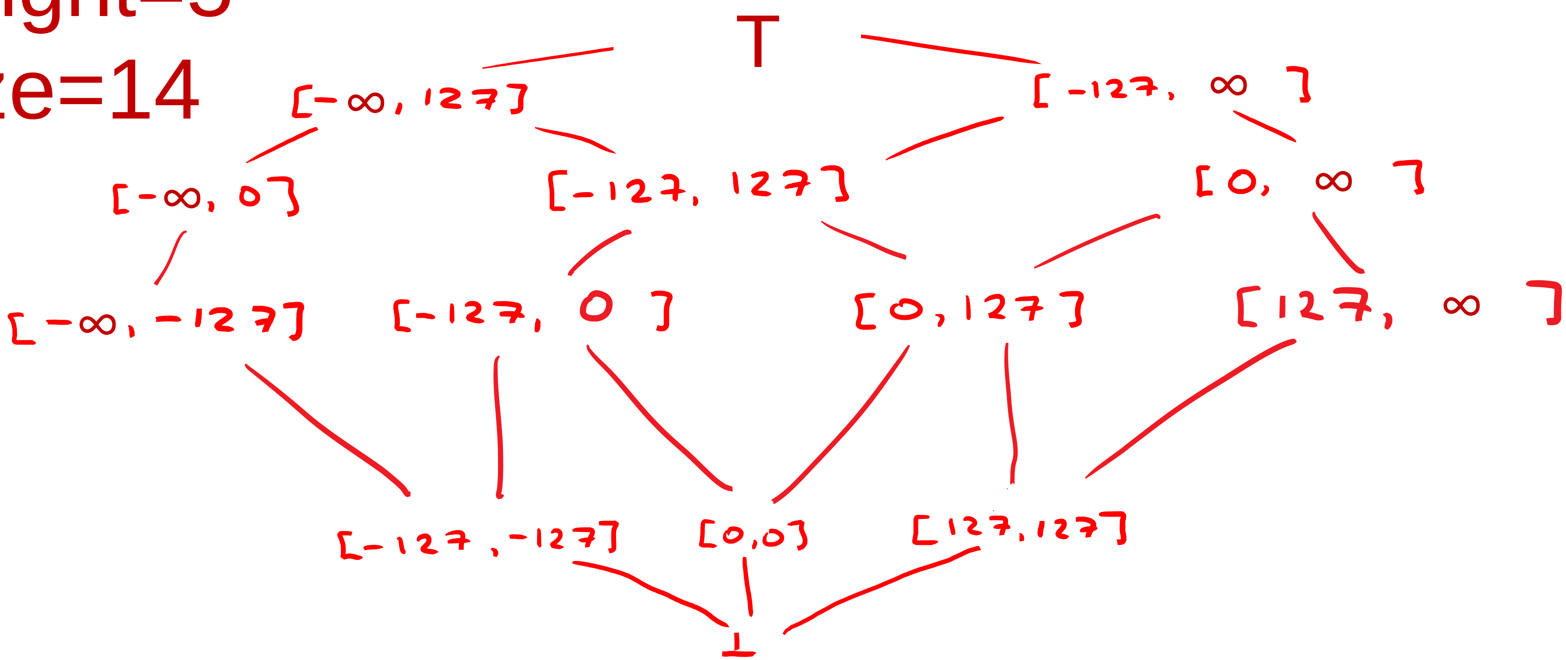
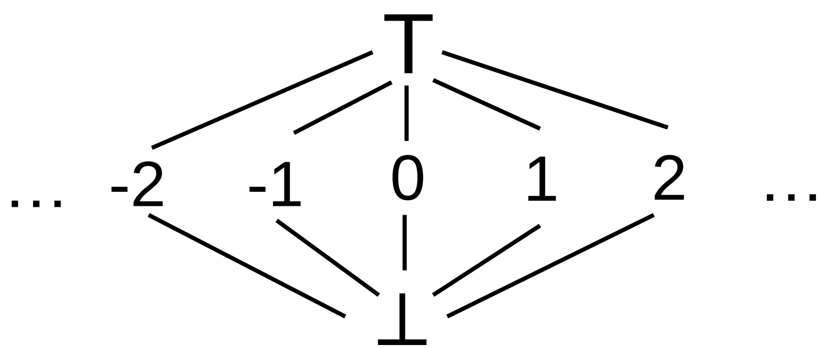
Abstract Interpretation Continued

Height of Lattice: Length of Max. Chain

height=5
size=14



height=2
size =∞



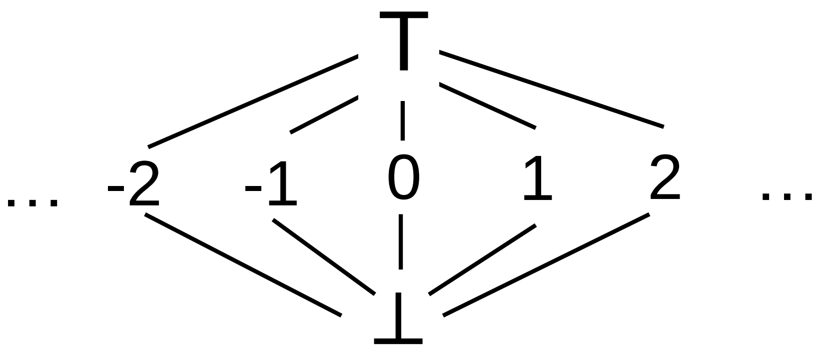
Chain of Length n

- 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: T

total changes: two (height)

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

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

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

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}

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

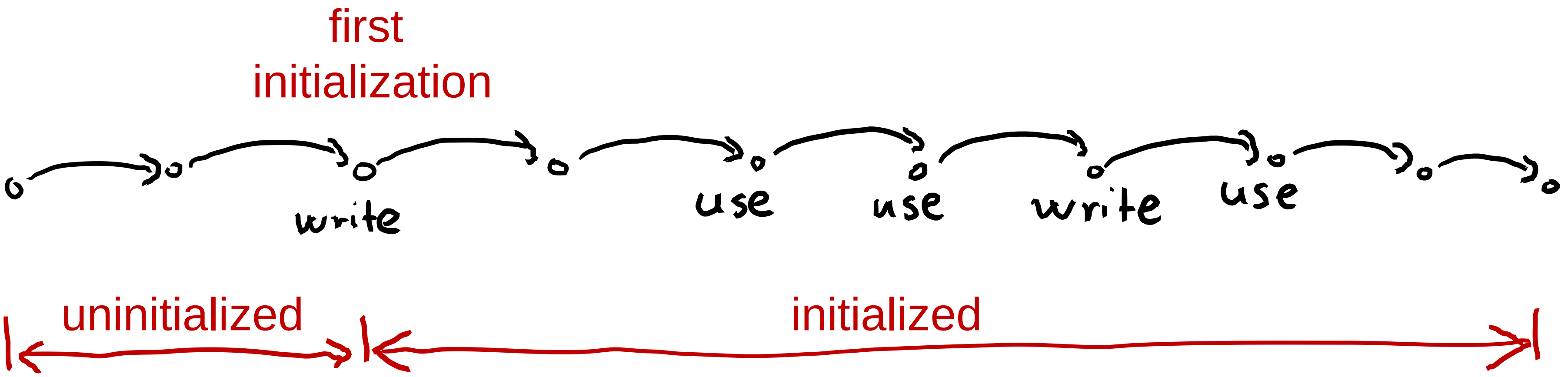
One possible chain of maximal length:

$\bot$

…

[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 undesirably affected by the value that was in the variable initially.

We can enforce such check using initialization analysis.

What does javac say to this?

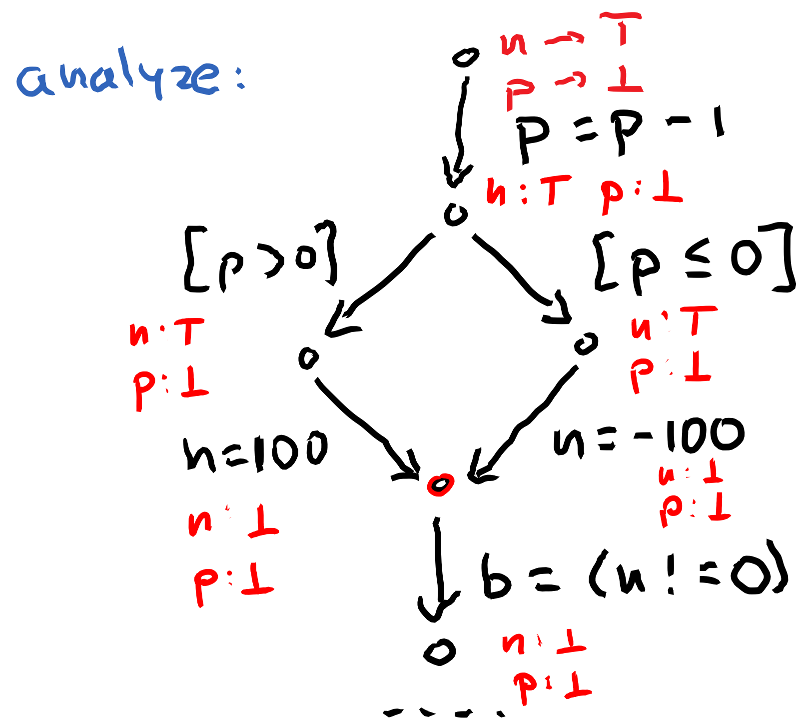
```
static void test(int p) {
    int n;
    p = p - 1;
    if (p > 0) {
        n = 100;
    }
    System.out.println("Hello!");
    if (p > 0) {
        while (n != 0) {
            System.out.println(n);
            n = n - p;
        }
    }
}
```

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

T ⊔ ⊥ = T

Run initialization analysis Ex.1

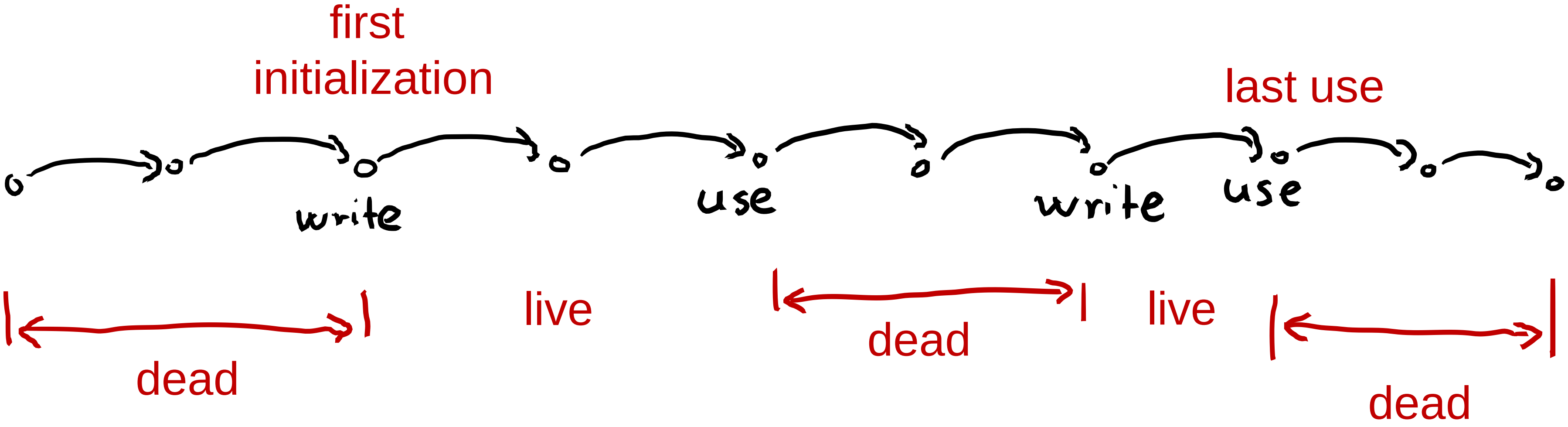
```
int n;
p = p - 1;
if (p > 0) {
    n = 100;
}
while (n != 0) {
    n = n - p;
}
```

Run initialization analysis Ex.2

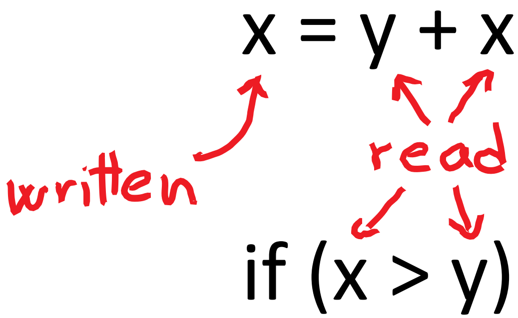
```
int n;
p = p - 1;
if (p > 0) {
    n = 100;
}
if (p > 0) {
    n = n - p;
}
```

Liveness Analysis

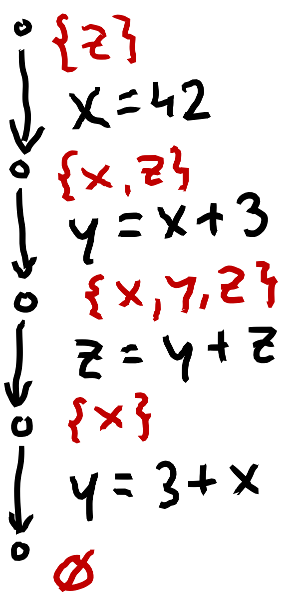
Variable is dead if its current value will not be used in the future. If there are no uses before it is reassigned or the execution ends, then the variable is surely dead at a given point.



What is Written and What Read



Example:



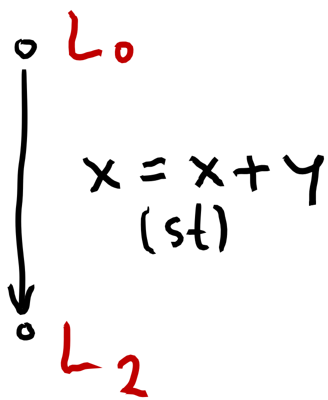
Purpose:

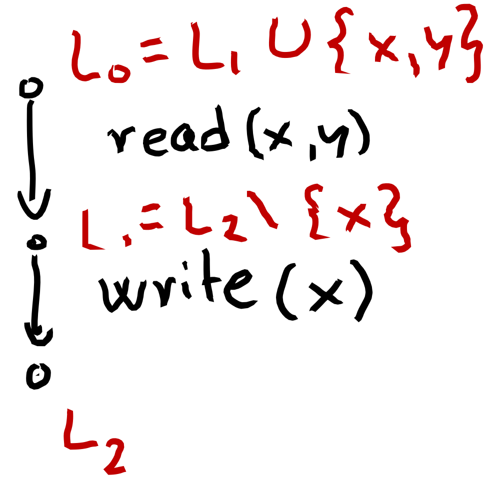
Register allocation:

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

How Transfer Functions Look

L_0 - set of live variables





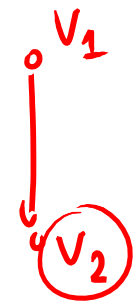
$$L_0 = (L_2 \setminus \{x\}) \cup \{x, y\}$$

Generally

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

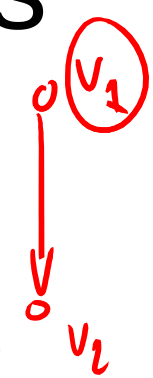
Initialization: Forward Analysis

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, y, xy, z, yz, xz, res1

∅

```
x = m[0]
```

{x}

```
y = m[1]
```

{x, y}

```
xy = x * y
```

{x, y, xy}

```
z = m[2]
```

{x, z, y, xy}

```
yz = y*z
```

{x, z, xy, yz}

```
xz = x*z
```

{xz, xy, yz}

```
res1 = xy + yz
```

{res1, xz}

```
m[3] = res1 + xz
```

∅

Register Machines

Better for most purposes than stack machines

- closer to modern CPUs (RISC architecture)
- closer to control-flow graphs
- simpler than stack machine (but register set is finite)

Examples:

ARM architecture

RISC V: http://riscv.org/

Directly Addressable RAM

large - GB, slow even with cache

A few fast registers

R0,R1,…,R31

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

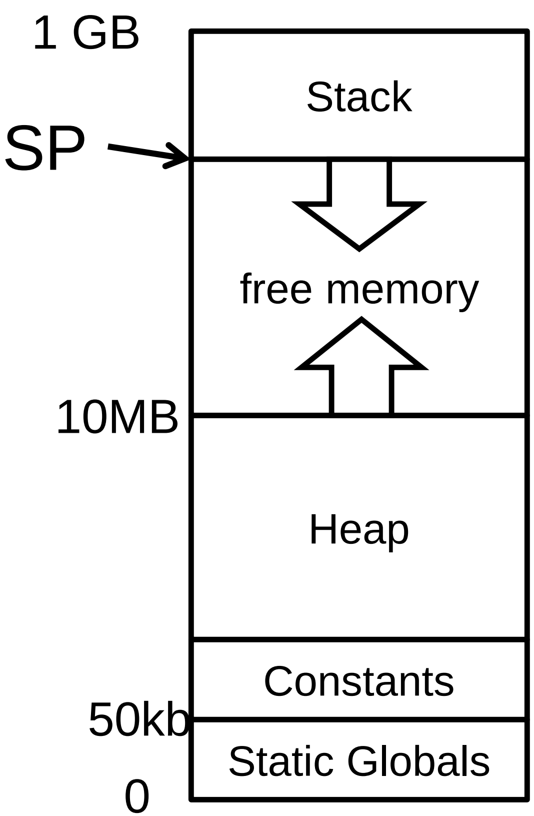
- heap need not be contiguous; can request more memory from the OS if needed
- stack grows downwards

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

Stack Machine vs General Register Machine Code Naïve Correct Translation

JVM:

i32.mul

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?

7 variables: x,y,z,xy,yz,xz,res1

```
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
```

can do it with 5 only!

```
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 Register Allocation

program:

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

live variable analysis result:

{} {x} {x,y} {y,x,xy} {y,z,x,xy} {x,z,xy,yz} {xy,yz,xz} {r,xz} {}

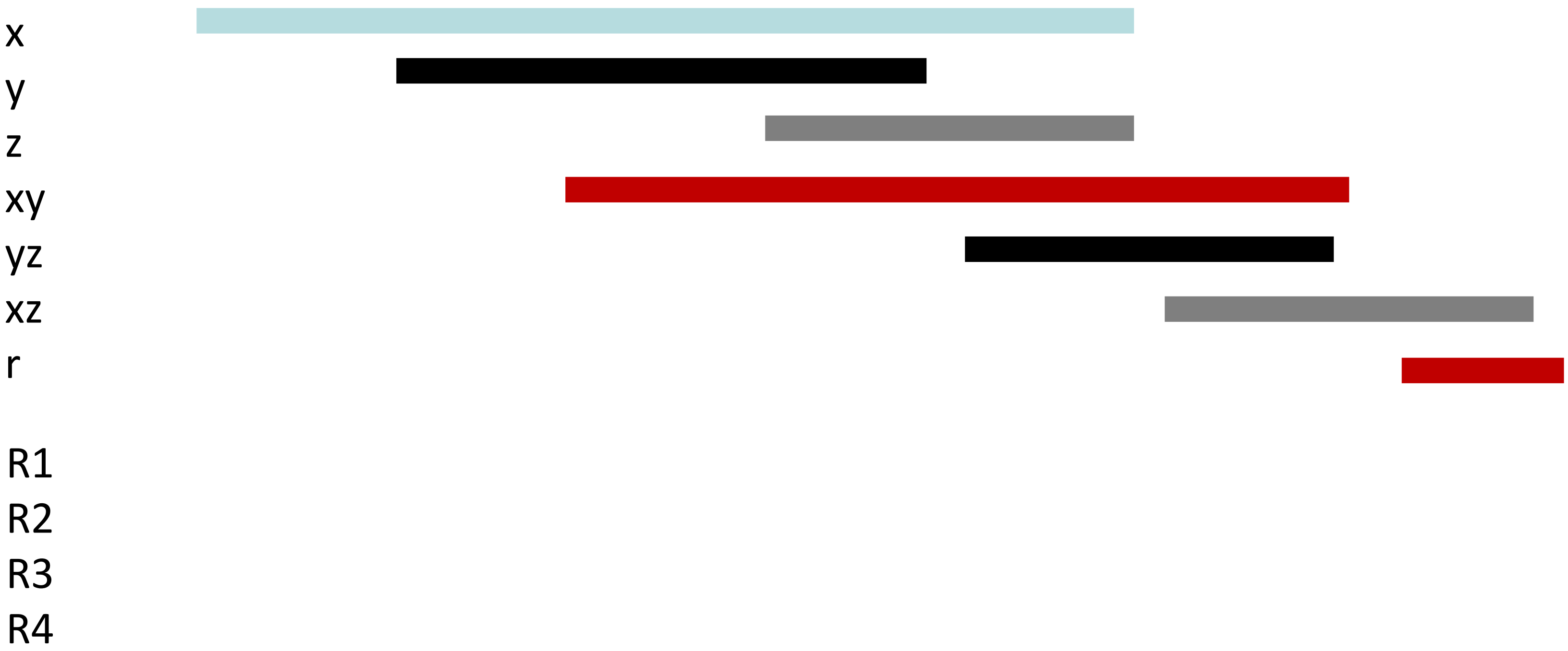
Color Variables
Avoid Overlap of Same Colors

program:

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

live variable analysis result:

{} {x} {x,y} {y,x,xy} {y,z,x,xy} {x,z,xy,yz} {xy,yz,xz} {r,xz} {}



Each color denotes a register

4 registers are enough for this program

Color Variables
Avoid Overlap of Same Colors

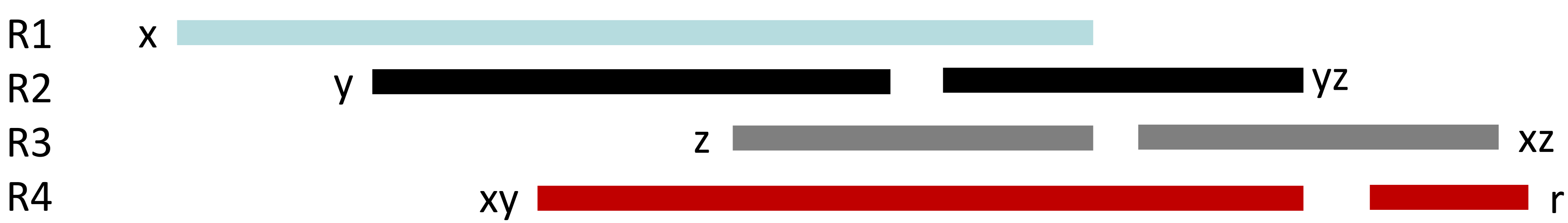
program:

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

live variable analysis result:

{} {x} {x,y} {y,x,xy} {y,z,x,xy} {x,z,xy,yz} {xy,yz,xz} {r,xz} {}

x
y
z
xy
yz
xz
r



Each color denotes a register

4 registers are enough for this 7-variable program

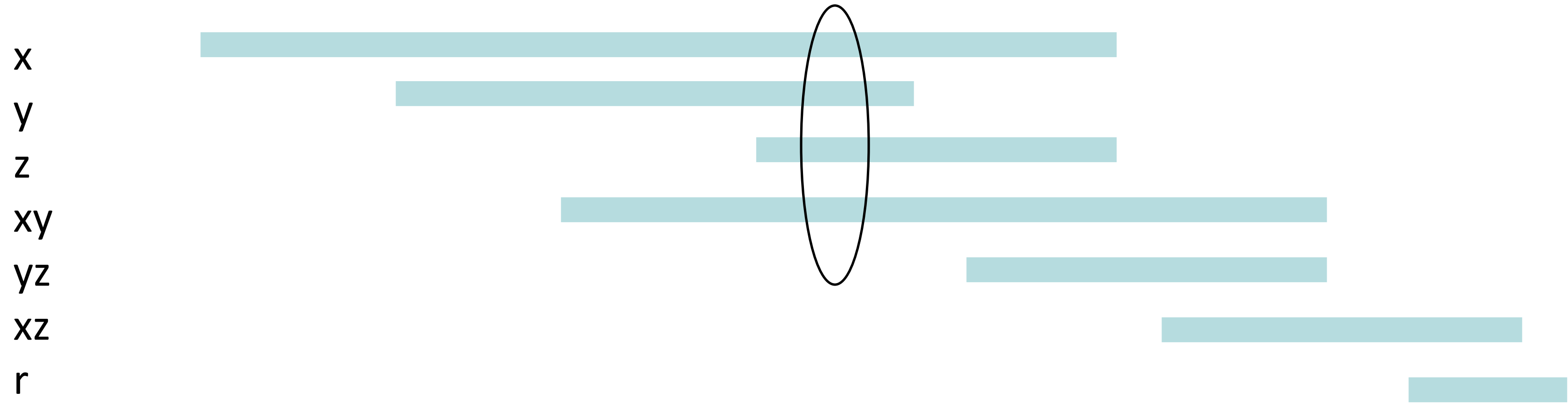
How to assign colors to variables?

program:

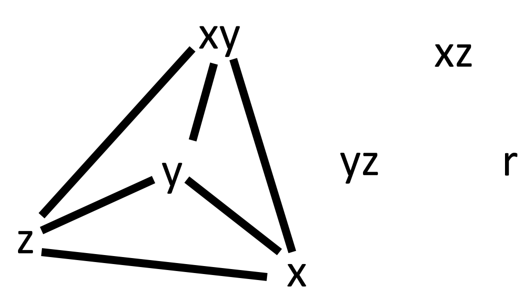
x = m[0]; y = m[1]; xy = x*y; z = m[2]; yz = y*z; xz = x*z; r = xy + yz; m[3] = r + xz

live variable analysis result:

{} {x} {x,y} {y,x,xy} **{y,z,x,xy}** {x,z,xy,yz} {xy,yz,xz} {r,xz} {}



For each pair of variables determine if their lifetime overlaps = there is a point at which they are both alive. Construct **interference graph**



Edges between members of each set

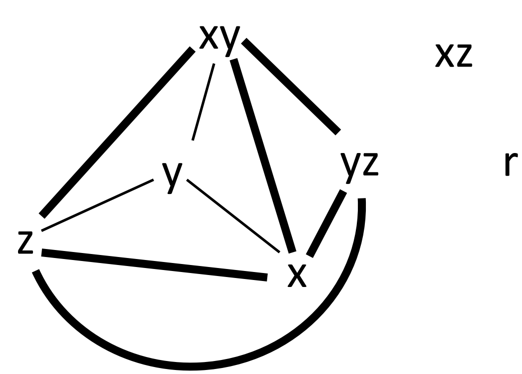
program:

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

live variable analysis result:

{} {x} {x,y} {y,x,xy} {y,z,x,xy} {x,z,xy,yz} {xy,yz,xz} {r,xz} {}

For each pair of variables determine if their lifetime overlaps = there is a point at which they are both alive. Construct **interference graph**



Final interference graph

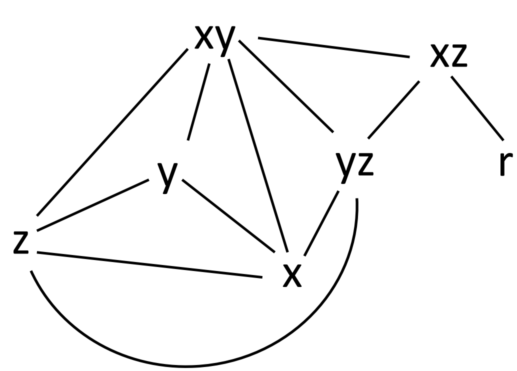
program:

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

live variable analysis result:

{}   {x}   {x,y}   {y,x,xy}   {y,z,x,xy}   {x,z,xy,yz}   {xy,yz,xz}   {r,xz}   {}

For each pair of variables determine if their lifetime overlaps = there is a point at which they are both alive. Construct **interference graph**



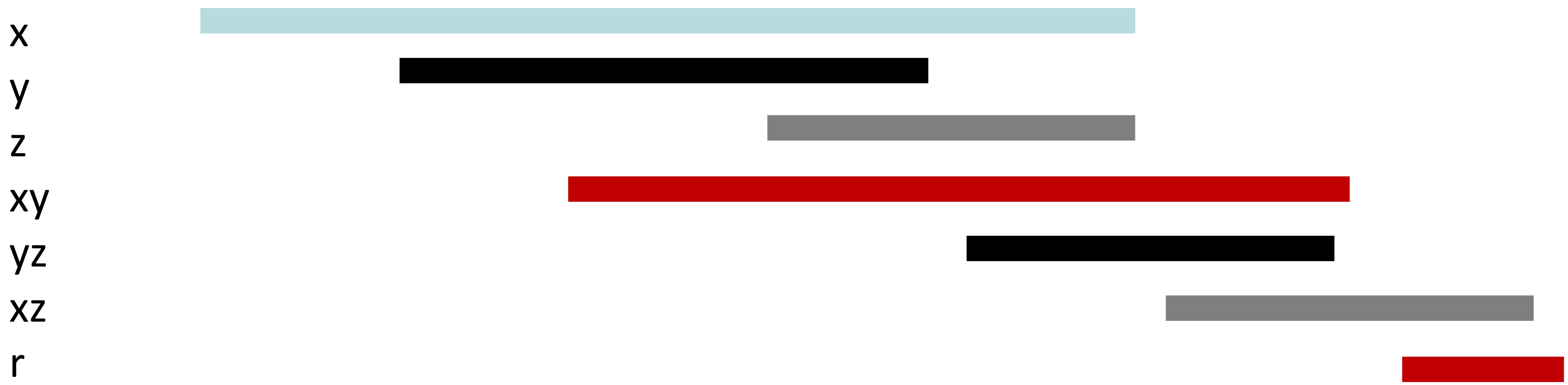
Coloring interference graph

program:

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

live variable analysis result:

{} {x} {x,y} {y,x,xy} {y,z,x,xy} {x,z,xy,yz} {xy,yz,xz} {r,xz} {}

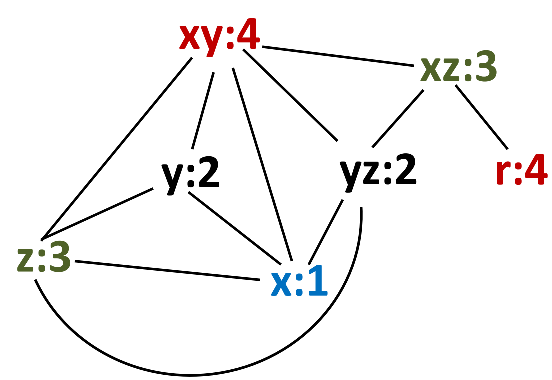


Need to assign colors (register numbers) to nodes such that:

if there is an edge between nodes,

then those nodes have different colors.

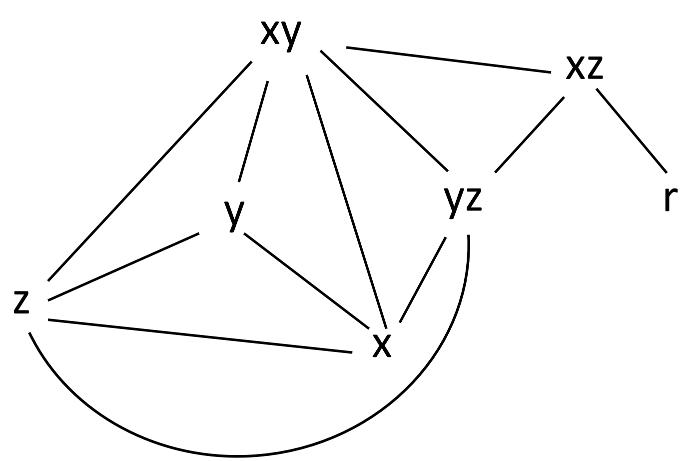
→ standard graph vertex coloring problem



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

All we need to do is solve graph coloring problem



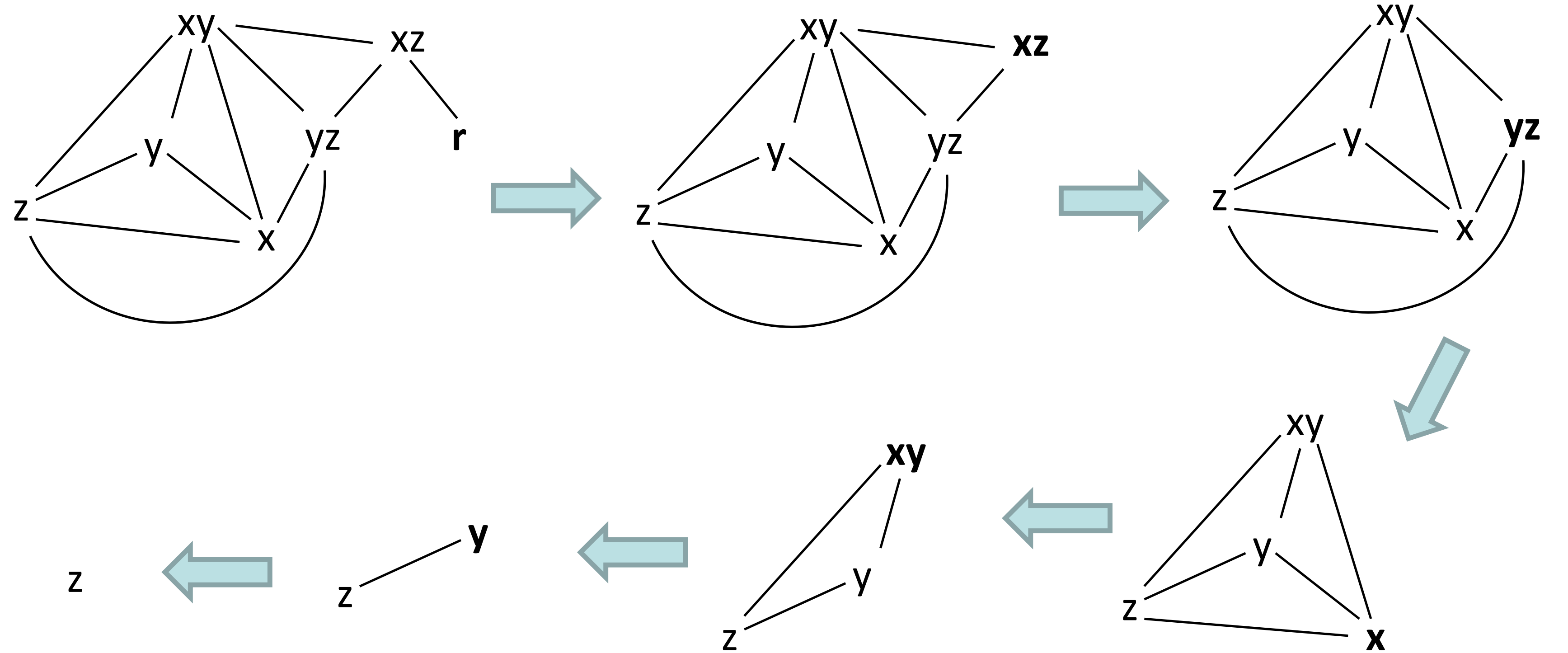
- NP hard
- In practice, we have heuristics that work for typical graphs
- If we cannot fit it all variables into registers, perform a **spill:**

  store variable into memory and load later when needed

Heuristic for Coloring with K Colors

Simplify:

If there is a node with less than K neighbors, we will always be able to color it!
So we can remove such node from the graph (if it exists, otherwise remove other node)
This reduces graph size. It is useful, even though incomplete
(e.g. planar can be colored by at most 4 colors, yet can have nodes with many neighbors)



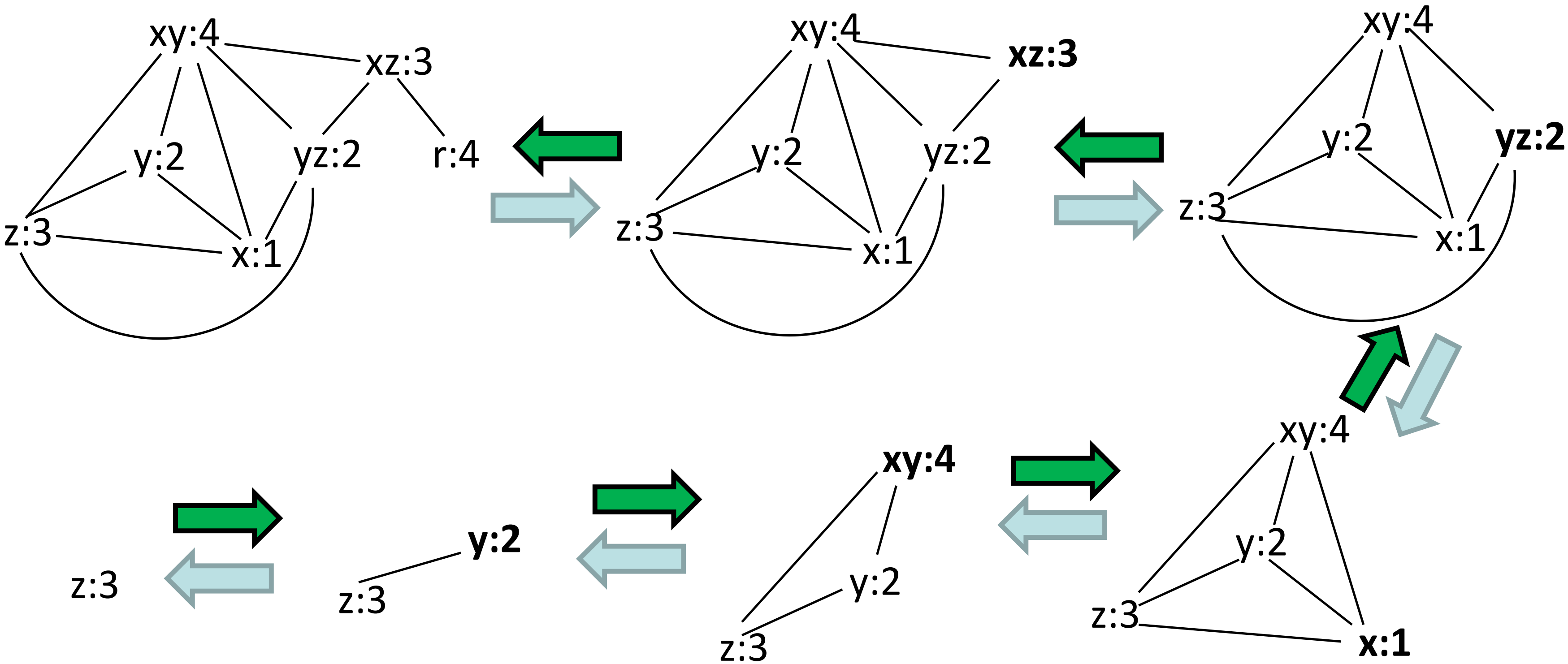
Heuristic for Coloring with K Colors

Select

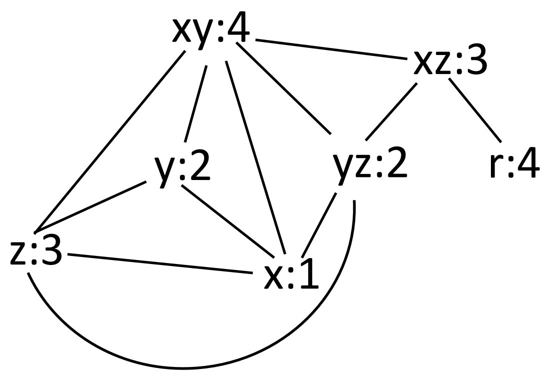
Assign colors backwards, adding nodes that were removed

If the node was removed because it had <K neighbors, we will always find a color

if there are multiple possibilities, we can choose any color



Use Computed Registers



x = m[0]

y = m[1]

xy = x * y

z = m[2]

yz = y*z

xz = x*z

r = xy + yz

m[3] = res1 + xz

R1 = m[0]

R2 = m[1]

R4 = R1*R2

R3 = m[2]

R2 = R2*R3

R3 = R1*R3

R4 = R4 + R2

m[3] = R4 + R3

Summary of Heuristic for Coloring

Simplify (forward, safe):
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

Potential Spill (forward, speculative):
If every node has K or more neighbors, we still remove one of them
we mark it as node for **potential** spilling. Then remove it and continue

Select (backward):
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 not, insert instructions to save and load values from memory (**actual spill**).
Restart with new graph (a graph is now easier to color as we killed a variable)

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!

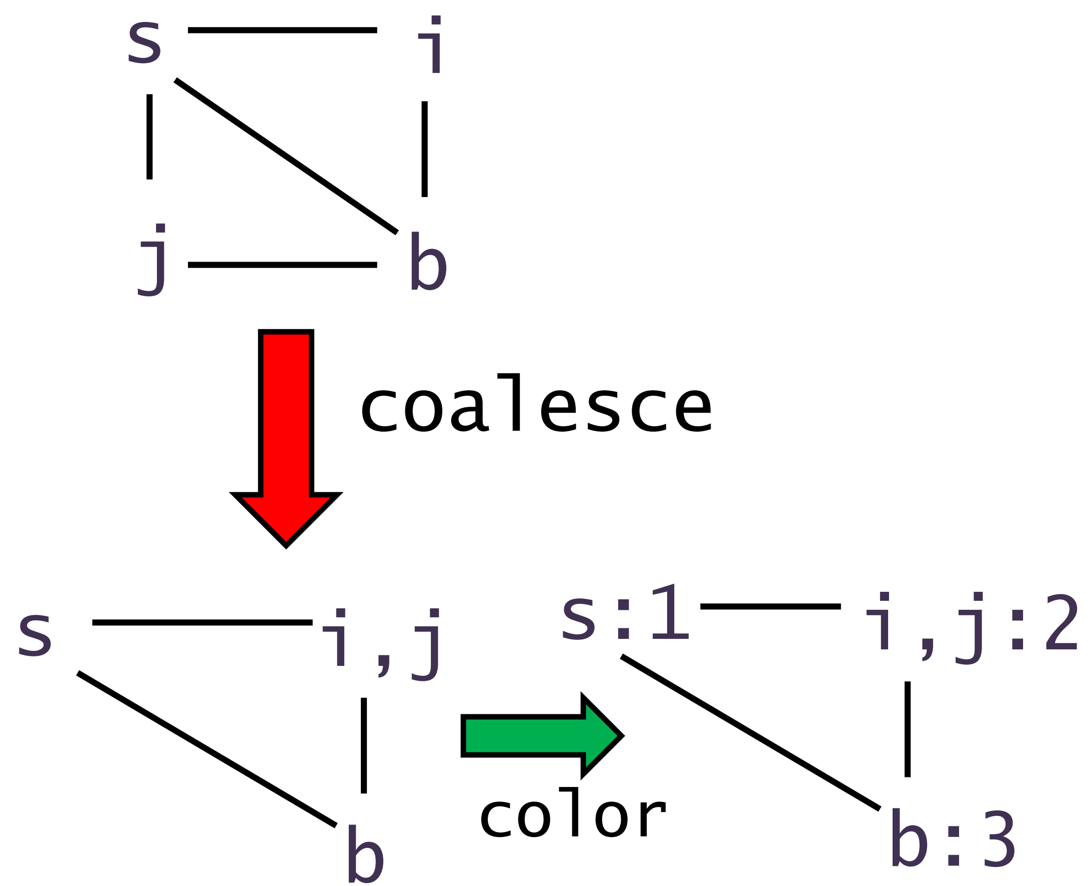
Conservative coalescing: coalesce only if merged node of tmp1 and tmp2 will have a small degree so that we are sure that we will be able to color it (e.g. resulting node has degree < K)

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

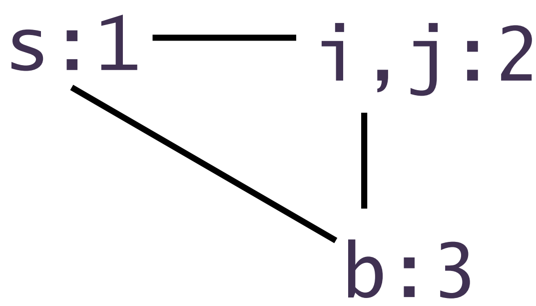
```
i = 0

s = s + i

i = i + b

j = i

s = s + j + b

j = j + 1
```

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

```
    {s,b}
i = 0
    {s,i,b}
s = s + i
    {s,i,b}
i = i + b
    {s,i,b}
j = i
    {s,j,b}
s = s + j + b
    {j}
j = j + 1
    {}
```



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




```
i = 0
s = s + i
i = i + b
j = i  // puf!
s = s + j + b
j = j + 1
```

```
R2 = 0
R1 = R1 + R2
R2 = R2 + R3
R1 = R1 + R2 + R3
R2 = R2 + 1
```