YSC4230: Programming Language Design and Implementation

Week 13: Register Allocation
Current Research in PLDI
Wrap-Up

Ilya Sergey

ilya.sergey@yale-nus.edu.sg

Previous Lectures: Liveness

- A variable v is *live* at a program point if v is defined before the program point and used after it.
- Liveness is defined in terms of where variables are *defined* and where variables are *used*
- Liveness analysis: Compute the live variables between each statement.
  - May be *conservative* (i.e. it may claim a variable is live when it isn't) so because that's a safe approximation
  - To be useful, it should be more *precise* than simple scoping rules.

Simple Liveness Analysis Algorithm

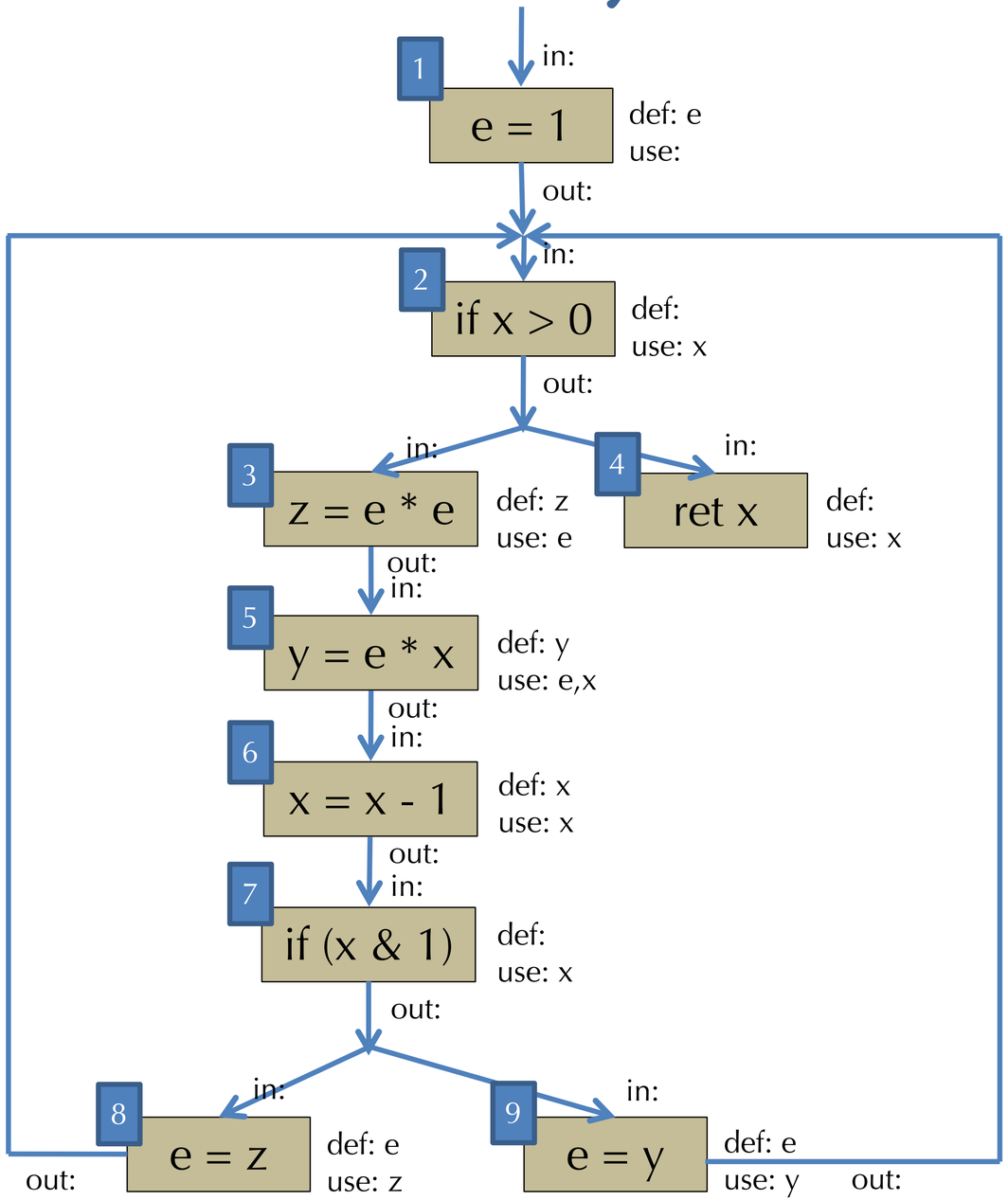
for all n, in[n] := ∅, out[n] := ∅
repeat until no change in ‘in’ and ‘out’
    for all n
      out[n] := $\bigcup_{n' \in succ[n]}$ in[n’]
      in[n] := use[n] ∪ (out[n] - def[n])
    end
end

- Finds a *fixpoint* of the in and out equations.

Example Liveness Analysis

- Example flow graph:

```
e = 1;
while(x>0) {
  z = e * e;
  y = e * x;
  x = x – 1;
  if (x & 1) {
    e = z;
  } else {
    e = y;
  }
}
return x;
```



Example Liveness Analysis

Each iteration update:

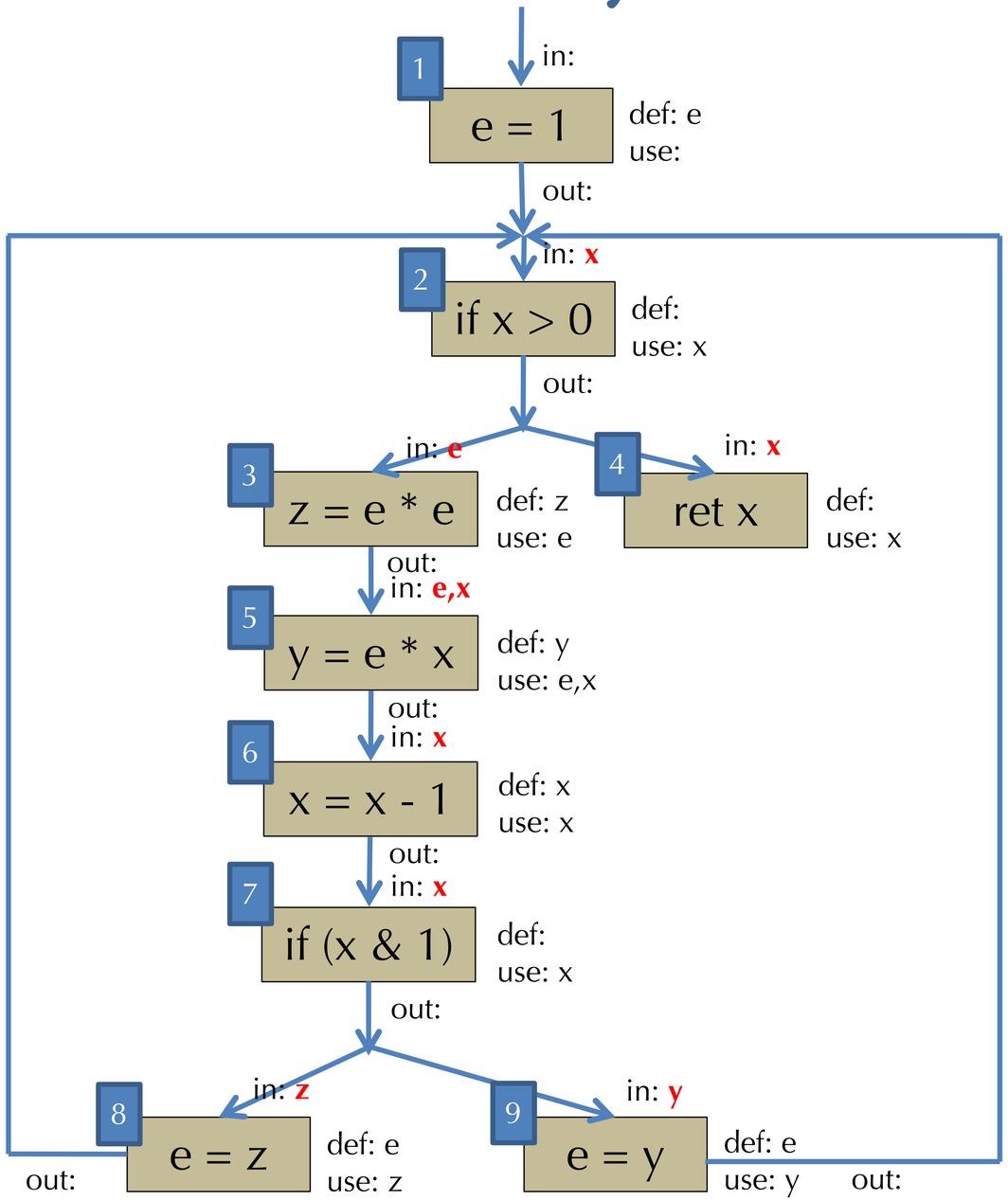
$out[n] := \bigcup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 1:

in[2] = x
in[3] = e
in[4] = x
in[5] = e,x
in[6] = x
in[7] = x
in[8] = z
in[9] = y

(showing only updates that make a change)

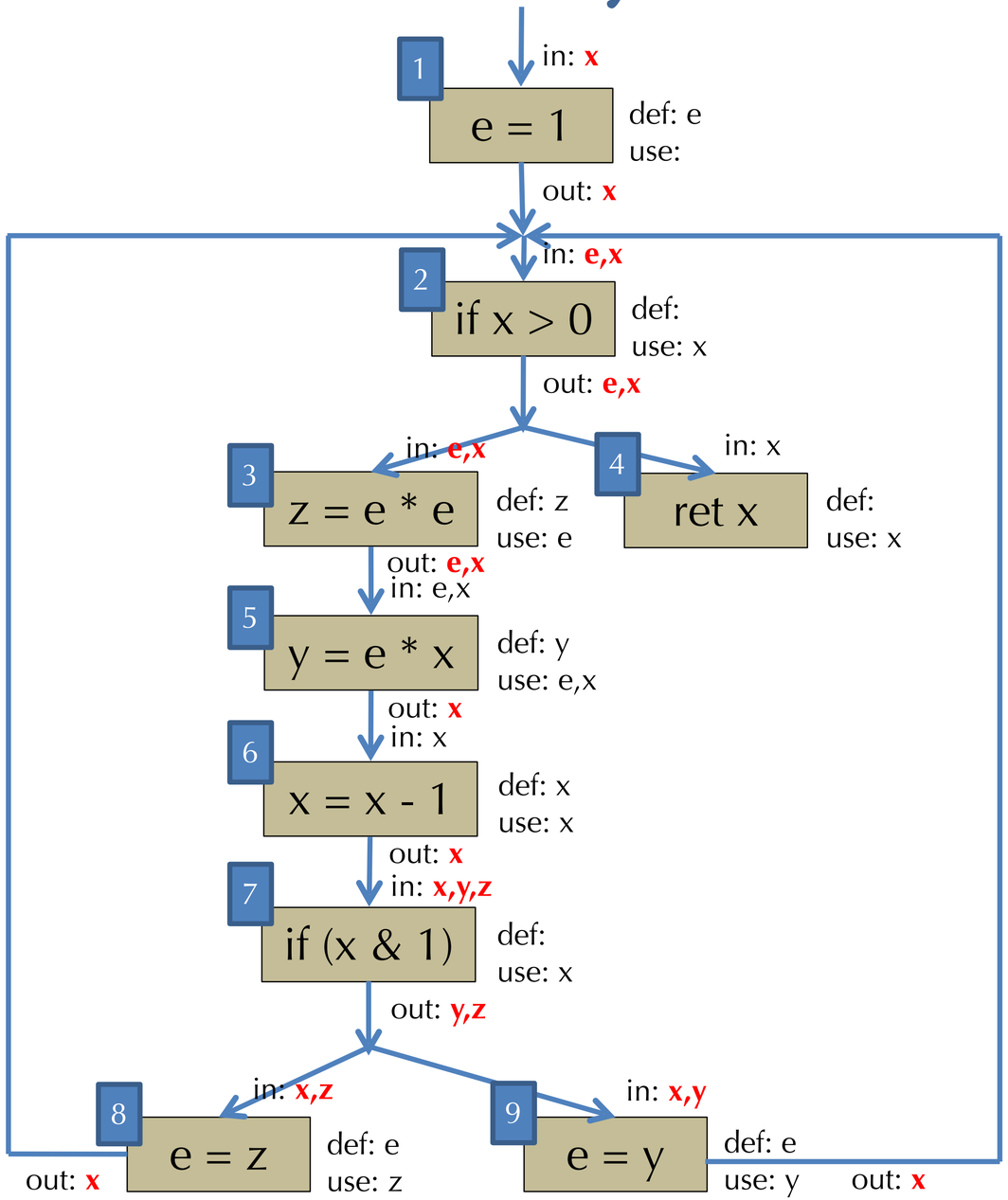


Example Liveness Analysis

Each iteration update:
out[n] := $\cup_{n' \in succ[n]}$ in[n']
in[n] := use[n] ∪ (out[n] - def[n])

- Iteration 2:

out[1]= x
in[1] = x
out[2] = e,x
in[2] = e,x
out[3] = e,x
in[3] = e,x
out[5] = x
out[6] = x
out[7] = z,y
in[7] = x,z,y
out[8] = x
in[8] = x,z
out[9] = x
in[9] = x,y



Example Liveness Analysis

Each iteration update:

$out[n] := \cup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 3:

out[1]= e,x

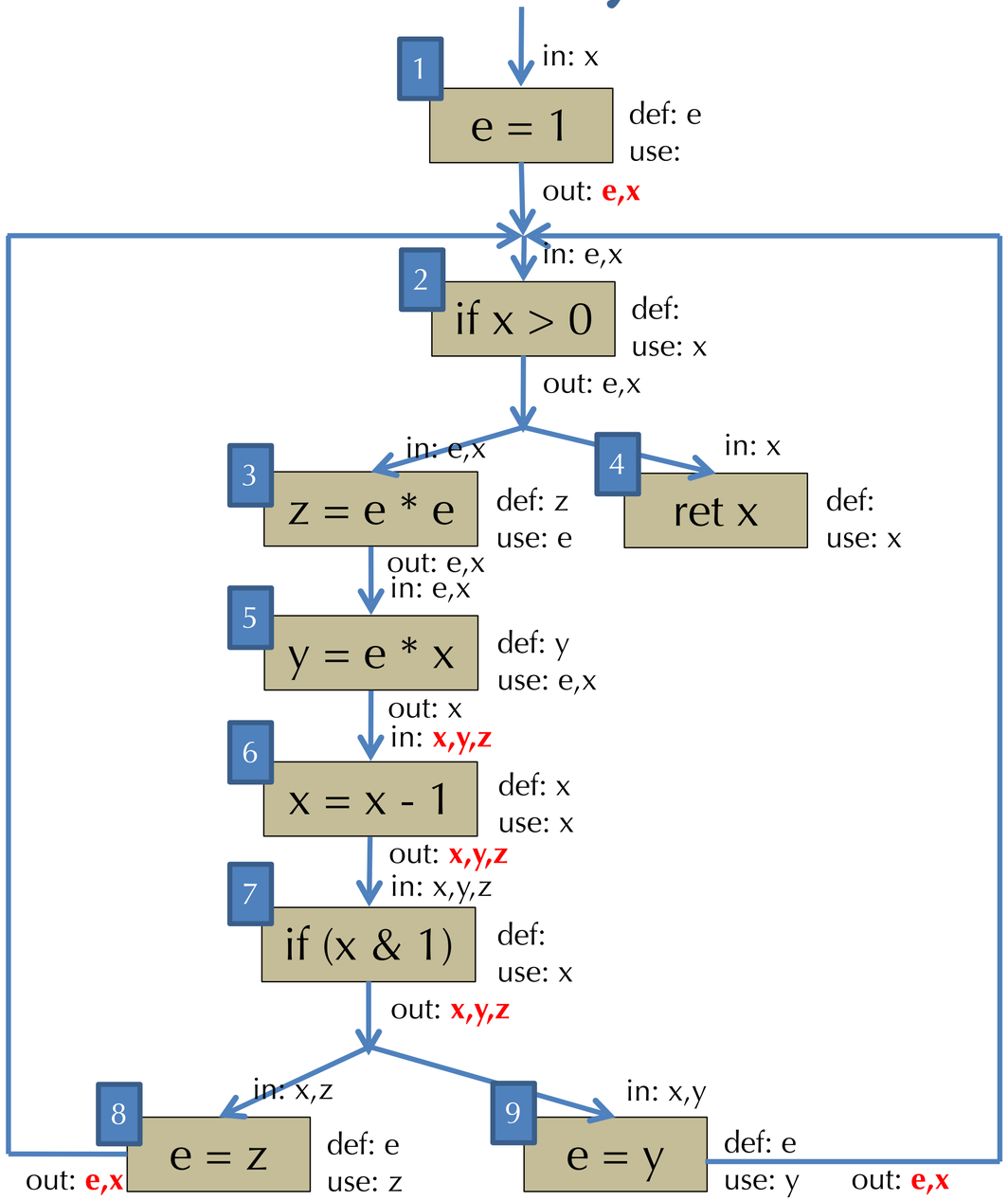
out[6]= x,y,z

in[6]= x,y,z

out[7]= x,y,z

out[8]= e,x

out[9]= e,x



Example Liveness Analysis

Each iteration update:

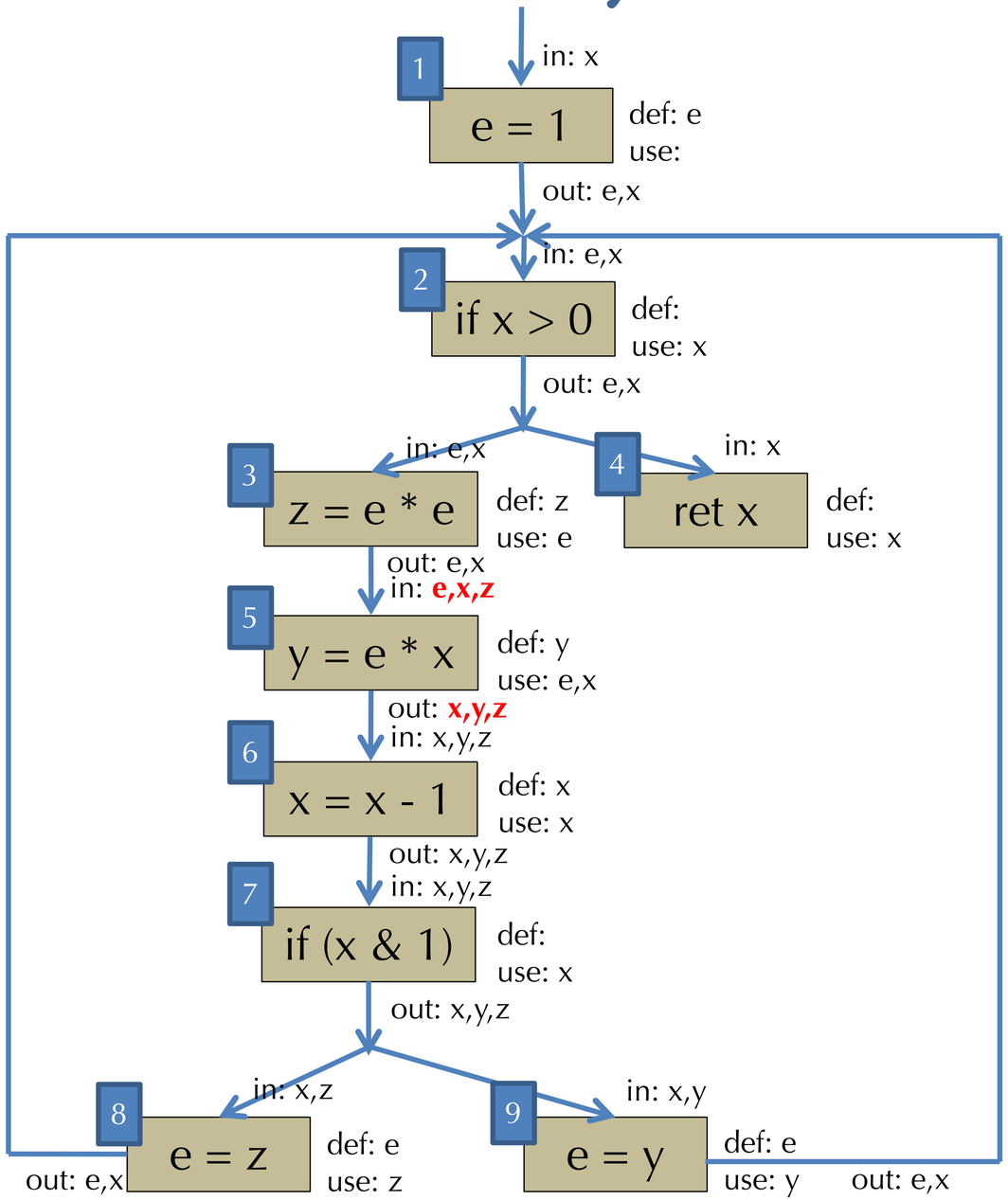
$out[n] := \cup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

- Iteration 4:

out[5]= x,y,z

in[5]= e,x,z



Example Liveness Analysis

Each iteration update:

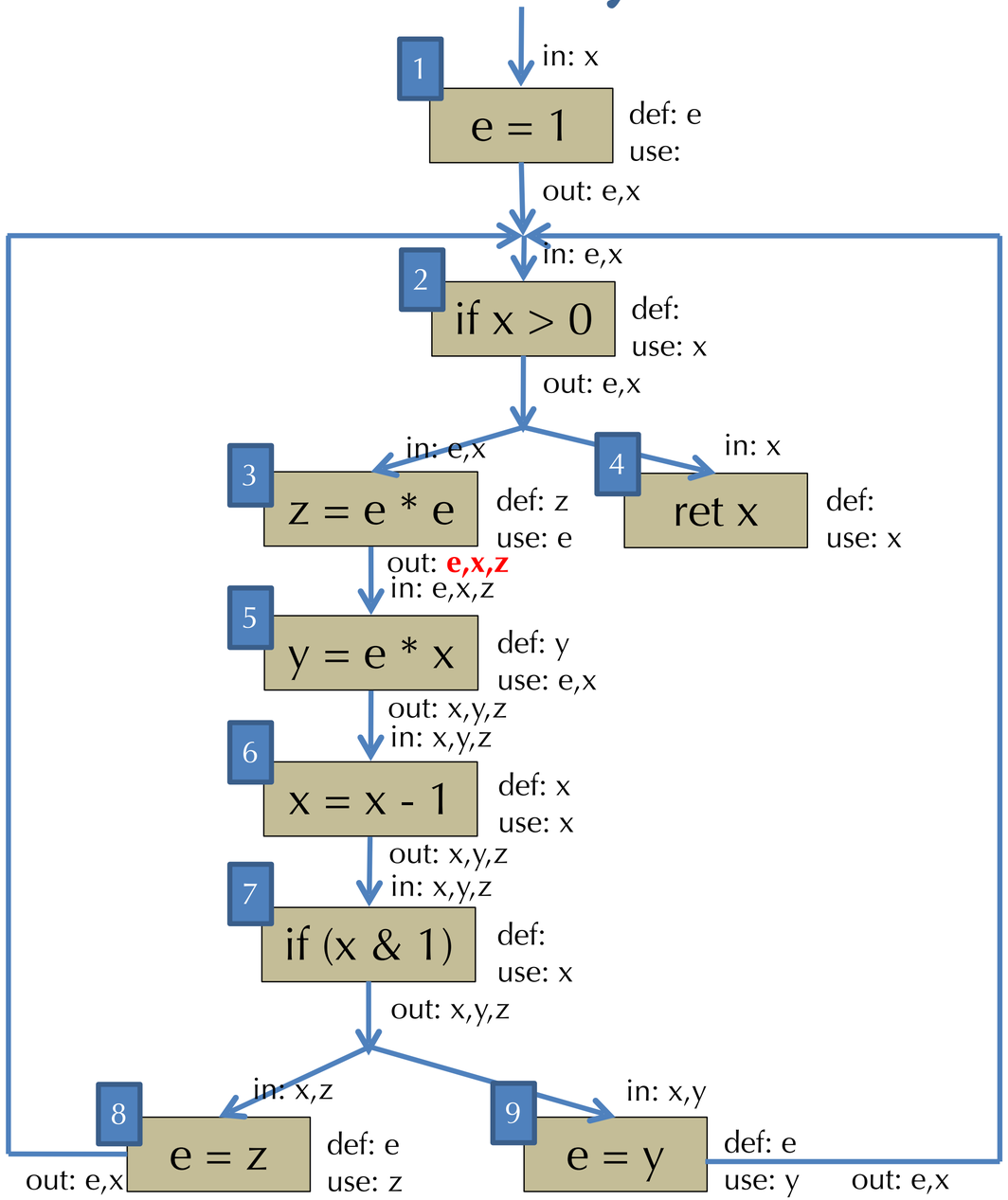
out[n] := $\cup_{n' \in succ[n]}$ in[n']

in[n] := use[n] ∪ (out[n] - def[n])

- Iteration 5:

out[3]= e,x,z

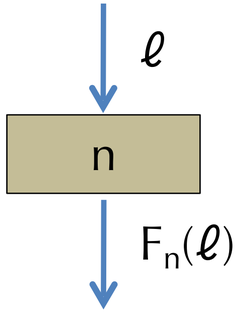
Done!

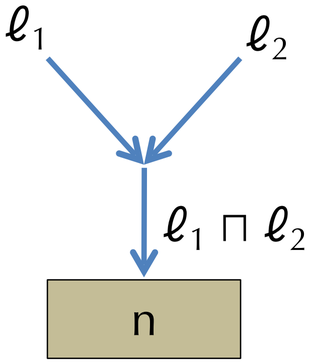


(Forward) Dataflow Analysis Framework

A forward dataflow analysis can be characterized by:

1. A domain of dataflow values $\mathcal{L}$
   - e.g. $\mathcal{L}$ = the powerset of all variables
   - Think of $\ell \in \mathcal{L}$ as a property, then "z $\in \ell$" means "z has the property"
2. For each node n, a ***flow function*** $F_n : \mathcal{L} \rightarrow \mathcal{L}$
   - So far we've seen $F_n(\ell) = gen[n] \cup (\ell - kill[n])$
   - So: $out[n] = F_n(in[n])$
   - "If ℓ is a property that holds before the node n, then $F_n(\ell)$ holds after n"
3. A ***combining*** operator $\sqcap$
   - "If we know *either* ℓ_1 *or* ℓ_2 holds on entry to node n, we know at most $\ell_1 \sqcap \ell_2$"
   - $in[n] := \bigsqcap_{n' \in pred[n]} out[n']$





Generic Iterative (Forward) Analysis

for all n, in[n] := $\top$, out[n] := $\top$

repeat until no change

  for all n

    in[n] := $\bigsqcap_{n' \in pred[n]} out[n']$

    out[n] := $F_n(in[n])$

  end

end

- Here, $\top \in \mathcal{L}$ (“top”) represents having the “maximum” amount of information.
  - Having “more” information enables more optimizations
  - “Maximum” amount could be inconsistent with the constraints, so we can’t keep it. :-(
  - Iteration refines the answer, eliminating inconsistencies

Structure of $\mathcal{L}$

- The domain has structure that reflects the "amount" of information for each dataflow value.

- Some dataflow values are more informative than others:
  - Write $\ell_1 \sqsubseteq \ell_2$ whenever ℓ_2 provides at least as much information as ℓ_1.
  - The dataflow value ℓ_2 is "better" for enabling optimizations.

- Example 1: for available expressions analysis, *larger* sets of nodes are *more informative*.
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination.
  - So: $\ell_1 \sqsubseteq \ell_2$ if and only if $\ell_1 \subseteq \ell_2$

- Example 2: for liveness analysis, *smaller* sets of variables are more informative.
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments.
  - So: $\ell_1 \sqsubseteq \ell_2$ if and only if $\ell_1 \supseteq \ell_2$

$\mathcal{L}$ as a Partial Order

- $\mathcal{L}$ is a *partial order* defined by the ordering relation $\sqsubseteq$.
- A partial order is an ordered set.

- Some of the elements might be *incomparable*.
  - That is, there might be $\ell_1, \ell_2 \in \mathcal{L}$ such that neither $\ell_1 \sqsubseteq \ell_2$ nor $\ell_2 \sqsubseteq \ell_1$

- Properties of a partial order:
  - *Reflexivity:* $\ell \sqsubseteq \ell$
  - *Transitivity:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_3$ implies $\ell_1 \sqsubseteq \ell_2$
  - *Anti-symmetry:* $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$ implies $\ell_1 = \ell_2$

- Examples:
  - Integers ordered by $\leq$
  - Types ordered by $<$:
  - Sets ordered by $\subseteq$ or $\supseteq$

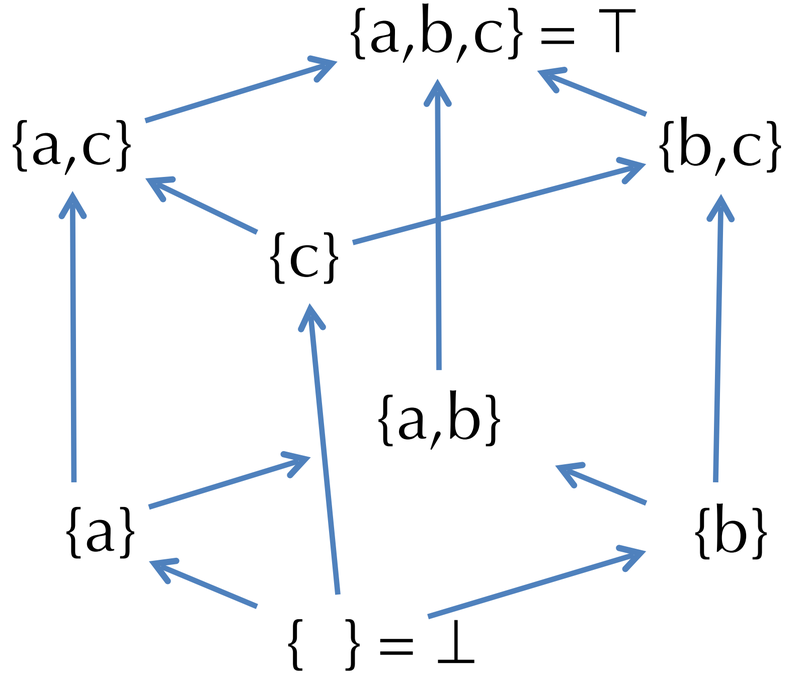
Subsets of {a,b,c} ordered by ⊆

ℓ_2

$\uparrow$ $\ell_1 \sqsubseteq \ell_2$

ℓ_1

Partial orders are often presented as a Hasse diagram.



Height is 3

order $\sqsubseteq$ is $\subseteq$ meet $\sqcap$ is $\cap$ join $\sqcup$ is $\cup$

Meets and Joins

- The *combining* operator $\sqcap$ is called the "meet" operation.
- It constructs the *greatest lower bound*:
  - $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_1$ and $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_2$

    "the meet is a lower bound"
  - If $\ell \sqsubseteq \ell_1$ and $\ell \sqsubseteq \ell_2$ then $\ell \sqsubseteq \ell_1 \sqcap \ell_2$

    "there is no greater lower bound"

- Dually, the $\sqcup$ operator is called the "join" operation.
- It constructs the *least upper bound*:
  - $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$

    "the join is an upper bound"
  - If $\ell_1 \sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$ then $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$

    "there is no smaller upper bound"

- A partial order that has all meets and joins is called a *lattice*.
  - If it has just meets, it's called a *meet semi-lattice*.

Another Way to Describe the (Forward) Algorithm

- Algorithm repeatedly computes (for each node n):
  - $out[n] := F_n(in[n])$

- Equivalently: $out[n] := F_n(\bigsqcap_{n' \in pred[n]} out[n'])$
  - By definition of $in[n]$

- We can write this as a simultaneous update of the vector of $out[n]$ values:
  - Let $x_n = out[n]$
  - Let $\mathbf{X} = (x_1, x_2, \ldots, x_n)$ it's a vector of points in $\mathcal{L}$ corresponding to CFG nodes
  - $\mathbf{F}(\mathbf{X}) = (F_1(\bigsqcap_{j \in pred[1]} out[j]), F_2(\bigsqcap_{j \in pred[2]} out[j]), \ldots, F_n(\bigsqcap_{j \in pred[n]} out[j]))$

- Any solution to the constraints is a *fixpoint* **X** of **F**
  - i.e. $\mathbf{F}(\mathbf{X}) = \mathbf{X}$

Iteration Computes Fixpoints

- Let $\mathbf{X}_0 = (\top, \top, \ldots, \top)$
- Each loop through the algorithm apply **F** to the old vector:
  $\mathbf{X}_1 = \mathbf{F}(\mathbf{X}_0)$
  $\mathbf{X}_2 = \mathbf{F}(\mathbf{X}_1)$
  …
- $\mathbf{F}^{k+1}(\mathbf{X}) = \mathbf{F}(\mathbf{F}^{k}(\mathbf{X}))$
- A fixpoint is reached when $\mathbf{F}^{k}(\mathbf{X}) = \mathbf{F}^{k+1}(\mathbf{X})$
  - That's when the algorithm stops.

- Wanted: a *maximal* fixpoint
  - Because that one is more informative/useful for performing optimizations

Monotonicity & Termination

- Each flow function F_n maps lattice elements to lattice elements; to be sensible is should be *monotonic*:

- $F : \mathcal{L} \rightarrow \mathcal{L}$ is *monotonic* iff:
  $\ell_1 \sqsubseteq \ell_2$ implies that $F(\ell_1) \sqsubseteq F(\ell_2)$
  - Intuitively: "If you have more information entering a node, then you have more information leaving the node."

- Monotonicity lifts point-wise to the function: $\mathbf{F} : \mathcal{L}^n \rightarrow \mathcal{L}^n$
  - vector $(x_1, x_2, \ldots, x_n) \sqsubseteq (y_1, y_2, \ldots, y_n)$ iff $x_i \sqsubseteq y_i$ for each i

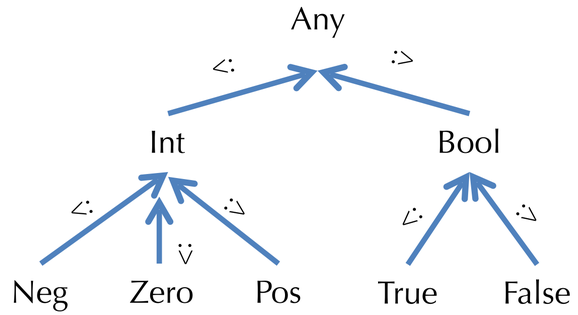
- Note that **F** is consistent: $\mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\ldots \sqsubseteq \mathbf{F}(\mathbf{F}(\mathbf{X}_0)) \sqsubseteq \mathbf{F}(\mathbf{X}_0) \sqsubseteq \mathbf{X}_0$

- Therefore, # steps needed to reach a fixpoint is at most the height H of $\mathcal{L}$ times the number of nodes: $O(H_n)$ — height of the lattice

Building Lattices?

- Information about individual nodes or variables can be lifted *pointwise:*
  - If $\mathcal{L}$ is a lattice, then so is $\{ f : X \to \mathcal{L} \}$ where $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$.

- Like *types*, the dataflow lattices are *static approximations* to the dynamic behavior:
  - Could pick a lattice based on subtyping:
  - Or other information:



- Points in the lattice are sometimes called dataflow "*facts*"

More on Fixpoint Solutions

- Remember constructing LL(1) parse tables

$$
\begin{aligned}
T &\longmapsto S\$ \\
S &\longmapsto ES' \\
S' &\longmapsto \varepsilon \\
S' &\longmapsto + S \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
$$

- First(T) = First(S)
- First(S) = First(E)
- First(S′) = { + }
- First(E) = { number, ‘(‘ }

- Follow(S′) = Follow(S)
- Follow(S) = { $, ‘)’ } ∪ Follow(S′)

Then: we want the *least* solution to this system of set equations… a *fixpoint* computation. More on these later in the course.

Now: This solution is obtained by starting from taking all First/Follow as ∅ and then iterating the equations until *fixpoint* is reached.

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ num.		⟼ (S)		

Dataflow Analysis: Summary

- Many dataflow analyses fit into a common framework.

- Key idea: *iterative solution* of a system of equations over a *lattice* of *facts* (constraints).
  - Iteration terminates if flow functions are *monotonic*.
  - Solution is obtained as the *greatest* fixpoint is reached via the *meet* operation ($\sqcap$).

- In the literature, sometimes the definition of the analysis lattice is *reversed*:
  - The most useful/precise information is represented by the bottom element ($\bot$)
  - Solution is obtained as the *least* fixpoint via iterative application of *join* operator ($\sqcup$)
  - The two definitions are equivalent modulo the (semi-)lattice *direction*.

Implementation

- See HW6
  - Generic analysis is to be defined in **solver.ml**
  - Control-Flow Graphs are defined in **cfg.ml**

- Analysis example: **liveness.ml**

- Printing analysis results, e.g., liveness:

  ./printanalysis.native -live llprograms/analysis2.ll

Register Allocation Problem

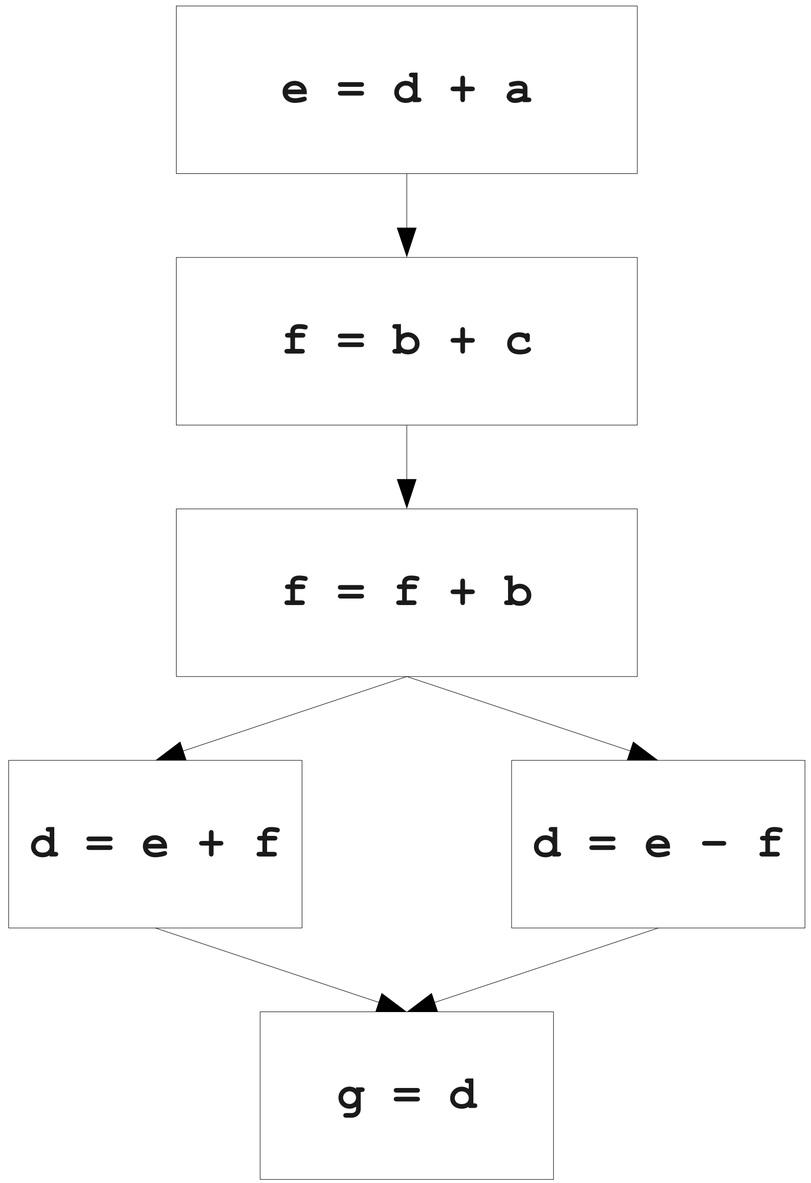
- Given: an IR program that uses an unbounded number of temporaries
  - e.g. the uids of our LLVM programs

- Find: a mapping from temporaries to machine registers such that
  - program semantics is preserved (i.e. the behaviour is the same)
  - register usage is maximised
  - moves between registers are minimised
  - calling conventions / architecture requirements are obeyed

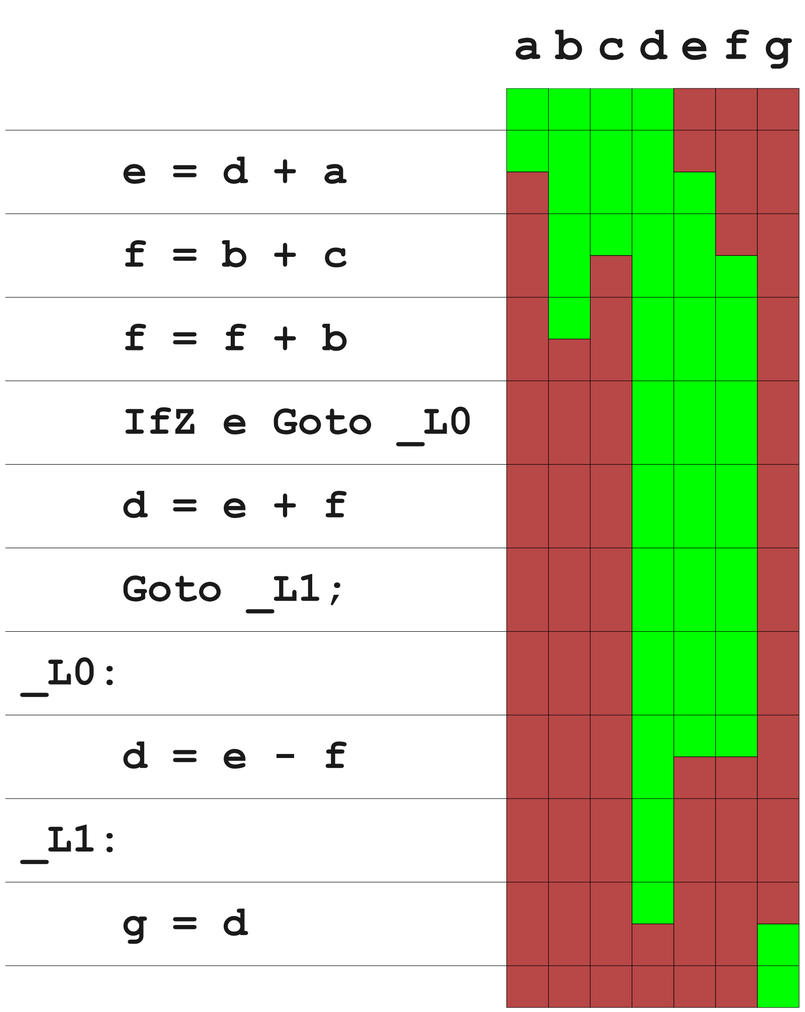
- Stack *Spilling*
  - If there are k registers available and m > k temporaries are *live* at the same time, then not all of them will fit into registers.
  - So: “spill” the excess temporaries to the stack.

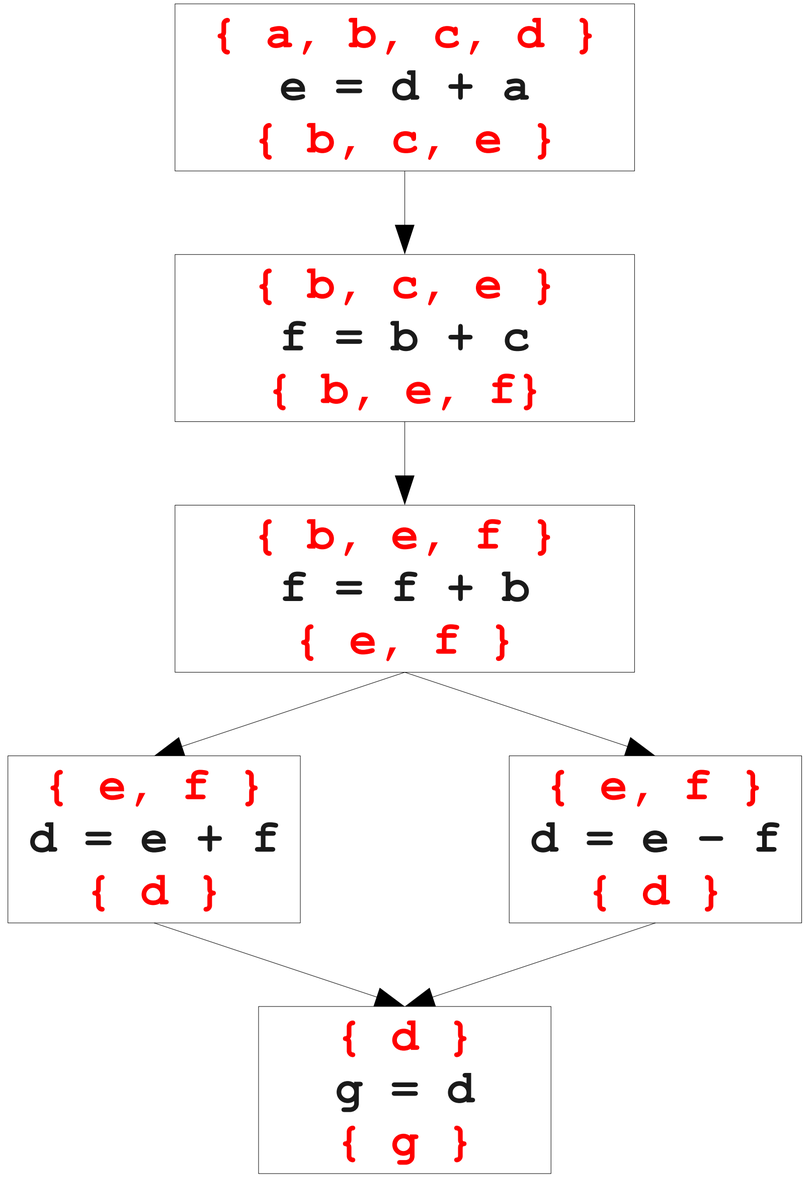
Linear-Scan Register Allocation

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



Linear-Scan Register Allocation

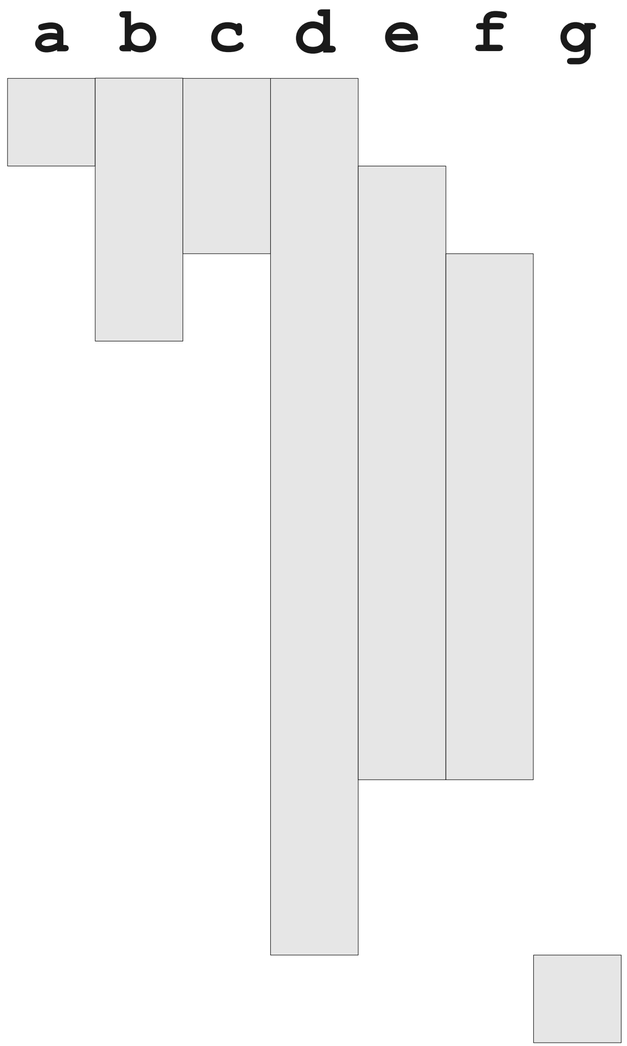




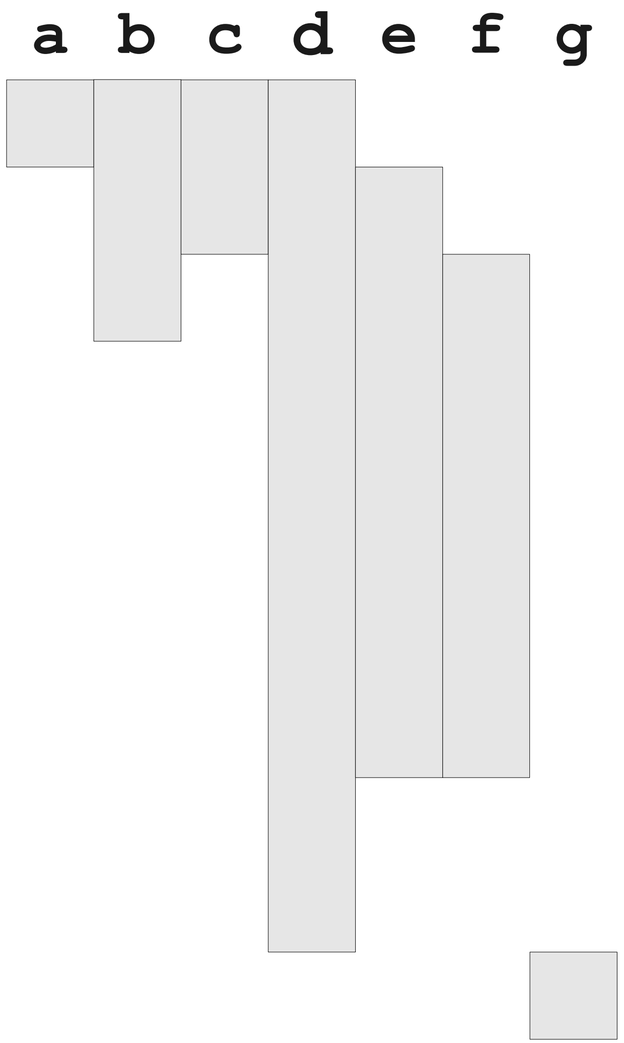
Linear-Scan Register Allocation

Idea: sweep the program *top-down*,
allocating registers for *live* variables and *evicting* non-live ones.

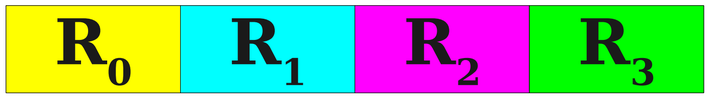
Linear-Scan Register Allocation



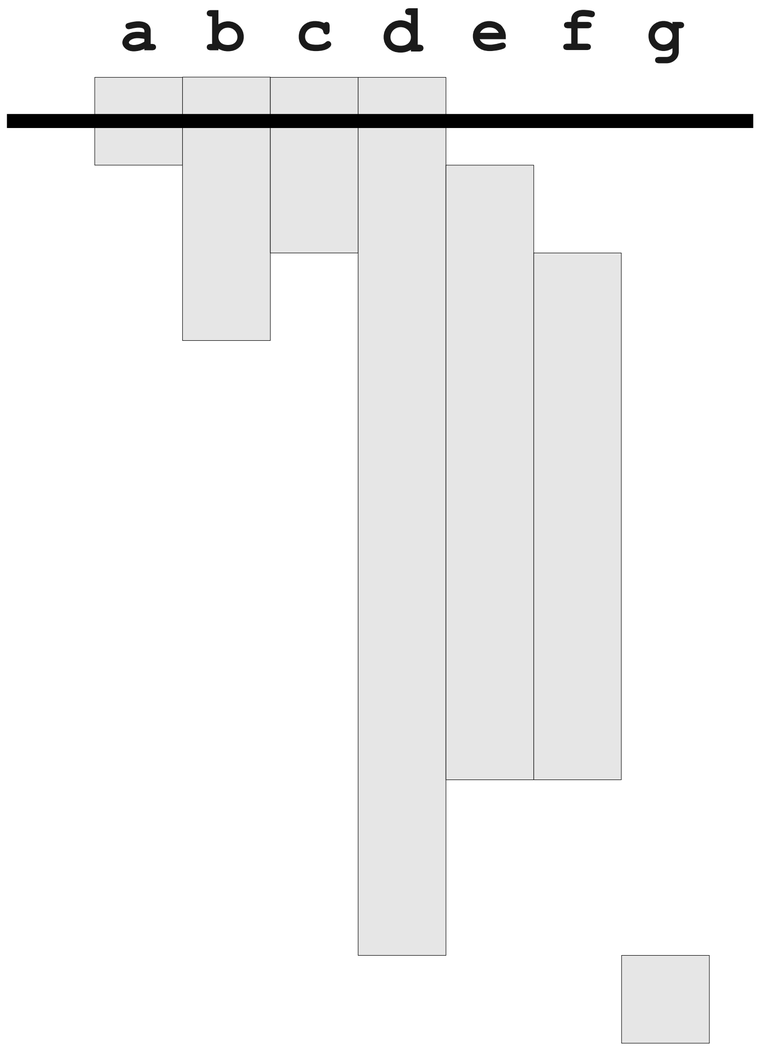
Linear-Scan Register Allocation



Free Registers



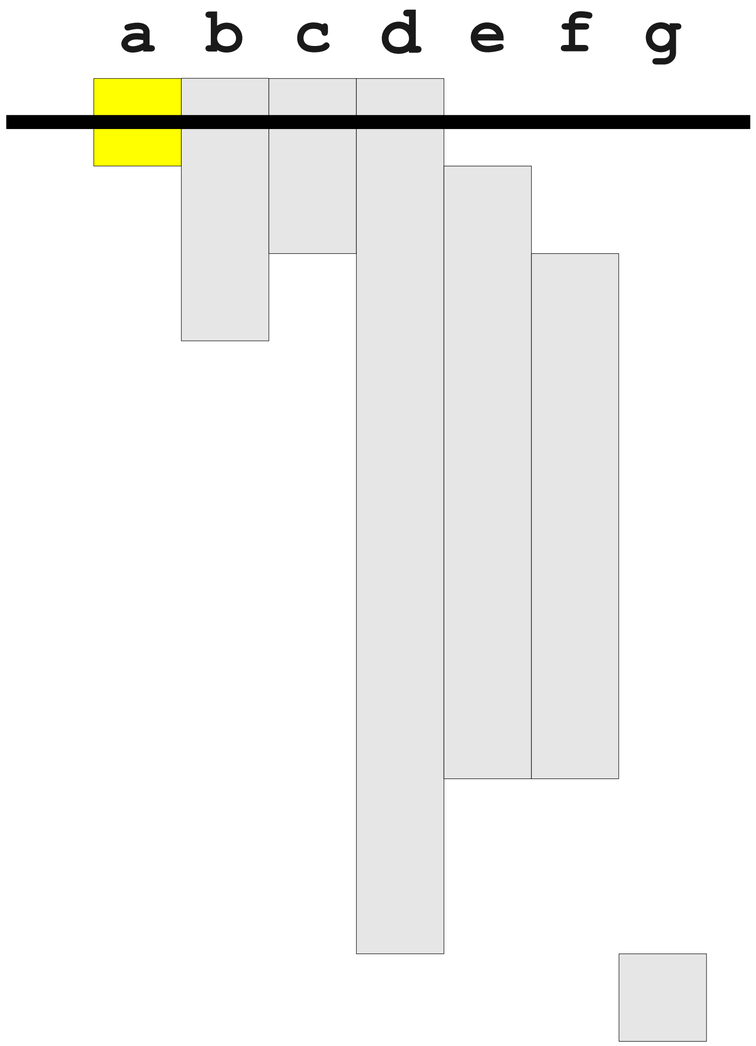
Linear-Scan Register Allocation



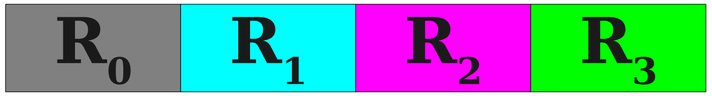
Free Registers

R_0	R_1	R_2	R_3

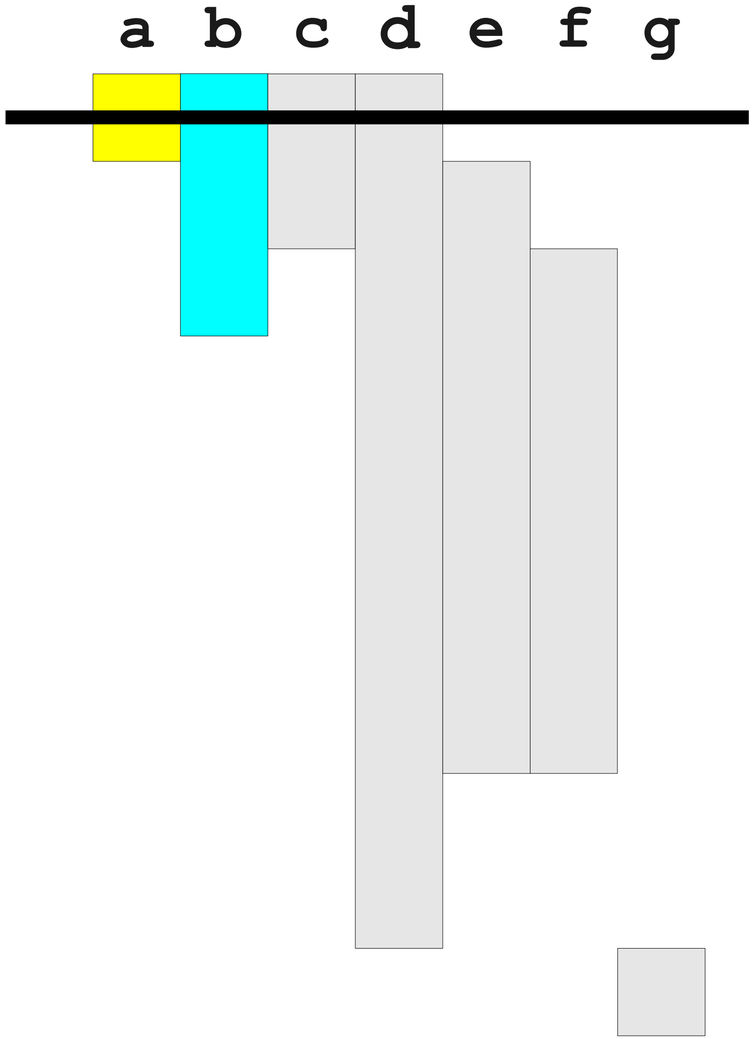
Linear-Scan Register Allocation



Free Registers

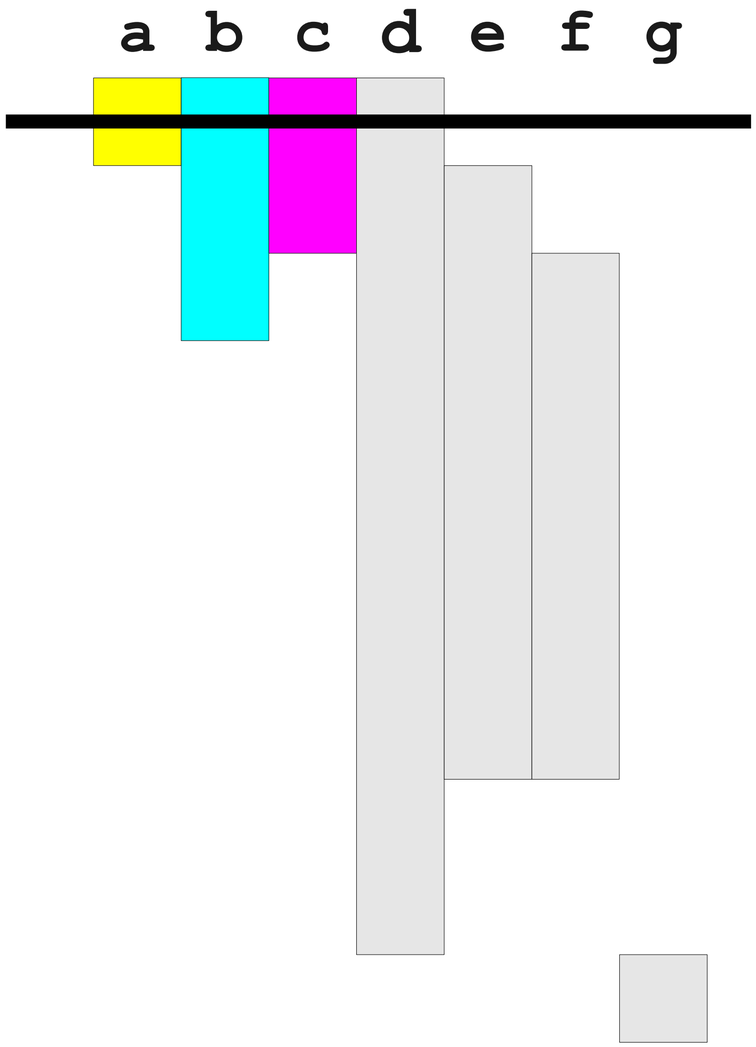


Linear-Scan Register Allocation



Free Registers

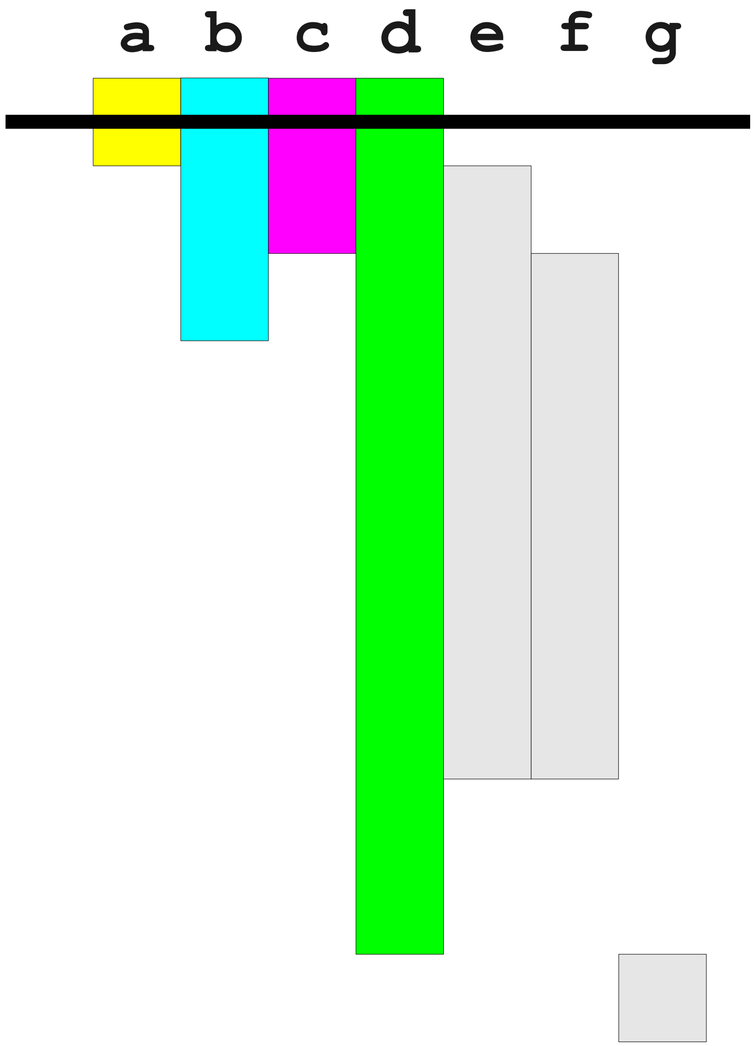
Linear-Scan Register Allocation



Free Registers

R_0	R_1	R_2	R_3

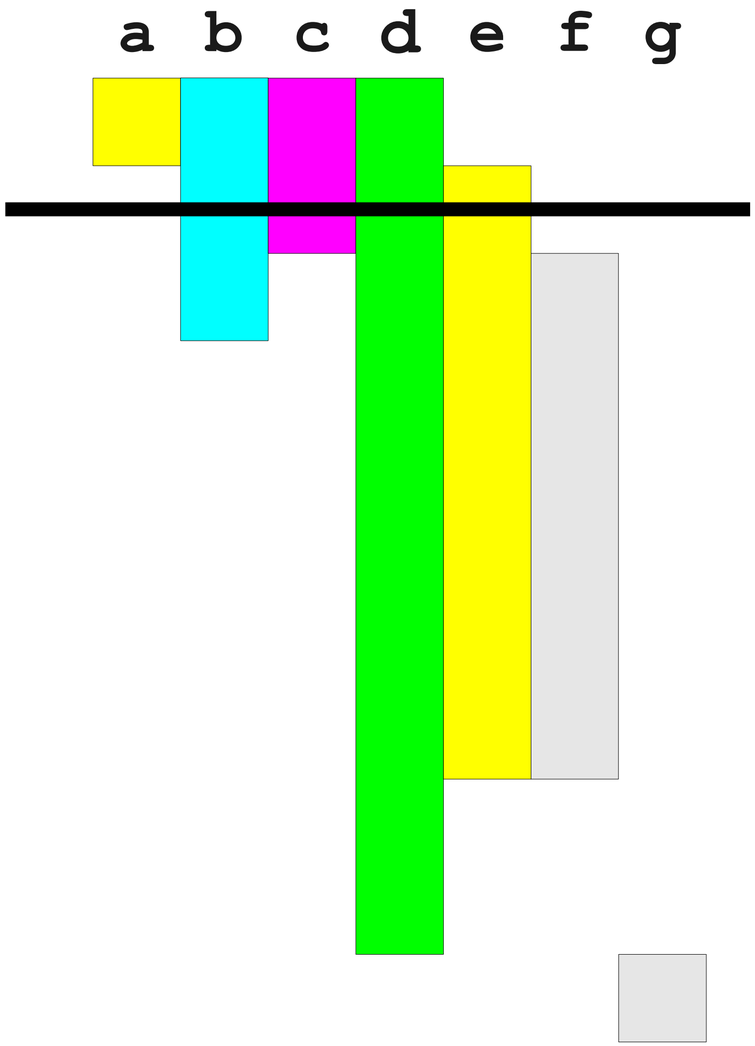
Linear-Scan Register Allocation



Free Registers

R_0	R_1	R_2	R_2

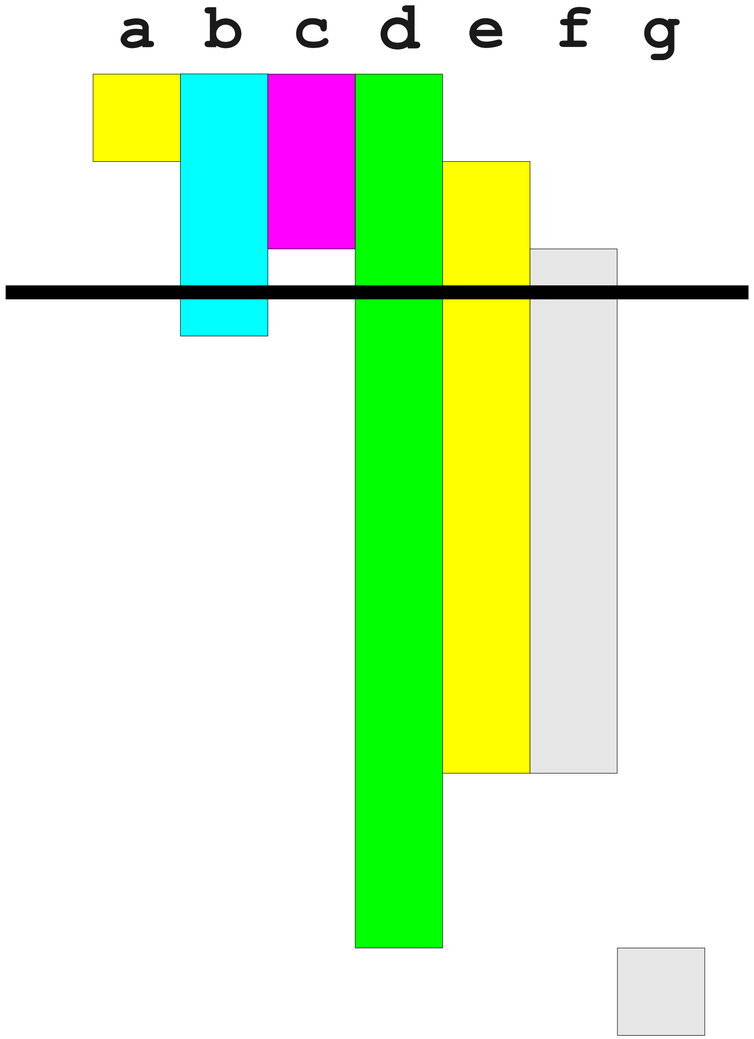
Linear-Scan Register Allocation



Free Registers

R_0	R_1	R_2	R_2

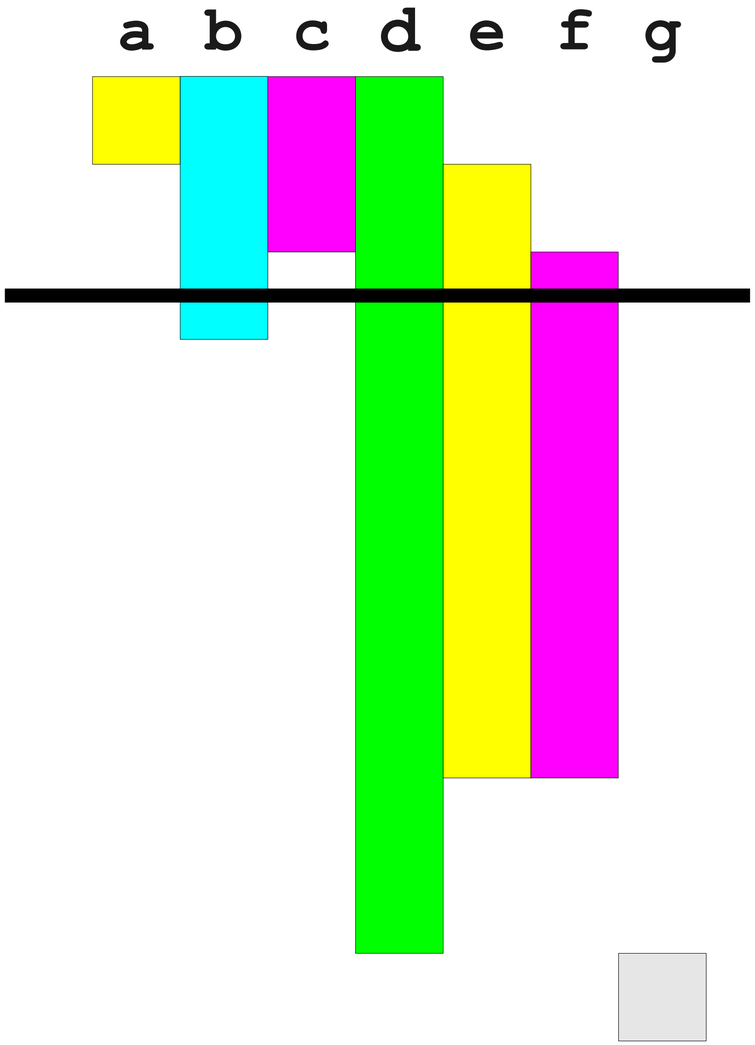
Linear-Scan Register Allocation



Free Registers

R_0	R_1	R_2	R_2

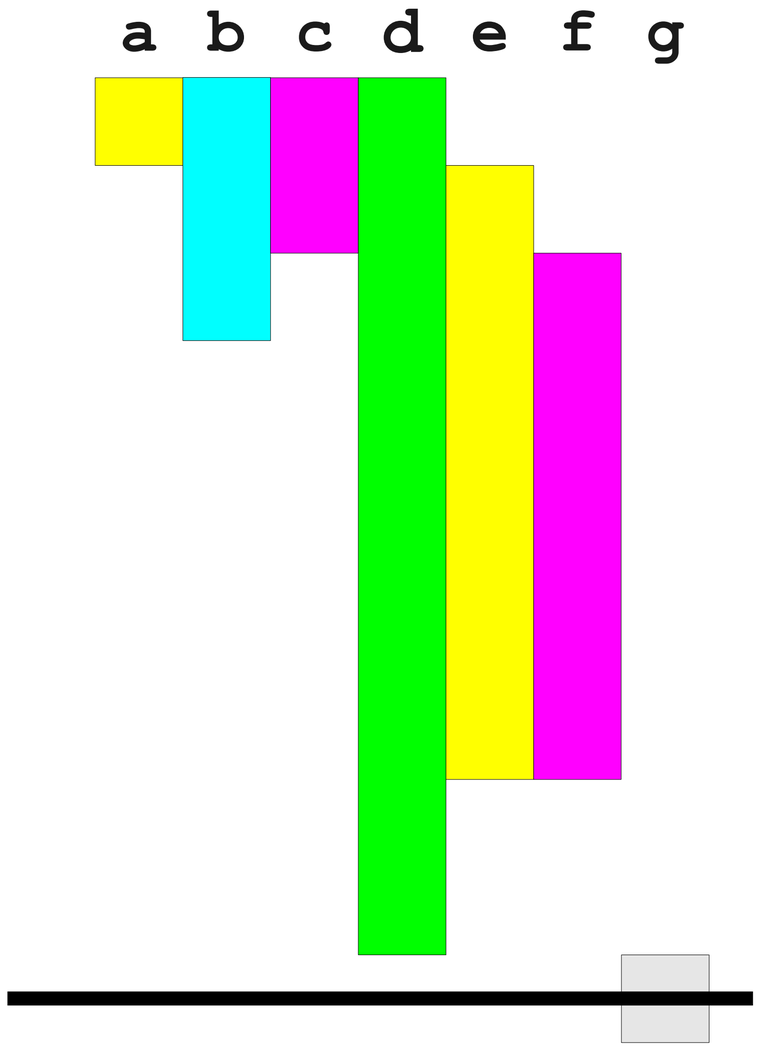
Linear-Scan Register Allocation



Free Registers

R_0	R_1	R_2	R_2

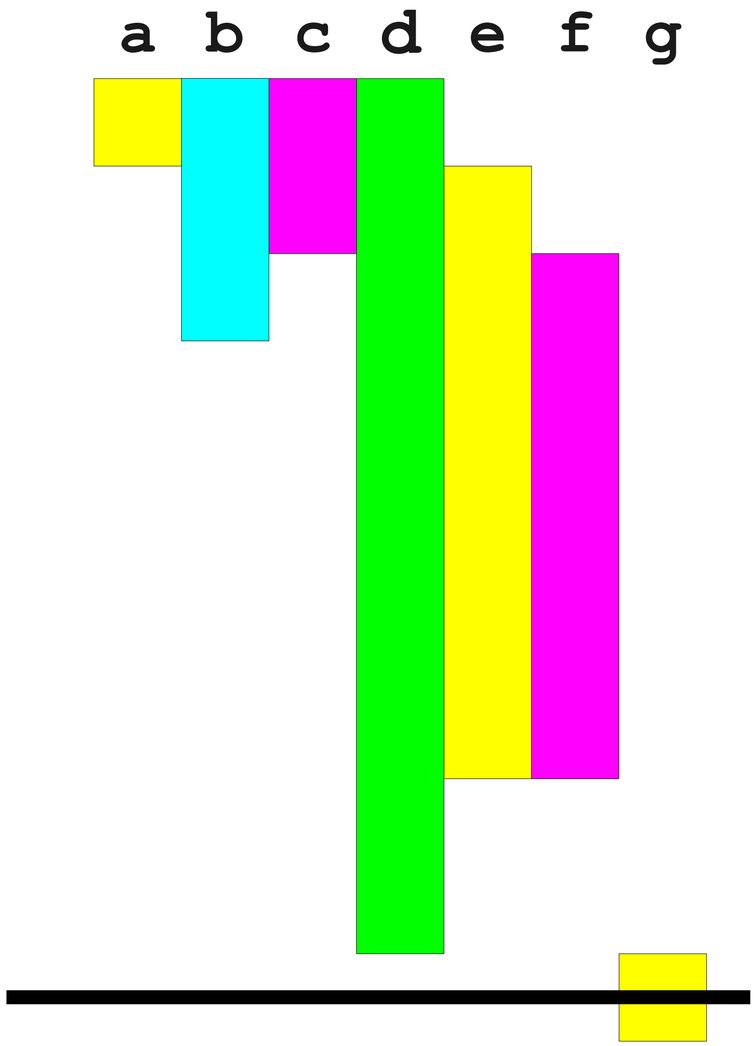
Linear-Scan Register Allocation



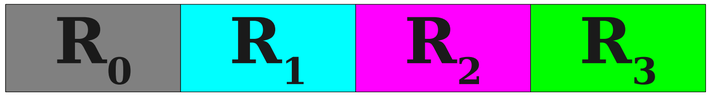
Free Registers

R_0	R_1	R_2	R_3

Linear-Scan Register Allocation



Free Registers



Linear-Scan Register Allocation

Simple, *greedy* register-allocation strategy:

1. Compute liveness information: `live(x)`
   - recall: `live(x)` is the set of uids that are live on entry to `x`'s definition
2. Let `regs` be the set of usable registers
   - usually reserve a couple for spill code (offloading to stack) [our implementation uses rax,rcx]
3. Maintain "layout" `uid_loc` that maps uids to locations
   - locations include registers and stack slots n, starting at n=0
4. Scan through the program. For each instruction that defines a uid `x`
   - `used = {r | reg r = uid_loc(y)` s.t. `y ∈ live(x)}`
   - `available = regs - used`
   - If `available` is empty: *// no registers available, spill*
     `uid_loc(x) := slot n ; n = n + 1`
   - Otherwise, pick `r` in `available`: *// choose an available register*
     `uid_loc(x) := reg r`

Linear-Scan Register Allocation

- Advantages
  - Very efficient (after computing live intervals, runs in linear time)
  - Produces good code in many instances.
  - Allocation step works in one pass; can generate code during iteration.
  - Often used in JIT compilers like Java HotSpot.

- Pitfalls
  - Doesn't always choose a very good strategy due to *greediness*.
  - Doesn't work well with branching.

Linear Scan and Branching

```
a = d = c = 0
b = 1
if (?) {
  a = b + 1
  c = a + b
  print(a)
} else {
  d = b + 2
  c = d + b
  print(d)
}
return c
```

```
a = d = c = 0
// live = {  }
b = 1
```

// live = { b }	// live = { b }
`a = b + 1`	`d = b + 2`
// live = { a, b }	// live = { d, b }
`c = a + b`	`c = d + b`
// live = { a, c }	// live = { d, c }
`print(a)`	`print(d)`
// live = { c }	// live = { c }

```
return c
```

Let’s have a short break

Graph Colouring

Register Allocation via Colour Graphs

Basic process:

1. Compute liveness information for each temporary (%uid).

2. Create an *interference graph*:
   - Nodes are temporary variables (%uids).
   - There is an *edge* between node **n** and **m** if **n** is *live* at the same time as **m**

3. Try to colour the graph
   - Each colour corresponds to a register

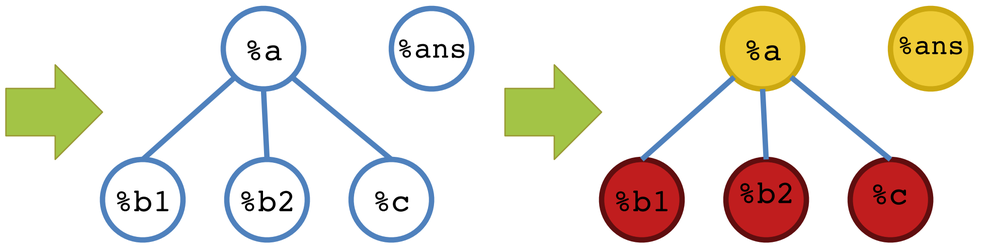
4. In case **Step 3** fails, "spill" a temporary to the stack and repeat the whole process.

5. Rewrite the program to use registers

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables *interfere* if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph colouring*.
  - A graph colouring assigns each node in the graph a colour (register)
  - Any two nodes connected by an edge must have different colours.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



Interference Graph

2-Colouring of the graph:
red = r8
yellow = r9

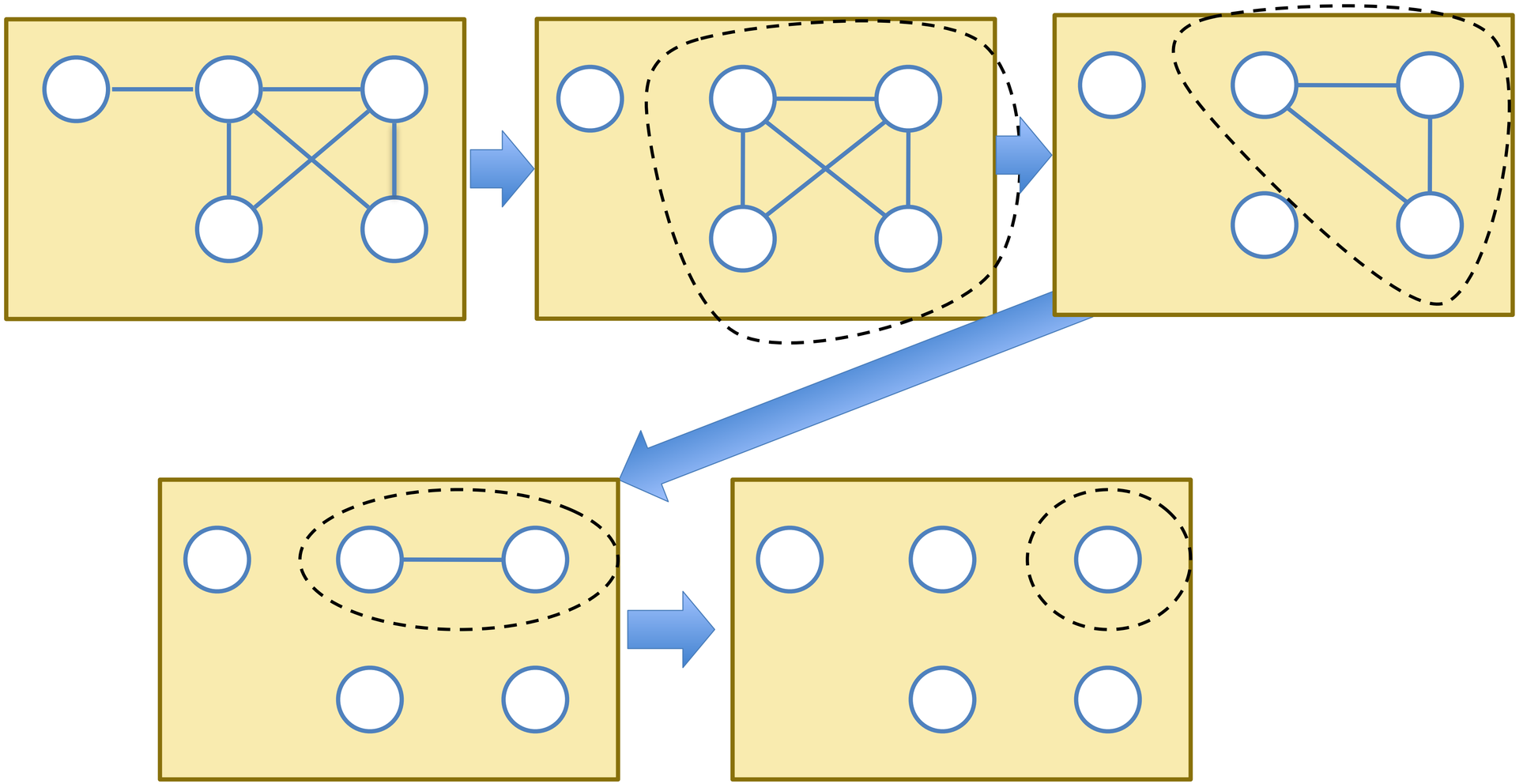
Register Allocation Questions

- Can we efficiently find a k-*colouring* of the graph whenever possible?
  - Answer: in general the problem is NP-complete (it requires search)
  - But, we can do an efficient approximation using *heuristics*.
- How do we assign registers to colours?
  - If we do this in a smart way, we can eliminate many redundant MOV instructions.
- What do we do when there aren't enough colours/registers?
  - We have to use stack space, but how do we do this effectively?

Colouring a Graph: Kempe's Algorithm

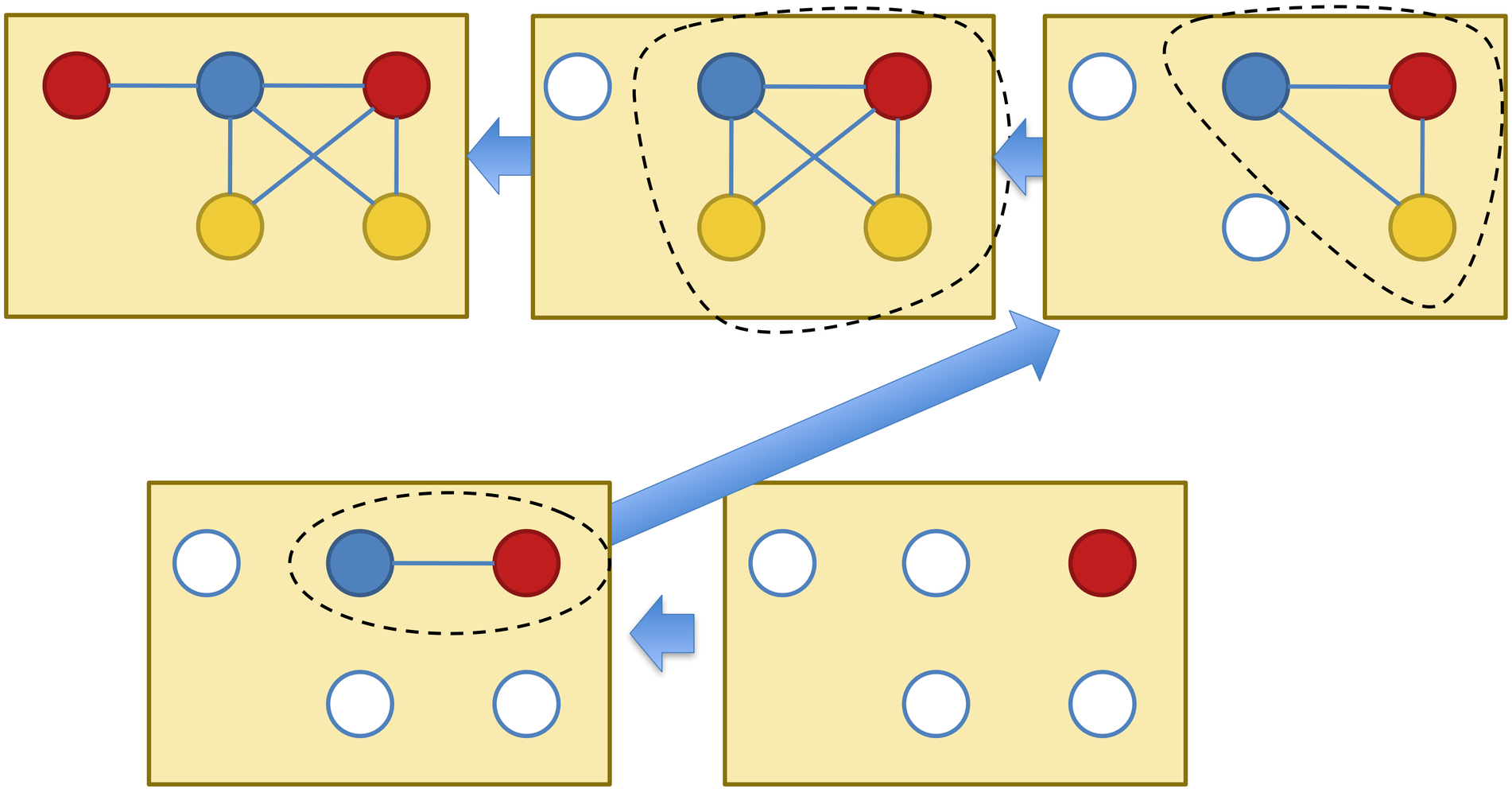
- Kempe [1879] provides this algorithm for K-coloring a graph.
- It's a recursive algorithm that works in three steps:
- Step 1: Find a node with degree < K and cut it out of the graph.
  - Remove the nodes and edges.
  - This is called *simplifying* the graph
- Step 2: Recursively K-colour the remaining subgraph
- Step 3: When remaining graph is coloured, there must be at least one free colour available for the deleted node (since its degree was < K). Pick such a colour.

Example: 3-colour this Graph



Recursing Down the Simplified Graphs

Example: 3-colour this Graph

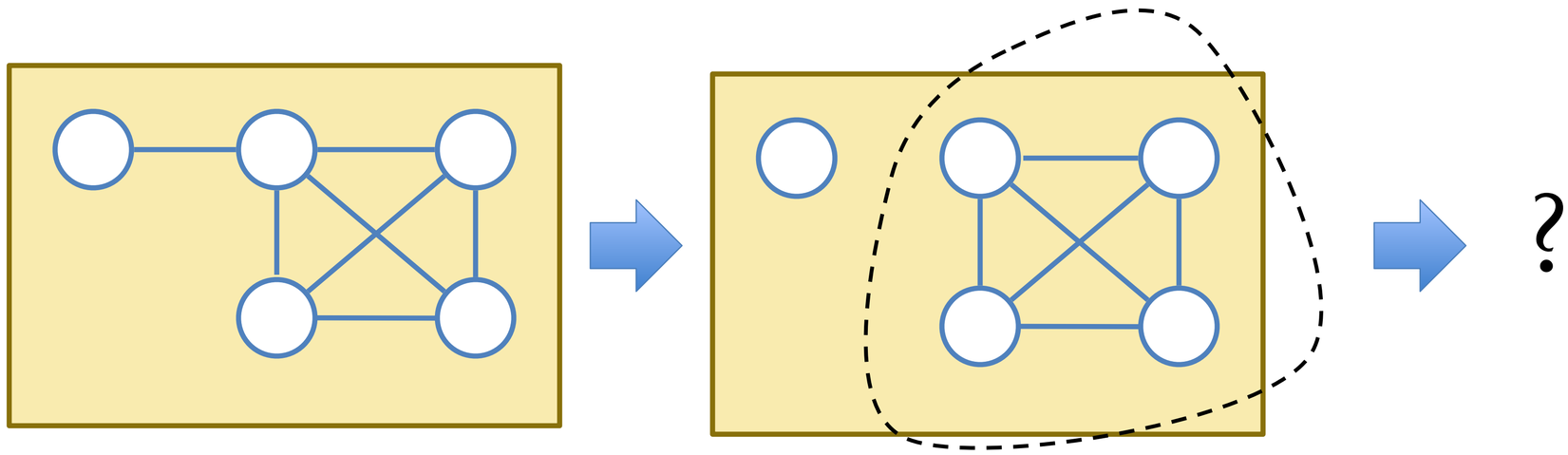


Assigning colours on the way back up.

Failure of the Algorithm

- If the graph cannot be coloured, it will simplify to a graph where every node has at least K neighbours.
  - This can happen even when the graph is K-colourable!
  - This is a symptom of NP-hardness (it requires search)

- Example: When trying to 3-colour this graph:



Spilling

- Idea: If we can't K-colour the graph, we need to store one temporary variable on the stack.

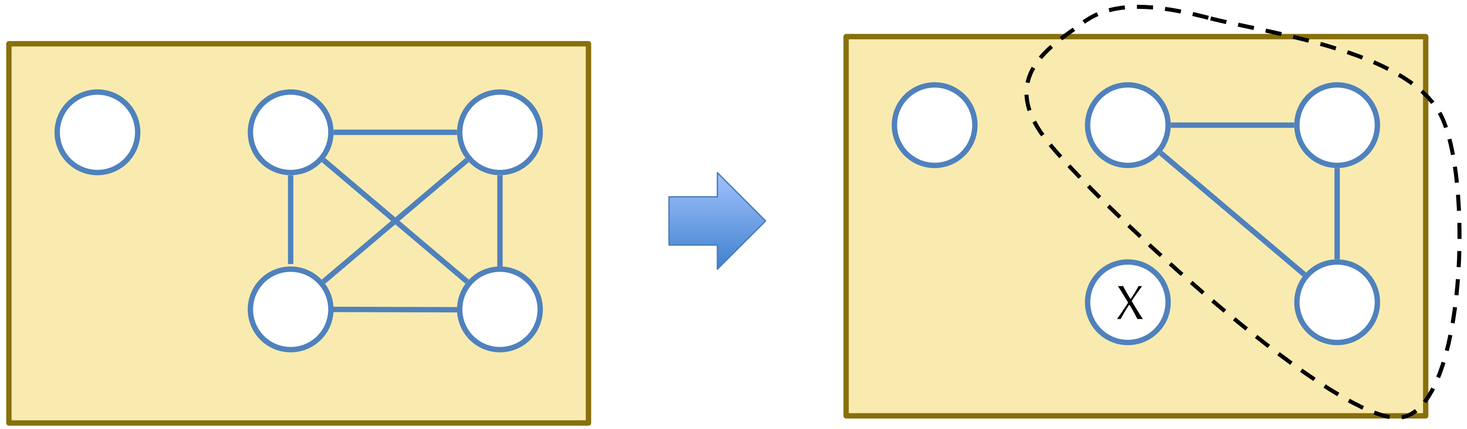
- Which variable to spill?
  - Pick one that isn't used very frequently
  - Pick one that isn't used in a (deeply nested) loop
  - Pick one that has high interference (since removing it will make the graph easier to colour)

- In practice: some weighted combination of these criteria

- When colouring:
  - Mark the node as spilled
  - Remove it from the graph
  - Keep recursively colouring

Spilling, Pictorially

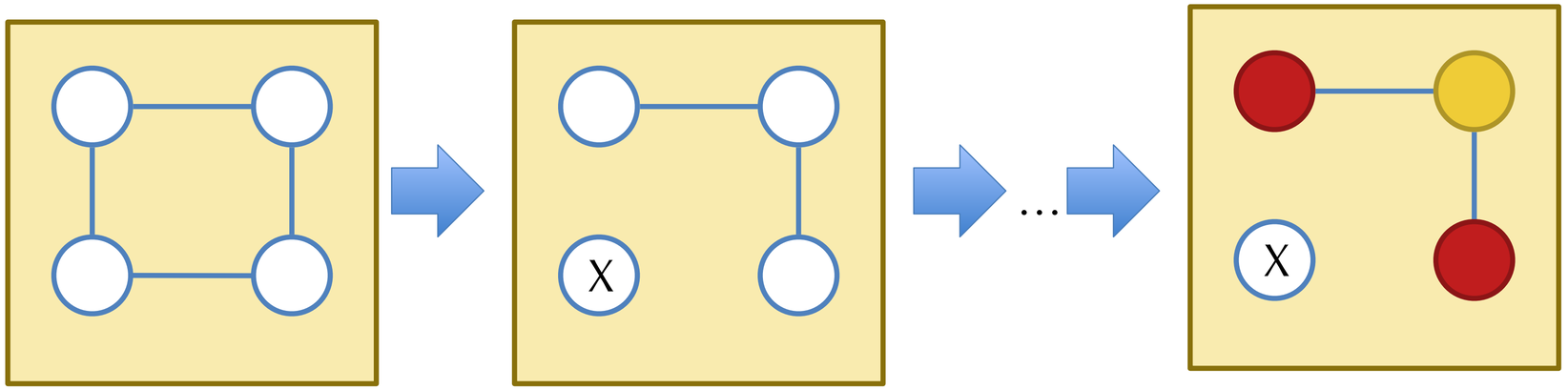
- Select a node to spill
- Mark it and remove it from the graph
- Continue colouring



Optimistic Colouring

- Sometimes it is possible to colour a node marked for spilling.
  - If we get “lucky” with the choices of colours made earlier.

- Example: When 2-colouring this graph, we don’t have a node with degree < 2



- Even though the node was marked for spilling, we can colour it.

- So: on the way down, mark for spilling, but don’t actually spill…

Precoloured Nodes

- Some variables must be pre-assigned to registers.
  - E.g. on X86 the multiplication instruction: IMul must define %rax
  - The “Call” instruction should kill the caller-save registers %rax, %rcx, %rdx.
  - Any temporary variable live across a call interferes with the caller-save registers.

- To properly allocate temporaries, we treat registers as nodes in the interference graph with pre-assigned colours.
  - Pre-coloured nodes can’t be removed during simplification.
  - Trick: Treat pre-coloured nodes as having “infinite” degree in the interference graph – this guarantees they won’t be simplified.
  - When the graph is empty except the pre-coloured nodes, we have reached the point where we start colouring the rest of the nodes.

Picking Good Colours

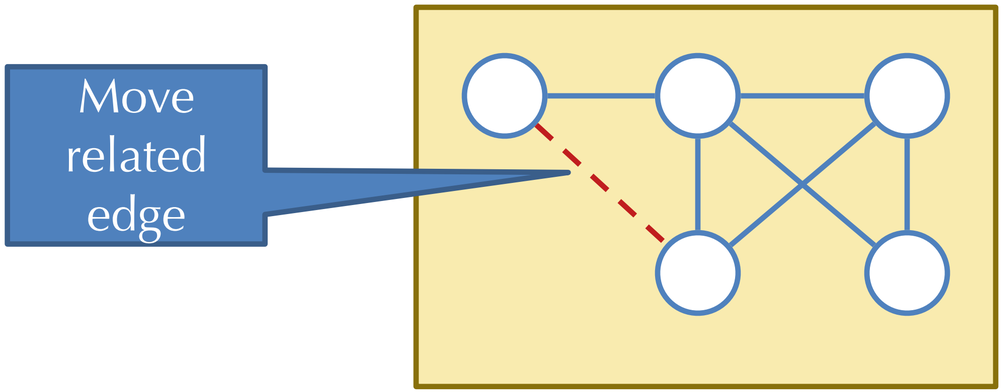
- When choosing colours during the colouring phase, *any* choice is semantically correct, but some choices are better for performance.

- Example:
  `%t1 = %t2`
  - If t1 and t2 can be assigned the same register (colour) then this move is redundant and can be eliminated.

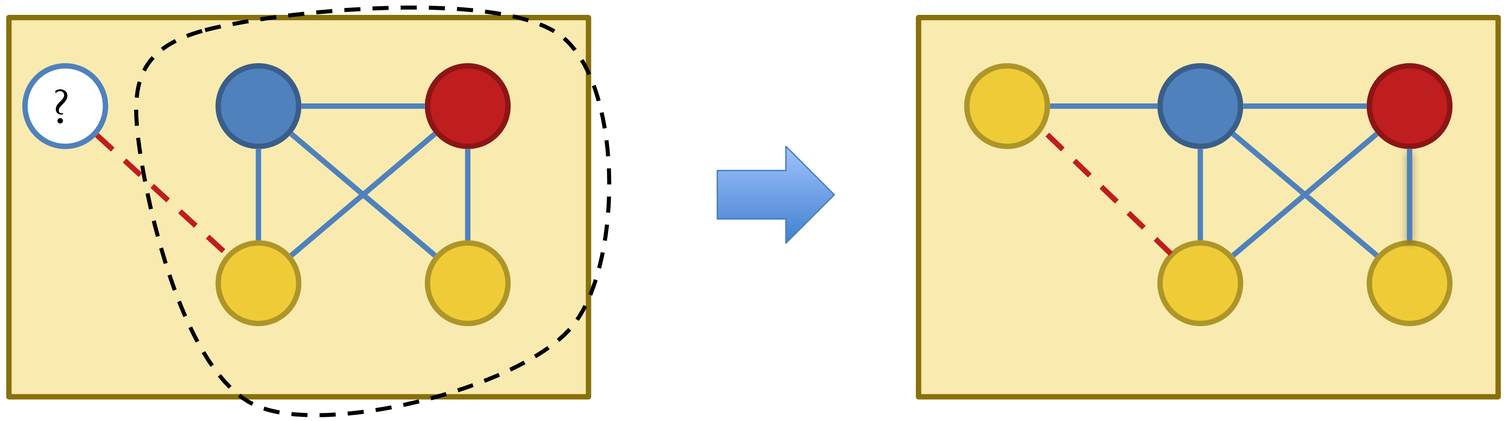
- A simple colour choosing strategy that helps eliminate such moves:
  - Add a new kind of “move related” edge between the nodes for t1 and t2 in the interference graph.
  - When choosing a colour for t1 (or t2), if possible pick a colour of an already coloured node reachable by a move-related edge.

Example Colour Choice

- Consider 3-colouring this graph, where the dashed edge indicates that there is a Mov from one temporary to another.

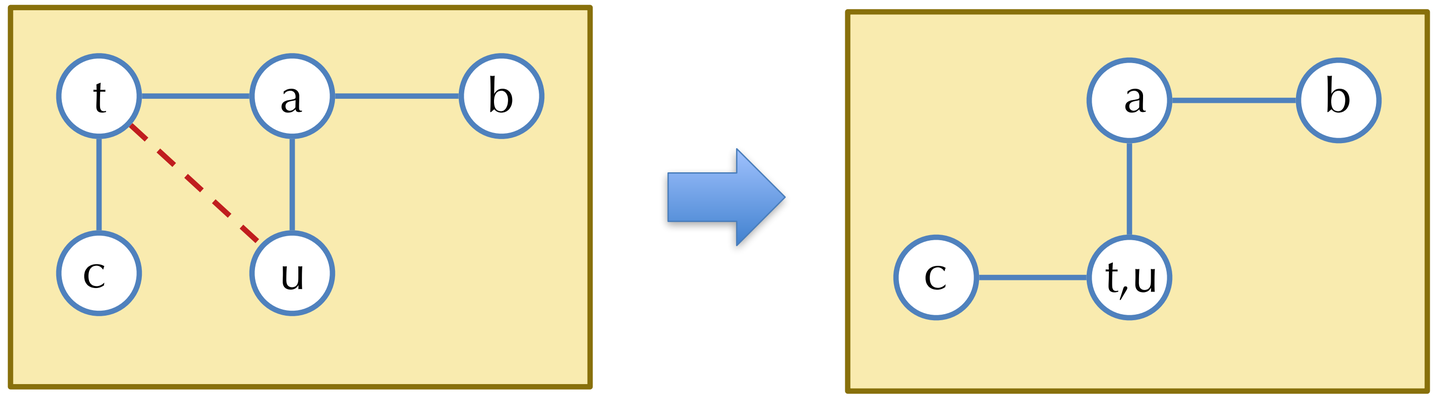


- After colouring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.

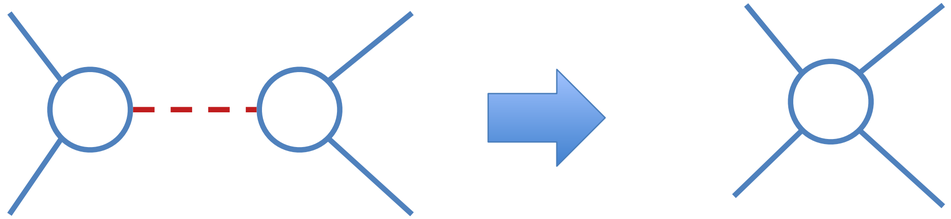


Coalescing Interference Graphs

- A more aggressive strategy is to *coalesce* nodes of the interference graph if they are connected by move-related edges.
  - Coalescing the nodes *forces* the two temporaries to be assigned the same register.



- Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated.
- Problem: coalescing can sometimes increase the degree of a node.



Conservative Coalescing

- Two strategies are guaranteed to preserve the k-colorability of the interference graph.
- *Brigg's strategy*: It's safe to coalesce **x & y** if the resulting node will have fewer than k neighbours (with degree ≥ k).
- *George's strategy:* We can safely coalesce **x & y** if for every neighbour **t** of **x**, either **t** already interferes with **y** or t has degree **< k**.

Complete Register Allocation Algorithm

1. Build interference graph (pre-colour nodes as necessary).
   - Add move related edges

2. Reduce the graph (building a stack of nodes to color).
   a. Simplify the graph as much as possible without removing nodes that are move-related (i.e. have a move-related neighbour). Remaining nodes are high degree or move-related.
   b. Coalesce move-related nodes using Brigg's or George's strategy.
   c. Coalescing can reveal more nodes that can be simplified, so repeat 2.a and 2.b until no node can be simplified or coalesced.
   d. If no nodes can be coalesced *freeze* (remove) a move-related edge and keep trying to simplify/coalesce.

3. If there are non-precoloured nodes left, mark one for spilling, remove it from the graph and continue doing step 2.

4. When only pre-coloured node remain, start colouring (popping simplified nodes off the top of the stack).
   a. If a node must be spilled, insert spill code as on slide "*Example Spill Code*" and rerun the whole register allocation algorithm, starting at step 1.

Was it worth it?

Demo: Register allocation in HW6

For HW6

- HW 6 implements two naive register allocation strategies:
- `no_reg_layout`: spill all registers to the stack
- `greedy_layout`: puts the first few uids in available registers and spills the rest. It uses liveness information to recycle available registers when their current value becomes dead (see the slides above).
- Your job: do “better” than these via graph colouring.
- Quality Metric:
  - the total number of memory accesses in x86 program, which is the sum of:
    - the number of Ind2 and Ind3 operands
    - the number of Push and Pop instructions
  - shorter code is better

Another break?

Current Research in PLDI

Validating Compilers

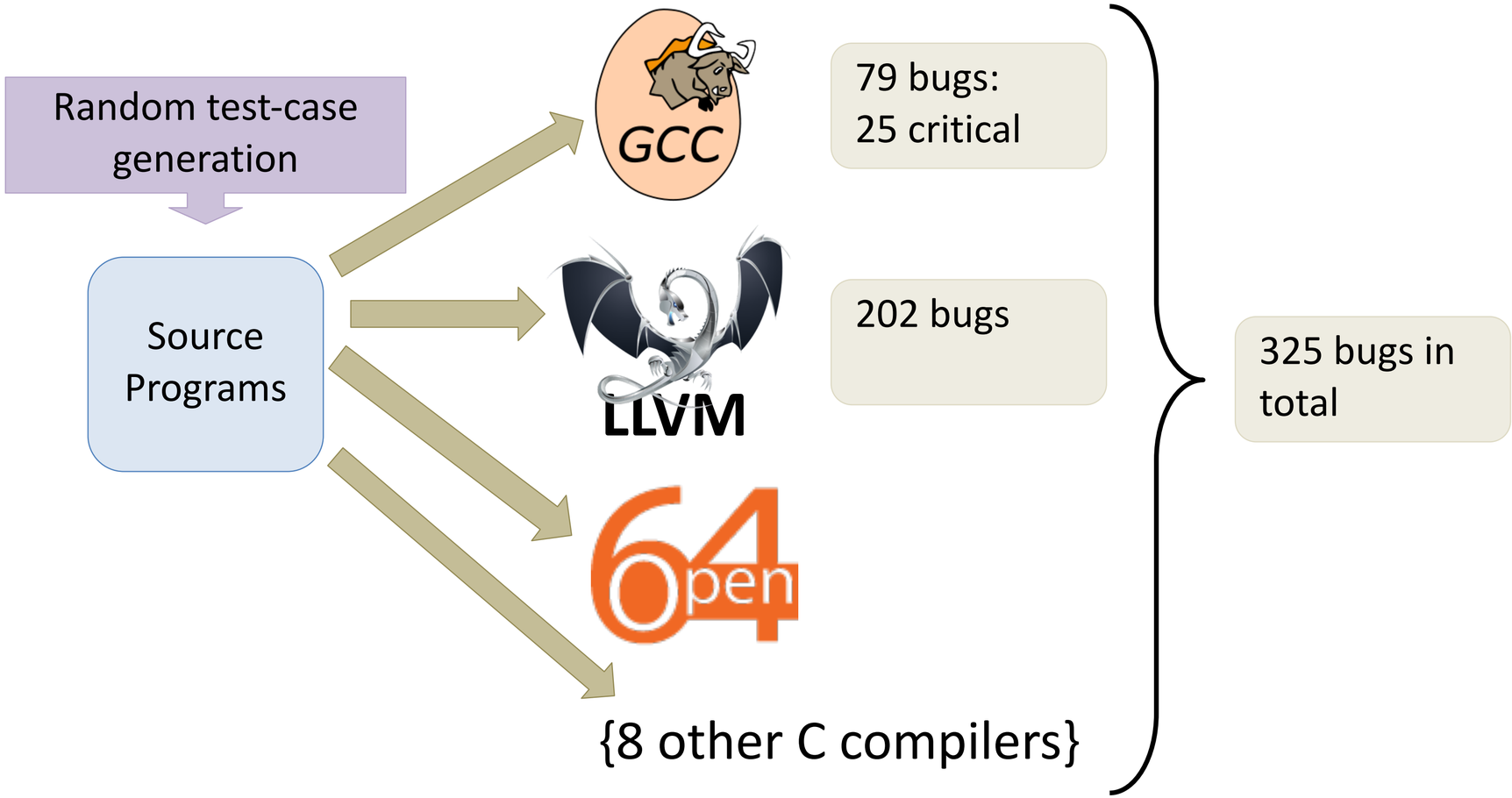
- The job of a compiler is to *translate from the syntax of one language to another, but preserve the semantics*.

- Compiler correctness is critical
  - Trustworthiness of every component built in a compiled language depends on trustworthiness of the compiler

- Compilers tend to be well-engineered and well-tested,
  but that does not mean they are *bug-free*.

Testing Compilers

Finding and Understanding Bugs in C Compilers.Yang et al. PLDI 2011



Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

(in PLDI 2011)

Compilers should be correct.

To improve the quality of C compilers, we created Csmith, a **randomized test-case generation tool**, and spent **three years** using it to find compiler bugs.

During this period we reported **more than 325 previously unknown bugs** to compiler developers.

The striking thing about our **CompCert** results is that the middle-end bugs we found in all other compilers are **absent**.

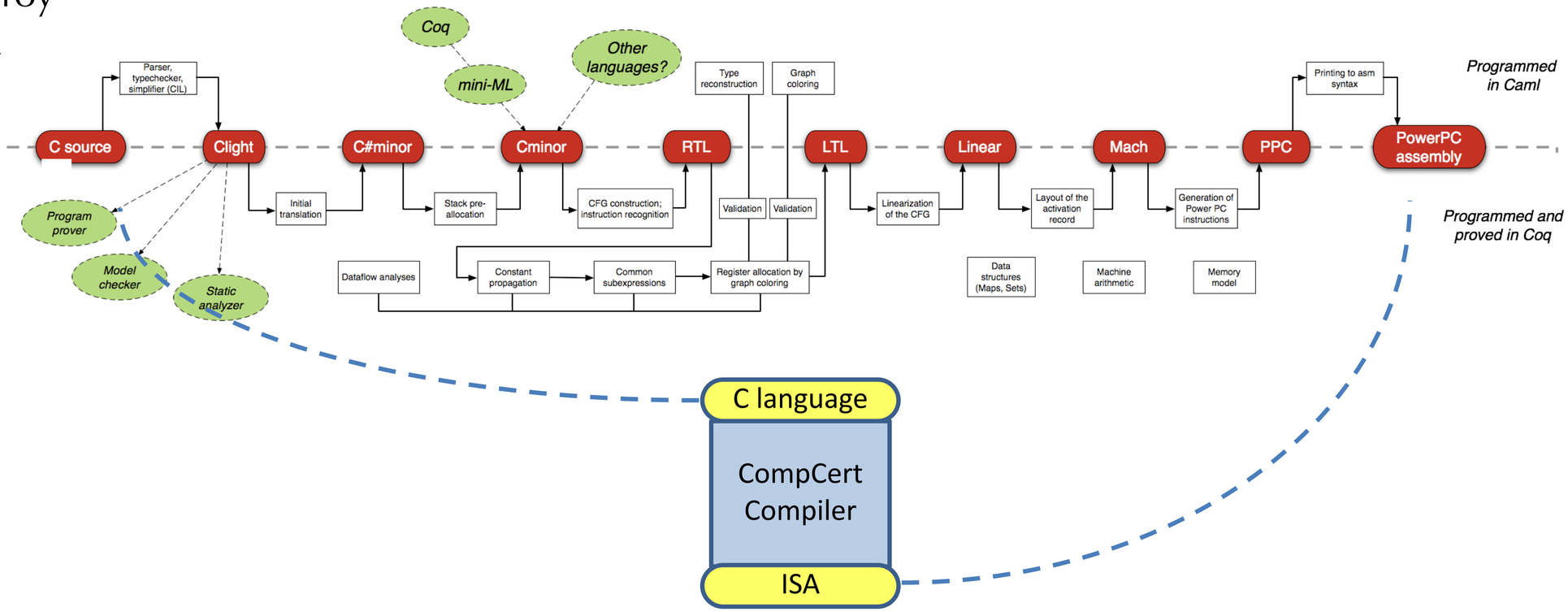
As of early 2011, the under-development version of **CompCert** is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.

The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

Verified Compilation

Xavier Leroy
INRIA

CompCert (2006-now):
Optimising C Compiler,
proved correct *end-to-end*
with machine-checked proof in Coq



Comparing Behaviours

- Consider two programs P1 and P2 possibly in different languages.
  - e.g. P1 is an Oat program, P2 is its compilation to LL

- The semantics of the languages associate to each program a set of observable behaviours:

$$\mathbf{B}(P1) \text{ and } \mathbf{B}(P2)$$

- Note: $|\mathbf{B}(P)| = 1$ if P is deterministic, > 1 otherwise

What is Observable?

- For C-like languages:

  observable behavior ::=
  | terminates(st) (i.e. observe the final state)
  | diverges
  | goeswrong

- For pure functional languages:

  observable behavior ::=
  | terminates(v) (i.e. observe the final value)
  | diverges
  | goeswrong

What about I/O?

- Add a *trace* of input-output events performed:

```
          t    ::=  []  |  e :: t          (finite traces)
coind.    T    ::=  []  |  e :: T          (finite and infinite traces)
```

```
observable behavior ::=
          | terminates(t, st)     (end in state st after trace t)
          | diverges(T)            (loop, producing trace T)
          | goeswrong(t)
```

Examples

- P1:
  print(1); / st ⇒ terminates(out(1)::[],st)

- P2:
  print(1); print(2); / st
  ⇒ terminates(out(1)::out(2)::[],st)

- P3:
  WHILE true DO print(1) END / st
  ⇒ diverges(out(1)::out(1)::…)

- So **B**(P1) ≠ **B**(P2) ≠ **B**(P3)

Bisimulation

- Two programs P1 and P2 are *bisimilar* whenever:

$$\mathbf{B}(\text{P1}) \;=\; \mathbf{B}(\text{P2})$$

- The two programs are completely indistinguishable.

- But… this is often too strong in practice.

Compilation Reduces Nondeterminism

- Some languages (like C) have underspecified behaviours:
  - Example: order of evaluation of expressions f() + g()

- Concurrent programs often permit nondeterminism
  - Classic optimizations can reduce this nondeterminism
  - Example:

    a := x + 1; b := x + 1 || x := x+1

    vs.

    a := x + 1; b := a || x := x+1

- LLVM explicitly allows nondeterminism:
  - undef values (not part of LLVM lite)

Backward Simulation

- Program P2 can exhibit fewer behaviours than P1:

$$\mathbf{B}(\text{P1}) \supseteq \mathbf{B}(\text{P2})$$

- All of the behaviours of P2 are permitted by P1, though some of them may have been eliminated.

- Also called *refinement*.

Related Research Topics

Automated Parallelisation

- Moore's law: processor advances double speed every 18 months
- Moore's law ended in 2006 for single-threaded applications
- Started to hit fundamental limits in how small transistors can be
- Processor manufacturers shifted to multi-core processors
- Need new compiler technology to take advantage of multi-core – automatically find and exploit opportunities for parallel execution

Program Analysis

- The goal of a program analysis is to *answer questions about the run-time behaviour of software*

- In compilers: *data flow analysis*, *control flow analysis*
  - Typical goal: determine whether an optimisation is safe

- Research in program analysis has shifted to more sophisticated properties:
  - *Numerical analyses*, e.g., find geometric regions that contain reachable values for integer variables.
    Can be used to verify absence of buffer overflows.
  - *Shape analyses* – determine whether a data structure in the heap is a list, a tree, a graph,...
    Can be used to verify memory safety.
  - *Resource analyses* – e.g., find a conservative upper bound on the run-time complexity of a loop.
    Can be used to find timing side-channel attacks.
  - *Concurrency analysis*: find all data races in a multi-threaded program.

- Industrial program analysis:
  - Static Driver Verifier (Microsoft): finds bugs in device driver code
  - Infer (Facebook): proves memory safety & finds race conditions
  - Astrée (AbsInt): static analyser for safety-critcal embedded code (e.g.,automotive& aerospace applications)

Program Verification and Synthesis

- *Verification:* Given a program and a specification, prove that the program satisfies the specification

- *Synthesis:* Given a specification, find a program that satisfies the specification
  - Kind of a "compilation on steroids" from language of specifications to a programming language

```
void swap(loc x, loc y)
```

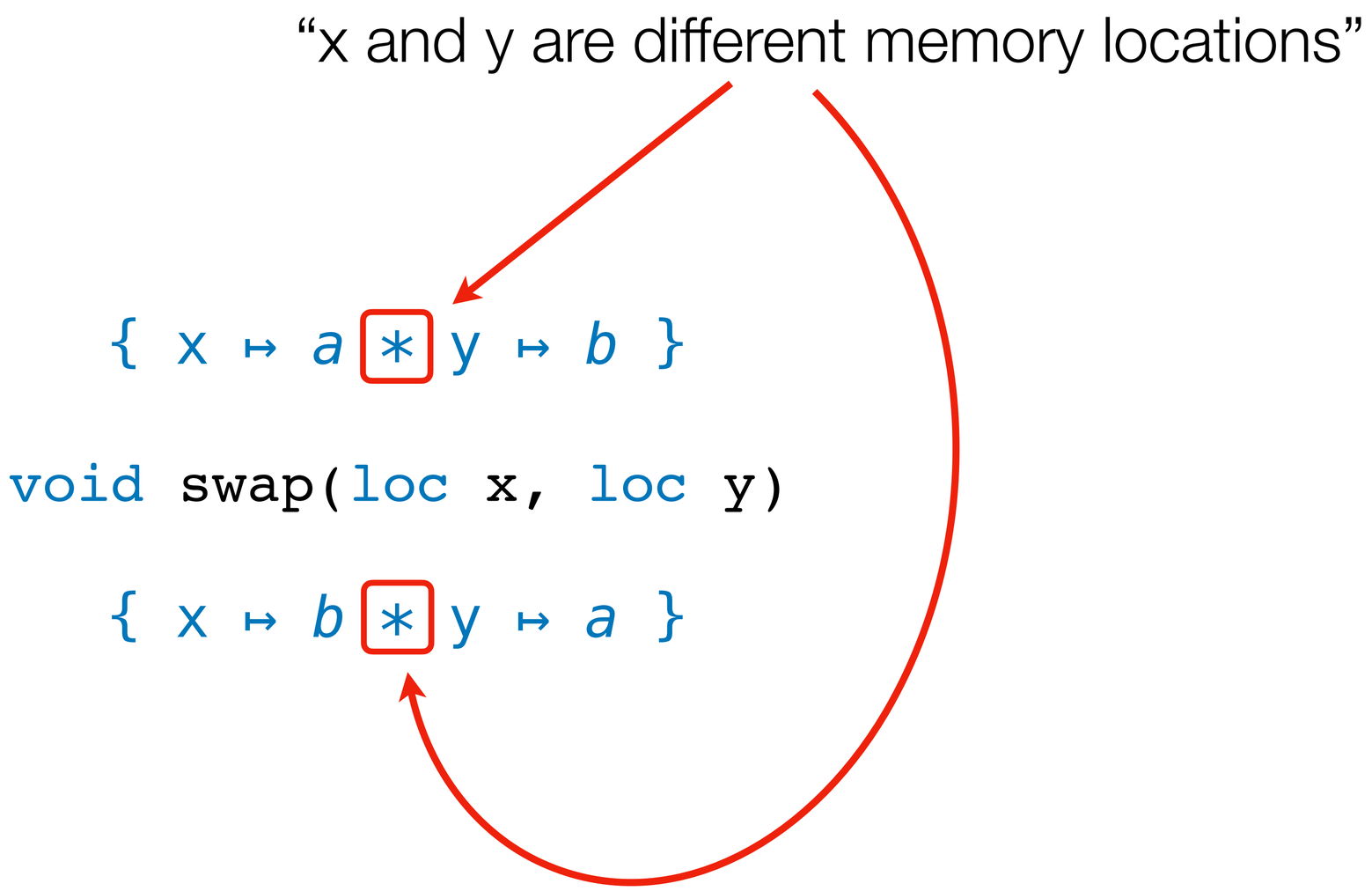
$$\{\ x \mapsto a \wedge y \mapsto b\ \}$$

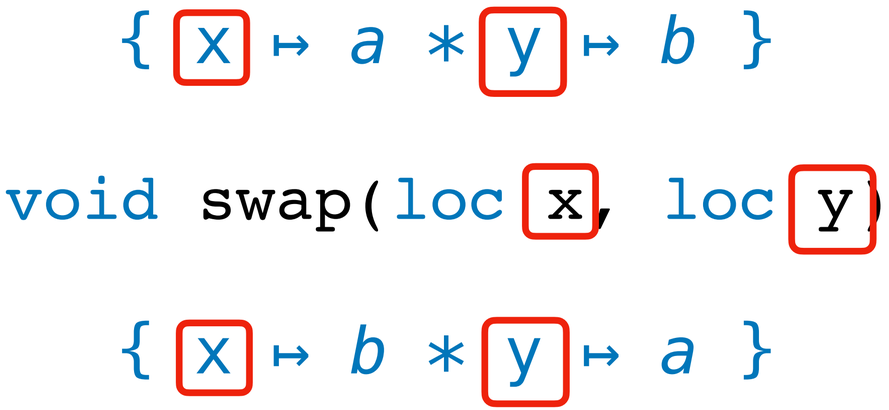
```
void swap(loc x, loc y)
```

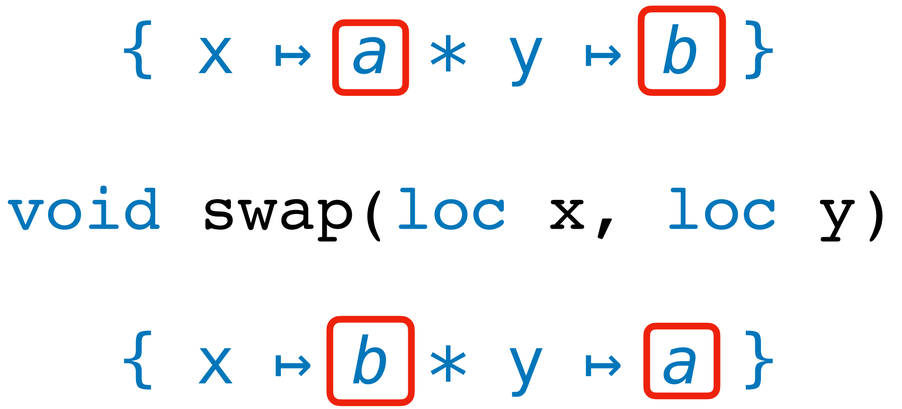
$$\{\ x \mapsto a \wedge y \mapsto b\ \}$$

```
void swap(loc x, loc y)
```

$$\{\ x \mapsto b \wedge y \mapsto a\ \}$$


“x and y are different memory locations”
{ x ↦ a * y ↦ b }
void swap(loc x, loc y)
{ x ↦ b * y ↦ a }


{ x ↦ a * y ↦ b }
void swap(loc x, loc y)
{ x ↦ b * y ↦ a }


{ x ↦ a * y ↦ b }
void swap(loc x, loc y)
{ x ↦ b * y ↦ a }

$$\{\ x \mapsto \boxed{a} * y \mapsto b\ \}$$

??

$$\{\ x \mapsto b * y \mapsto \boxed{a}\ \}$$

```
let a2 = *x;
```

$$\{\ x \mapsto a2 * y \mapsto \boxed{b}\ \}$$

??

$$\{\ x \mapsto \boxed{b} * y \mapsto a2\ \}$$

```
let a2 = *x;
let b2 = *y;
{ x ↦ a2 ∗ y ↦ b2 }
??
{ x ↦ b2 ∗ y ↦ a2 }
```

```
let a2 = *x;
let b2 = *y;
*x = b2;
```

$$\{\ x \mapsto b2 * y \mapsto b2\ \}$$

??

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

```
let a2 = *x;
let b2 = *y;
*x = b2;
*y = a2;
```

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

??

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

```
let a2 = *x;
let b2 = *y;
*x = b2;
*y = a2;
```

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

??

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

$$x \mapsto b2 * y \mapsto a2 \ \vdash\ x \mapsto b2 * y \mapsto a2$$

```
let a2 = *x;
let b2 = *y;
*x = b2;
*y = a2;
```

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

??

$$\{\ x \mapsto b2 * y \mapsto a2\ \}$$

$$x \mapsto b2 * y \mapsto a2 \ \vdash\ x \mapsto b2 * y \mapsto a2$$

```
void swap(loc x, loc y) {
        let a2 = *x;
        let b2 = *y;
        *x = b2;
        *y = a2;
}
```

Transforming Entailment

$$P \rightsquigarrow Q$$

There *exists* a program **c**, such that
for any initial state satisfying **P**,
c, after it terminates,
will transform to a state satisfying **Q**.

$x \mapsto a \quad \twoheadrightarrow \quad x \mapsto 42$

"Proof": `*x = 42`

$$\Gamma; \{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid \texttt{skip} \qquad \text{(Emp)}$$

$$\frac{a \in \mathrm{GV}(\Gamma, P, Q) \qquad y \text{ is fresh} \qquad \Gamma, y ; [y/a]\{\, x \mapsto y * P \,\} \rightsquigarrow [y/a]\{\, Q \,\} \mid \texttt{c}}{\Gamma; \{\, x \mapsto a * P \,\} \rightsquigarrow \{\, Q \,\} \mid \texttt{let y = *x; c}} \text{(Read)}$$

$$\frac{\mathrm{EV}(\Gamma, P, Q) \cap \mathit{Vars}(R) = \varnothing \qquad \Gamma ; \{\, P \,\} \rightsquigarrow \{\, Q \,\} \mid \texttt{c}}{\Gamma; \{\, P * R \,\} \rightsquigarrow \{\, Q * R \,\} \mid \texttt{c}} \text{(Frame)}$$

$$\frac{\mathit{Vars}(e) \subseteq \Gamma \qquad \Gamma ; \{\, x \mapsto e * P \,\} \rightsquigarrow \{\, x \mapsto e * Q \,\} \mid \texttt{c}}{\Gamma; \{\, x \mapsto - * P \,\} \rightsquigarrow \{\, x \mapsto e * Q \,\} \mid \texttt{*x = e; c}} \text{(Write)}$$

$$\{ x \mapsto a * y \mapsto b \}$$

```
void swap(loc x, loc y)
```

$$\{ x \mapsto b * y \mapsto a \}$$

$$\{ x, y \} ; \{ x \mapsto a * y \mapsto b \} \twoheadrightarrow \{ x \mapsto b * y \mapsto a \} \quad | \quad \texttt{??}$$

$$\frac{\{\, x, y, a2 \,\} ; \{\, x \mapsto a2 * y \mapsto b \,\} \twoheadrightarrow \{\, x \mapsto b * y \mapsto a2 \,\} \mid \texttt{??}}{\{\, x, y \,\} ; \{\, x \mapsto a * y \mapsto b \,\} \twoheadrightarrow \{\, x \mapsto b * y \mapsto a \,\} \mid \texttt{let a2 = *x; ??}} \text{ (Read)}$$

$$\frac{\{\, x, y, a2, b2 \,\} ;\ \{\, x \mapsto a2 * y \mapsto b2 \,\} \ \twoheadrightarrow\ \{\, x \mapsto b2 * y \mapsto a2 \,\} \ \mid\ \texttt{??}}{\{\, x, y, a2 \,\} ;\ \{\, x \mapsto a2 * y \mapsto b \,\} \ \twoheadrightarrow\ \{\, x \mapsto b * y \mapsto a2 \,\} \ \mid\ \texttt{let b2 = *y; ??}}\ \text{(Read)}$$

$$\frac{\{\, x, y, a2 \,\} ;\ \{\, x \mapsto a2 * y \mapsto b \,\} \ \twoheadrightarrow\ \{\, x \mapsto b * y \mapsto a2 \,\} \ \mid\ \texttt{let b2 = *y; ??}}{\{\, x, y \,\} ;\ \{\, x \mapsto a * y \mapsto b \,\} \ \twoheadrightarrow\ \{\, x \mapsto b * y \mapsto a \,\} \ \mid\ \texttt{let a2 = *x; ??}}\ \text{(Read)}$$

$$\frac{\{\, x, y, a2, b2 \,\} ; \{\, x \mapsto b2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{??}}{\{\, x, y, a2, b2 \,\} ; \{\, x \mapsto a2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{*x = b2; ??}}\ \text{(Write)}$$

$$\frac{\{\, x, y, a2, b2 \,\} ; \{\, x \mapsto a2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{*x = b2; ??}}{\{\, x, y, a2 \,\} ; \{\, x \mapsto a2 * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a2 \,\} \mid \texttt{let b2 = *y; ??}}\ \text{(Read)}$$

$$\frac{\{\, x, y, a2 \,\} ; \{\, x \mapsto a2 * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a2 \,\} \mid \texttt{let b2 = *y; ??}}{\{\, x, y \,\} ; \{\, x \mapsto a * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a \,\} \mid \texttt{let a2 = *x; ??}}\ \text{(Read)}$$

$$\{\, x, y, a2, b2 \,\} ;\ \{\, y \mapsto b2 \,\} \twoheadrightarrow \{\, y \mapsto a2 \,\} \mid \texttt{??}$$

(Frame)

$$\{\, x, y, a2, b2 \,\} ;\ \{\, x \mapsto b2 * y \mapsto b2 \,\} \twoheadrightarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{??}$$

(Write)

$$\{\, x, y, a2, b2 \,\} ;\ \{\, x \mapsto a2 * y \mapsto b2 \,\} \twoheadrightarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{*x = b2; ??}$$

(Read)

$$\{\, x, y, a2 \,\} ;\ \{\, x \mapsto a2 * y \mapsto b \,\} \twoheadrightarrow \{\, x \mapsto b * y \mapsto a2 \,\} \mid \texttt{let b2 = *y; ??}$$

(Read)

$$\{\, x, y \,\} ;\ \{\, x \mapsto a * y \mapsto b \,\} \twoheadrightarrow \{\, x \mapsto b * y \mapsto a \,\} \mid \texttt{let a2 = *x; ??}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{\,x, y, a2, b2\,\};\ \{\,y \mapsto a2\,\} \rightsquigarrow \{\,y \mapsto a2\,\} \mid \texttt{??}}{\{\,x, y, a2, b2\,\};\ \{\,y \mapsto b2\,\} \rightsquigarrow \{\,y \mapsto a2\,\} \mid \texttt{*y = a2; ??}}\ \text{(Write)}}{\{\,x, y, a2, b2\,\};\ \{\,x \mapsto b2 * y \mapsto b2\,\} \rightsquigarrow \{\,x \mapsto b2 * y \mapsto a2\,\} \mid \texttt{??}}\ \text{(Frame)}}{\{\,x, y, a2, b2\,\};\ \{\,x \mapsto a2 * y \mapsto b2\,\} \rightsquigarrow \{\,x \mapsto b2 * y \mapsto a2\,\} \mid \texttt{*x = b2; ??}}\ \text{(Write)}}{\{\,x, y, a2\,\};\ \{\,x \mapsto a2 * y \mapsto b\,\} \rightsquigarrow \{\,x \mapsto b * y \mapsto a2\,\} \mid \texttt{let b2 = *y; ??}}\ \text{(Read)}}{\{\,x, y\,\};\ \{\,x \mapsto a * y \mapsto b\,\} \rightsquigarrow \{\,x \mapsto b * y \mapsto a\,\} \mid \texttt{let a2 = *x; ??}}\ \text{(Read)}$$

$$\frac{\{\ x, y, a2, b2\ \};\ \{\ \mathsf{emp}\ \}\ \rightsquigarrow\ \{\ \mathsf{emp}\ \}\ \mid\ \texttt{??}}{\{\ x, y, a2, b2\ \};\ \{\ y \mapsto a2\ \}\ \rightsquigarrow\ \{\ y \mapsto a2\ \}\ \mid\ \texttt{??}}\ \text{(Frame)}$$

$$\frac{\{\ x, y, a2, b2\ \};\ \{\ y \mapsto a2\ \}\ \rightsquigarrow\ \{\ y \mapsto a2\ \}\ \mid\ \texttt{??}}{\{\ x, y, a2, b2\ \};\ \{\ y \mapsto b2\ \}\ \rightsquigarrow\ \{\ y \mapsto a2\ \}\ \mid\ \texttt{*y = a2; ??}}\ \text{(Write)}$$

$$\frac{\{\ x, y, a2, b2\ \};\ \{\ y \mapsto b2\ \}\ \rightsquigarrow\ \{\ y \mapsto a2\ \}\ \mid\ \texttt{*y = a2; ??}}{\{\ x, y, a2, b2\ \};\ \{\ x \mapsto b2 * y \mapsto b2\ \}\ \rightsquigarrow\ \{\ x \mapsto b2 * y \mapsto a2\ \}\ \mid\ \texttt{??}}\ \text{(Frame)}$$

$$\frac{\{\ x, y, a2, b2\ \};\ \{\ x \mapsto b2 * y \mapsto b2\ \}\ \rightsquigarrow\ \{\ x \mapsto b2 * y \mapsto a2\ \}\ \mid\ \texttt{??}}{\{\ x, y, a2, b2\ \};\ \{\ x \mapsto a2 * y \mapsto b2\ \}\ \rightsquigarrow\ \{\ x \mapsto b2 * y \mapsto a2\ \}\ \mid\ \texttt{*x = b2; ??}}\ \text{(Write)}$$

$$\frac{\{\ x, y, a2, b2\ \};\ \{\ x \mapsto a2 * y \mapsto b2\ \}\ \rightsquigarrow\ \{\ x \mapsto b2 * y \mapsto a2\ \}\ \mid\ \texttt{*x = b2; ??}}{\{\ x, y, a2\ \};\ \{\ x \mapsto a2 * y \mapsto b\ \}\ \rightsquigarrow\ \{\ x \mapsto b * y \mapsto a2\ \}\ \mid\ \texttt{let b2 = *y; ??}}\ \text{(Read)}$$

$$\frac{\{\ x, y, a2\ \};\ \{\ x \mapsto a2 * y \mapsto b\ \}\ \rightsquigarrow\ \{\ x \mapsto b * y \mapsto a2\ \}\ \mid\ \texttt{let b2 = *y; ??}}{\{\ x, y\ \};\ \{\ x \mapsto a * y \mapsto b\ \}\ \rightsquigarrow\ \{\ x \mapsto b * y \mapsto a\ \}\ \mid\ \texttt{let a2 = *x; ??}}\ \text{(Read)}$$

$$\frac{}{\{\, x, y, a2, b2 \,\};\ \{\, \mathsf{emp} \,\} \rightsquigarrow \{\, \mathsf{emp} \,\} \mid \texttt{skip}}\ (\text{Emp})$$

$$\frac{\{\, x, y, a2, b2 \,\};\ \{\, \mathsf{emp} \,\} \rightsquigarrow \{\, \mathsf{emp} \,\} \mid \texttt{skip}}{\{\, x, y, a2, b2 \,\};\ \{\, y \mapsto a2 \,\} \rightsquigarrow \{\, y \mapsto a2 \,\} \mid \texttt{??}}\ (\text{Frame})$$

$$\frac{\{\, x, y, a2, b2 \,\};\ \{\, y \mapsto a2 \,\} \rightsquigarrow \{\, y \mapsto a2 \,\} \mid \texttt{??}}{\{\, x, y, a2, b2 \,\};\ \{\, y \mapsto b2 \,\} \rightsquigarrow \{\, y \mapsto a2 \,\} \mid \texttt{*y = a2;}\ \texttt{??}}\ (\text{Write})$$

$$\frac{\{\, x, y, a2, b2 \,\};\ \{\, y \mapsto b2 \,\} \rightsquigarrow \{\, y \mapsto a2 \,\} \mid \texttt{*y = a2;}\ \texttt{??}}{\{\, x, y, a2, b2 \,\};\ \{\, x \mapsto b2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{??}}\ (\text{Frame})$$

$$\frac{\{\, x, y, a2, b2 \,\};\ \{\, x \mapsto b2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{??}}{\{\, x, y, a2, b2 \,\};\ \{\, x \mapsto a2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{*x = b2;}\ \texttt{??}}\ (\text{Write})$$

$$\frac{\{\, x, y, a2, b2 \,\};\ \{\, x \mapsto a2 * y \mapsto b2 \,\} \rightsquigarrow \{\, x \mapsto b2 * y \mapsto a2 \,\} \mid \texttt{*x = b2;}\ \texttt{??}}{\{\, x, y, a2 \,\};\ \{\, x \mapsto a2 * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a2 \,\} \mid \texttt{let b2 = *y;}\ \texttt{??}}\ (\text{Read})$$

$$\frac{\{\, x, y, a2 \,\};\ \{\, x \mapsto a2 * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a2 \,\} \mid \texttt{let b2 = *y;}\ \texttt{??}}{\{\, x, y \,\};\ \{\, x \mapsto a * y \mapsto b \,\} \rightsquigarrow \{\, x \mapsto b * y \mapsto a \,\} \mid \texttt{let a2 = *x;}\ \texttt{??}}\ (\text{Read})$$

```
void swap(loc x, loc y) {
        let a2 = *x;
        let b2 = *y;
        *x = b2;
        *y = a2;
}
```

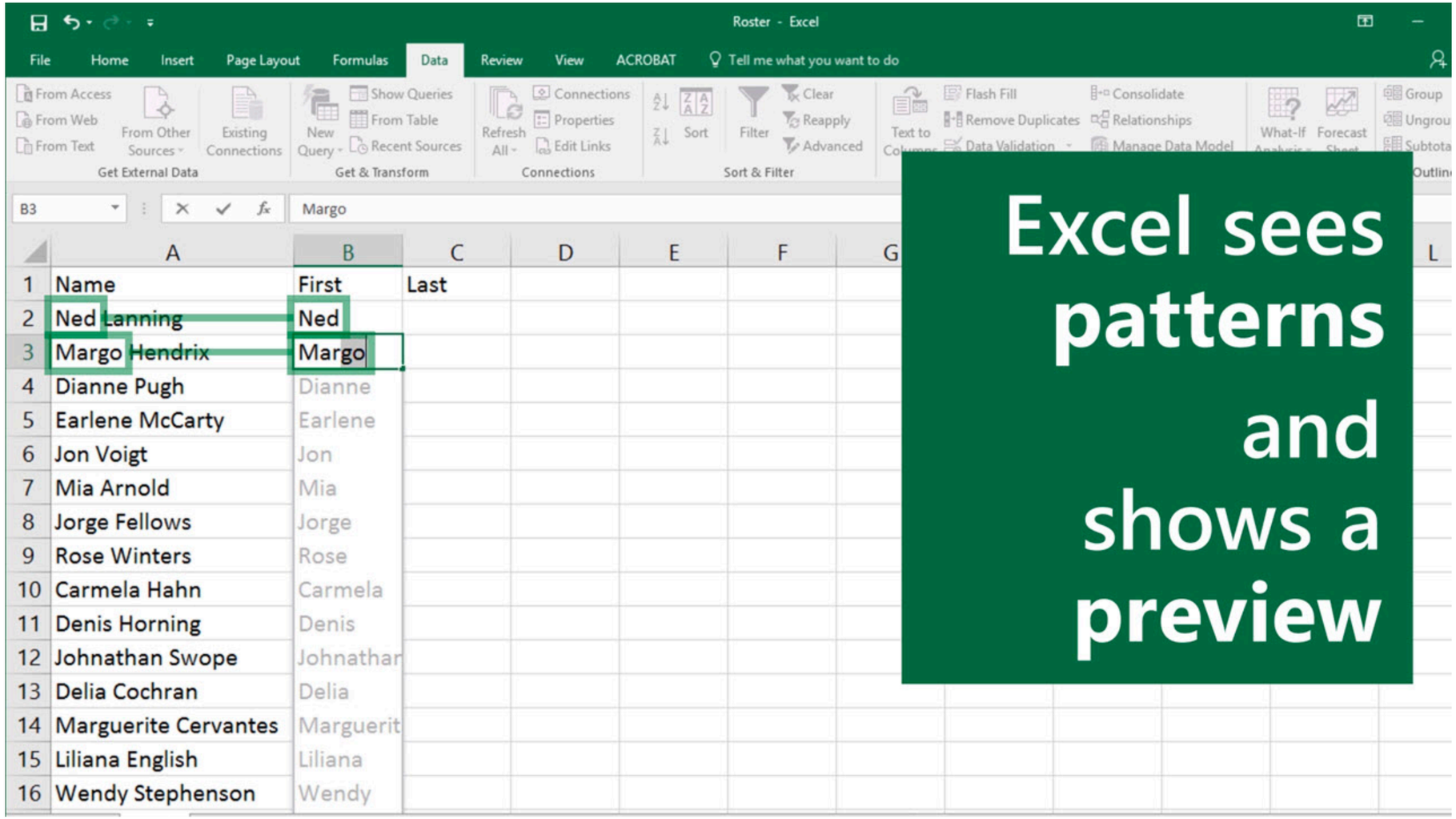
What Can be Synthesised

Group	*Description*	*Code*	*Code/Spec*	*Time*	*T-phase*	*T-inv*	*T-fail*	*T-com*	*T-all*	*T-IS*
Integers	swap two	12	0.9x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	min of two[2]	10	0.7x	0.1	0.1	0.1	< 0.1	0.1	0.2	
Linked List	length[1,2]	21	1.2x	0.4	0.9	0.5	0.4	0.6	1.4	29x
	max[1]	27	1.7x	0.6	0.8	0.5	0.4	0.4	0.8	20x
	min[1]	27	1.7x	0.5	0.9	0.5	0.4	0.5	1.2	49x
	singleton[2]	11	0.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	dispose	11	2.8x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	initialize	13	1.4x	< 0.1	0.1	0.1	< 0.1	0.1	< 0.1	
	copy[3]	35	2.5x	0.2	0.3	0.3	0.1	0.2	-	
	append[3]	19	1.1x	0.2	0.3	0.3	0.2	0.3	0.7	
	delete[3]	44	2.6x	0.7	0.5	0.3	0.2	0.3	0.7	
Sorted list	prepend[1]	11	0.3x	0.2	1.4	83.5	0.1	0.1	-	48x
	insert[1]	58	1.2x	4.8	-	-	-	5.0	-	6x
	insertion sort[1]	28	1.3x	1.1	1.8	1.3	1.2	1.2	74.2	82x
Tree	size	38	2.7x	0.2	0.3	0.2	0.2	0.2	0.3	
	dispose	16	4.0x	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	
	copy	55	3.9x	0.4	49.8	-	0.8	1.4	-	
	flatten w/append	48	4.0x	0.4	0.6	0.5	0.4	0.4	0.6	
	flatten w/acc	35	1.9x	0.6	1.7	0.7	0.5	0.6	-	
BST	insert[1]	58	1.2x	31.9	-	-	-	-	-	11x
	rotate left[1]	15	0.1x	37.7	-	-	-	-	-	0.5x
	rotate right[1]	15	0.1x	17.2	-	-	-	-	-	0.8x

[1] From (Qiu and Solar-Lezama 2017) [2] From (Leino and Milicevic 2012) [3] From (Qiu et al. 2013)

More Program Synthesis

Excel® FlashFill



	A	B	C
1	Name	First	Last
2	Ned Lanning	Ned	
3	Margo Hendrix	Margo	
4	Dianne Pugh	Dianne	
5	Earlene McCarty	Earlene	
6	Jon Voigt	Jon	
7	Mia Arnold	Mia	
8	Jorge Fellows	Jorge	
9	Rose Winters	Rose	
10	Carmela Hahn	Carmela	
11	Denis Horning	Denis	
12	Johnathan Swope	Johnathar	
13	Delia Cochran	Delia	
14	Marguerite Cervantes	Marguerit	
15	Liliana English	Liliana	
16	Wendy Stephenson	Wendy	

Final Project

- Choose one of the 30+ papers (seminal ones or state of the art in PLDI research)

- Topics:
  - Low-Level and Intermediate-Level Languages
  - Interpreters and Program Transformations
  - Lexing and Parsing
  - Types and Type Systems
  - Code Analysis and Optimisations
  - Verified Compilers
  - Miscellanea (language design, compiler testing, program synthesis)

- Write a report summarising the paper:
  - Problem, motivation
  - Ideas, contributions
  - (Most important) Evaluation
    - Developed example using paper's theory (paper-and-pencil), or
    - Report on using the system/tool, or
    - A survey of the follow-up works and the paper's impact

Wrapping Up

Why YSC4230?

- We have learned (hopefully):
  - How high-level languages are implemented in machine language
  - (A subset of) Intel x86 architecture
  - (A subset of) LLVM
  - Lexing and parsing
  - Lambda-calculus and its extensions
  - A little about programming language semantics and type systems
  - How to implement a type checker
  - How to implement a program analyser
  - Practical applications of theory (logic, proofs, automata, graphs, lattices)
  - How to represent complex data structures in memory
  - How to write large working programs in OCaml
  - How to be a better programmer

Where else is this stuff applicable?

- Understanding hardware/software interface
  - Different devices have different instruction sets, programming models

- General programming
  - In C/C++, better understanding of how the compiler works can help you generate better code.
  - Ability to read assembly/LLVM output from compiler
  - Experience with functional programming gives you different ways to think about solving a problem
  - Knowledge of type systems helps you understand type errors in Java, Scala, OCaml, etc.

- Writing domain specific languages
  - lex/yacc very useful for little utilities
  - understanding abstract syntax specification
  - understanding typing rules
  - being able to write *your own* interpreter and compiler

Stuff we didn't Cover

- We skipped stuff at every level…
- Concrete syntax/parsing:
  - Much more to the theory of parsing… LR(*)
  - Good syntax is art not science!
- Source language features:
  - Exceptions, advanced type systems, type inference, dependent types, concurrency
- Intermediate languages:
  - Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
- Compilation:
  - Continuation-passing transformation, analyses for SSA, compiling OO classes, lambda-lifting, closure conversion
- Analysis and Optimisations:
  - Abstract interpretation, cache optimization, instruction selection/optimization
- Runtime support:
  - memory management, garbage collection

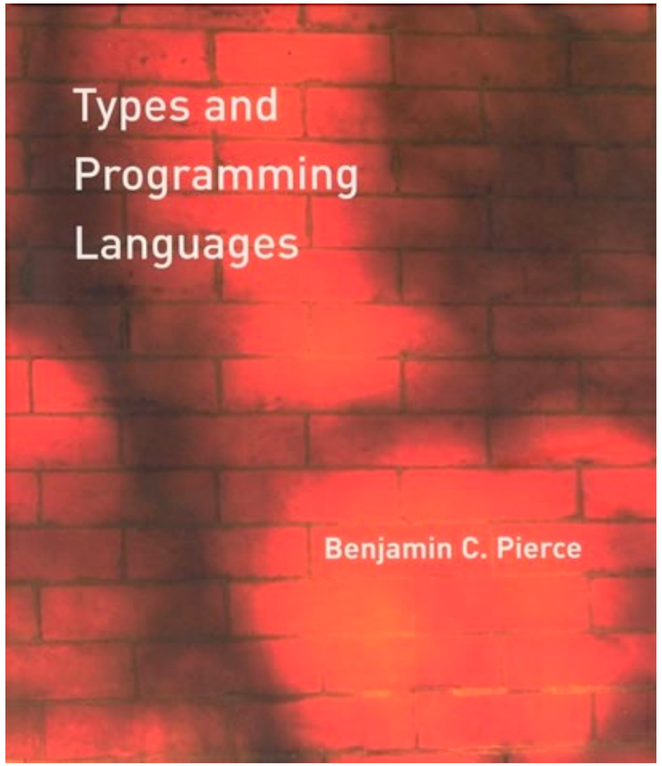
Where to go from here?

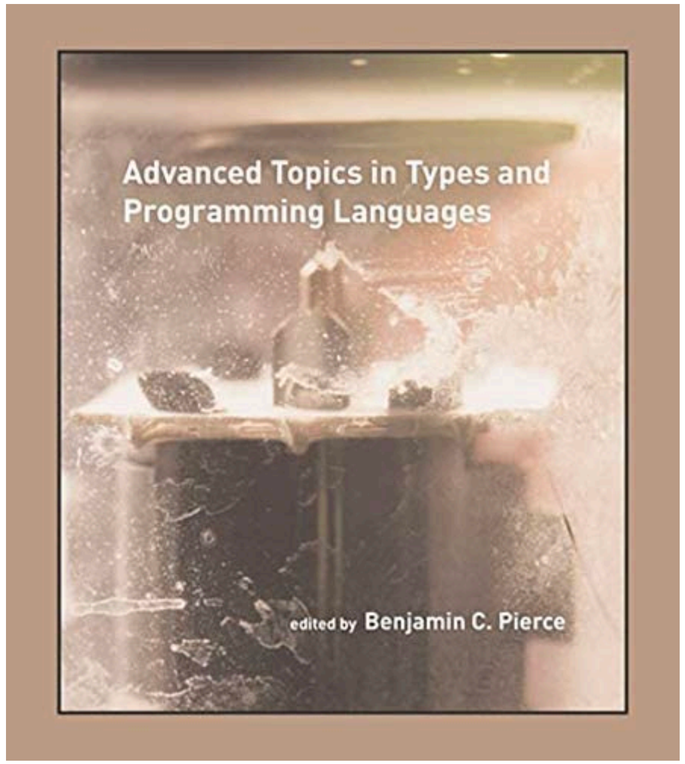
- Conferences (proceedings available on the web):
  - Programming Language Design and Implementation (PLDI)
  - Principles of Programming Languages (POPL)
  - Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  - International Conference on Functional Programming (ICFP)
  - European Symposium on Programming (ESOP)

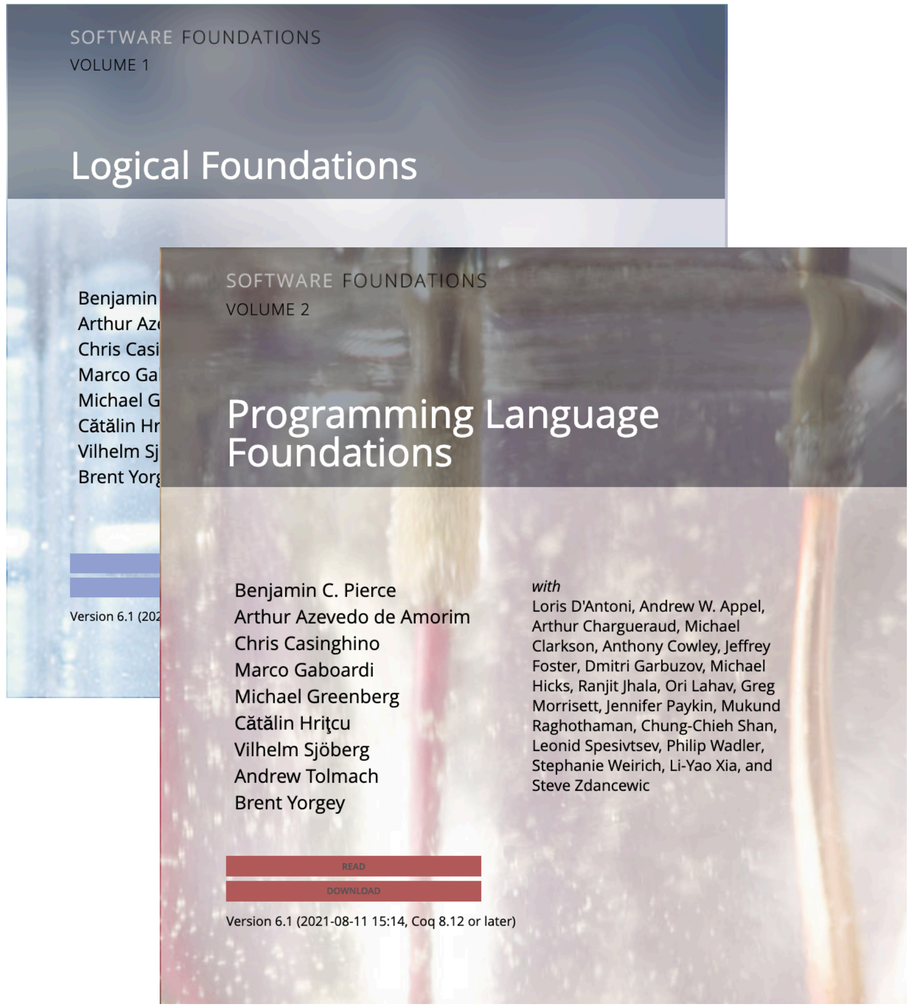
- Programming Language Mentoring Workshops (PLMW)
  - Affiliated with POPL/PLDI/OOPSLA/ICFP

- Technologies / Open Source Projects
  - Yacc, lex, bison, flex, …
  - LLVM – low level virtual machine
  - Java virtual machine (JVM), Microsoft's Common Language Runtime (CLR)
  - Languages: OCaml, F#, Haskell, Scala, Go, Rust, … Coq, Agda, …?

Further Reading - Types

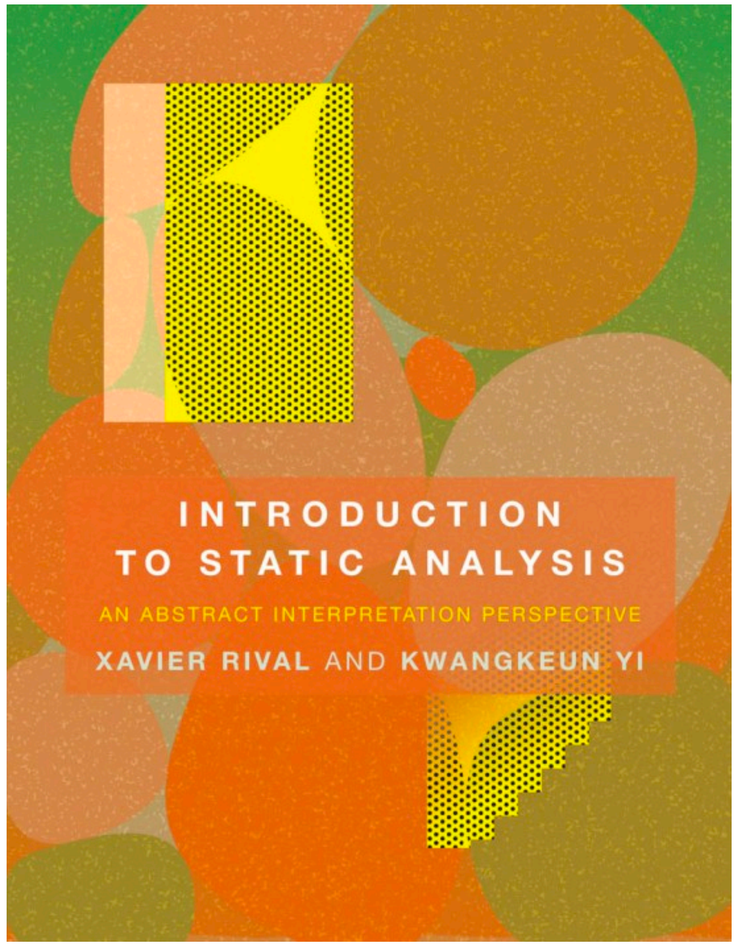


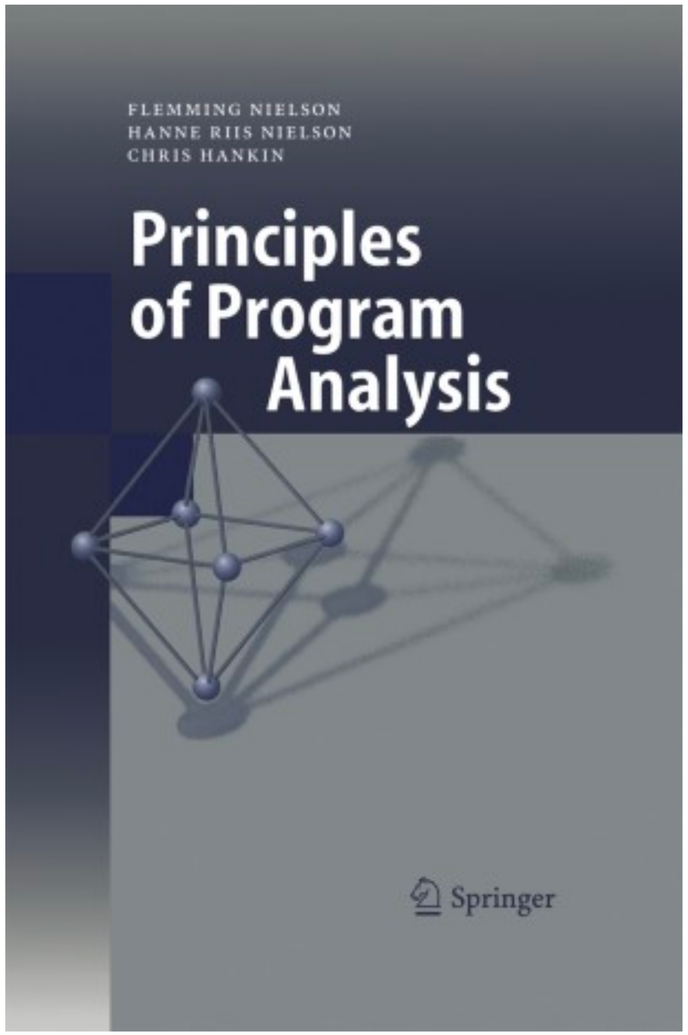




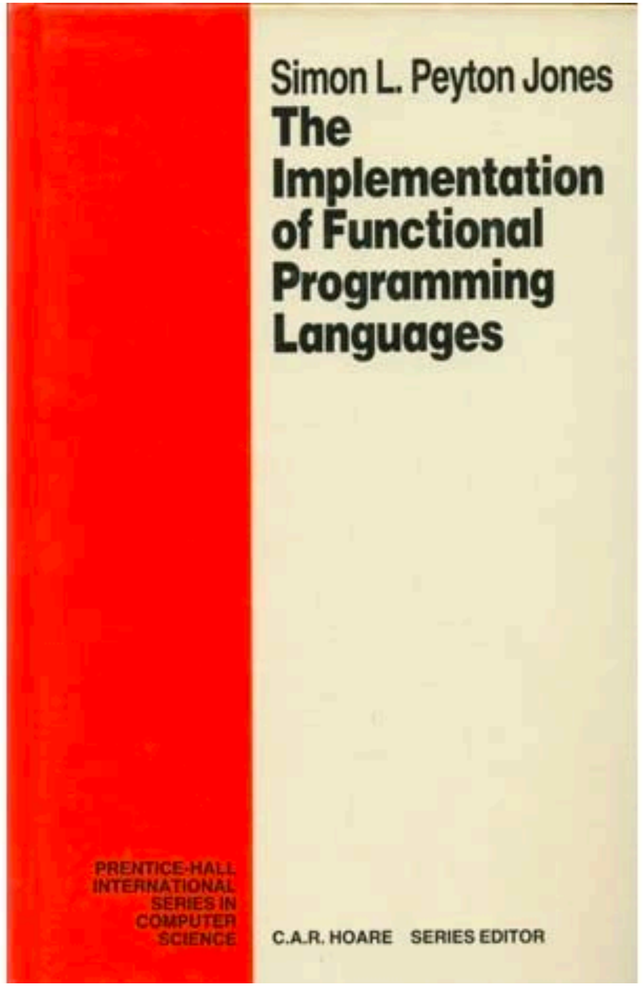
https://softwarefoundations.cis.upenn.edu/

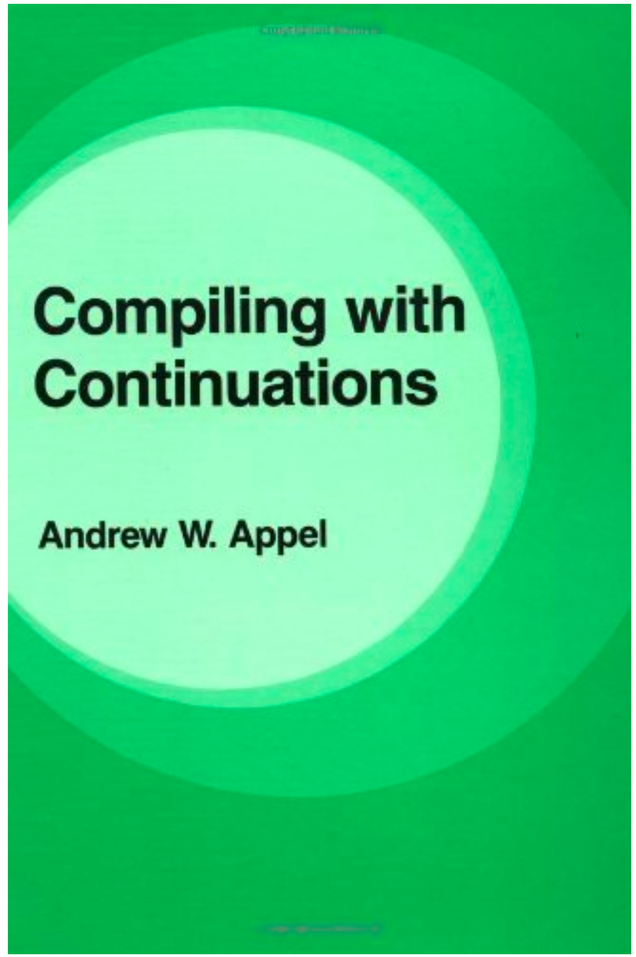
Further Reading - Analysis





Further Reading - Compilation Techniques





PL Classes at NUS School of Computing

- CS4215: Programming Language Implementation
  - semantics, type systems, automatic memory management, dynamic linking and just-in-time compilation, as features of modern execution systems

- CS5218: Principles and Practice of Program Analysis
  - foundations of static program analysis, abstract interpretation, lattice theory, analysis of higher-order languages

- CS6215: Advanced Topics in Program Analysis
  - symbolic execution, model checking, state-of-the-art industrial analysis tools, performance analysis

The End

Thanks!