

{a}
Ret a

End
CFGs with loops - iteration

\[
\begin{align*}
{\{c, d\}} \\
b &:= c + d; \\
c &:= c + d; \\
{\{b, c\}}
\end{align*}
\]

\[
\begin{align*}
{\{b, c\}} \\
a &:= b + c; \\
d &:= a + c; \\
{\{a, b, c\}}
\end{align*}
\]

\[
\begin{align*}
{\{a, b\}} \\
c &:= a + b; \\
{\{a, b, c\}}
\end{align*}
\]

\[
\begin{align*}
{\{a, b, c\}} \\
a &:= a + b; \\
d &:= b + c; \\
{\{a, c, d\}}
\end{align*}
\]

\[
\begin{align*}
{\{a\}} \\
\text{Ret } a
\end{align*}
\]

End
CFGs with loops - iteration

```
{c, d}
b := c + d;
c := c + d;
{a, b, c}

{b, c}
a := b + c;
d := a + c;
{a, b, c}

{a, b}
c := a + b;
{a, b, c}

{a, b, c}
a := a + b;
d := b + c;
{a, c, d}

{a}
Ret a
```

Start

End
CFGs with loops - iteration

Start

{a, c, d}
b := c + d;
c := c + d;
{a, b, c}

{b, c}
a := b + c;
d := a + c;
{a, b, c}

{a, b}
c := a + b;
{a, b, c}

{a, b, c}
a := a + b;
d := b + c;
{a, c, d}

{a, c, d}

{a, b, c}

{a}
Ret a

End
CFGs with loops – fixed point

Start

\{a, c, d\}
\[b := c + d;\]
\[c := c + d;\]
\{a, b, c\}

\{b, c\}
\[a := b + c;\]
\[d := a + c;\]
\{a, b, c\}

\{a, b\}
\[c := a + b;\]
\{a, b, c\}

\{a, b, c\}
\[a := a + b;\]
\[d := b + c;\]
\{a, c, d\}

\{a\}
Ret a

End
FORMALIZING
GLOBAL LIVENESS ANALYSIS
Global liveness analysis

• Initially, set $IN[C] = \{ \}$ for each command $C$
• Set $IN[end]$ to the set of variables known to be live on exit ($\{\}$ unless special assumptions)
  – Language-specific knowledge
• Repeat until no changes occur:
  – For each command $C$ in any order you'd like:
    • Set $OUT[C]$ to be the union of $IN[C_{\text{next}}]$ for each successor $C_{\text{next}}$ of $C$
    • Set $IN[C]$ to $(OUT[C] \setminus DEF(C)) \cup USE(C)$
• Yet another fixed-point iteration!
Global liveness analysis

\[ a := b + c \]

\[ \text{IN}[s] = (\text{OUT}[s] - \{a\}) \cup \{b, c\} \]

\[ \text{OUT}[s] = \text{IN}[s2] \cup \text{IN}[s3] \]

\[ \text{IN}[s2] \]

\[ \text{IN}[s3] \]

\[ s2 \]

\[ s3 \]
Why does this work?

• To show correctness, we need to show that
  – The algorithm eventually terminates, and
  – When it terminates, it has a sound answer
Termination argument

- ...?
Termination argument

• Once a variable is discovered to be live during some point of the analysis, it always stays live
• Only finitely many variables and finitely many places where a variable can become live
Soundness argument (sketch)

• Each individual rule, applied to some set, correctly updates liveness in that set.
• When computing the union of the set of live variables, a variable is only live if it was live on some path leaving the statement.
• So, locally, every step in the algorithm is correct.
• Does local correctness imply global correctness?
  – Yes under some conditions
  – Monotone dataflow
OPTIMIZATION CORRECTNESS
Revisiting semantic equivalence

• Recall our definition of semantic equivalence: for every pair of IL programs $P$ and $P'$ – they are equivalent if for every input state $m$,
  \[
  \llbracket P \rrbracket m = \llbracket P' \rrbracket m
  \]

• Does it work for $P$ and $P' = DCE(P)$?

<table>
<thead>
<tr>
<th>$P$</th>
<th>$P'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a := b$</td>
<td>$a := b$</td>
</tr>
<tr>
<td>$c := a$</td>
<td>$d := a$</td>
</tr>
<tr>
<td>$d := a + b$</td>
<td>$d := a$</td>
</tr>
<tr>
<td>$e := d$</td>
<td>$f := e$</td>
</tr>
<tr>
<td>$d := a$</td>
<td>Print(d)</td>
</tr>
<tr>
<td>Print(d)</td>
<td></td>
</tr>
</tbody>
</table>
Refining semantic equivalence

• **Definition:** For each IL program $P$, let $V=\text{ObservableVars}(P)$ be the variables appear in Ret and Call commands.

• **Definition:** For every pair of IL programs $P$ and $P'$ we say that they are **observably equivalent** if for every input state $m$, 

$$(\llbracket P \rrbracket m)|_{\text{ObservableVars}(P)} = (\llbracket P' \rrbracket m)|_{\text{ObservableVars}(P')}$$

• TODO: fix by defining trace semantics
Semantic equivalence for DCE

• **Lemma**: Let $DCE(P)=P'$ be a dead code elimination optimization applied to program $P$ then the following holds:
  $\text{ObservableVars}(P) = \text{ObservableVars}(P')$

• **Theorem**: $P$ and $DCE(P)$ are observably equivalent
Example

\[ P \]
\[
\begin{align*}
  a & := b \\
  c & := a \\
  d & := a + b \\
  e & := d \\
  d & := a \\
  f & := e \\
  \text{Print}(d)
\end{align*}
\]

\[ P' \]
\[
\begin{align*}
  a & := b \\
  d & := a \\
  \text{Print}(d)
\end{align*}
\]

\[ \text{ObservableVars}(P) = \{d\} \]

- For \( m = [\text{pc} \mapsto 1, \ a \mapsto 0, \ b \mapsto 1, \ c \mapsto 2, \ d \mapsto 3, \ e \mapsto 4, \ f \mapsto 5] \)
- At the function call
  for \( P \) we have \([\text{pc} \mapsto 6, \ a \mapsto 1, \ b \mapsto 1, \ c \mapsto 1, \ d \mapsto 1, \ e \mapsto 2, \ f \mapsto 2] | \{d\} = [d \mapsto 1] \)
  for \( P' \) we have \([\text{pc} \mapsto 3, \ a \mapsto 1, \ b \mapsto 1, \ c \mapsto 2, \ d \mapsto 1, \ e \mapsto 4, \ f \mapsto 5] | \{d\} = [d \mapsto 1] \)
- At the output:
  \[
  [P] \ m = [\text{pc} \mapsto 7, \ a \mapsto 1, \ b \mapsto 1, \ c \mapsto 1, \ d \mapsto 1, \ e \mapsto 2, \ f \mapsto 2] \\
  [P'] \ m = [\text{pc} \mapsto 4, \ a \mapsto 1, \ b \mapsto 1, \ c \mapsto 2, \ d \mapsto 1, \ e \mapsto 4, \ f \mapsto 5] \\
  ([P] \ m) | \{d\} = ([P'] \ m) | \{d\} = [d \mapsto 1] \]
Next lecture: Dataflow Analysis Framework