Range Analysis Domain for BigInt

Domain values D include:

- bounded intervals of integers: [a,b]
- intervals unbounded from one side $(-\infty,b]$, $[a,\infty)$
- empty set, denoted $\bot$
- the set of all integers, denoted T

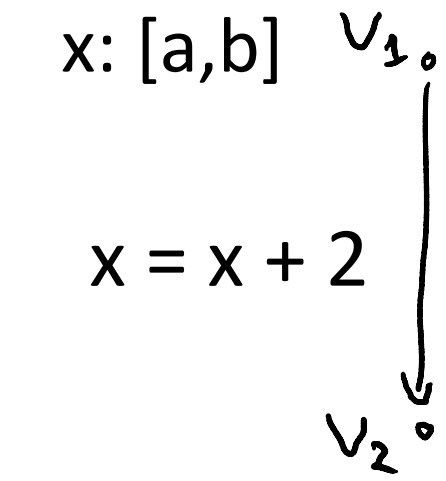
Formally, if **Z** denotes integers, then

$$D = \{\bot,T\} \cup \{ [a,b] \mid a,b \in \mathbf{Z}\}$$
$$\cup \{(-\infty,b] \mid b \in \mathbf{Z}\}$$
$$\cup \{[a,\infty) \mid a \in \mathbf{Z}\}$$

D is an infinite set!

Sequences in Analysis are Monotonically Growing

Transfer functions (describe how statements affect elements of D) should be monotonic:
if we start with a representation of a larger set of states, the representation of the resulting set of states should also be larger



$d_1 \le d_2$ implies $[[x=x+2]](d_1) \le [[x=x+2]](d_2)$

We start from $\bot$ everywhere except entry
So in first step, the values can only grow
$\bot \le d_2$ implies $[[x=x+2]](\bot) \le [[x=x+2]](d_2)$
Values computed in second step are also bigger
If $x_n = F^n(\bot)$ then $x_0 = \bot \le x_1$

$x_n \le x_{n+1}$ / F

$F(x_n) \le F(x_{n+1})$ i.e. $x_{n+1} \le x_{n+2}$

How Long Does Analysis Take?

- We explore this question by comparing
  - range analysis: maintain intervals
  - constant propagation:
    maintains indication whether the value is constant

Iterating Range Analysis

Find the **number** of updates **range analysis** needs to **stabilize** in the following code

```
x = 1
n = 1000
while (x < n) {
  x = x + 2
}
```

Iterating Range Analysis

Now, if we assume that any number can be entered from the user, what is now the number of steps?

```
x = 1
n = readInput()  //anything, so n becomes T
while (x < n) {
  x = x + 2
}
```

For unknown program inputs and unbounded domains, such analysis need not terminate!

One solution: “smaller” domain

Range Analysis with Finite Set of Endpoints

Pick a set W of "interesting" interval end-points

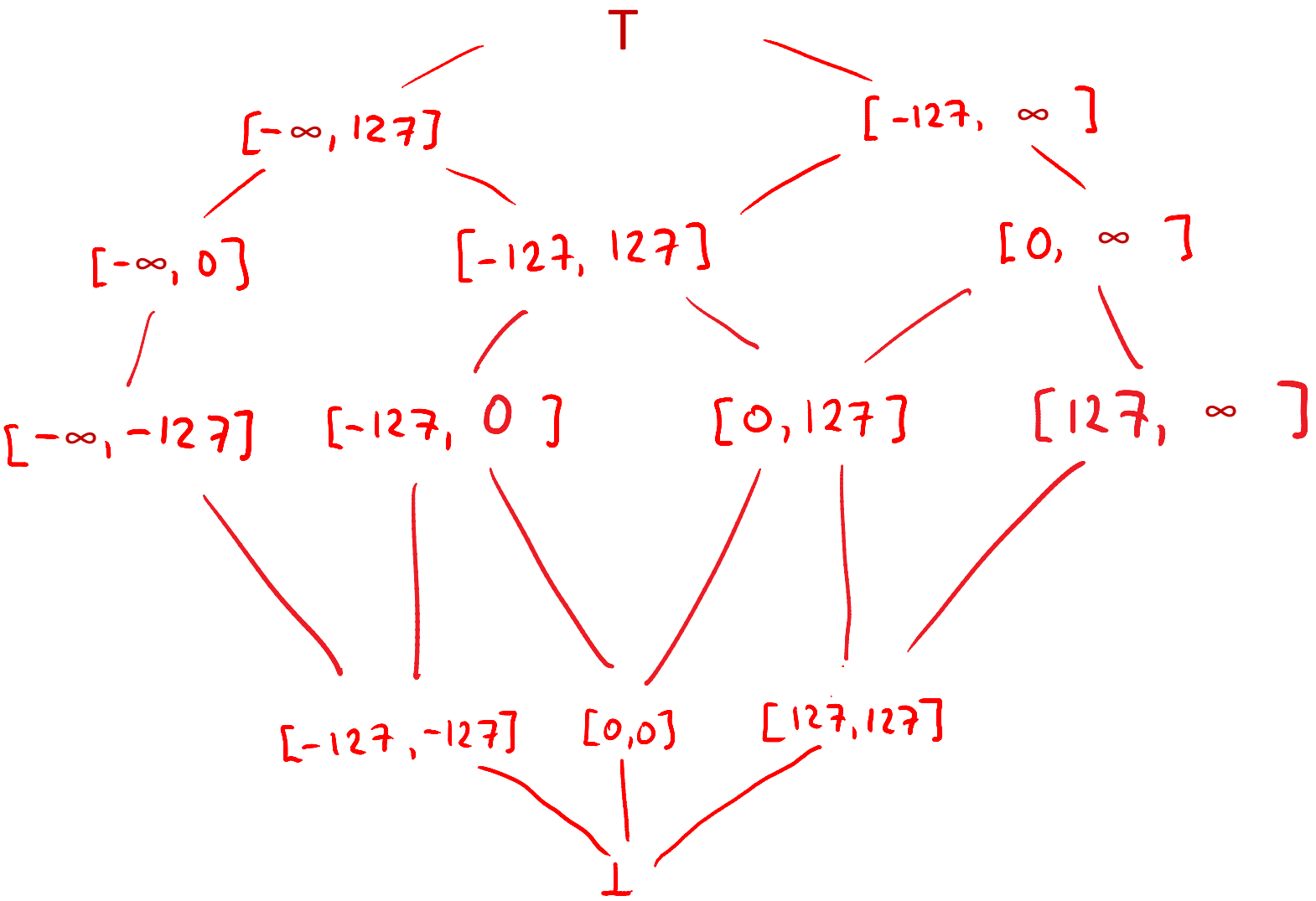
Example:

$$W = \{-128, 0, 127\}$$

$$D = \{\bot, T\} \cup \{ [a,b] \mid a,b \in W, a \leq b\} \cup \{(-\infty,b] \mid b \in W\} \cup \{[a,\infty) \mid a \in W\}$$

D is a finite set! How many elements does it have?

Domain Lattice Diagram for this W: 14 elements



Re-Run Analysis with Finite Endpoint Set

What is the number of updates?

```
x = 1
n = 1000
while (x < n) {
  x = x + 2
}
```

```
x = 1
n = readInt()
while (x < n) {
  x = x + 2
}
```

Constant Propagation Domain

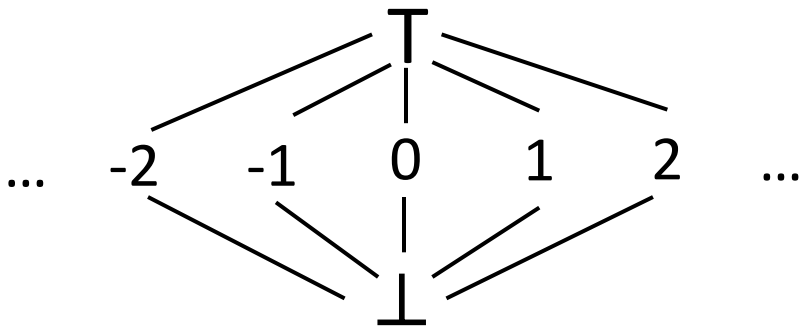
Domain values D are:

- intervals [a,a], denoted simply ‘a’
- empty set, denoted ⊥ and set of all integers T

Formally, if **Z** denotes integers, then

$$D = \{\bot, T\} \cup \{ a \mid a \in \mathbf{Z} \}$$

D is an infinite set



Constant Propagation Transfer Functions

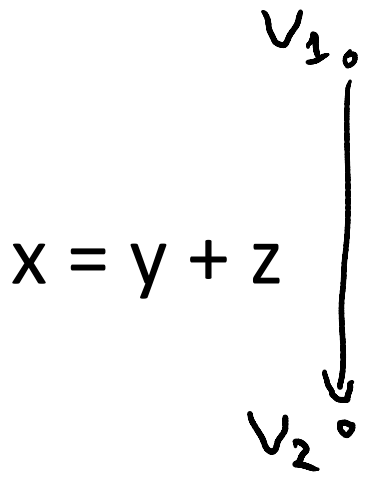
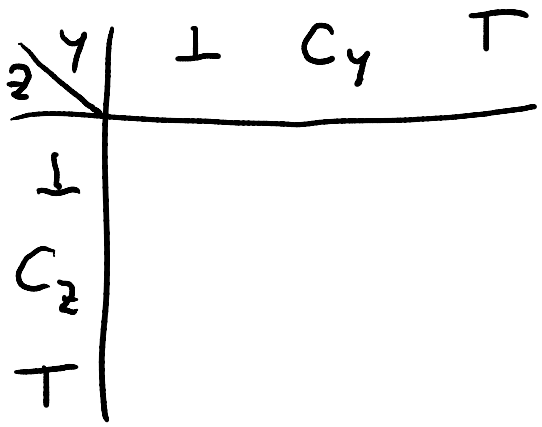


table for +:



For **each variable** (x,y,z) and **each CFG node (program point)** we store: ⊥ , a constant, or ⊤

```
abstract class Element
case class Top extends Element
case class Bot extends Element
case class Const(v:Int) extends Element
var facts : Map[Nodes,Map[VarNames,Element]]
```

what executes during analysis of x=y+z :

```
oldY = facts(v1)("y")
oldZ = facts(v1)("z")
newX = tableForPlus(oldY, oldZ)
facts(v2) = facts(v2) join facts(v1).updated("x", newX)
```

```
def tableForPlus(y:Element, z:Element) =
(x,y) match {
case (Const(cy),Const(cz)) => Const(cy+cz)
case (Bot,_) => Bot
case (_,Bot) => Bot
case (Top,Const(cz)) => Top
case (Const(cy),Top) => Top
}
```

Run Constant Propagation

What is the number of updates?

```
x = 1
n = 1000
while (x < n) {
  x = x + 2
}
```

```
x = 1
n = readInt()
while (x < n) {
  x = x + 2
}
```

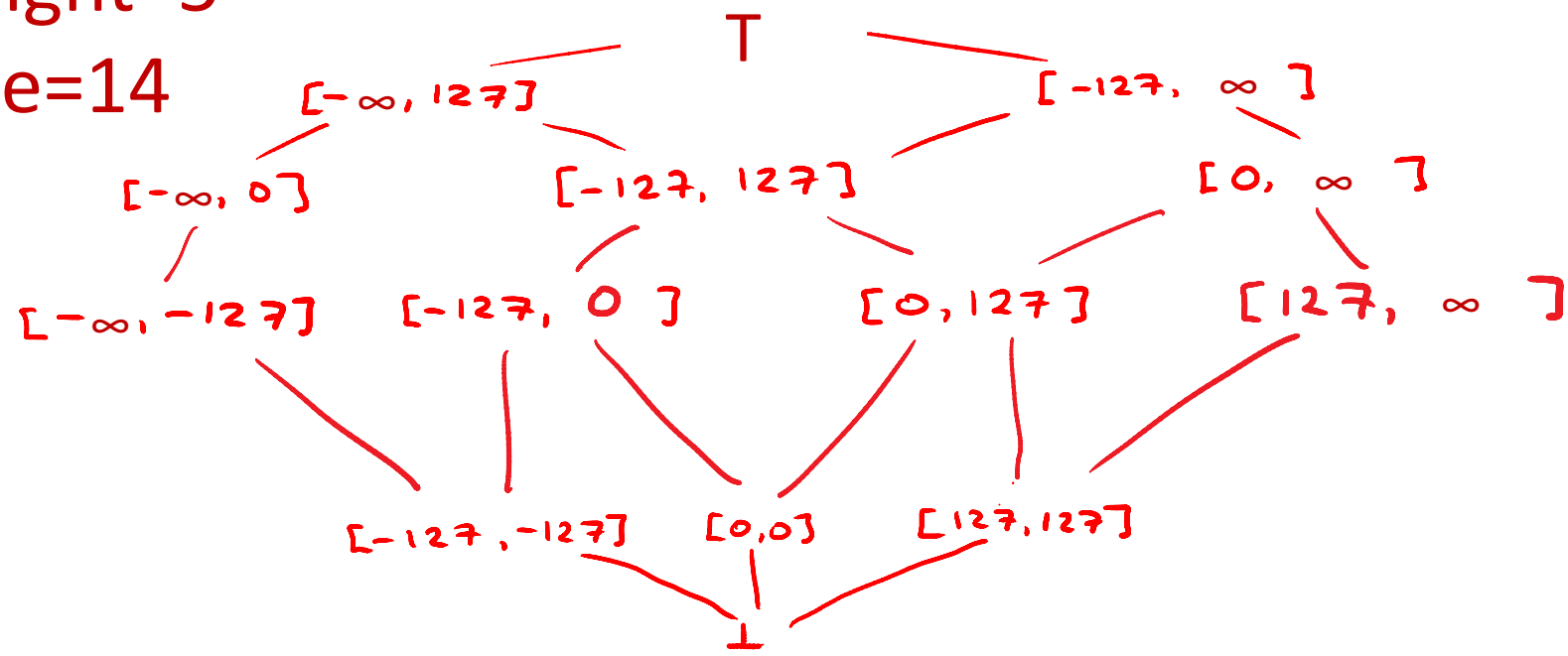
Observe

- Range analysis with W = {-128, 0, 127} has a finite domain
- Constant propagation has infinite domain (for every integer constant, one element)
- Yet, constant propagation finishes sooner!
  - it is not about the size of the domain
  - it is about the height

Height of Lattice: Length of Max. Chain

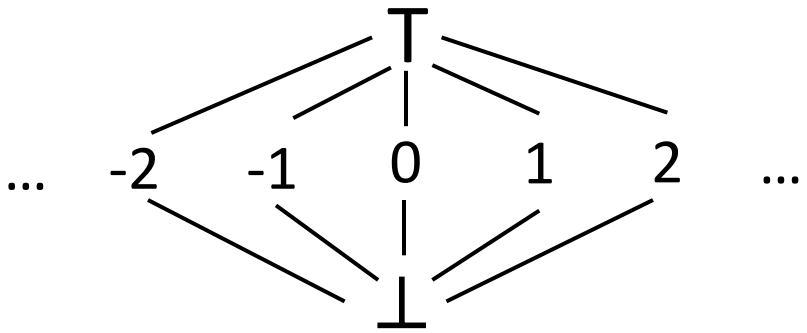
height=5

size=14



height=2

size =∞



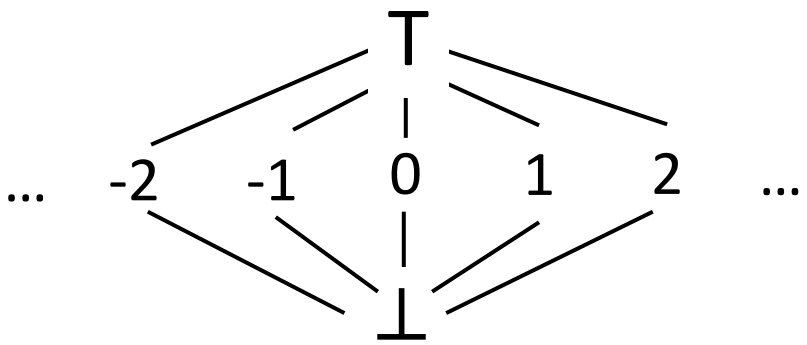
Chain of Length n

- A set of elements $x_0, x_1, \ldots, x_n$ in D that are linearly ordered, that is $x_0 < x_1 < \ldots < x_n$
- A lattice can have many chains. Its **height** is the maximum n for all the chains
- If there is no upper bound on lengths of chains, we say lattice has **infinite height**
- Any monotonic sequence of distinct elements has length at most equal to lattice height
  - including sequence occuring during analysis!
  - **such sequences are always monotonic**

In constant propagation, each value can change only twice

height=2

size =∞



consider value for x before assignment

- Initially: ⊥
- changes 1st time to: 1
- change 2nd time to: ⊥

total changes: two (height)

```
x = 1
n = 1000
while (x < n) {
  x = x + 2
}
```

var facts : **Map[Nodes,Map[VarNames,Element]]**

Total number of changes bounded by: **height·|Nodes| ·|Vars|**

Exercise

$\mathbf{B_{32}}$ – the set of all 32-bit integers

What is the upper bound for number of changes in the entire analysis for:

- 3 variables,
- 7 program points

for these two analyses:

1) constant propagation for constants from $\mathbf{B}_{32}$

2) The following domain D:

$$D = \{\bot\} \cup \{ [a,b] \mid a,b \in \mathbf{B_{32}}, a \leq b\}$$

Height of B_{32}

$D = \{\bot\} \cup \{ [a,b] \mid a,b \in \mathbf{B_{32}}, a \leq b\}$

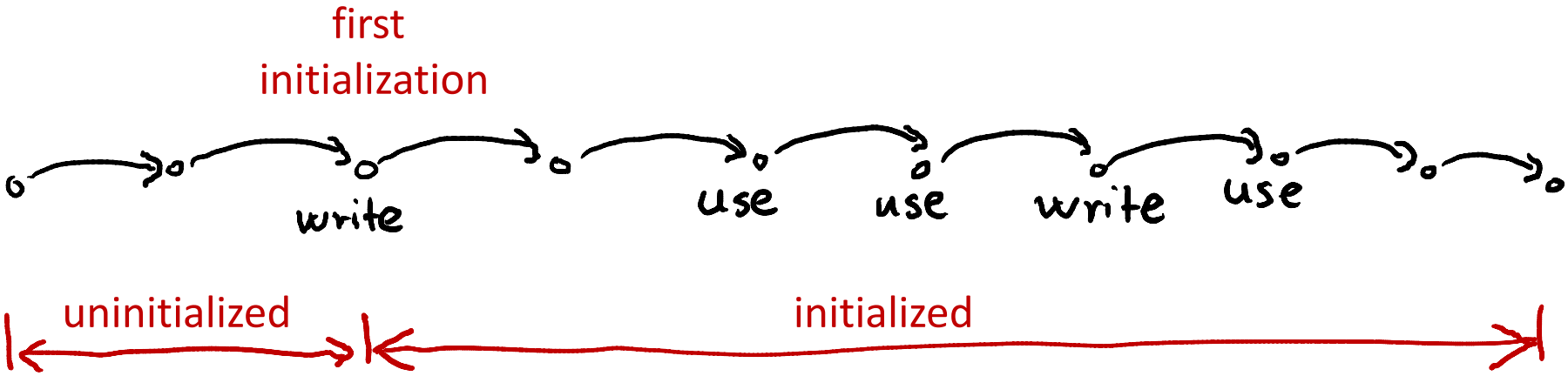
One possible chain of maximal length:

$\bot$

…

[MinInt,MaxInt]

Initialization Analysis



What does javac say to this:

```
class Test {
   static void test(int p) {
         int n;
         p = p - 1;
         if (p > 0) {
            n = 100;
         }
         while (n != 0) {
            System.out.println(n);
            n = n - p;
         }
   }
}
```

```
Test.java:8: variable n might not have been initialized
            while (n > 0) {
            ^
1 error
```

Program that compiles in java

```
class Test {
   static void test(int p) {
         int n;
         p = p - 1;
         if (p > 0) {
            n = 100;
         }
         else {
            n = -100;
         }
         while (n != 0) {
            System.out.println(n);
            n = n - p;
         }
   }
}
```

We would like variables to be initialized on all execution paths.

Otherwise, the program execution could be undesirable affected by the value that was in the variable initially.

We can enforce such check using initialization analysis.

What does javac say to this?

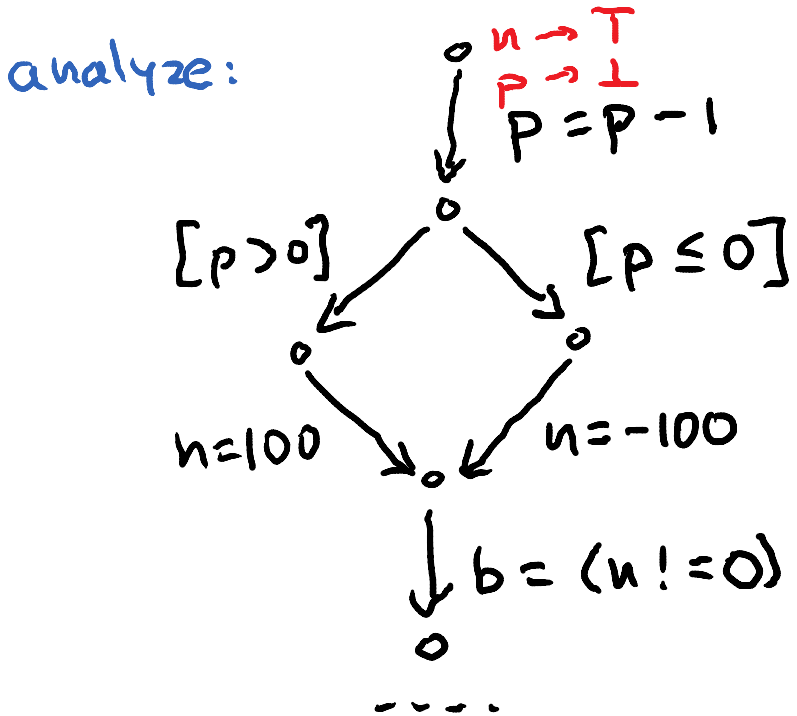
```
static void test(int p) {
        int n;
        p = p - 1;
        if (p > 0) {
            n = 100;
        }
        System.out.println("Hello!");
        if (p > 0) {
            while (n != 0) {
                System.out.println(n);
                n = n - p;
            }
        }
}
```

Initialization Analysis

```
class Test {
    static void test(int p) {
            int n;
            p = p - 1;
            if (p > 0) {
                n = 100;
            }
            else {
                n = -100;
            }
            while (n != 0) {
                System.out.println(n);
                n = n - p;
            }
    }
}
```

T indicates presence of flow from states where variable was not initialized:

- If variable is **possibly uninitialized**, we use T
- Otherwise (initialized, or unreachable): ⊥



If var occurs anywhere but left-hand side of assignment and has value T, report error

Sketch of Initialization Analysis

- Domain: for each variable, for each program point: D = {⊥,T}
- At program entry, local variables: T ; parameters: ⊥
- At other program points: each variable: ⊥
- An assignment x = e sets variable x to ⊥
- lub (join) of any value with T gives T
  - uninitialized values are contagious along paths
  - ⊥ value for x means there is definitely no possibility for accessing uninitialized value of x

Run initialization analysis Ex.1

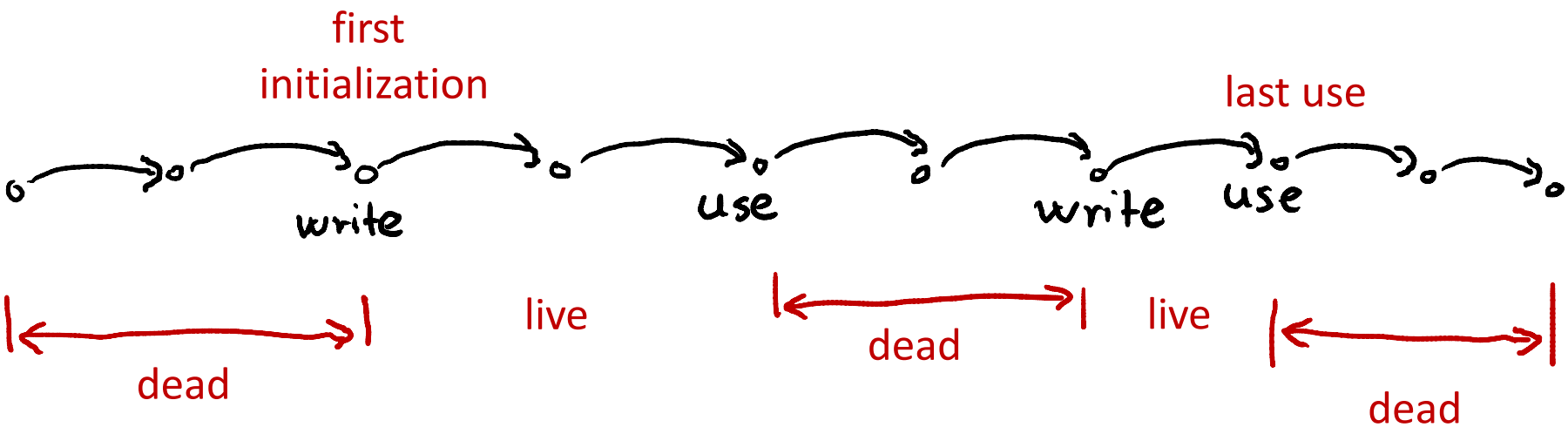
```
int n;
p = p - 1;
if (p > 0) {
    n = 100;
}
while (n != 0) {
    n = n - p;
}
```

Run initialization analysis Ex.2

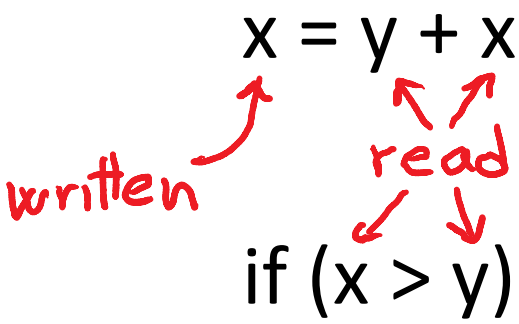
```
int n;
p = p - 1;
if (p > 0) {
    n = 100;
}
if (p > 0) {
    n = n - p;
}
```

Liveness Analysis

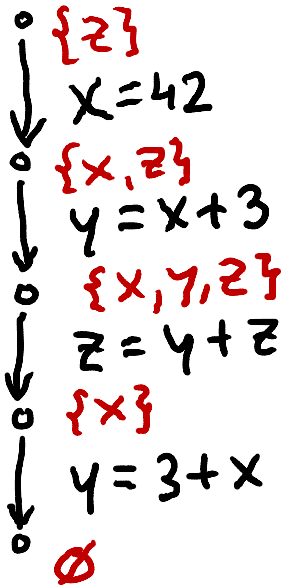
Variable is dead if its current value will not be used in the future. If there are no uses before it is reassigned or the execution ends, then the variable is surely dead at a given point.



What is Written and What Read



Example:



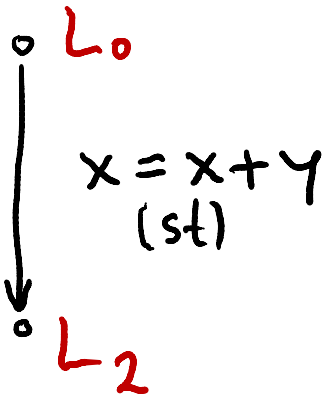
Purpose:

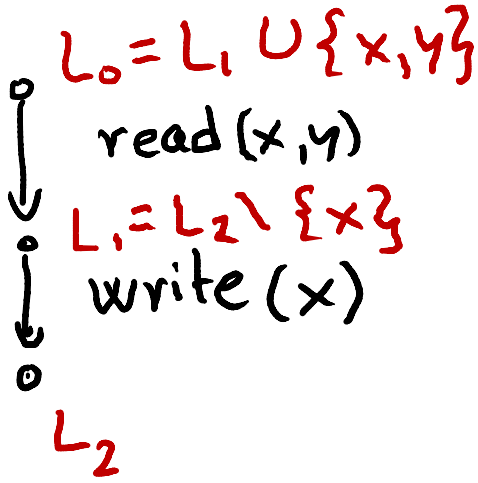
Register allocation:

find good way to decide which variable should go to which register at what point in time.

How Transfer Functions Look

L_c - set of live variables





$$L_0 = (L_2 \setminus \{x\}) \cup \{x, y\}$$

Generally

$$L_0 = (L_2 \setminus def(st)) \cup use(st)$$

Initialization: Forward Analysis

while (there was change)
  pick edge (v1,statmt,v2) from CFG
        such that facts(v1) has changed
  facts(v2)=facts(v2) **join** transferFun(statmt, facts(v1))
}

Liveness: Backward Analysis

while (there was change)
  pick edge (v1,statmt,v2) from CFG
        such that facts(v2) has changed
  facts(v1)=facts(v1) **join** transferFun(statmt, facts(v2))
}

Example

x = m[0]

y = m[1]

xy = x * y

z = m[2]

yz = y*z

xz = x*z

res1 = xy + yz

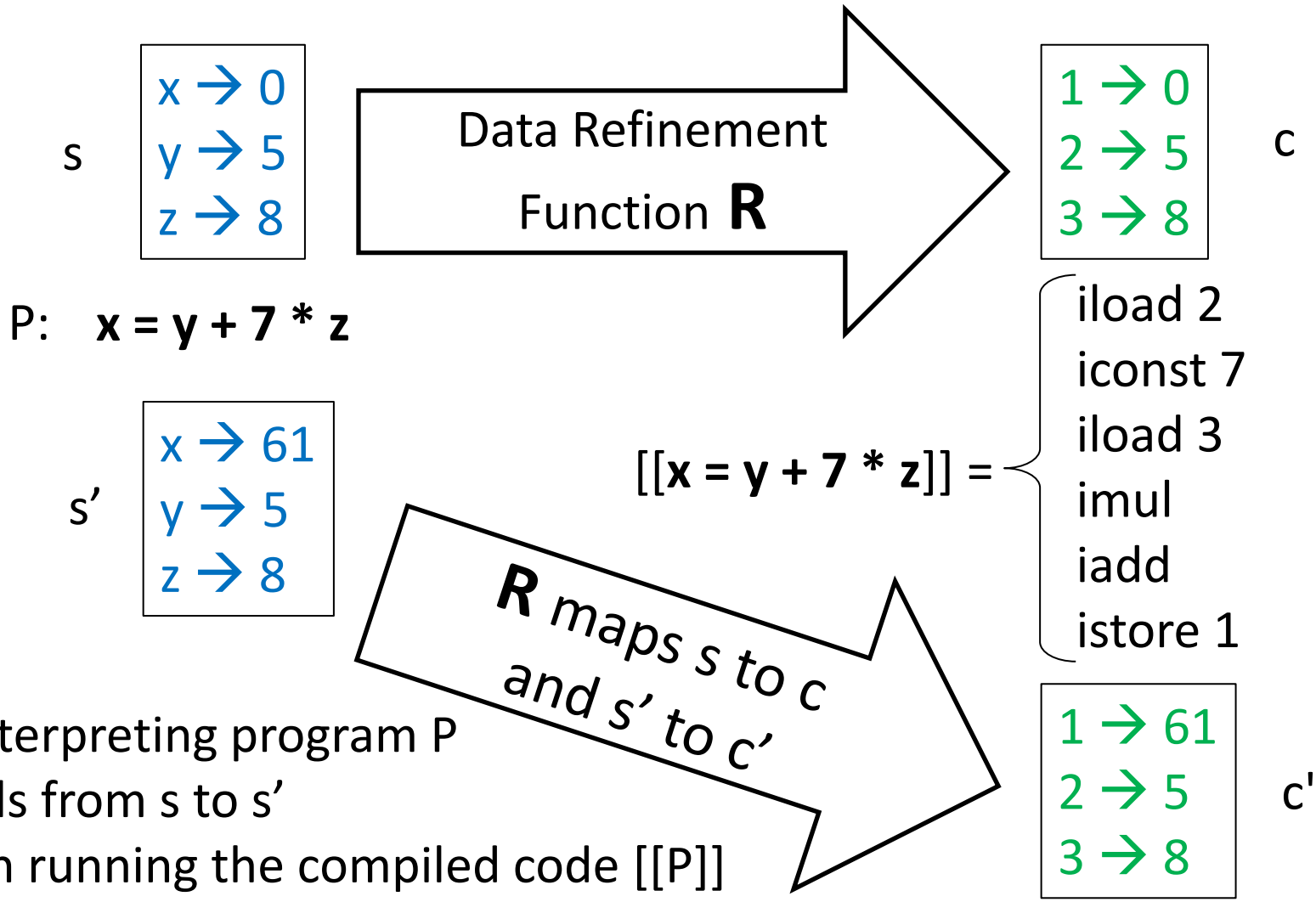
m[3] = res1 + xz

Data Representation Overview

Original and Target Program have Different Views of Program State

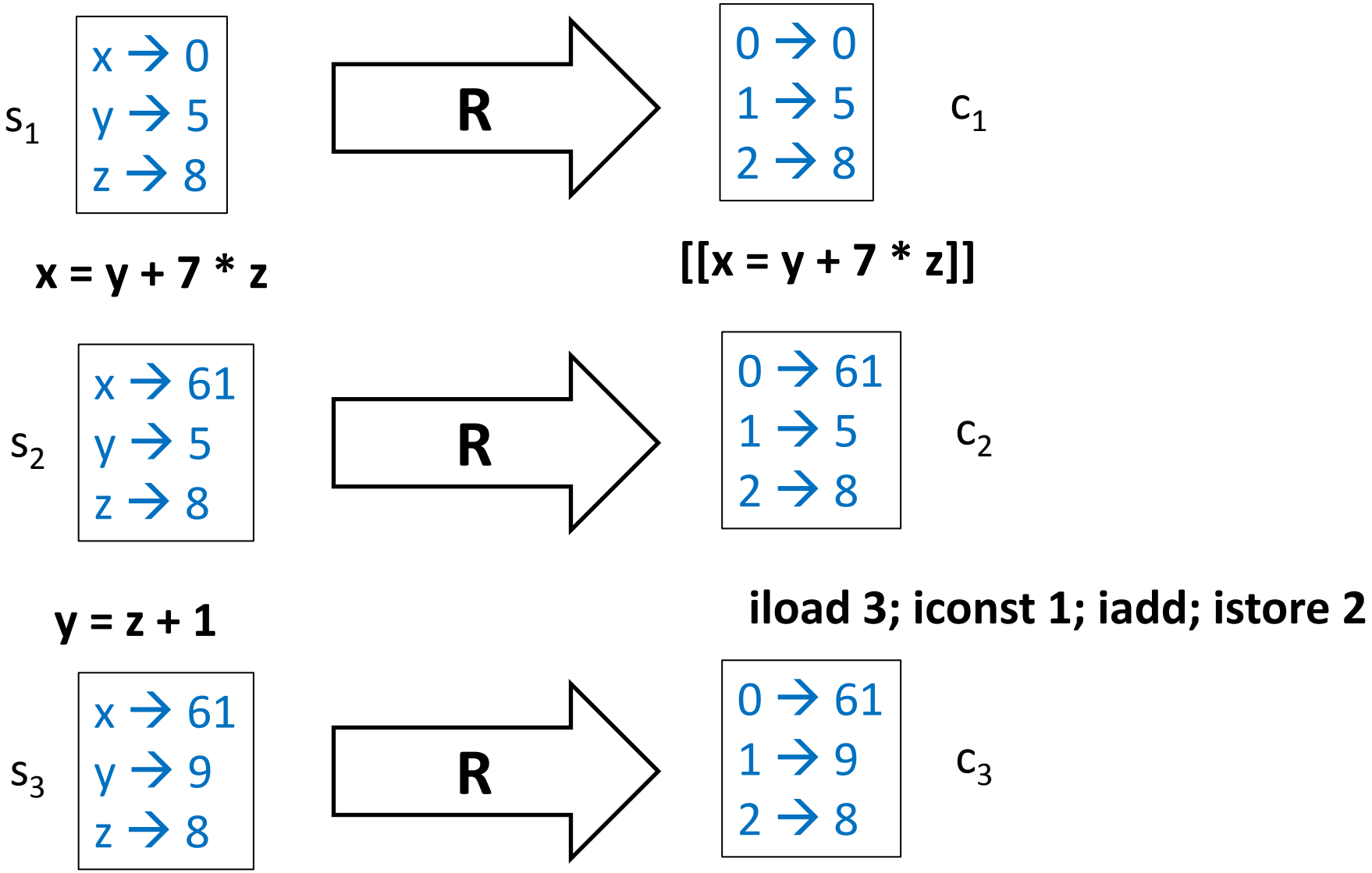
- Original program:
  - local variables given by names (any number of them)
  - each procedure execution has fresh space for its variables (even if it is recursive)
  - fields given by names
- Java Virtual Machine
  - local variables given by slots (0,1,2,…), any number
  - intermediate values stored in operand stack
  - each procedure **call** gets fresh slots and stack
  - fields given by names and object references
- **Machine code:** program state is a large arrays of bytes and a finite number of registers

Compilation Performs Automated Data Refinement



If interpreting program P
leads from s to s'
then running the compiled code [[P]]
leads from R(s) to R(s')

Inductive Argument for Correctness



(R may need to be a relation, not just function)

A Simple Theorem

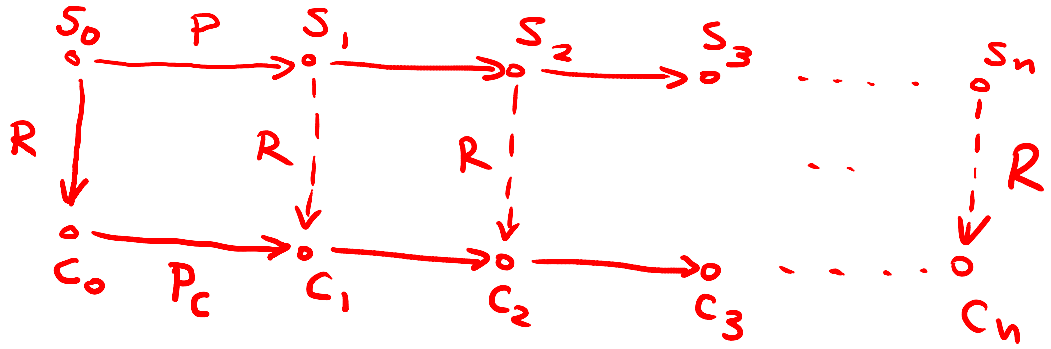
$P : S \rightarrow S$ is a program meaning function

$P_c : C \rightarrow C$ is meaning function for the compiled program

$R : S \rightarrow C$ is data representation function

Let $s_{n+1} = P(s_n)$, n = 0,1,… be interpreted execution

Let $c_{n+1} = P(c_n)$, n = 0,1,… be compiled execution



Theorem: If

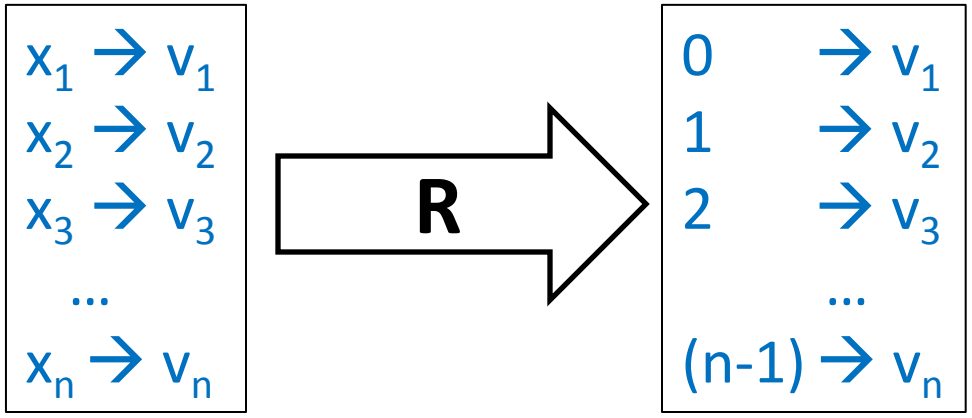
- $c_0 = R(s_0)$
- for all s, $P_c(R(s)) = R(P(s))$

then $c_n = R(c_n)$ for all n.

Proof: immediate, by induction. R is often called **simulation relation.**

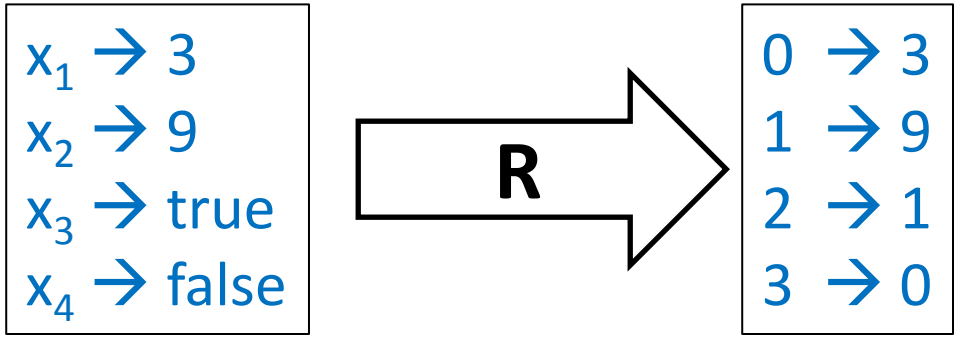
Example of a Simple R

- Let the receiver, the parameters, and local variables, in their order of declaration, be $x_1, x_2 \ldots x_n$
- Then R maps program state with only integers like this:



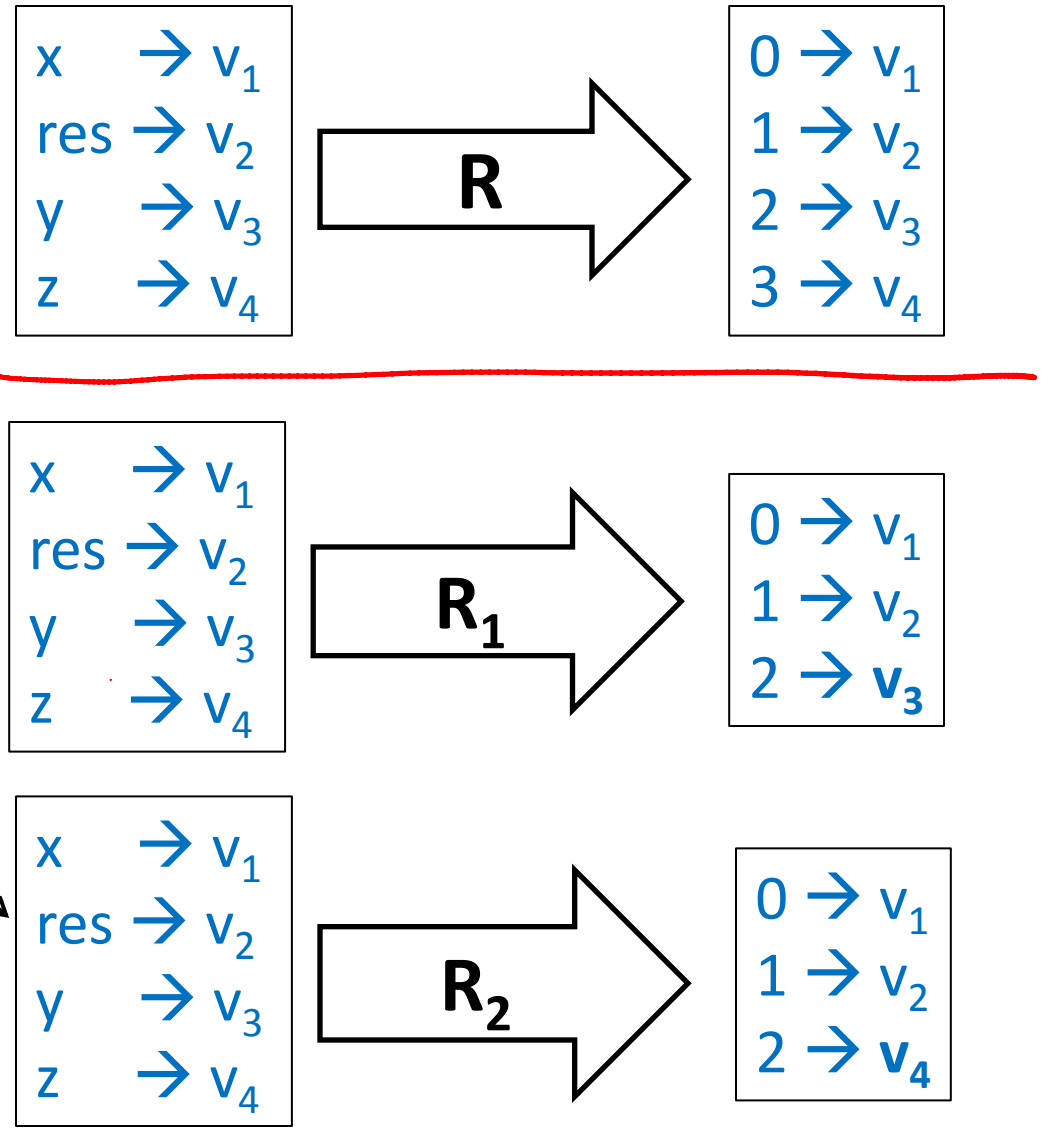
R for Booleans

- Let the received, the parameters, and local variables, in their order of declaration, be $x_1, x_2 \ldots x_n$
- Then R maps program state like this, where x_1 and x_2 are integers but x_3 and x_4 are Booleans:



R that depends on Program Point

```
def main(x:Int) {
  var res, y, z: Int
  if (x>0) {
    y = x + 1
    res = y
  } else {
     z = -x - 10
     res = z
  }
  return res;
}
```



Map y,z to same slot.

Consume fewer slots!

Packing Variables into Memory

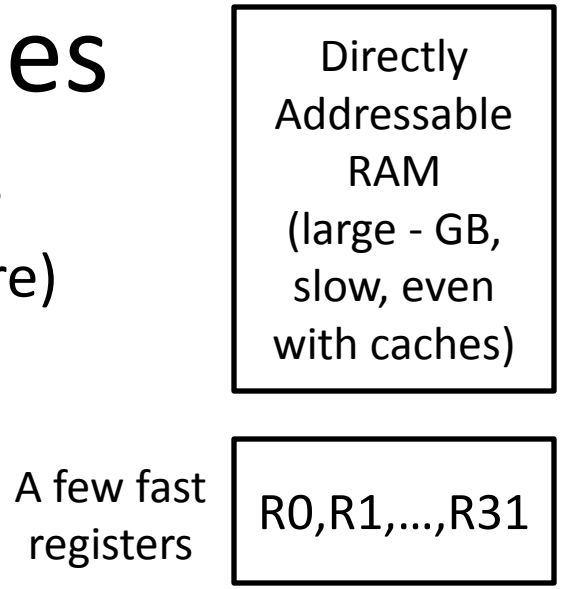
- If values are not used at the same time, we can store them in the same place
- This technique arises in
  - **Register allocation**: store frequently used values in a bounded number of fast registers
  - ‘malloc’ and ‘free’ manual memory management: *free* releases memory to be used for later objects
  - Garbage collection, e.g. for JVM, and .NET as well as languages that run on top of them (e.g. Scala)

Register Machines

Better for most purposes than stack machines

- closer to modern CPUs (RISC architecture)
- closer to control-flow graphs
- simpler than stack machine

Example: ARM architecture



Basic Instructions of Register Machines

$R_i \leftarrow Mem[R_j]$ load

$Mem[R_j] \leftarrow R_i$ store

$R_i \leftarrow R_j * R_k$ compute: for an operation *

Efficient register machine code uses as few loads and stores as possible.

State Mapped to Register Machine

Both dynamically allocated heap and stack expand

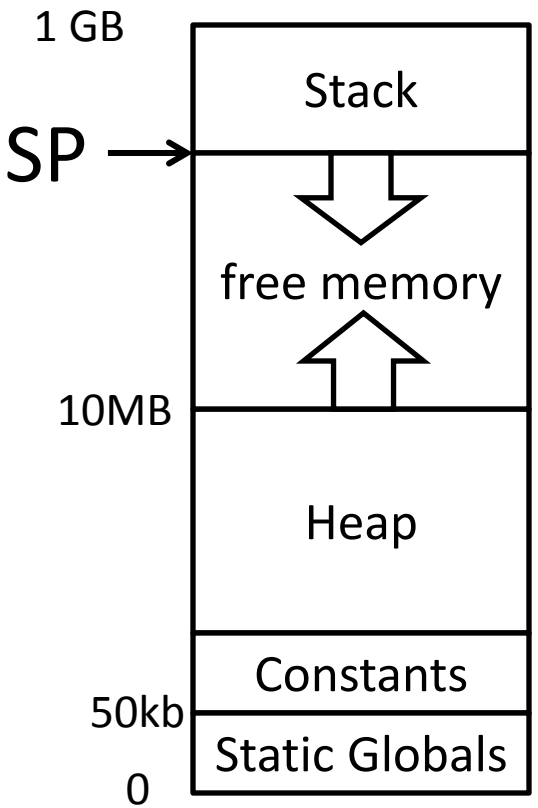
- heap need not be contiguous; can request more memory from the OS if needed
- stack grows downwards

Heap is more general:

- Can allocate, read/write, and deallocate, in any order
- Garbage Collector does deallocation automatically
  - Must be able to find free space among used one, group free blocks into larger ones (compaction),…

Stack is more efficient:

- allocation is simple: increment, decrement
- top of stack pointer (SP) is often a register
- if stack grows towards smaller addresses:
  - to allocate N bytes on stack (**push**): **SP := SP - N**
  - to deallocate N bytes on stack (**pop**): **SP := SP + N**



Exact picture may depend on hardware and OS

JVM vs General Register Machine Code Naïve Correct Translation

JVM:

imul

Register Machine:

R1 ← Mem[SP]

SP = SP + 4

R2 ← Mem[SP]

R2 ← R1 * R2

Mem[SP] ← R2

Register Allocation

How many variables?

x,y,z,xy,xz,res1

Do we need 6 distinct registers if we wish to avoid load and stores?

```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
xz = x*z
res1 = xy + yz
m[3] = res1 + xz
```

```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
y = x*z          // reuse y
x = xy + yz      // reuse x
m[3] = x + y
```

Idea of Graph Coloring

- Register Interference Graph (RIG):
  - indicates whether there exists a point of time where both variables are live
  - look at the sets of live variables at all progrma points after running live-variable analysis
  - if two variables occur together, draw an edge
  - we aim to assign different registers to such these variables
  - finding assignment of variables to K registers: corresponds to coloring graph using K colors

Graph Coloring Algorithm

Simplify
If there is a node with less than K neighbors, we will always be able to color it!
so we can remove it from the graph
This reduces graph size (it is incomplete)
e.g.Every planar can be colored by at most 4 colors (yet can have nodes with 100 neighbors)

Spill
If every node has K or more neighbors, we remove one of them
we mark it as node for potential spilling
then remove it and continue

Select
Assign colors backwards, adding nodes that were removed
If we find a node that was spilled, we check if we are lucky that we can color it
if yes, continue
if no, insert instructions to save and load values from memory
  restart with new graph (now we have graph that is easier to color, we killed a variable)

Run Register Allocation Ex.1 use 4 registers

```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
xz = x*z
res1 = xy + yz
m[3] = res1 + xz
```

Run Register Allocation Ex.2
use 3 registers

x = m[0]

y = m[1]

xy = x * y

z = m[2]

yz = y*z

xz = x*z

res1 = xy + yz

m[3] = res1 + xz

Conservative Coalescing

Suppose variables tmp1 and tmp2 are both assigned to the same register R and the program has an instruction:

tmp2 = tmp1

which moves the value of tmp1 into tmp2. This instruction then becomes

R = R

which can be simply omitted.

How to force a register allocator to assign tmp1 and tmp2 to same register?

merge the nodes for tmp1 and tmp2 in the interference graph!

this is called **coalescing**

But: if we coalesce non-interfering nodes when there are assignments, then our graph may become more difficult to color, and we may in fact need more registers!

Conservative coalescing: coalesce only if merged node of tmp1 and tmp2 will have a small degree so that we are sure that we will be able to color it.

Run Register Allocation Ex.3 use 4 registers, coallesce j=i

```
i = 0
s = s + i
i = i + b
j = i
s = s + j + b
j = j + 1
```