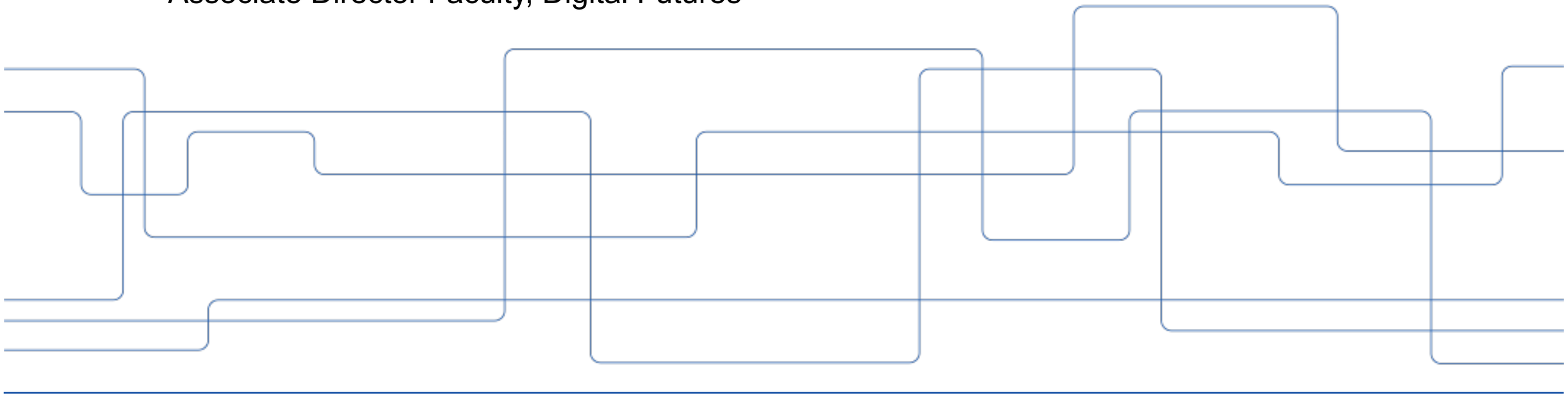
Compilers and Execution Environments (ID2202)

Fall 2021

Lecture 10: Liveness Analysis and Register Allocation

David Broman
Associate Professor, KTH Royal Institute of Technology
Associate Director Faculty, Digital Futures



Course Overview

Module 1 :Lexical Analysis, Syntax Analysis, and Semantic Analysis (Week 45-46)

LE 4: Lexical Analysis

LE 5: Synax Analysis

LE 6: Semantic Analysis and more on Parsing

Module 0 : Getting started (Week 44)

LE 1: Course Introduction

LE 2-3: Typed Functional Programming

Front End (analysis) → **Optimizer** → **Back End (synthesis)** → **Target Program**

Module 3 : Program Analysis and Optimizations (Weeks 49-50)

LE 10: Liveness Analysis and Register Allocation

LE 11: Dominators and Loop Optimization

LE 12: Advanced Optimizations

LE 13: Summary and Research Outlook

Module 2 : Code Generation and Runtime Environments (Weeks 47-48)

Execution Environment

LE 7: Linkers, Loaders, and Assembly Programming

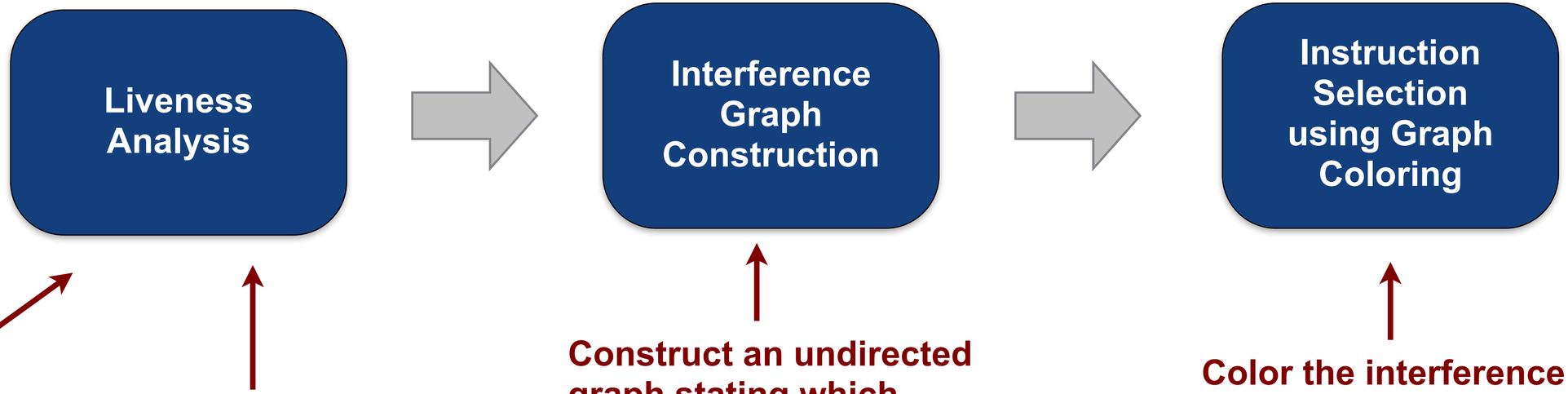
LE 8: Intermediate Representations and Instruction Selection

LE 9: Garbage Collection and Managed Runtime Environments

Part I
Liveness Analysis

Part II
Register Allocation using Graph Coloring

Overview - Instruction Selection



Discover which variables (virtual registers) that are live at the same time.

If not live at the same time, we can us the same physical register for different virtual registers.

Construct an undirected graph stating which variables interfere with each other.

Vertices = variables (virtual registers).

Edges = interference between variables.

Color the interference graph so that two connected nodes have different colors.

NP-complete problem, hence we use heuristics.

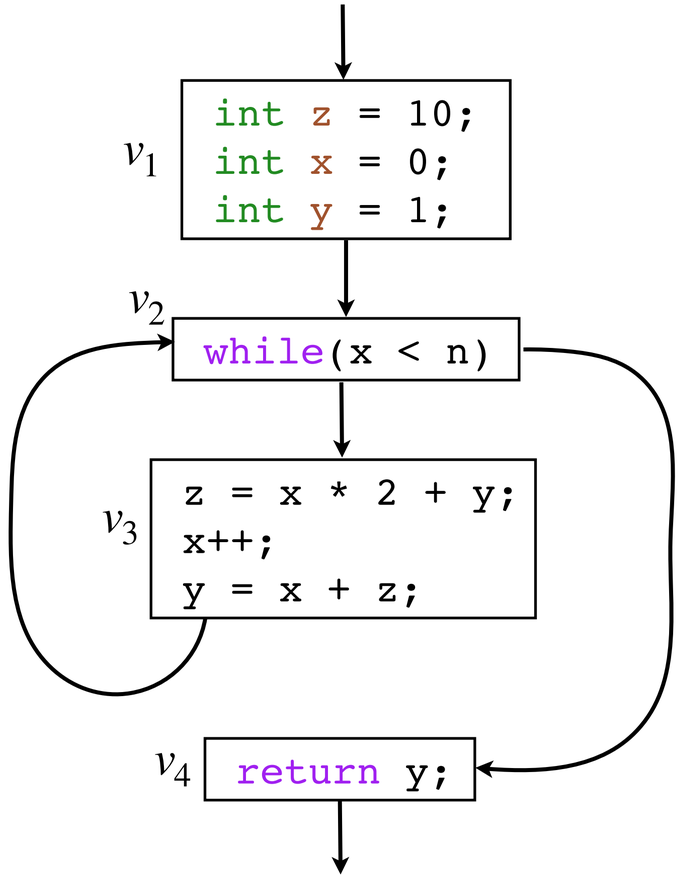
Exists many variants of register allocation using graph coloring. Commonly used in production compilers.

Alternatives: e.g., linear scan algorithms. Faster, but generally less good allocation. Good for JIT compilers.

Part I
Liveness Analysis

Control flow graph (CFG)

```
int foo(int n){
  int z = 10;
  int x = 0;
  int y = 1;
  while(x < n){
    z = x * 2 + y;
    x++;
    y = x + z;
  }
  return y;
}
```



CFG Definition

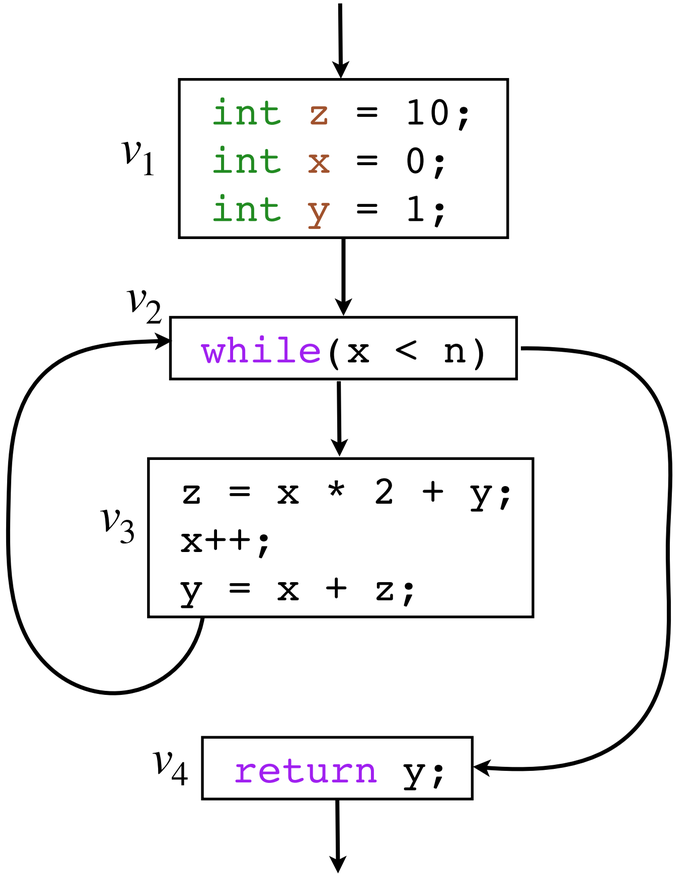
$$G = (V, E)$$

$$E = V \times V$$

$$V = \{v_1, v_2, v_3, v_4\}$$

$$E = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_2)\}$$

Help functions (predecessor and successor)



Definitions (pred and succ)

$$pred : V \to \mathcal{P}(V) \qquad pred(v) = \{w \in V \mid (w, v) \in E\}$$

$$succ : V \to \mathcal{P}(V) \qquad succ(v) = \{w \in V \mid (v, w) \in E\}$$

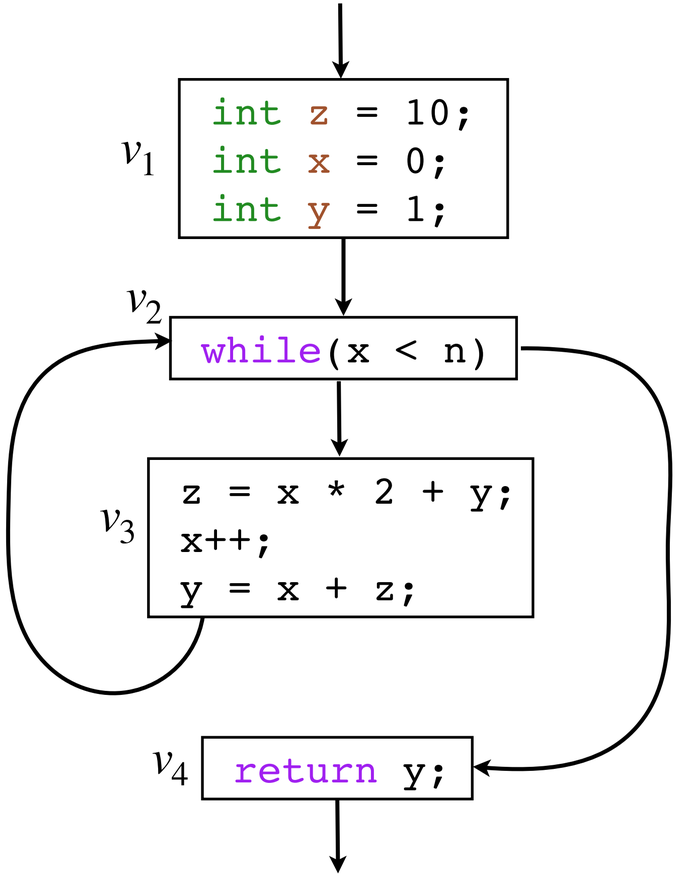
Examples

$$pred(v_1) = \emptyset \qquad succ(v_1) = \{v_2\}$$

Exercise: Define predecessor and successor for vertex number 2.

$$pred(v_2) = \{v_1, v_3\} \qquad succ(v_2) = \{v_3, v_4\}$$

Definition and use



$def(v_1) = \{z, x, y\}$
$use(v_1) = \emptyset$

$def(v_2) = \emptyset$
$use(v_2) = \{x, n\}$

$def(v_3) = \{z, x, y\}$
$use(v_3) = \{x, y\}$

$def(v_4) = \emptyset$
$use(v_4) = \{y\}$

Definitions (def and use)

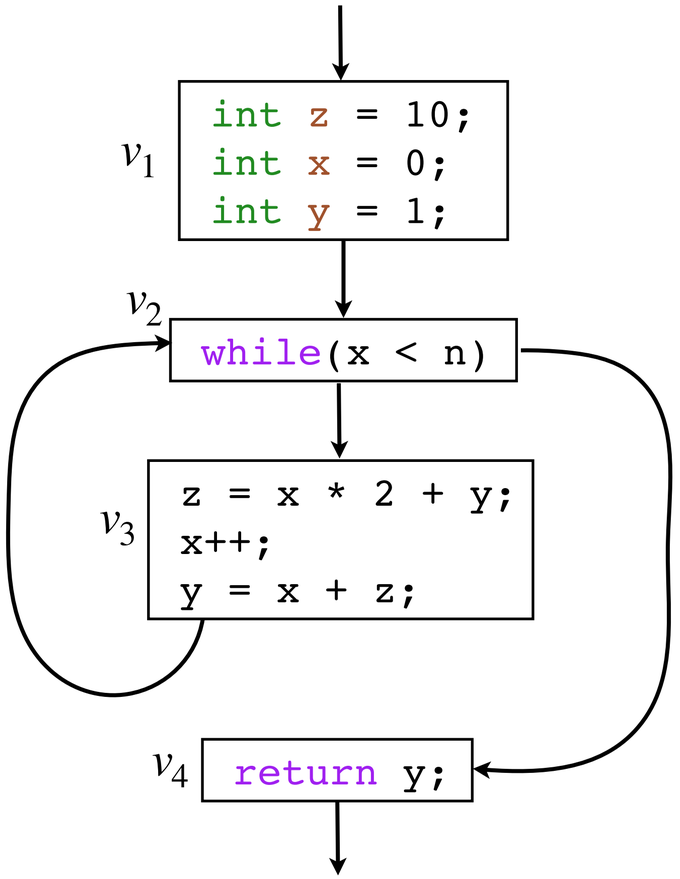
$def : V \rightarrow \mathcal{P}(X)$
$use : V \rightarrow \mathcal{P}(X)$

def **represents the set of variables that are defined in the block (vertex).**

use **of a vertex** v **is the set of variables that are used in the block,** ***before a potential redefinition.***

Note. z **is not used before redefinition.**

Live-in and live-out



$$def(v_1) = \{z, x, y\}$$
$$use(v_1) = \emptyset$$

$$def(v_2) = \emptyset$$
$$use(v_2) = \{x, n\}$$

$$def(v_3) = \{z, x, y\}$$
$$use(v_3) = \{x, y\}$$

$$def(v_4) = \emptyset$$
$$use(v_4) = \{y\}$$

Live variable: A variable is live on an edge if it contains a value that *may* be used in the future.

Liveness analysis: Compute live-in to a node and live-out from a node.

Live-in: The set of variables that are live on any incoming edge.

$$in : V \rightarrow \mathcal{P}(X)$$

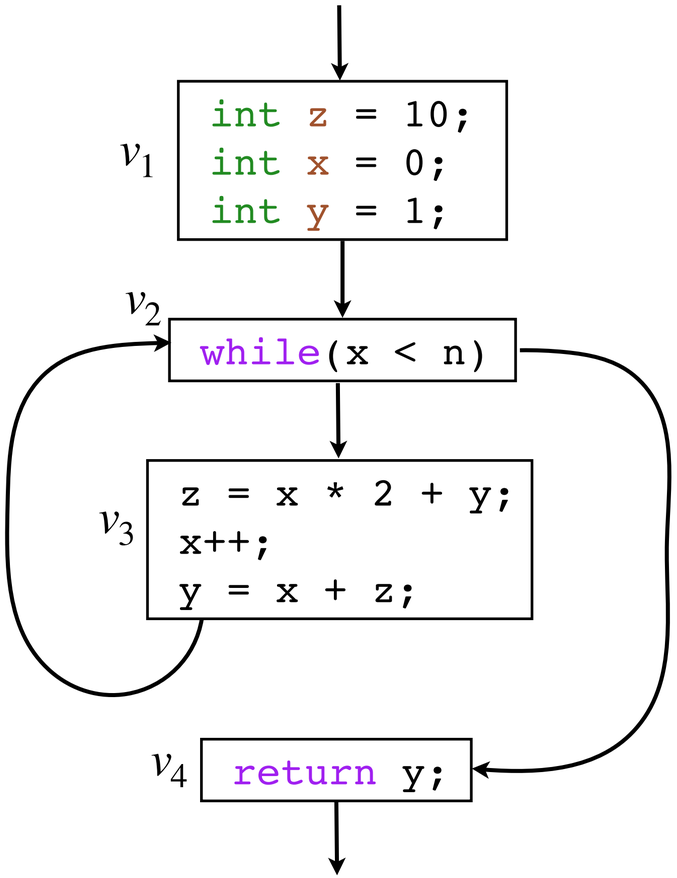
Live-out: The set of variables that are live on any outgoing edge.

$$out : V \rightarrow \mathcal{P}(X)$$

How do we compute live-in and live-out sets?

By using data-flow analysis!

Data-flow Equations



Data-flow equations

$$in(v) = use(v) \cup \left(out(v) - def(v)\right)$$

$$out(v) = \bigcup_{w \in succ(v)} in(w)$$

Algorithm (Liveness Analysis)

for each $v \in V$

$out[v] \leftarrow \emptyset$

$in[v] \leftarrow \emptyset$

while($true$)

for each $v \in V$

$out'[v] \leftarrow out[v]$

$in'[v] \leftarrow in[v]$

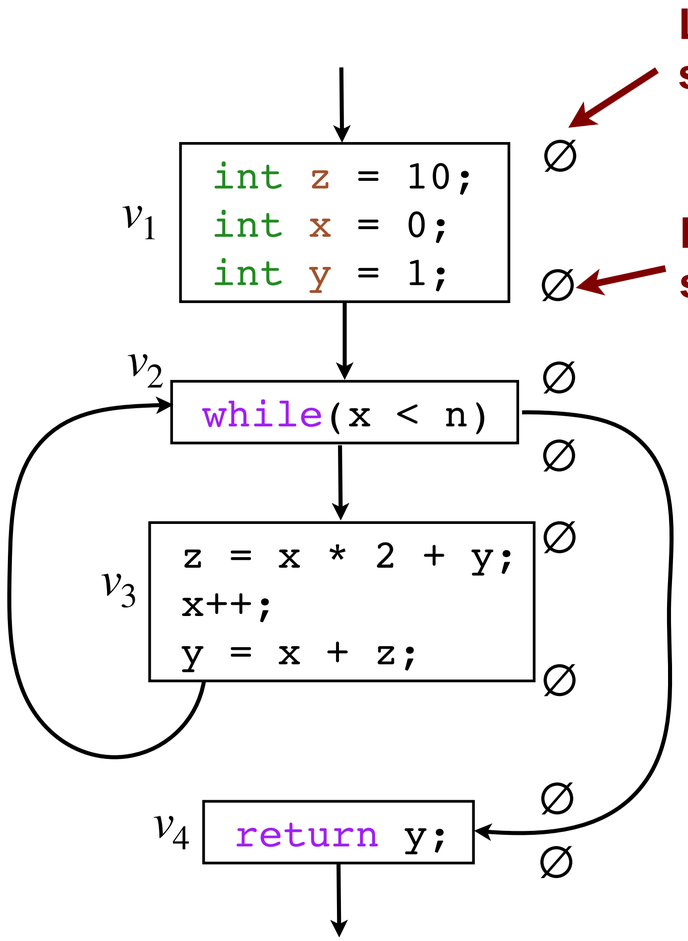
$$out[v] \leftarrow \bigcup_{w \in succ(v)} in[w]$$

$$in[v] \leftarrow use(v) \cup \left(out[v] - def(v)\right)$$

if $\forall v \in V \,.\, in[v] = in'[v] \wedge out[v] = out'[v]$ then

break

Solving Data-flow Equations (Initialization)



$$def(v_1) = \{z, x, y\}$$
$$use(v_1) = \emptyset$$

$$def(v_2) = \emptyset$$
$$use(v_2) = \{x, n\}$$

$$def(v_3) = \{z, x, y\}$$
$$use(v_3) = \{x, y\}$$

$$def(v_4) = \emptyset$$
$$use(v_4) = \{y\}$$

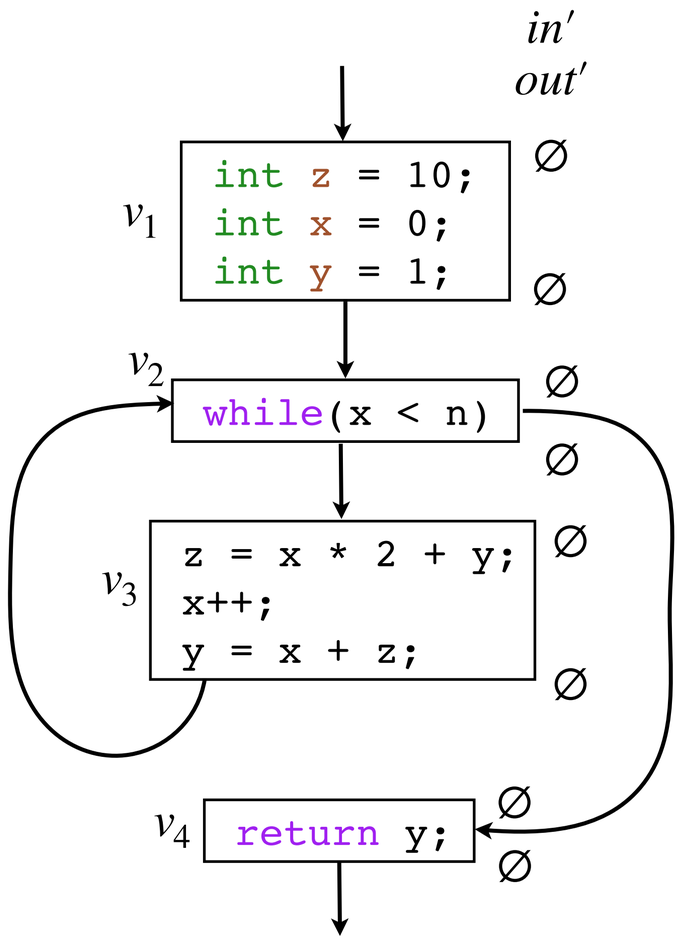
Algorithm (Liveness Analysis)

for each $v \in V$
$out[v] \leftarrow \emptyset$
$in[v] \leftarrow \emptyset$

Initialize live-in and live-out to empty sets.

while($true$)
for each $v \in V$
$out'[v] \leftarrow out[v]$
$in'[v] \leftarrow in[v]$
$out[v] \leftarrow \bigcup_{w \in succ(v)} in[w]$
$in[v] \leftarrow use(v) \cup (out[v] - def(v))$
if $\forall v \in V \,.\, in[v] = in'[v] \wedge out[v] = out'[v]$ then
break

Solving Data-flow Equations (Step 1)



	in' / out'	in / out	
v_1	$\emptyset$	$\{n\}$	$def(v_1) = \{z, x, y\}$
	$\emptyset$	$\{x, y, n\}$	$use(v_1) = \emptyset$
v_2	$\emptyset$	$\{x, y, n\}$	$def(v_2) = \emptyset$
	$\emptyset$	$\{x, y\}$	$use(v_2) = \{x, n\}$
v_3	$\emptyset$	$\{x, y\}$	$def(v_3) = \{z, x, y\}$
	$\emptyset$	$\emptyset$	$use(v_3) = \{x, y\}$
v_4	$\emptyset$	$\{y\}$	$def(v_4) = \emptyset$
	$\emptyset$	$\emptyset$	$use(v_4) = \{y\}$

2. More efficient to compute in backward order, from out to in.

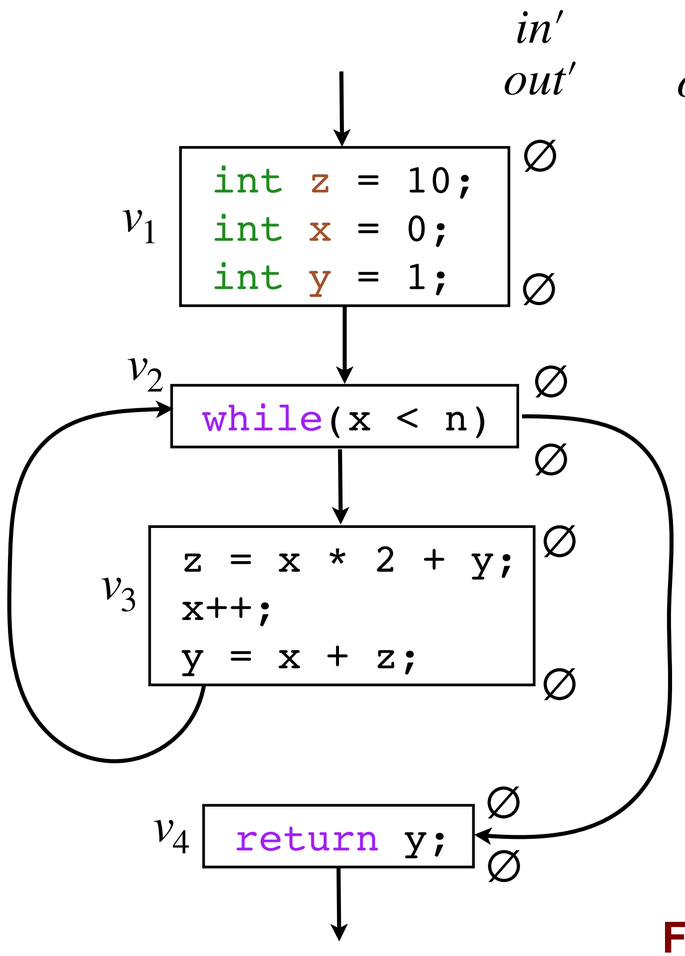
Algorithm (Liveness Analysis)

for each $v \in V$
  $out[v] \leftarrow \emptyset$
  $in[v] \leftarrow \emptyset$

1. The order of processed nodes does not matter for correctness, but for performance!

while($true$)
  for each $v \in V$
    $out'[v] \leftarrow out[v]$
    $in'[v] \leftarrow in[v]$
    $out[v] \leftarrow \bigcup_{w \in succ(v)} in[w]$
    $in[v] \leftarrow use(v) \cup \left(out[v] - def(v)\right)$
  if $\forall v \in V \,.\, in[v] = in'[v] \wedge out[v] = out'[v]$ then
    break

Solving Data-flow Equations (Step 1, check)



	in' / out'	in / out	
v_1	$\varnothing$	$\{n\}$	$def(v_1) = \{z, x, y\}$
	$\varnothing$	$\{x, y, n\}$	$use(v_1) = \varnothing$
v_2	$\varnothing$	$\{x, y, n\}$	$def(v_2) = \varnothing$
	$\varnothing$	$\{x, y\}$	$use(v_2) = \{x, n\}$
v_3	$\varnothing$	$\{x, y\}$	$def(v_3) = \{z, x, y\}$
	$\varnothing$	$\varnothing$	$use(v_3) = \{x, y\}$
v_4	$\varnothing$	$\{y\}$	$def(v_4) = \varnothing$
	$\varnothing$	$\varnothing$	$use(v_4) = \{y\}$

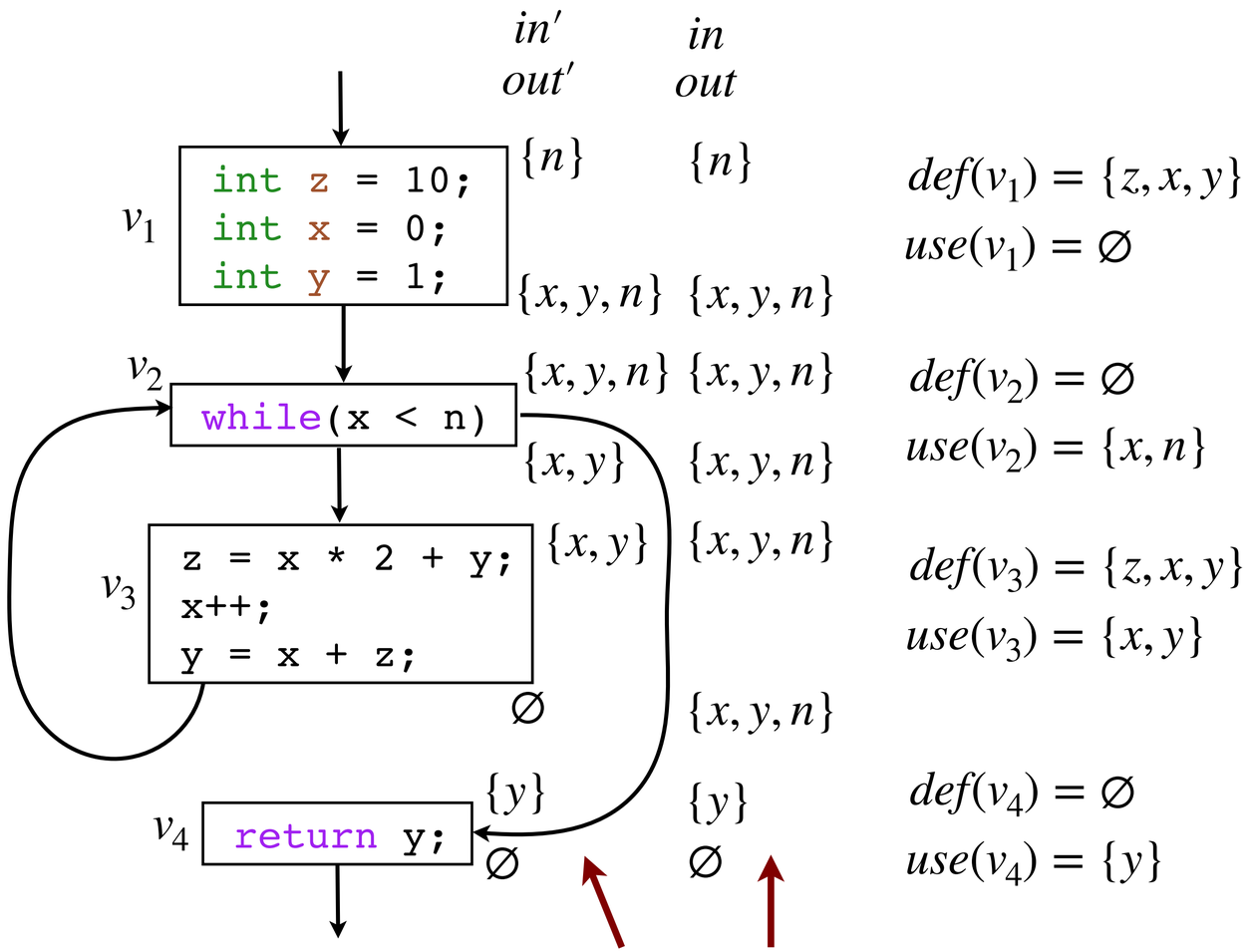
Fixed-point computation. If no change, break out of the while loop. In this case, continue.

Algorithm (Liveness Analysis)

for each $v \in V$
  $out[v] \leftarrow \varnothing$
  $in[v] \leftarrow \varnothing$

while($true$)
  for each $v \in V$
    $out'[v] \leftarrow out[v]$
    $in'[v] \leftarrow in[v]$
    $out[v] \leftarrow \bigcup_{w \in succ(v)} in[w]$
    $in[v] \leftarrow use(v) \cup (out[v] - def(v))$
    if $\forall v \in V \,.\, in[v] = in'[v] \wedge out[v] = out'[v]$ then
      break

Solving Data-flow Equations (Step 2 and check)



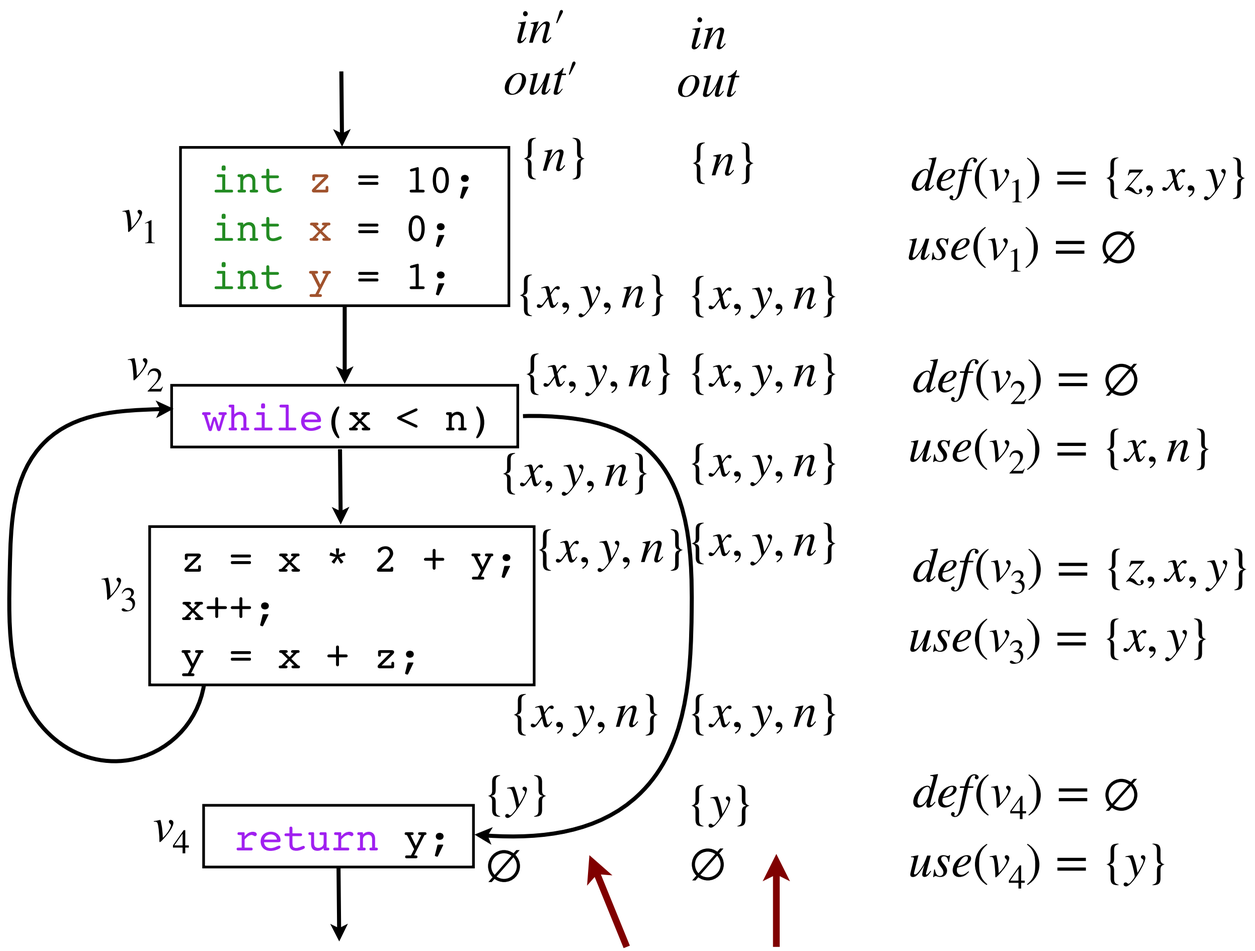
Algorithm (Liveness Analysis)

for each $v \in V$
$out[v] \leftarrow \varnothing$
$in[v] \leftarrow \varnothing$

Exercise: Compute new live-in and live-out variable sets.

while($true$)
for each $v \in V$
$out'[v] \leftarrow out[v]$
$in'[v] \leftarrow in[v]$
$out[v] \leftarrow \bigcup_{w \in succ(v)} in[w]$
$in[v] \leftarrow use(v) \cup (out[v] - def(v))$
if $\forall v \in V \,.\, in[v] = in'[v] \wedge out[v] = out'[v]$ then
break

Solving Data-flow Equations (Step 3 and check)



Algorithm (Liveness Analysis)

for each $v \in V$

$out[v] \leftarrow \emptyset$

$in[v] \leftarrow \emptyset$

while($true$)

for each $v \in V$

$out'[v] \leftarrow out[v]$

$in'[v] \leftarrow in[v]$

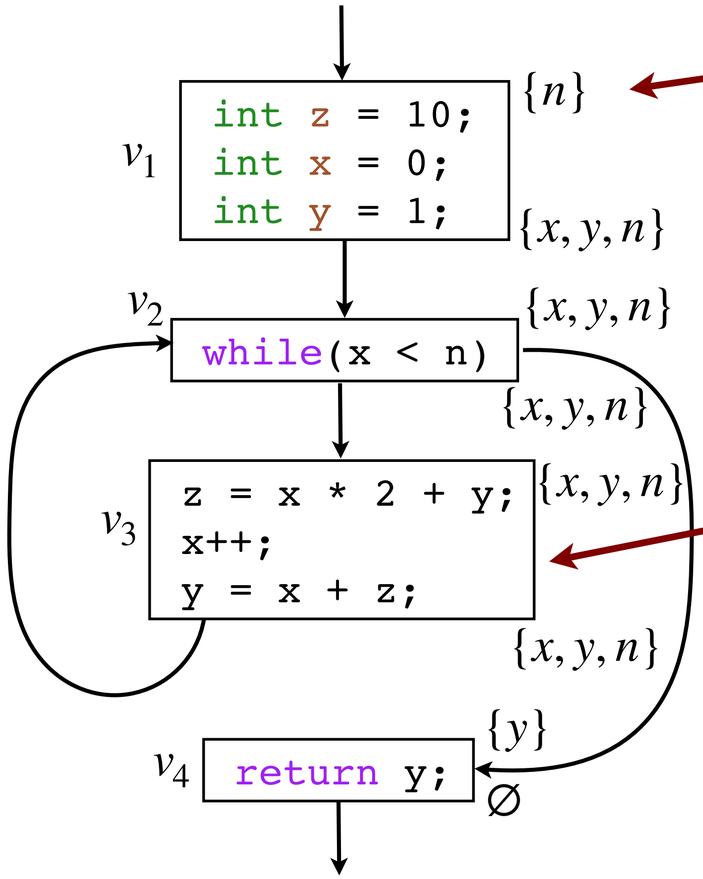
$$out[v] \leftarrow \bigcup_{w \in succ(v)} in[w]$$

$$in[v] \leftarrow use(v) \cup \left(out[v] - def(v)\right)$$

if $\forall v \in V \,.\, in[v] = in'[v] \wedge out[v] = out'[v]$ then

break

Solving Data-flow Equations (Observations)



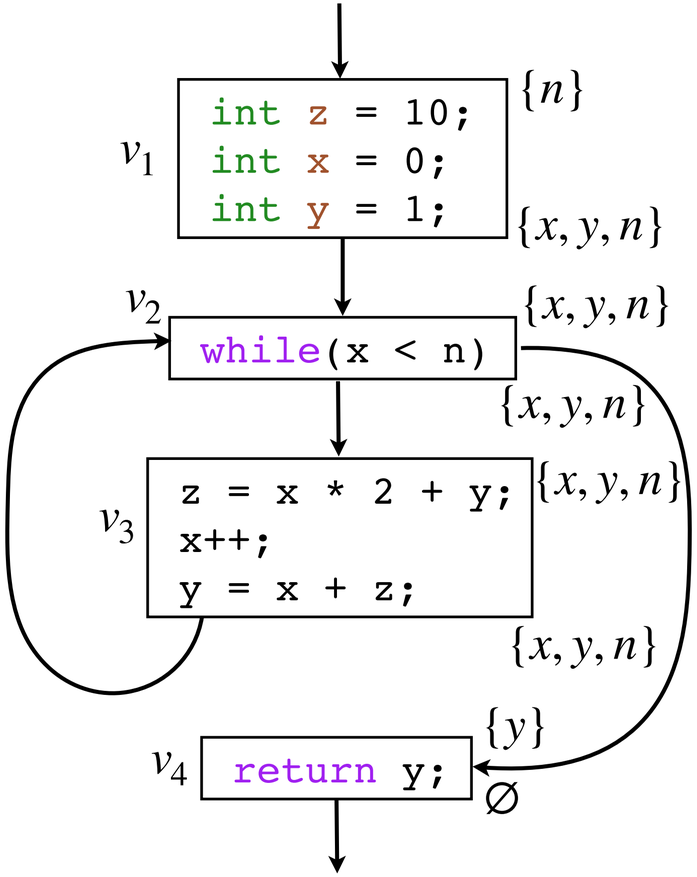
Variable n is live in the whole program (except for in the return statement). Variable n is a parameter.

Variables z is not live-in or live-out anywhere. It is only used within the basic block v_3.

Variables $\{x, y, n\}$ are live-out because of the back loop to v_2.

```
int foo(int n){
  int z = 10;
  int x = 0;
  int y = 1;
  while(x < n){
    z = x * 2 + y;
    x++;
    y = x + z;
  }
  return y;
}
```

Interference Graph



Recall: we have a large set of variables (or virtual registers), but a limited number of physical registers.

If two variables are not live at the same time, we can reuse the same physical register.

Hence, we want to specify when two variables interfere with each other.

Interference graph

Nodes (vertices) represent the variables (virtual registers)

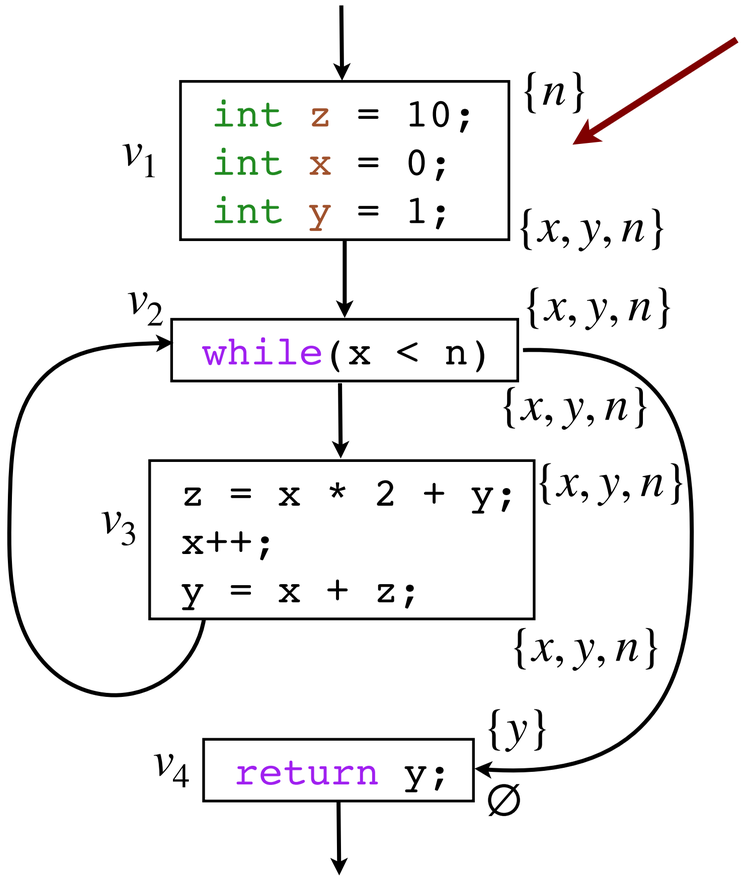
An edge between two nodes states that these two variables interfere with each other.

Possible cause of interference:

- Two variables are live at the same time.
- Special case handling of variables.

Interference Graph

Problem already in the first block.
All variables interfere with each other.



$def(v_1) = \{z, x, y\}$

$def(v_2) = \emptyset$

$def(v_3) = \{z, x, y\}$

$def(v_4) = \emptyset$

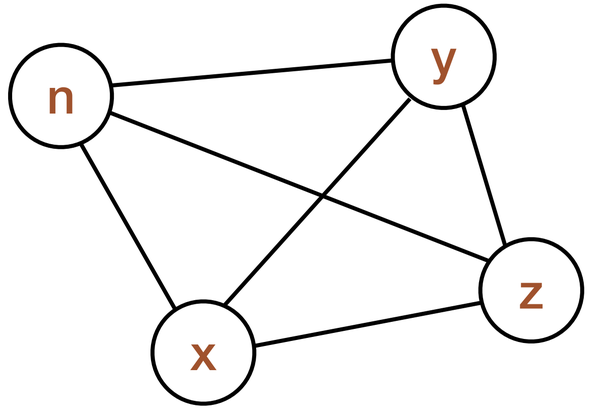
Algorithm (Interference Graph Construction)

1. For each variable $x \in X$, create a unique vertex
2. For each vertex $v \in V$ do:

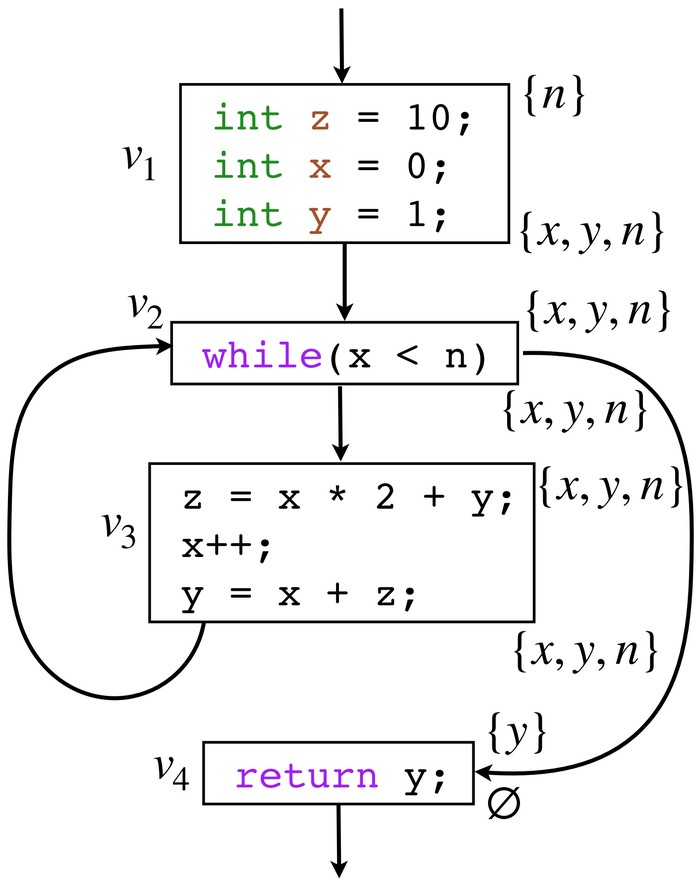
   For each $a \in def(v)$ do:

   For each $b \in out(v)$ do:

   Add an edge $\{a, b\}$



Liveness Analysis at the Instruction Level

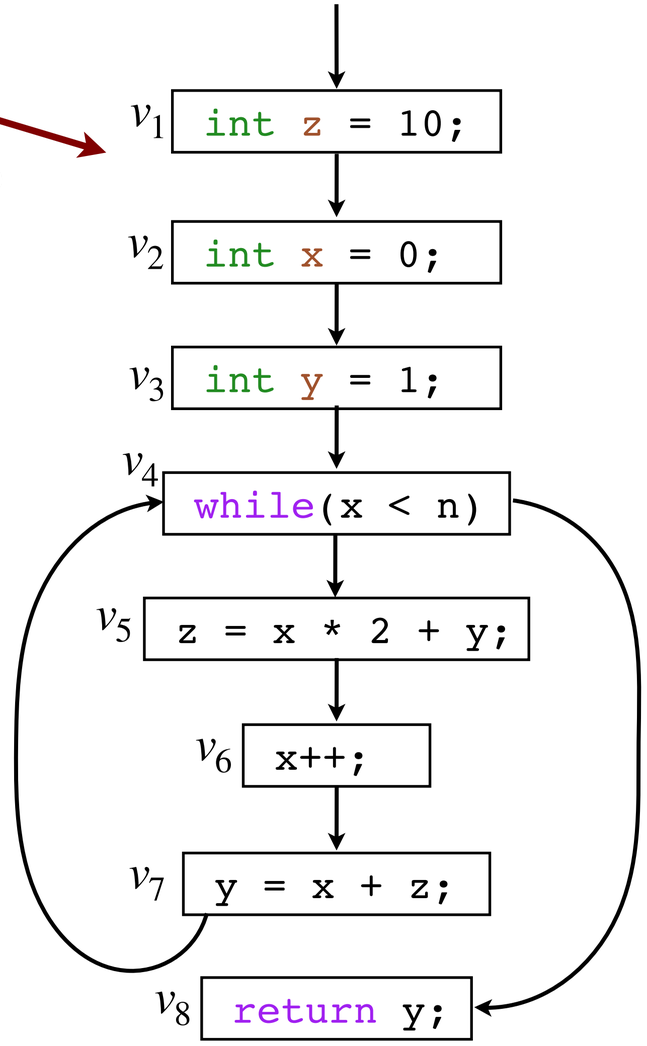


Why not perform the analysis at instruction level?

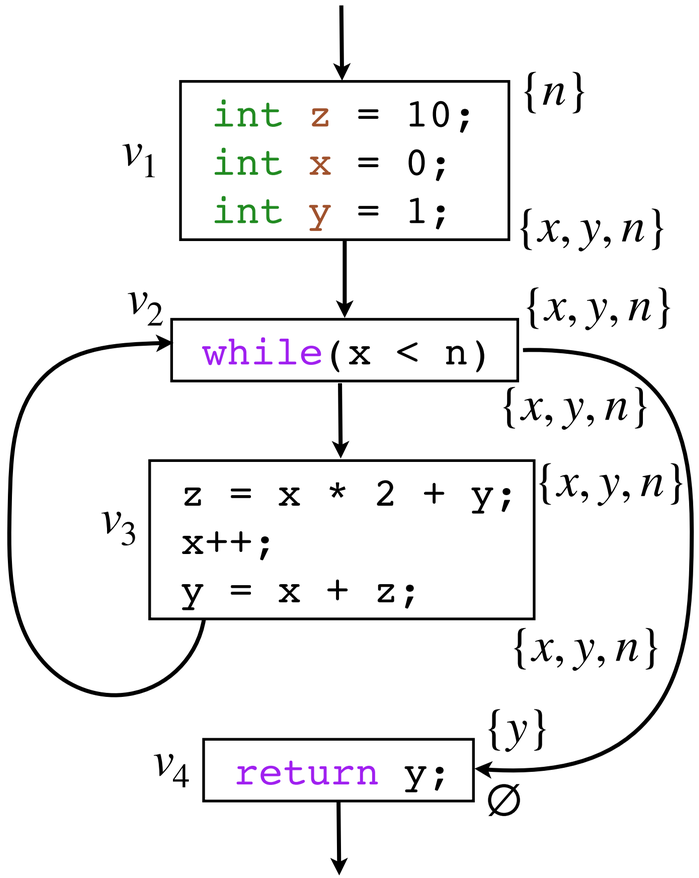
Because of performance. Worst case $O(n^4)$, when n is the max number of nodes and variables.

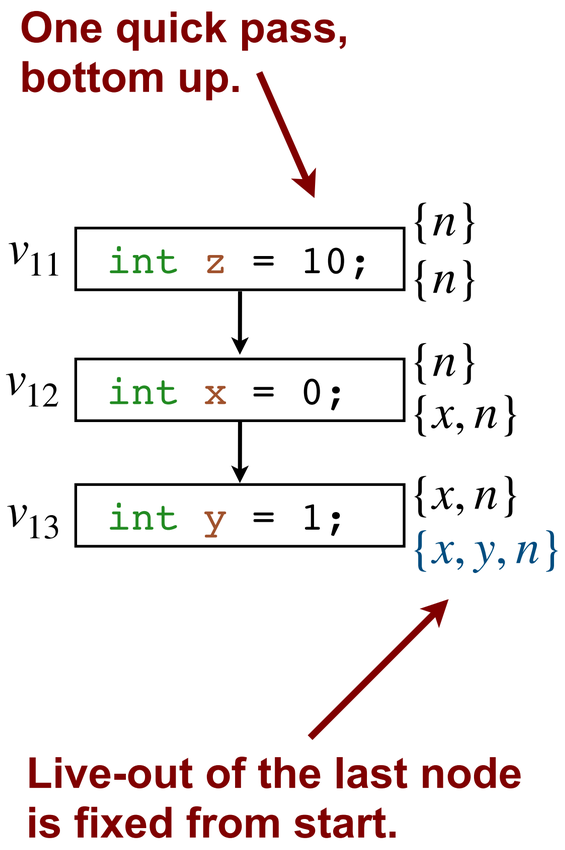
Alternative

1. Perform analysis first at the CFG-level (nodes are basic blocks).
2. For each basic block with more than one instruction, perform the analysis at the instruction level.



Liveness Analysis at the Instruction Level





Algorithm (Liveness Analysis)

for each $v \in V$

$$out[v] \leftarrow \emptyset$$
$$in[v] \leftarrow \emptyset$$

while($true$)

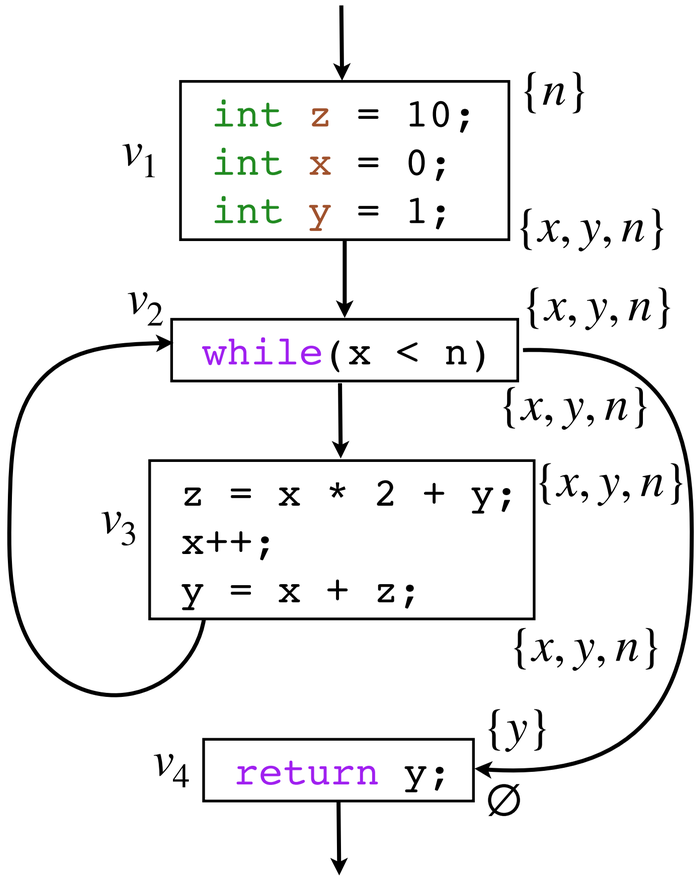
for each $v \in V$

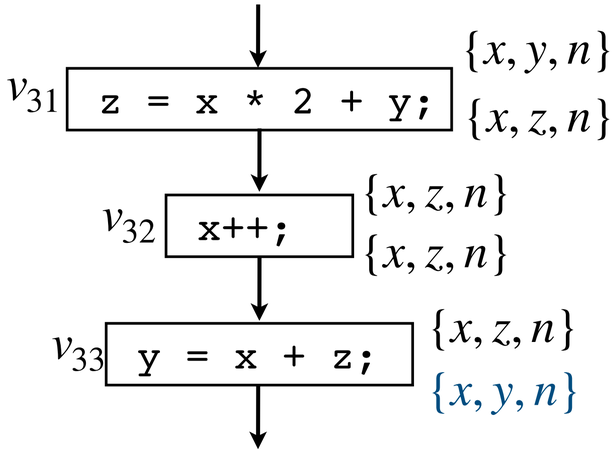
$$out'[v] \leftarrow out[v]$$
$$in'[v] \leftarrow in[v]$$
$$out[v] \leftarrow \bigcup_{w \in succ(v)} in[w]$$
$$in[v] \leftarrow use(v) \cup \left(out[v] - def(v)\right)$$

if $\forall v \in V \,.\, in[v] = in'[v] \wedge out[v] = out'[v]$ then

break

Liveness Analysis at the Instruction Level





Algorithm (Liveness Analysis)

for each $v \in V$

$$out[v] \leftarrow \emptyset$$
$$in[v] \leftarrow \emptyset$$

while($true$)

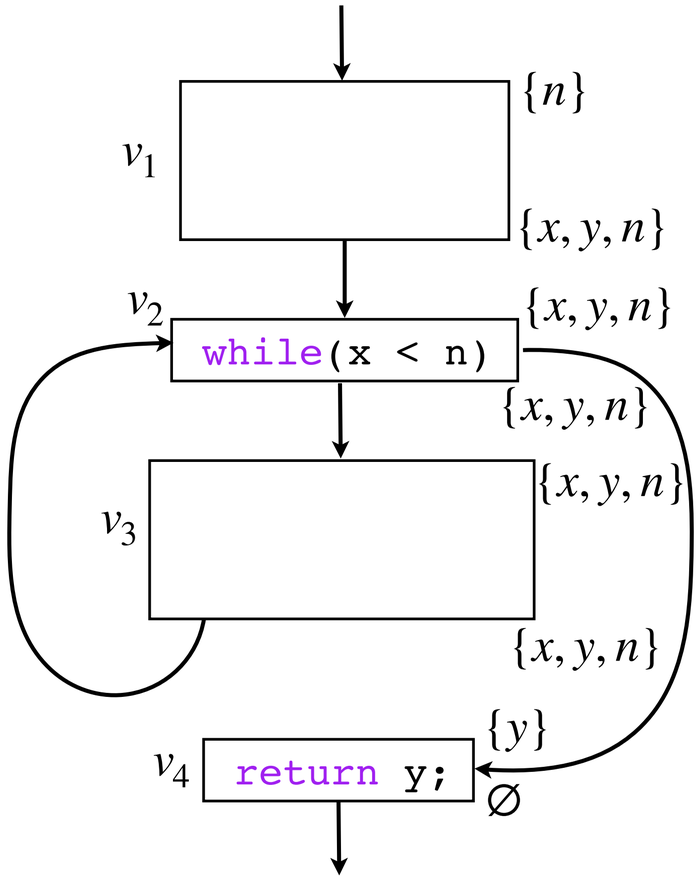
for each $v \in V$

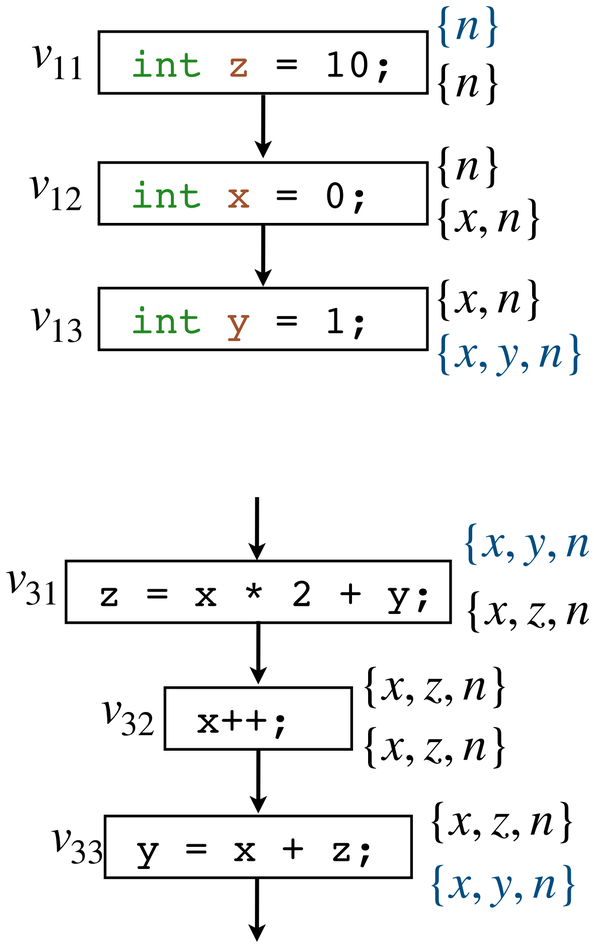
$$out'[v] \leftarrow out[v]$$
$$in'[v] \leftarrow in[v]$$
$$out[v] \leftarrow \bigcup_{w \in succ(v)} in[w]$$
$$in[v] \leftarrow use(v) \cup \left(out[v] - def(v)\right)$$

if $\forall v \in V \,.\, in[v] = in'[v] \wedge out[v] = out'[v]$ then

break

Liveness Analysis at the Instruction Level





Algorithm (Interference Graph Construction)

1. For each variable $x \in X$, create a unique vertex
2. For each vertex $v \in V$ do:

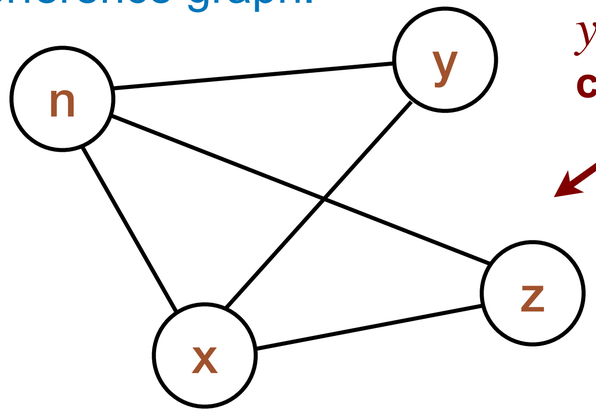
   For each $a \in def(v)$ do:

   For each $b \in out(v)$ do:

   Add an edge $\{a, b\}$

Exercise:
Compute the interference graph.



Note: Variables y and z are not conflicting.

Part II
Register Allocation using Graph Coloring

Register Allocation Using Graph Coloring

Algorithm (Graph Coloring by Simplification)

The machine has k number of physical registers

0. $S \leftarrow \emptyset$; $R \leftarrow$ Empty stack
1. **Construct** the interference graph G
   using liveness analysis, assuming S actual spill.
   $S' \leftarrow S$
2. **Simplify** by graph coloring heuristic:
   repeat
      if there exists a node $v \in V$
                                  where $|edges(v)| < k$ then
         push $(v, edges(v),$ color$)$ on stack R
         $G \leftarrow (V - \{v\}, E - edges(v))$
      else
         push $(w, edges(w),$ spill$)$ on stack S, where $w \in V'$
         $G \leftarrow (V - \{w\}, E - edges(w))$
   until $G = (\emptyset, \emptyset)$
3. **Assign** colors (registers) or actual spills.
   while(stack$R$ is not empty)
      $t \leftarrow$ pop item from stack R
      if t matches $(v, E',$ color$)$ then
         $G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$
         $c[v] \leftarrow n$ where $\forall w \in edges(v) \,.\, n \neq c[w]$
      else t matches $(v, E',$ spill$)$
         if $|\{c[w] \;:\; w \in strip(edges(v), V)\}| < k$ then
            $G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$
            $c[v] \leftarrow n$ where $\forall w \in strip(edges(v), V) \,.\, n \neq c[w]$
         else
            $S \leftarrow S \cup \{v\}$
4. **Restart** at step 1 if $S \neq S'$

$$strip(E, V) = \{(x, y) : (x, y) \in E \wedge x \in V \wedge y \in V\}$$

(The algorithm is based Chap 11 in Apple, 1998)

Register Allocation Using Graph Coloring

Algorithm (Graph Coloring by Simplification)

The machine has k number of physical registers

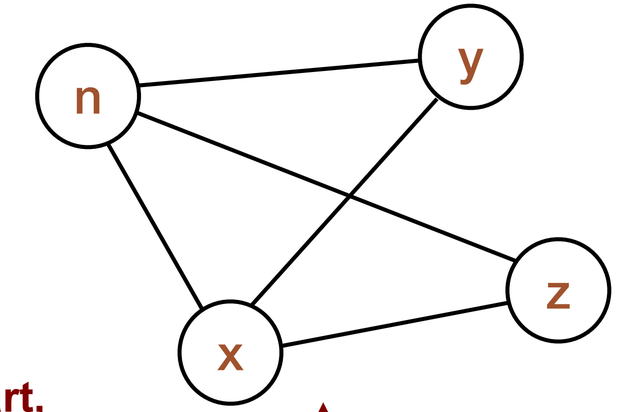
0. $S \leftarrow \emptyset$; $R \leftarrow$ Empty stack
1. **Construct** the interference graph G
   using liveness analysis, assuming S actual spill.
   $S' \leftarrow S$
2. **Simplify** by graph coloring heuristic:
   repeat
      if there exists a node $v \in V$
                  where $|edges(v)| < k$ then
         push $(v, edges(v), \text{color})$ on stack R
         $G \leftarrow (V - \{v\}, E - edges(v))$
      else
         push $(w, edges(w), \text{spill})$ on stack S, where $w \in V$
         $G \leftarrow (V - \{w\}, E - edges(w))$
   until $G = (\emptyset, \emptyset)$

Example: Assume we have variables x, y, n, z, and $k = 3$ physical registers.

1. Construct

$S = \{\}$

$R = []$



As a start, we have no actual spill.

The stack is initially empty.

We construct the interference graph using liveness analysis.

Register Allocation Using Graph Coloring

Algorithm (Graph Coloring by Simplification)

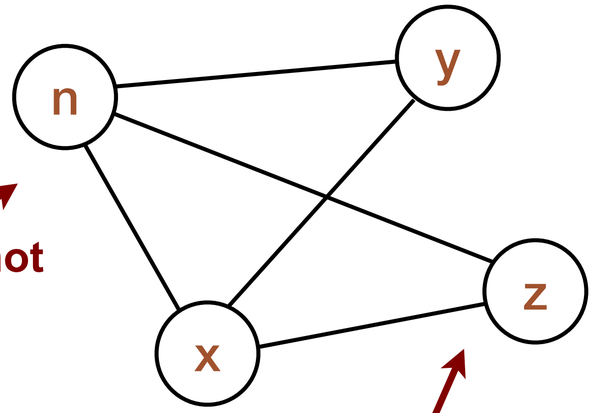
The machine has k number of physical registers

0. $S \leftarrow \emptyset$; $R \leftarrow$ Empty stack
1. **Construct** the interference graph G
   using liveness analysis, assuming S actual spill.
   $S' \leftarrow S$
2. **Simplify** by graph coloring heuristic:

repeat
  if there exists a node $v \in V$
    where $|edges(v)| < k$ then
    push $(v, edges(v), \text{color})$ on stack R
    $G \leftarrow (V - \{v\}, E - edges(v))$
  else
    push $(w, edges(w), \text{spill})$ on stack S, where $w \in V$
    $G \leftarrow (V - \{w\}, E - edges(w))$
until $G = (\emptyset, \emptyset)$

Example: Assume we have variables x, y, n, z, and $k = 3$ physical registers

Simplify:
Before iteration 1
$S = \{\}$
$R = []$

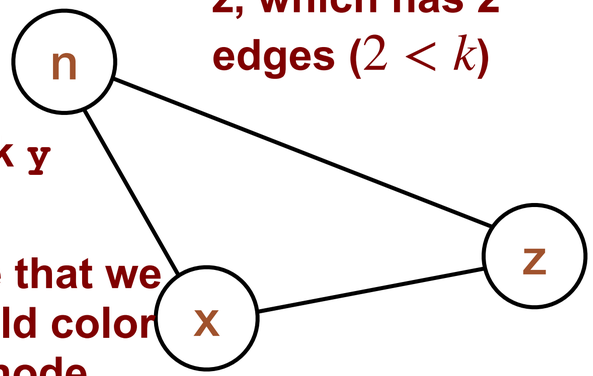


1. We cannot pick n or x

2. We can pick y or z, which has 2 edges ($2 < k$)

Simplify:
After iteration 1
$S = \{\}$
$R = [(\text{y}, \_, \text{color})]$

3. We pick y



To save space, we do not write the edges

State that we should color this node

Register Allocation Using Graph Coloring

Algorithm (Graph Coloring by Simplification)

The machine has k number of physical registers

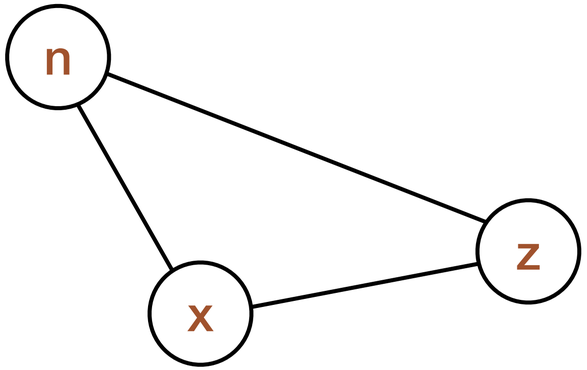
0. $S \leftarrow \emptyset$; $R \leftarrow$ Empty stack
1. **Construct** the interference graph G
   using liveness analysis, assuming S actual spill.
   $S' \leftarrow S$
2. **Simplify** by graph coloring heuristic:

repeat
  if there exists a node $v \in V$
    where $|edges(v)| < k$ then
    push $(v, edges(v), \text{color})$ on stack R
    $G \leftarrow (V - \{v\}, E - edges(v))$
  else
    push $(w, edges(w), \text{spill})$ on stack S, where $w \in V$
    $G \leftarrow (V - \{w\}, E - edges(w))$
until $G = (\emptyset, \emptyset)$

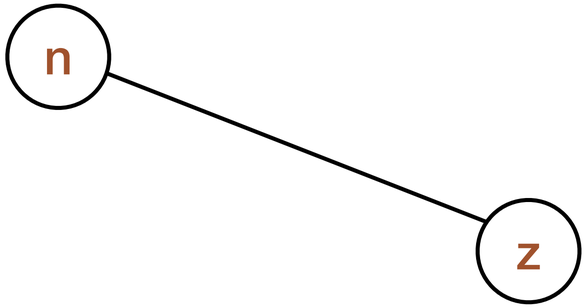
Example: Assume we have variables x, y, n, z, and $k = 3$ physical registers

Simplify:
Before iteration 2
$S = \{\}$
$R = [(\text{y}, \_, \text{color})]$

We can pick any node. Let's pic x.



Simplify:
After iteration 2
$S = \{\}$
$R = [(\text{x}, \_, \text{color}),$
$(\text{y}, \_, \text{color})]$



Register Allocation Using Graph Coloring

Algorithm (Graph Coloring by Simplification)

The machine has k number of physical registers

0. $S \leftarrow \emptyset$; $R \leftarrow$ Empty stack
1. **Construct** the interference graph G
   using liveness analysis, assuming S actual spill.
   $S' \leftarrow S$
2. **Simplify** by graph coloring heuristic:

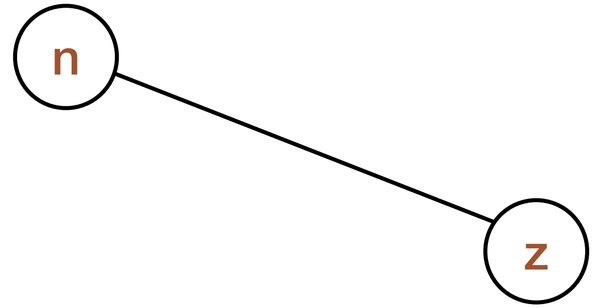
repeat
  if there exists a node $v \in V$
    where $|edges(v)| < k$ then
    push $(v, edges(v), \text{color})$ on stack R
    $G \leftarrow (V - \{v\}, E - edges(v))$
  else
    push $(w, edges(w), \text{spill})$ on stack S, where $w \in V$
    $G \leftarrow (V - \{w\}, E - edges(w))$
until $G = (\emptyset, \emptyset)$

Example: Assume we have variables x, y, n, z, and $k = 3$ physical registers

Simplify:
Before iteration 3

$S = \{\}$

$R = [(\text{x}, \_, \text{color}),$
$(\text{y}, \_, \text{color})]$



We proceed with step 3 and 4 in the same way.

Simplify:
After iteration 4

$S = \{\}$

$R = [(\text{n}, \_, \text{color}),$
$(\text{z}, \_, \text{color}),$
$(\text{x}, \_, \text{color}),$
$(\text{y}, \_, \text{color})]$

Register Allocation Using Graph Coloring

Assign:
Before iteration 1

$S = \{\}$

$R = [(\text{n}, \_, \text{color}),$
$(\text{z}, \_, \text{color}),$
$(\text{x}, \_, \text{color}),$
$(\text{y}, \_, \text{color})]$

Assign:
After iteration 1

$S = \{\}$

$R = [(\text{z}, \_, \text{color}),$
$(\text{x}, \_, \text{color}),$
$(\text{y}, \_, \text{color})]$

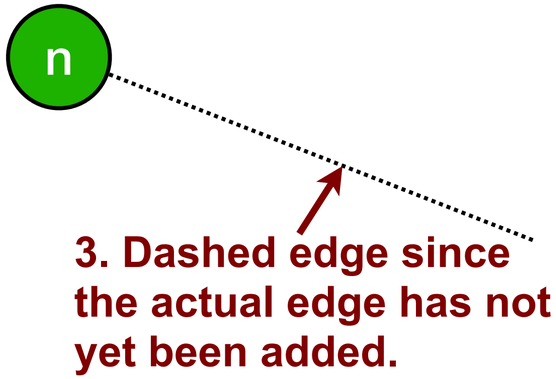
$c = \{n \mapsto 1\}$

$t = (\text{n}, \_, \text{color})$

1. Pop from the stack, gets into the true branch (we should color)

2. Note that G is empty since the last step.

Pick a fresh register (1 in this case). Assigns register 1 to variable n



3. Dashed edge since the actual edge has not yet been added.

3. **Assign** colors (registers) or actual spills.

4. We have no neighbor vertex with the same color

while(stack R is not empty)

$t \leftarrow$ pop item from stack R
if t matches (v, E', color) then
$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$
$c[v] \leftarrow n$ where $\forall w \in edges(v) \,.\, n \neq c[w]$
else t matches (v, E', spill)
if $|\{c[w] \;:\; w \in strip(edges(v), V)\}| < k$ then
$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$
$c[v] \leftarrow n$ where $\forall w \in strip(edges(v), V) \,.\, n \neq c[w]$
else
$S \leftarrow S \cup \{v\}$

4. **Restart** at step 1 if $S \neq S'$

$$strip(E, V) = \{(x, y) : (x, y) \in E \wedge x \in V \wedge y \in V\}$$

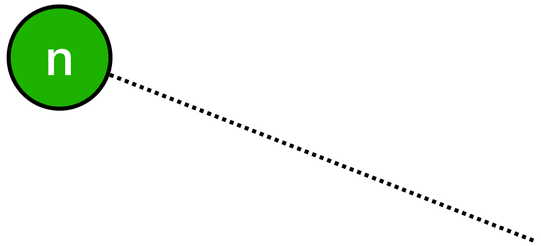
Register Allocation Using Graph Coloring

Assign:
Before iteration 2

$S = \{\}$

$R = [(\text{z}, \_, \text{color}),$
$(\text{x}, \_, \text{color}),$
$(\text{y}, \_, \text{color})]$

$c = \{n \mapsto 1\}$



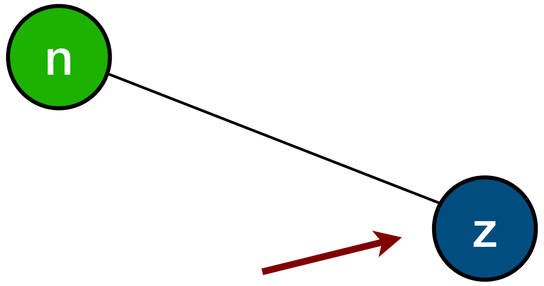
$t = (\text{z}, \_, \text{color})$

Assign:
After iteration 2

$S = \{\}$

$R = [(\text{x}, \_, \text{color}),$
$(\text{y}, \_, \text{color})]$

$c = \{n \mapsto 1, z \mapsto 2\}$



Z is colored and given register 2

Register 1 **Register 2**

3. **Assign** colors (registers) or actual spills.

while(stack$R$ is not empty)

$t \leftarrow$ pop item from stack R

if t matches (v, E', color) then

$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$

$c[v] \leftarrow n$ where $\forall w \in edges(v) \,.\, n \neq c[w]$

else t matches (v, E', spill)

if $|\{c[w] \;:\; w \in strip(edges(v), V)\}| < k$ then

$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$

$c[v] \leftarrow n$ where $\forall w \in strip(edges(v), V) \,.\, n \neq c[w]$

else

$S \leftarrow S \cup \{v\}$

4. **Restart** at step 1 if $S \neq S'$

$$strip(E, V) = \{(x, y) : (x, y) \in E \wedge x \in V \wedge y \in V\}$$

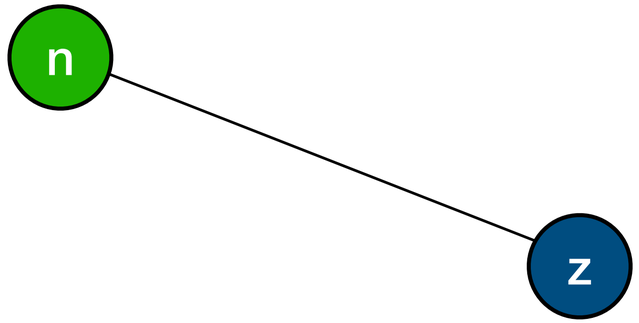
Register Allocation Using Graph Coloring

Assign:
Before iteration 3

$S = \{\}$

$R = [(\text{x}, \_, \text{color}),$
$(\text{y}, \_, \text{color})]$

$c = \{n \mapsto 1,\ z \mapsto 2\}$ $\quad t = (\text{x}, \_, \text{color})$

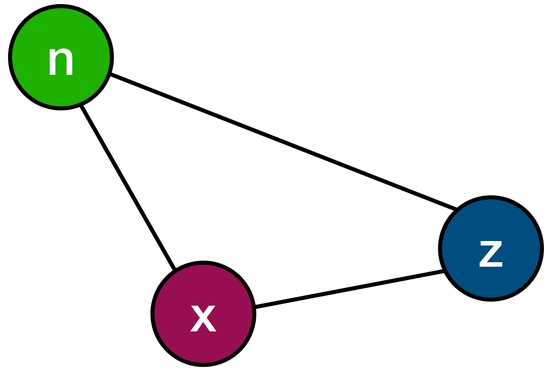


Assign:
After iteration 3

$S = \{\}$

$R = [(\text{y}, \_, \text{color})]$

$c = \{n \mapsto 1,\ z \mapsto 2\ \ x \mapsto 3\}$



Register 1 Register 2 Register 3

3. **Assign** colors (registers) or actual spills.

while(stack$R$ is not empty)

$t \leftarrow$ pop item from stack R

if t matches (v, E', color) then

$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$

$c[v] \leftarrow n$ where $\forall w \in edges(v)\,.\,n \neq c[w]$

else t matches (v, E', spill)

if $|\{c[w] \ : \ w \in strip(edges(v), V)\}| < k$ then

$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$

$c[v] \leftarrow n$ where $\forall w \in strip(edges(v), V)\,.\,n \neq c[w]$

else

$S \leftarrow S \cup \{v\}$

4. **Restart** at step 1 if $S \neq S'$

$$strip(E, V) = \{(x, y) : (x, y) \in E \wedge x \in V \wedge y \in V\}$$

Register Allocation Using Graph Coloring

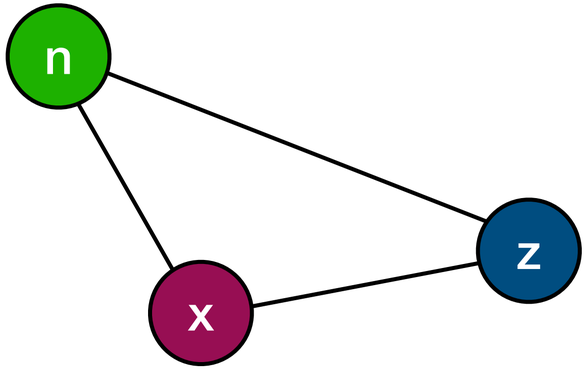
Assign:
Before iteration 4

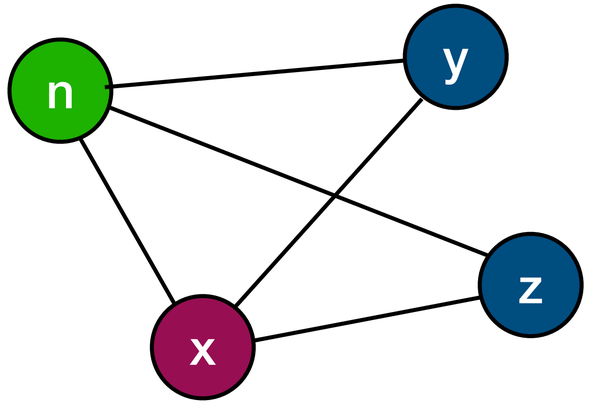
$S = \{\}$

$R = [(\text{y}, \_, \text{color})]$

$c = \{n \mapsto 1,\ z \mapsto 2\ \ x \mapsto 3\}$

$t = (\text{y}, \_, \text{color})$





Register 1 **Register 2** **Register 3**

3. **Assign** colors (registers) or actual spills.

while(stack$R$ is not empty)

$t \leftarrow$ pop item from stack R

if t matches (v, E', color) then

$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$

$c[v] \leftarrow n$ where $\forall w \in edges(v)\,.\,n \neq c[w]$

else t matches (v, E', spill)

if $|\{c[w] \ : \ w \in strip(edges(v), V)\}| < k$ then

$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$

$c[v] \leftarrow n$ where $\forall w \in strip(edges(v), V)\,.\,n \neq c[w]$

else

$S \leftarrow S \cup \{v\}$

4. **Restart** at step 1 if $S \neq S'$

$$strip(E, V) = \{(x, y) : (x, y) \in E \wedge x \in V \wedge y \in V\}$$

Exercise: Which color (register) should y have?
Must be blue, already connected to n and x

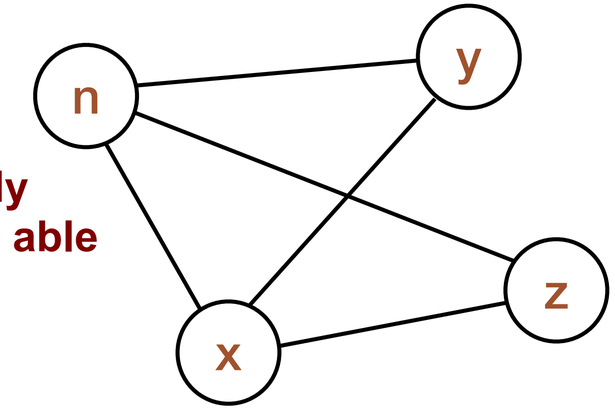
Register Allocation Using Graph Coloring

Algorithm (Graph Coloring by Simplification)

The machine has k number of physical registers

0. $S \leftarrow \emptyset$; $R \leftarrow$ Empty stack
1. **Construct** the interference graph G
   using liveness analysis, assuming S actual spill.
   $S' \leftarrow S$
2. **Simplify** by graph coloring heuristic:
   repeat
      if there exists a node $v \in V$
                  where $|edges(v)| < k$ then
         push $(v, edges(v), \text{color})$ on stack R
         $G \leftarrow (V - \{v\}, E - edges(v))$
      else
         push $(w, edges(w), \text{spill})$ on stack S, where $w \in V$
         $G \leftarrow (V - \{w\}, E - edges(w))$
   until $G = (\emptyset, \emptyset)$

Example: Assume we have variables x, y, n, z, and $k = 2$ physical registers.



We can see directly that we will not be able to color without spilling.

We can select any node for spill, but it is good to select a node with high degree.

Pushes on the stack in the same way, but mark each element as potential `spill`.

Register Allocation Using Graph Coloring

3. **Assign** colors (registers) or actual spills.

while(stack$R$ is not empty)

$t \leftarrow$ pop item from stack R

if t matches (v, E', color) then

$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$

$c[v] \leftarrow n$ where $\forall w \in edges(v) \,.\, n \neq c[w]$

else t matches (v, E', spill)

if $|\{c[w] \;:\; w \in strip(edges(v), V)\}| < k$ then

$G \leftarrow (V \cup \{v\}, E \cup strip(E', V))$

$c[v] \leftarrow n$ where $\forall w \in strip(edges(v), V) \,.\, n \neq c[w]$

else

$S \leftarrow S \cup \{v\}$

4. **Restart** at step 1 if $S \neq S'$

$$strip(E, V) = \{(x, y) : (x, y) \in E \wedge x \in V \wedge y \in V\}$$

When spilled potential `stack` elements are discovered, we can do optimistic coloring (it might not need to be spilled.

If not, it is an actual spill, which is marked in set S.

If we have new spills, we start at step 1, and redo the whole liveness analysis.

Exercise: Why do we restart?

Answer: Spilling uses new temporaries, which will interfere with other variables (small live ranges though).

Handling of special registers?

How should we handle the following x86 registers?

Parameter registers: following the standard calling convention, registers rdi, rsi, rdx, rcx, r8 and r9 cannot easily be used as a general purpose register.

Special register. For instance, rax and rdx for multiplication, cl for shifting, rsp for stack pointer etc.

Caller-save and callee-save registers. If a variable is live across several function calls, callee-saved registers should be used, but if it is not live across functions, caller-saved registers should be used.

In both cases, these physical registers cannot be used for coloring.

Two cases:

1. When they hold the parameter values of the function being called.
2. When a function is placing argument values in these registers and calls another function.

Note: only when these variables are live!

Note: in x86, all registers that are not callee-saved can be seen as caller saved.

Precolored Virtual Registers

Solution: Precoloring

Parameter registers: following the standard calling convention, registers rdi, rsi, rdx, rcx, r8 and r9 cannot easily be used as a general purpose register.

Special register. For instance, rax and rdx for multiplication, cl for shifting, rsp for stack pointer etc.

Caller-save and callee-save registers. If a variable is live across several function calls, callee-saved registers should be used, but if it is not live across functions, caller-saved registers should be used.

This is how parameter registers and special registers are included.

Forces variables that are live across function called to use callee saved registers.

Procedure (Precoloring of nodes)

1. Before liveness analysis, include all physical registers also as special *precolored variables*.
2. Include the precolored variables when creating def and use sets.
3. Let all `call` instructions interfere with caller-save registers (by letting the interfering variables be in the def set for the `call` instruction.
4. Before graph coloring, give all pre-colored nodes (variables) a unique color (hence the name).
5. In the interference graph, make all pre-colored nodes interfere with each other.

Coalescing

Instruction selection can result in many extra `mov` instructions.

Removing redundant `mov` instructions, and thus using the same registers is called coalescing.

Intuition:

For a given `mov` instruction, if there is no edge in the interference graph for the source and destination registers, then the two registers do not interfere.

Conclusion:
the nodes for the source and destination can be coalesced = a new node where the edges for the new node are the union of the two previous nodes' edges.

Problem: coalesced nodes are more constrained. Graphs may not be colourable any more.

Register allocation with coalescing:

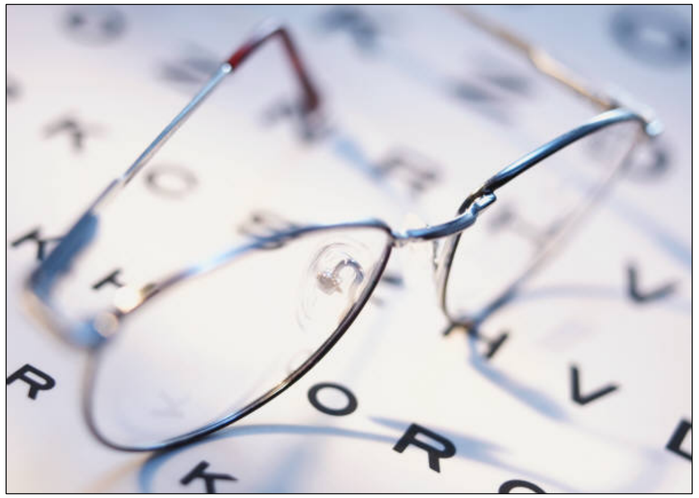
A more complicated approach, where simplification and coalescing typically are performed in an iterative manner.

Some more hacking, soon?

Part I
Liveness Analysis

Part II
Register Allocation using Graph Coloring

Reading Guidelines - Module 3



- Register allocation C&T Chapters 13.1-13.4.
- Instruction Scheduling 12.1 - 1.2
- Data-flow analysis C&T Chapters 9.1 - 9.2
- Static Single Assignment (SSA) C&T Chapter 9.3
- Optimization C&T Chapters 8.1 - 8.7 See as background reading.

C&T = the course book by Cooper and Torczon

Reading Guidelines
See the course webpage
for more information.

Conclusions

Some key take away points:

- **Liveness analysis** can be used in many places in a compiler. Goal: to find when variables are live.

- **Register allocation using graph coloring** is a standard technique for register allocation. It makes use of an interference graph.

Thanks for listening!