Tentative syllabus

Front End
- Scanning
- Top-down Parsing (LL)
- Bottom-up Parsing (LR)

Intermediate Representation
- Operational Semantics
- Lowering

Optimizations
- Dataflow Analysis
- Loop Optimizations

Code Generation
- Register Allocation

mid-term
exam
Previously

• Extending While/IL syntax with procedures
• 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
Theory to the rescue

• Building up all of the machinery to design the liveness analysis was tricky
• The key ideas, however, are mostly independent of the analysis:
  – We need to be able to compute functions describing the behavior of each statement
  – We need to be able to merge several subcomputations together
  – We need an initial value for all of the basic blocks
• There is a beautiful formalism that captures many of these properties: dataflow framework
AGENDA

• Dataflow analysis framework
  – Join semilattices
  – Chaotic iteration
  – Monotone transformers

• More optimizations
  – Reminder: Available expressions → common sub-expression elimination + copy propagation
  – Constant propagation → constant folding
  – Reaching definition → loop invariant code motion
LATTICE THEORY
Partial order

- A **partial order** is a pair \((V, \sqsubseteq)\)
- \(V\) is a set of abstract elements called **domain**
- \(\sqsubseteq : V \times V\) is a binary relation that is
  - Reflexive: \(x \sqsubseteq x\)
  - Antisymmetric: if \(x \sqsubseteq y\) and \(y \sqsubseteq x\), then \(x = y\)
  - Transitive: if \(x \sqsubseteq y\) and \(y \sqsubseteq z\), then \(x \sqsubseteq z\)
Liveness partial order

• An IL program $P$ induces the partial order $(2^{\text{Vars}(P)}, \subseteq)$
  – $\subseteq$ is reflexive: $X \subseteq X$
  – $\subseteq$ is antisymmetric: $X \subseteq Y$ and $Y \subseteq X$ implies $X=Y$
  – $\subseteq$ is transitive: $X \subseteq Y$ and $Y \subseteq Z$ implies $X \subseteq Z$
Partial order for liveness (Hasse diagram)

\[ Vars(P) = \{a, b, c\} \]

Greater

Bottom element

Vars(P) = \{a, b, c\}
Partial order for liveness (Hasse diagram)

Vars(P) = \{a, b, c\}

Least precise

{a, b, c}

{a, b}

{a, c}

{b, c}

{a}

{b}

{c}

{}
Join operation

• Let \((V, \sqsubseteq)\) be a partial order

• An operation \(\sqcup : V \times V \rightarrow V\) is a join if it is
  – **commutative**: \(x \sqcup y = y \sqcup x\)
  – **associative**: \((x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)\)
  – **idempotent**: \(x \sqcup x = x\)

• If \(x \sqcup y = z\), we say that \(z\) is the **join** or of \(x\) and \(y\)

• Intuitively the join of two elements represents combining information from two elements by an overapproximation
  – Actually, the **Least Upper Bound**
Join semilattices

• A **join semilattice** is a partial order equipped with a join operation and a bottom element $(V, \sqsubseteq, \sqcup, \bot)$

• There is a unique bottom element $\bot$, which is smaller than all other elements
  – The bottom element represents “no information *yet*” or “most precise value”

• There is also usually a top element $\top$, which is greater than all other elements
  – “Most conservative value”
Join semilattices in program analysis

- The elements of a join semilattice represent (infinite) sets of states
  - We compute one element per program location
- Join approximates the information from two different paths
- The bottom element usually stands for an empty set of states
- The top element stands for the set of all possible states
What is the join of \{b\} and \{c\}?
What is the join of \{b\} and \{c\}?
What is the join of \{b\} and \{a, c\}?
What is the join of \{b\} and \{a, c\}?
What is the join of \{a\} and \{a, b\}?
What is the join of \{a\} and \{a,b\}?
A join semilattice for liveness

• Sets of live variables and the set union operation
• Idempotent:
  – \( x \cup x = x \)
• Commutative:
  – \( x \cup y = y \cup x \)
• Associative:
  – \( (x \cup y) \cup z = x \cup (y \cup z) \)
• Bottom element:
  – The empty set: \( \emptyset \cup x = x \)
• What is the ordering over these elements?
Join semilattices and orderings

• Every join semilattice \((V, \sqcup)\) induces an ordering relationship \(\sqsubseteq\) over its elements

• Define \(x \sqsubseteq y\) iff \(x \sqcup y = y\)

• Exercise: prove
  – Reflexivity: \(x \sqsubseteq x\)
  – Antisymmetry: If \(x \sqsubseteq y\) and \(y \sqsubseteq x\), then \(x = y\)
  – Transitivity: If \(x \sqsubseteq y\) and \(y \sqsubseteq z\), then \(x \sqsubseteq z\)
DATAFLOW FRAMEWORK
Semilattices and program analysis

- Semilattices naturally solve many of the problems we encounter in global analysis
- How do we combine information from multiple basic blocks?  
  - …?
- What value do we give to basic blocks we haven't seen yet?  
  - …?
- How do we know that the algorithm always terminates?  
  - …?
Semilattices and program analysis

• Semilattices naturally solve many of the problems we encounter in global analysis

• How do we combine information from multiple basic blocks?
  – Take the join of all information from those blocks

• What value do we give to basic blocks we haven't seen yet?
  – Use the bottom element

• How do we know that the algorithm always terminates?
  – Actually, we still don't! More on that later
A general framework

• A global analysis is a tuple \((D, V, \sqcap, F, I)\)
  – \(D\) is a direction (forward or backward)
    – The order to visit statements within a basic block, not the order in which to visit the basic blocks
  – \(V\) is a set of values (domain)
  – \(\sqcap\) is a join operator over those values
  – \(F = \{F[c] : V \rightarrow V \mid c \in P\}\)
    is a set of transfer functions, one per command \(c\)
  – \(I\) is an initial value
    – If \(D=\)forward then we set \(I\) in the start node, otherwise we set \(I\) in the end node
Liveness as a dataflow problem

• For a program $P$, define a dataflow problem as
  $\text{DataflowProblem} = (D, V, \sqcup, F, I)$

• Liveness =
  $\text{(backward, }$
  \mathcal{2}^\text{Vars}(P)$,
  \cup,$
  \{\lambda \; v. \; (v \setminus \text{DEF}(c)) \cup \text{USE}(c) \mid c \in P\},$
  \{\} )$
Running global analyses

- Let $P$ be an IL program
- Assume that $(D, V, \sqcup, F, I)$ is a forward analysis
- Set $\text{OUT}[c] = \bot$ for all commands $c \in P$
- Set $\text{OUT}[\text{start}] = I$
- Repeat until no values change:
  - For each statement $c$ with predecessors $p_1, p_2, \ldots, p_n$:
    - Set $\text{IN}[c] = \text{OUT}[p_1] \sqcup \text{OUT}[p_2] \sqcup \ldots \sqcup \text{OUT}[p_n]$
    - Set $\text{OUT}[c] = F[c] \left( \text{IN}[c] \right)$
- The order of this iteration does not matter:
  Chaotic iteration
Generic algorithm for D=forward

Set OUT[c] = ⊥ for all statements c
Set OUT[start] = I
Repeat until no values change:
For each c ∈ P with predecessors p₁,...,pₙ:
• Set IN[c] = OUT[p₁] △ ... △ OUT[pₙ]
• Set OUT[c] = F[c] (IN[c])

Set IN[c] = {} for all statements c
Set IN[end] = {}
Repeat until no values change:
For each c ∈ P with successors s₁,...,sₙ:
• Set OUT[c] = IN[s₁] △ ... △ IN[sₙ]
• Set IN[c] = (OUT[c]\DEF(c)) △ USE(c)
What is this framework good for?

• This form of analysis is called the dataflow framework
• Can be used to easily prove an analysis is sound
• With certain restrictions, can be used to prove that an analysis eventually terminates
  – Again, more on that later
TERMINATION CONDITIONS
Height of a lattice

• An increasing chain is a sequence of elements
  \( \bot \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \ldots \sqsubseteq a_k \)
  – The length of such a chain is \( k \)
• The height of a lattice is the length of the maximal increasing chain
• For liveness with \( n \) program variables:
  – \( \{\} \subset \{v_1\} \subset \{v_1,v_2\} \subset \ldots \subset \{v_1,\ldots,v_n\} \)
Monotone transfer functions

• A transfer function $f$ is monotone iff
  if $x \subseteq y$, then $f(x) \subseteq f(y)$

• Intuitively, if you know less information about a program point, you can't "gain back" more information about that program point

• Many transfer functions are monotone, including those for liveness and constant propagation

• Note: Monotonicity does not mean that $x \subseteq f(x)$
  – (This is a different property called extensivity)
Liveness and monotonicity

• A transfer function $f$ is **monotone** iff
  
  if $x \subseteq y$, then $f(x) \subseteq f(y)$

• Recall our transfer function for $a:=b+c$ is
  
  $F[a:=b+c](X) = (X \setminus \{a\}) \cup \{b, c\}$

• Recall that our join operator is set union and induces an ordering relationship
  
  $X \subseteq Y$ iff $X \subseteq Y$

• Is this monotone?
Termination theorem

• **Theorem 1:** A dataflow analysis with a *finite-height join semilattice* and family of *monotone transfer functions* always terminates

• Proof sketch:
  – The join operator can only bring values up
  – Transfer functions can never lower values back down below where they were in the past (monotonicity)
  – Values cannot increase indefinitely (finite height)
Soundness and optimality theorem

• **Theorem 2:** A dataflow analysis with a **finite-height join semilattice** and family of **monotone transfer functions** **always yields a unique solution** – the least fixed point in the lattice
DISTRIBUTIVITY
An “optimality” result

• A transfer function $f$ is distributive if
  \[ f(a \sqcup b) = f(a) \sqcup f(b) \]
  for every domain elements $a$ and $b$

• If all transfer functions are distributive then the fixed-point solution is equal to the solution computed by joining results from all (potentially infinite) control-flow paths
  – Join over all paths
An “optimality” result

• A transfer function $f$ is distributive if $f(a \sqcup b) = f(a) \sqcup f(b)$ for every domain elements $a$ and $b$

• If all transfer functions are distributive then the fixed-point solution is equal to the solution computed by joining results from all (potentially infinite) control-flow paths
  – Join over all paths

• Optimal if we ignore program conditions
  – Pretend all control-flow paths can be executed by the program

• Which analyses use distributive functions?
COMMON SUBEXPRESSION ELIMINATION
CSE Example

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

\[ b := a \times a; \]
\[ c := a \times a; \]
\[ d := b + c; \]
\[ e := b + b; \]
CSE Example

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

Common sub-expression elimination
Common Subexpression Elimination

• If we have two variable assignments
  \( v_1 := a \ op \ b \)
  ...
  \( v_2 := a \ op \ b \)
• and the values of \( v_1 \), \( a \), and \( b \) have not changed between the assignments, rewrite the code as
  \( v_1 := a \ op \ b \)
  ...
  \( v_2 := v_1 \)
• Eliminates useless recalculation
• Paves the way for later optimizations
COPY PROPAGATION
CP Example

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

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

Copy propagation
Copy Propagation

• If we have a variable assignment
  \(x := y\)
  then as long as \(x\) and \(y\) are not reassigned, we can replacing \(x\) with \(y\) in any expression
  Example 1: \(a := t+x\) rewrites to \(a := t+y\)
  Example 2: \(\text{foo}(x)\) rewrites into \(\text{foo}(y)\)

• Notation: \(a := R\) rewrites to \(a := R[y/x]\)
  \(\text{foo}(a_1,\ldots,x, \ldots, a_n)\) rewrites to
  \(\text{foo}(a_1,\ldots,y, \ldots, a_n)\)
AVAILABLE EXPRESSIONS
Available expressions

• Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program

• An expression $a \leftarrow b \text{ op } c$ is called available at program location $L$ if variable $a$ holds the value of $b \text{ op } c$ at that location
  – Similarly for $a \leftarrow b$

• In common subexpression elimination, we replace an available expression ($b \text{ op } c$) by the variable holding its value ($a$)

• In copy propagation, we replace the use of a variable ($a$) by the available expression it holds ($b$)
Finding available expressions

• Compute for each program location $L$ a set of expressions $AE$ of the forms $a \leftarrow b \ op \ c$ and $a \leftarrow b$ that are definitely available there.

• Whenever we execute a statement $a := b \ op \ c$:
  - Any expression holding $a$ is invalidated.
  - The expression $a \leftarrow b \ op \ c$ becomes available.
Available expressions step

\[ a := b + c \]

\[ \text{AE}_{\text{out}} = (\text{AE}_{\text{in}} \setminus \{e \mid e \text{ contains } a\}) \cup \{a \leftarrow b + c\} \]

Expressions of the forms
\[ a \leftarrow \ldots \text{ and } x \leftarrow \ldots a \ldots \]

Provided that \(a\) and \(b\) and \(a\) and \(c\) are different pairs of variables.
Available expressions example

{  }

a := b;
{ a←b  }

c := b;
{ a←b, c←b  }

d := a + b;
{ a←b, c←b, d←a+b  }

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

d := b;
{ a←b, c←b, d←b, e←a+b  }

f := a + b;
{ a←b, c←b, d←b, e←a+b, f←a+b  }
Available expressions problem

• For a program $P$, define a dataflow problem as
  \[ \text{DataflowProblem} = (D, V, \sqsubseteq, F, I) \]

• \( \text{AvailExprs}(P) = \{ x \leftarrow R \in P \mid x := R \in P, \right. \)
  \[ R \text{ is not Call foo(...),} \]
  \[ x \notin \text{Vars}(R) \} \]

\( \text{AvailableExpressions} = \)
\[ (\text{forward}, 2^{\text{AvailExprs}(P)}, \cap, \{ F_{AE}[c] \mid c \in P \}, \{ \}) \]
Available expressions transformer

- \( \text{AE}_{\text{out}} = F_{\text{AE}}[C](\text{AE}_{\text{in}}) \)

<table>
<thead>
<tr>
<th>Command Type</th>
<th>( \text{AE}_{\text{out}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>( \text{AE}_{\text{in}} )</td>
</tr>
<tr>
<td>( x := R )</td>
<td>( \left( \text{AE}_{\text{in}} \setminus { \text{ae} \mid x \text{ appears in } \text{ae} } \right) \cup { x\leftarrow R \mid x \notin R } )</td>
</tr>
<tr>
<td>Goto ( l' )</td>
<td>( \text{AE}_{\text{in}} )</td>
</tr>
<tr>
<td>IfZ ( x ) Goto ( l' )</td>
<td>( \langle \text{AE}<em>{\text{in}}, \text{AE}</em>{\text{in}} \rangle )</td>
</tr>
<tr>
<td>IfNZ ( x ) Goto ( l' )</td>
<td>( \langle \text{AE}<em>{\text{in}}, \text{AE}</em>{\text{in}} \rangle )</td>
</tr>
<tr>
<td>Call ( f(x_1, \ldots, x_n) )</td>
<td>( \text{AE}_{\text{in}} )</td>
</tr>
<tr>
<td>Ret ( x )</td>
<td>( \text{AE}_{\text{in}} )</td>
</tr>
</tbody>
</table>

Avoid adding expressions like \( x\leftarrow x \), \( x\leftarrow x+1 \).
Optimizing via available expressions

- **Common sub-expression elimination**
  - If \( \{ \ldots \ t \leftarrow y \text{ op } z \ldots \} \ x := y \text{ op } z \)
  - Transform statement into \( x := t \)

- **Copy propagation**
  - If \( \{ \ldots \ y \leftarrow t \ldots \} \ x:=R \)
  - Transform statement into \( x := R[t/y] \)
  - (and same for function calls)
  - E.g., can transform the statement \( x := y \text{ op } z \)
    into \( x := t \text{ op } z \)

- **Note:** same for \( x := y \)
CONSTANT PROPAGATION
Constant propagation

• **Constant folding** is an optimization that replaces each variable that is known to be a constant value with that constant
  – Replace \( x := 5 \); \( y := x + 7 \) with \( x := 5 \); \( y := 12 \)

• Statically evaluate expressions and replace with the resulting constant whenever possible
  – Right-hand side of assignments
  – Conditions
Constant propagation analysis

• In order to do a constant propagation, we need to track the values assigned to a variable at each program point
• Every variable will either
  – Never have a value assigned to it,
  – Have a single constant value assigned to it,
  – Have two or more constant values assigned to it, or
  – Have a known non-constant value
  – Our analysis will propagate this information throughout a CFG to identify locations where a value is constant
Properties of constant propagation

• For now, consider just some single variable \( x \)
• At each point in the program, we know one of three things about the value of \( x \):
  – We have never seen a value for \( x \)
  – \( x \) is definitely a constant and has value \( k \)
  – \( x \) is not a constant, since it's been assigned two values or assigned a value that we know isn't a constant
• Note that the first and last of these are **not** the same!
  – The last one means that there may be a way for \( x \) to have multiple values
  – The first one means that \( x \) never had a value at all
Defining a join operator

• The join of any two different constants is **Not-a-Constant**
  – (If the variable might have two different values on entry to a statement, it cannot be a constant)

• The join of **Not a Constant** and any other value is **Not-a-Constant**
  – (If on some path the value is known not to be a constant, then on entry to a statement its value can't possibly be a constant)

• The join of **Undefined** and any other value is that other value
  – (If x has no value on some path and does have a value on some other path, we can just pretend it always had the assigned value)
  – **Note:** in the semantics we defined for IL variables always have values
A semilattice for constant propagation

- One possible semilattice for this analysis is shown here (for each variable):

The lattice is infinitely wide
A semilattice for constant propagation

• One possible semilattice for this analysis is shown here (for each variable):

```
Not-a-constant

-2  -1  0  1  2

Undefined
```

• Note:
  • The join of any two different constants is Not-a-Constant
  • The join of Not a Constant and any other value is Not-a-Constant
  • The join of Undefined and any other value is that other value
A semilattice for constant propagation

• One possible semilattice for this analysis is shown here (for each variable):


• Note:
  • The join of any two different constants is \( \top \)
  • The join of \( \top \) and any other value is \( \top \)
  • The join of \( \bot \) and any other value is that other value
Constant propagation problem

• For a program $P$, define a dataflow problem as
  $\text{DataflowProblem} = (D, V, \sqcap, F, I)$

• $\text{ConstExprs}(P) = (\text{Vars} \cup \text{Temp}) \rightarrow (\mathbb{Z} \cup \{\bot, \top\})$

ConstantPropagation =
(forward, ConstExprs($P$), $\sqcap_{CP}$, $\{F_{CP}[c] \mid c \in P\}$, $\lambda \ x. \top$)
What is the join operator?

• For a program $P$, define a dataflow problem as
  \[
  \text{DataflowProblem} = (D, V, \sqcup, F, I)
  \]

• $\text{ConstExprs}(P) = (\text{Vars} \cup \text{Temp}) \rightarrow (Z \cup \{\bot, \top\})$

\[
\text{ConstantPropagation} = \\
(\text{forward}, \text{ConstExprs}(P), \sqcup_{CP}, \{F_{CP}[c] \mid c \in P\}, \lambda x.\top)
\]

\[
[x \mapsto 1, y \mapsto 2, z \mapsto \bot, w \mapsto 5] \sqcup_{CP} [x \mapsto 1, y \mapsto 3, z \mapsto 4, w \mapsto \top] = ?
\]
Constant propagation transformer

- \( \text{CP}_{\text{out}} = F_{\text{CP}}[c](\text{CP}_{\text{in}}) \)

<table>
<thead>
<tr>
<th>Command Type</th>
<th>( \text{AE}_{\text{out}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>( \text{CP}_{\text{in}} )</td>
</tr>
<tr>
<td>( x := R )</td>
<td>if ( R=n ) then ( \text{CP}<em>{\text{in}}[x\rightarrow n] ) else ( \text{CP}</em>{\text{in}}[x\rightarrow \top] )</td>
</tr>
<tr>
<td>Goto ( l' )</td>
<td>( \text{CP}_{\text{in}} )</td>
</tr>
<tr>
<td>IfZ ( x ) Goto ( l' )</td>
<td>( \langle \text{CP}<em>{\text{in}}, \text{CP}</em>{\text{in}} \rangle )</td>
</tr>
<tr>
<td>IfNZ ( x ) Goto ( l' )</td>
<td>( \langle \text{CP}<em>{\text{in}}, \text{CP}</em>{\text{in}} \rangle )</td>
</tr>
<tr>
<td>Call ( f(x_1, \ldots, x_n) )</td>
<td>( \text{CP}_{\text{in}} )</td>
</tr>
<tr>
<td>Ret ( x )</td>
<td>( \text{CP}_{\text{in}} )</td>
</tr>
</tbody>
</table>
A more precise transformer

- \( \mathbf{C P}_{out} = F_{CP}^*[c](\mathbf{C P}_{in}) \)

<table>
<thead>
<tr>
<th>Command Type</th>
<th>( \mathbf{C P}_{out} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>( \mathbf{C P}_{in} )</td>
</tr>
<tr>
<td>( x := R )</td>
<td>( \mathbf{C P}<em>{in}[x\mapsto\text{eval}(R, \mathbf{C P}</em>{in})] )</td>
</tr>
<tr>
<td>Goto ( l' )</td>
<td>( \mathbf{C P}_{in} )</td>
</tr>
<tr>
<td>IfZ ( x ) Goto ( l' )</td>
<td>if ( \mathbf{C P}<em>{in}(x)=0 ) ( \langle \mathbf{C P}</em>{in}, \bot \rangle ) else if ( \mathbf{C P}<em>{in}(x)\neq0 ) ( \langle \bot, \mathbf{C P}</em>{in} \rangle ) else ( \langle \mathbf{C P}<em>{in}[x\mapsto0], \mathbf{C P}</em>{in} \rangle )</td>
</tr>
<tr>
<td>IfNZ ( x ) Goto ( l' )</td>
<td>if ( \mathbf{C P}<em>{in}(x)=0 ) ( \langle \bot, \mathbf{C P}</em>{in} \rangle ) else if ( \mathbf{C P}<em>{in}(x)\neq0 ) ( \langle \mathbf{C P}</em>{in}, \bot \rangle ) else ( \langle \mathbf{C P}<em>{in}, \mathbf{C P}</em>{in}[x\mapsto0] \rangle )</td>
</tr>
<tr>
<td>Call ( f(x_1, \ldots, x_n) )</td>
<td>( \mathbf{C P}_{in} )</td>
</tr>
<tr>
<td>Ret ( x )</td>
<td>( \mathbf{C P}_{in} )</td>
</tr>
</tbody>
</table>
eval \( R, \ CP_{\text{in}} \)

- \( \text{eval}(a+b, \ CP_{\text{in}}) = n+k \) if \( CP_{\text{in}}(a)=n \) and \( CP_{\text{in}}(b)=k \)

- ...
RUNNING GLOBAL CONSTANT PROPAGATION EXAMPLE
Input CFG

entry

x := 6;

y := x;
z := y;
w := x;
z := x;

x := 4;

exit
Setting initial values

entry
\( x = y = z = w = \top \)

\( x := 6; \)
\( x = y = z = w = \bot \)

\( y := x; \)
\( x = y = z = w = \bot \)

\( z := y; \)
\( x = y = z = w = \bot \)

\( w := x; \)
\( x = y = z = w = \bot \)

\( z := x; \)
\( x = y = z = w = \bot \)

exit
\( x := 4; \)
\( x = y = z = w = \bot \)
Iteration 1

entry
x=y=z=w=T

x := 6;
x=y=z=w=⊥

y := x;
x=y=z=w=⊥

z := y;
x=y=z=w=⊥

w := x;
x=y=z=w=⊥

z := x;
x=y=z=w=⊥

x := 4;
x=y=z=w=⊥

exit
Iteration 1

entry

x:=y:=z:=w:=\top

x:=y:=z:=w:=\bot

y := x;

x:=y:=z:=w:=\bot

z := y;

x:=y:=z:=w:=\bot

w := x;

x:=y:=z:=w:=\bot

z := x;

x:=y:=z:=w:=\bot

x := 4;

x:=y:=z:=w:=\bot

exit
entry
x=y=z=w=\top

x = y = z = w = \top
x := 6;

x = 6
y = z = w = \top

y := x;
x = y = z = w = \bot

z := y;
x = y = z = w = \bot

z := x;
x = y = z = w = \bot

w := x;
x = y = z = w = \bot

x := 4;
x = y = z = w = \bot

exit
entry
x=y=z=w=\top

x=y=z=w=\top

y := x;
x=y=z=w=\bot

z := y;x=y=z=w=\bot

w := x;
x=y=z=w=\bot

z := x;
x=y=z=w=\bot

x := 4;
x=y=z=w=\bot

exit
Iteration 2

entry

x=y=z=w=\top

x=y=z=w=\top

x=6 \ y=z=w=\top

y := x;

x=6 \ y=z=w=\bot

z := y;

x=6 \ y=z=w=\bot

w := x;

x=6 \ y=z=w=\bot

z := x;

x=6 \ y=z=w=\bot

z := x;

x=6 \ y=z=w=\bot

x := 4;

x=6 \ y=z=w=\bot

exit
entry
x=y=z=w=\top

x=y=z=w=\top
y := x;
x=y=6 z=w=\top

w := x;
x=y=z=w=\bot

z := x;
x=y=z=w=\bot

x := 4;
x=y=z=w=\bot

exit
Iteration 3

entry
x=y=z=w=\top

x=6 \ y=z=w=\top
y := x;
x=y=6 \ z=w=\top

w := x;
x=y=z=w=\bot

z := x;
x=y=z=w=\bot

exit
x := 4;
x=y=z=w=\bot
entry
x = y = z = w = T

x = y = z = w = T
y := x;
x = y = 6 z = w = T

x = 6 y = z = w = T

w := x;
x = y = z = w = ⊥

z := x;
x = y = z = w = ⊥

x := 4;
x = y = z = w = ⊥
Iteration 3

entry
x=y=z=w=⊤

x=6 y=z=w=⊤
y := x;
x=y=6 z=w=⊤

w := x;
x=y=z=w=⊥
z := x;
x=y=z=w=⊥
x := 4;
x=y=z=w=⊥

exit

x=y=z=w=⊤
Iteration 4

entry
x=y=z=w=\top

x=y=z=w=\top
x := 6;
x=6_y=z=w=\top

x=6 y=z=w=\top
y := x;
x=y=6 z=w=\top

x=6 y=z=w=\top
z := y;
x=6 y=z=w=\top

w := x;
x=y=z=w=\bot

z := x;
x=y=z=w=\bot

y=6 \lor y=\top \text{ gives what?}

x := 4;
x=y=z=w=\bot

exit
Iteration 4

entry

\[
x = y = z = w = \top
\]

\[
x = 6; \\
x = 6 \quad y = z = w = \top
\]

\[
x = 6 \quad y = z = w = \top \\
y := x; \\
x = y = 6 \quad z = w = \top
\]

\[
x = 6 \quad y = z = w = \top \\
z := y; \\
x = 6 \quad y = z = w = \top
\]

\[
x = 6 \quad y = z = w = \top \\
w := x; \\
x = y = z = w = \bot
\]

\[
z := x; \\
x = y = z = w = \bot
\]

exit

\[
x := 4; \\
x = y = z = w = \bot
\]
Iteration 4

entry
\[ x = y = z = w = \top \]

\[ x = 6; \]
\[ x = 6 \quad y = z = w = \top \]

\[ x = 6 \quad y = z = w = \top \]
\[ y := x; \]
\[ x = y = 6 \quad z = w = \top \]

\[ x = 6 \quad y = z = w = \top \]
\[ z := y; \]
\[ x = 6 \quad y = z = w = \top \]

\[ x = 6 \quad y = z = w = \top \]
\[ w := x; \]
\[ x = w = 6 \quad y = z = \top \]

\[ z := x; \]
\[ x = y = z = w = \bot \]

exit
\[ x := 4; \]
\[ x = y = z = w = \bot \]
Iteration 5

entry
x=y=z=w=\top

x=y=z=w=\top
x := 6;
x=6 \ y=z=w=\top

x=6 \ y=z=w=\top
y := x;
x=y=6 \ z=w=\top

x=6 \ y=z=w=\top
z := y;
x=6 \ y=z=w=\top

x=6 \ y=z=w=\top
w := x;
x=w=6 \ y=z=\top

x=6 \ y=z=w=\top

z := x;
x=y=z=w=\bot

x=6 \ y=z=w=\bot

exit
x := 4;
x=y=z=w=\bot

x=6 \ y=z=w=\bot
gives what?
Y=\bot \ \cup \ y=\top \ gives \ what?
Iteration 5

entry
x=y=z=w=1

x=6 y=z=w=1
y := x;
x=y=6 z=w=1

x=6 y=z=w=1
w := x;
x=w=6 y=z=1

x=w=6 y=z=1
z := x;
x=y=z=w=1

x := 4;
x=y=z=w=1

exit
Iteration 5

entry
x=y=z=w=\top

x=y=z=w=\top
y := x;
x=y=6 z=w=\top

x=6 y=z=w=\top
w := x;
x=w=6 y=z=\top

x=w=6 y=z=\top
z := x;
x=w=z=6 y=\bot

x := 4;
x=y=z=w=\bot

exit
Iteration 6

entry
x=y=z=w=1

x=y=z=w=1
y := x;
x=6 y=z=w=1

x=6 y=z=w=1
w := x;
x=w=6 y=z=1

x=w=6 y=z=1
z := x;
x=w=z=6 y=1

x=w=z=6 y=1
x := 4;
x=y=z=w=⊥

exit
Iteration 6

entry
x=y=z=w=\top

x=6 \ y=z=w=\top
y := x;
x=y=6 \ z=w=\top

x=6 \ y=z=w=\top
w := x;
x=w=6 \ y=z=\top

x=w=6 \ y=z=\top
z := x;
x=w=z=6 \ y=\top

x=w=z=6 \ y=\top
x := 4;
x=4 \ w=z=6 \ y=\top

exit
Iteration 7

entry
x=y=z=w=T

x=6 y=z=w=T
y := x;
x=y=6 z=w=T

x=6 y=z=w=T
w := x;
x=w=6 y=z=T

x=w=6 y=z=T
z := x;
x=w=z=6 y=T

x=w=z=6 y=T
x := 4;
x=4 w=z=6 y=T

exit
Iteration 7

entry
x=y=z=w=\top

x=y=z=w=\top
x := 6;
x=6 y=z=w=\top

x=6 y=z=w=\top
y := x;
x=y=6 z=w=\top

x=6 y=z=w=\top
z := y;
x=6 y=z=w=\top

x=6 y=z=w=\top
w := x;
x=w=6 y=z=\top

w=6 x=y=z=\top
z := x;
x=w=z=6 y=\top

x=w=z=6 y=\top
x := 4;
x=4 w=z=6 y=\top
Iteration 7

entry
x=y=z=w=\top

x=6 \ y=z=w=\top
y := x;
x=y=6 \ z=w=\top

x=6 \ y=z=w=\top
w := x;
x=w=6 \ y=z=\top

w=6 \ x=y=z=\top
z := x;
w=6 \ x=y=z=\top

x=w=z=6 \ y=\top
x := 4;
x=4 \ w=z=6 \ y=\top

exit
Iteration 8

entry
x=y=z=w=\top

x=6 \ y=z=w=\top
y := x;
x=y=6 \ z=w=\top

x=6 \ y=z=w=\top
w := x;
x=w=6 \ y=z=\top

w=6 \ x=y=z=\top
z := x;
w=6 \ x=y=z=\top

x=w=z=6 \ y=\top
x := 4;
x=4 \ w=z=6 \ y=\top

exit
Iteration 8

entry
x=y=z=w=\top

x=6 \ y=z=w=\top
y := x;
x=x=6 \ z=w=\top

x=6 \ y=z=w=\top
w := x;
x=x=w=6 \ y=z=\bot

w=6 \ x=y=z=\top
z := x;
w=w=6 \ x=y=z=\top

w=6 \ x=y=z=\top
x := 4;
x=x=4 \ w=6 \ y=z=\bot

exit
Iteration 9 – no change

entry

\[ x = y = z = w = T \]

\[ x = y = z = w = T \]
\[ x := 6; \]
\[ x = 6 \quad y = z = w = T \]

\[ x = 6 \quad y = z = w = T \]
\[ y := x; \]
\[ x = y = 6 \quad z = w = T \]

\[ x = 6 \quad y = z = w = T \]
\[ z := y; \]
\[ x = 6 \quad y = z = w = T \]

\[ x = 6 \quad y = z = w = T \]
\[ w := x; \]
\[ x = w = 6 \quad y = z = T \]

\[ w = 6 \quad x = y = z = T \]
\[ z := x; \]
\[ w = 6 \quad x = y = z = T \]

\[ w = 6 \quad x = y = z = T \]
\[ x := 4; \]
\[ x = 4 \quad w = 6 \quad y = z = T \]

exit
Iteration 10 – no change

entry
x=y=z=w=T

x=6 y=z=w=T
y := x;
x=y=6 z=w=T

x=6 y=z=w=T
z := y;
x=6 y=z=w=T

x=6 y=z=w=T
w := x;
x=w=6 y=z=T

w=6 x=y=z=T
z := x;
w=6 x=y=z=T

w=6 x=y=z=T
x := 4;
x=4 w=6 y=z=T

exit
Iteration 11 – no change

entry
x=y=z=w=\top

x=y=z=w=\top
y := x;
x=y=6 z=w=\top

x=6 y=z=w=\top
w := x;
x=w=6 y=z=\top

w=6 x=y=z=\top
z := x;
w=6 x=y=z=\top

w=6 x=y=z=\top
x := 4;
x=4 w=6 y=z=\top

exit
Iteration 12 – no change

entry
x=y=z=w=\top

x=y=z=w=\top
y := x;
x=y=6 \land z=w=\top

x=6 \land y=z=w=\top
w := x;
x=w=6 \land y=z=\top

w=6 \land x=y=z=\top
z := x;
w=6 \land x=y=z=\top

w=6 \land x=y=z=\top
x := 4;
x=4 \land w=6 \land y=z=\top

exit
Iteration 13 – no change

entry
x=y=z=w=1

x=6 y=z=w=1
y := x;
x=y=6 z=w=1

x=6 y=z=w=1
w := x;
x=w=6 y=z=1

w=6 x=y=z=1
z := x;
w=6 x=y=z=1

w=6 x=y=z=1
x := 4;
x=4 w=6 y=z=1

exit
Analysis fixed point

entry
\[ x=y=z=w=\top \]

\[ x=6 \quad y=z=w=\top \]
\[ y := x; \quad x=y=6 \quad z=w=\top \]

\[ x=6 \quad y=z=w=\top \]
\[ z := y; \quad x=6 \quad y=z=w=\top \]

\[ x=6 \quad y=z=w=\top \]
\[ w := x; \quad x=w=6 \quad y=z=\top \]

\[ w=6 \quad x=y=z=\top \]
\[ z := x; \quad w=6 \quad x=y=z=\top \]

w=6  x=y=z=\top
x := 4;
x=4  w=6  y=z=\top

exit

x=y=z=w=\top
Transformation opportunities

entry
x=y=z=w=T

x=6  y=z=w=T
y := x;
x=x=6  z=w=T

x=6  y=z=w=T
z := y;
x=x=6  y=z=w=T

x=6  y=z=w=T
w := x;
x=x=6  y=z=T

w=6  x=y=z=T
z := x;
w=w=6  x=y=z=T

w=6  x=y=z=T
x := 4;
x=x=4  w=6  y=z=T

exit
Next lecture: Loop Optimizations
Fully detailed liveness transformer

- \( L V_{in} = F_{L V}[C](L V_{out}) \)

<table>
<thead>
<tr>
<th>Command Type</th>
<th>( A E_{out} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>( L V_{out} )</td>
</tr>
<tr>
<td>( x := R )</td>
<td>( (L V_{out} \setminus {x}) \cup Vars(R) )</td>
</tr>
<tr>
<td>Goto ( l' )</td>
<td>( L V_{out} )</td>
</tr>
<tr>
<td>IfZ ( x ) Goto ( l' )</td>
<td>( L V_{out} \cup {x} )</td>
</tr>
<tr>
<td>IfNZ ( x ) Goto ( l' )</td>
<td>( L V_{out} \cup {x} )</td>
</tr>
<tr>
<td>Call ( f(x_1, \ldots, x_n) )</td>
<td>( L V_{out} \cup {x_1, \ldots, x_n} )</td>
</tr>
<tr>
<td>Ret ( x )</td>
<td>( L V_{out} \cup {x} )</td>
</tr>
</tbody>
</table>