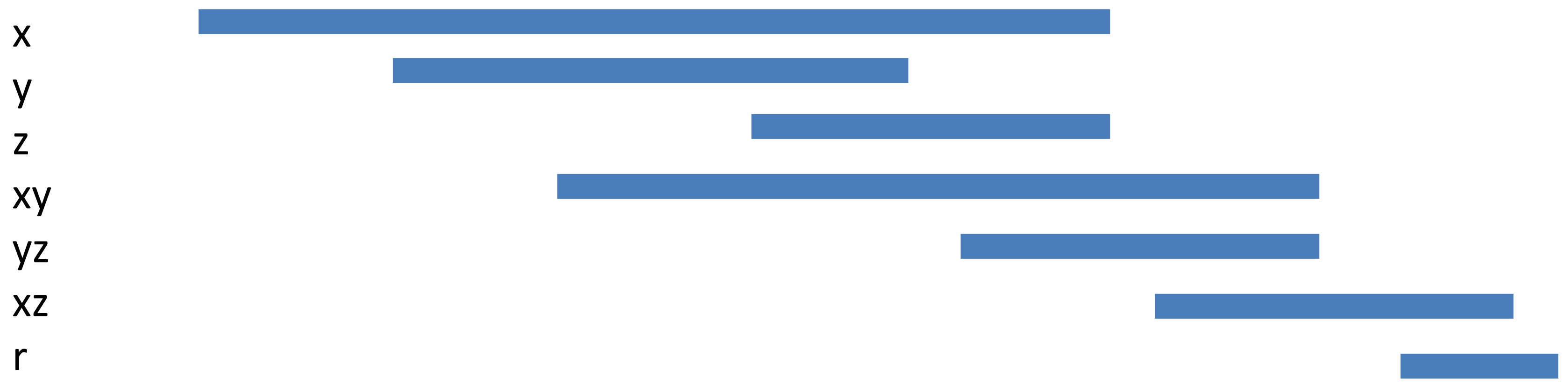
Idea of Register Allocation

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} {}



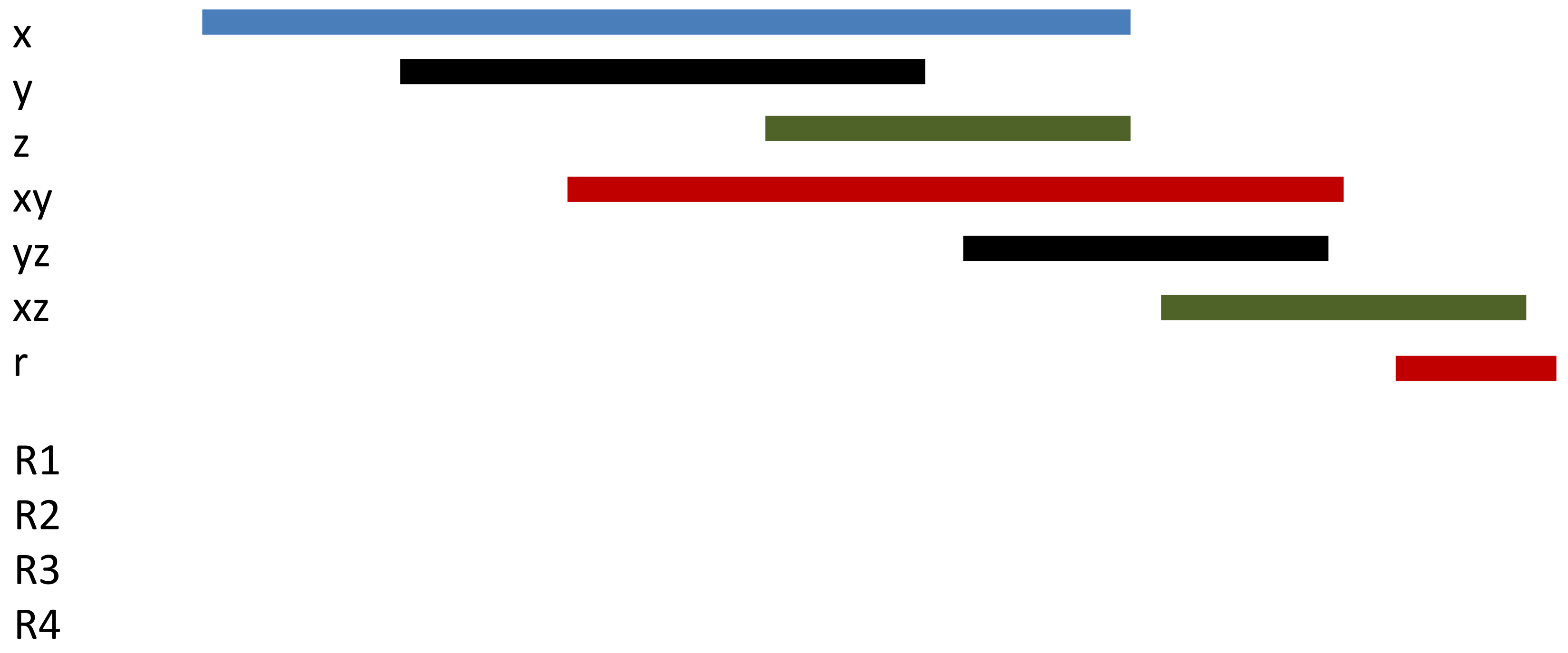
Color Variables
Avoid Overlap of Same Colors

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

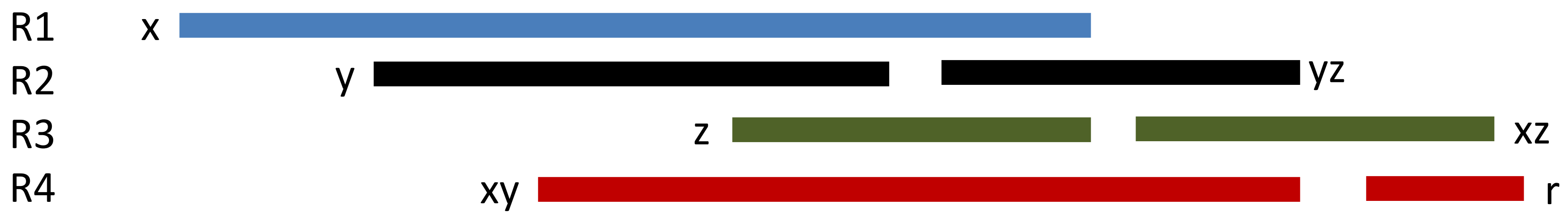
program:

```
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

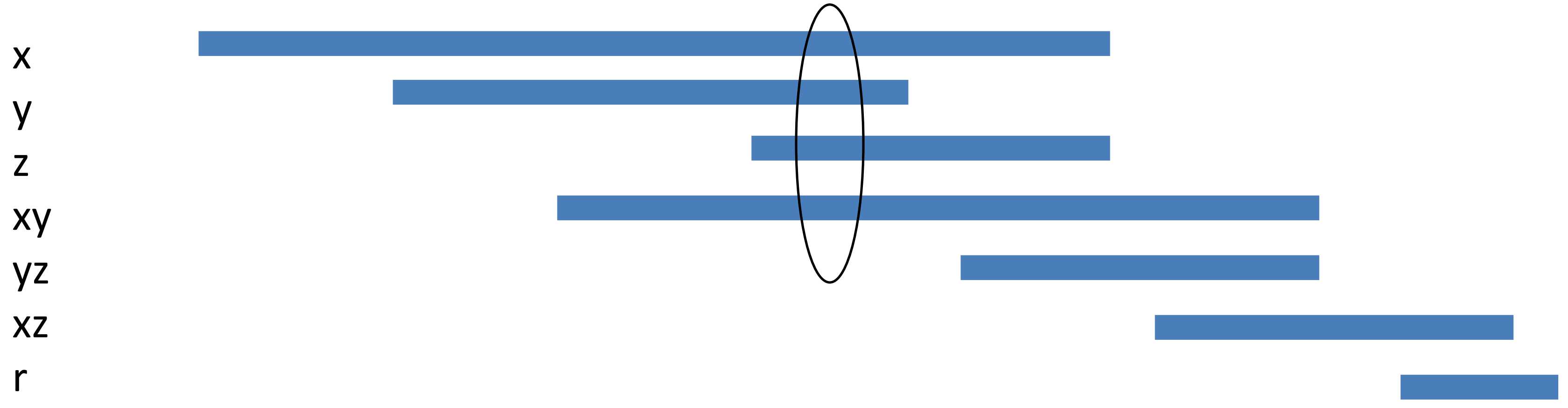
How to assign colors to variables?

program:

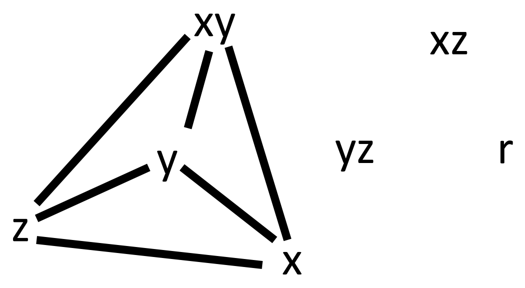
```
x = m[0];   y = m[1];   xy = x*y;   z = m[2];   yz = y*z;   xz = x*z;   r = xy + yz;   m[3] = r + xz
```

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**



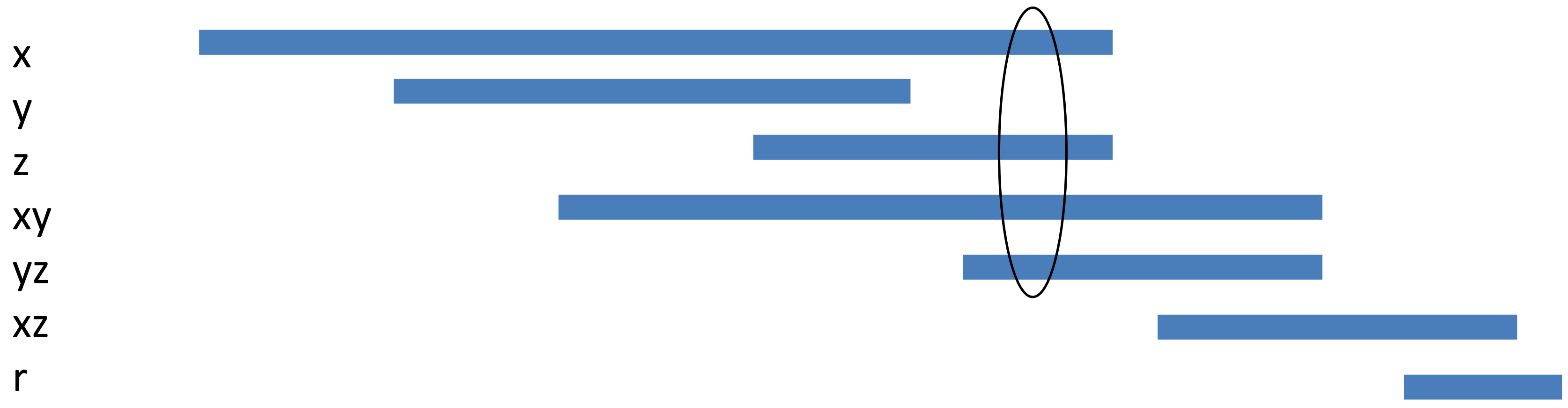
Edges between members of each set

program:

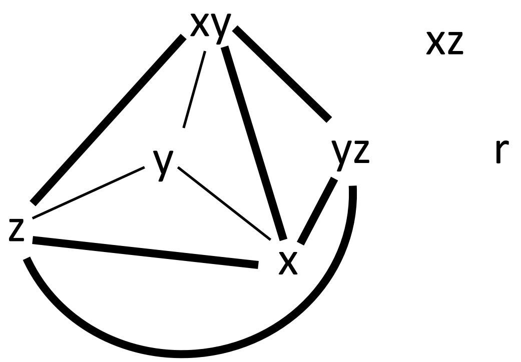
```
x = m[0];   y = m[1];   xy = x*y;   z = m[2];   yz = y*z;   xz = x*z;   r = xy + yz;   m[3] = r + xz
```

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**



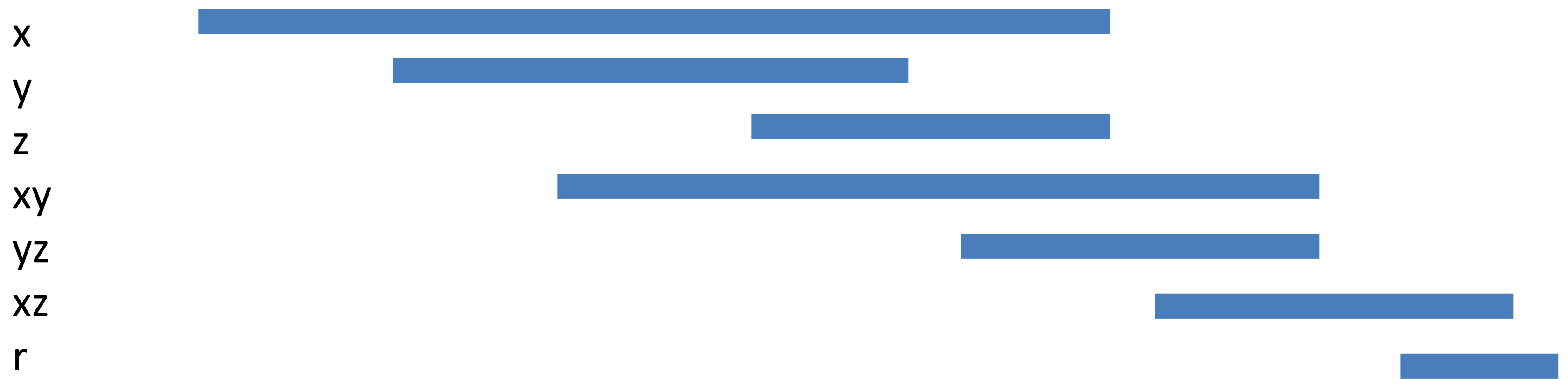
Final interference graph

program:

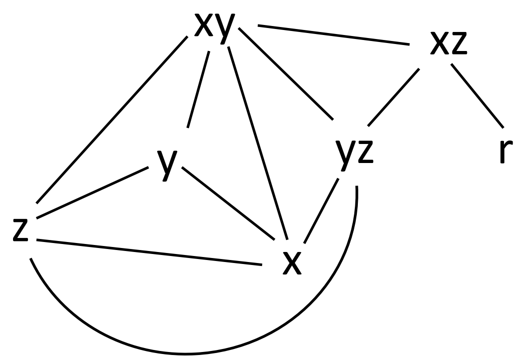
```
x = m[0];    y = m[1];    xy = x*y;    z = m[2];    yz = y*z;    xz = x*z;    r = xy + yz;    m[3] = r + xz
```

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**



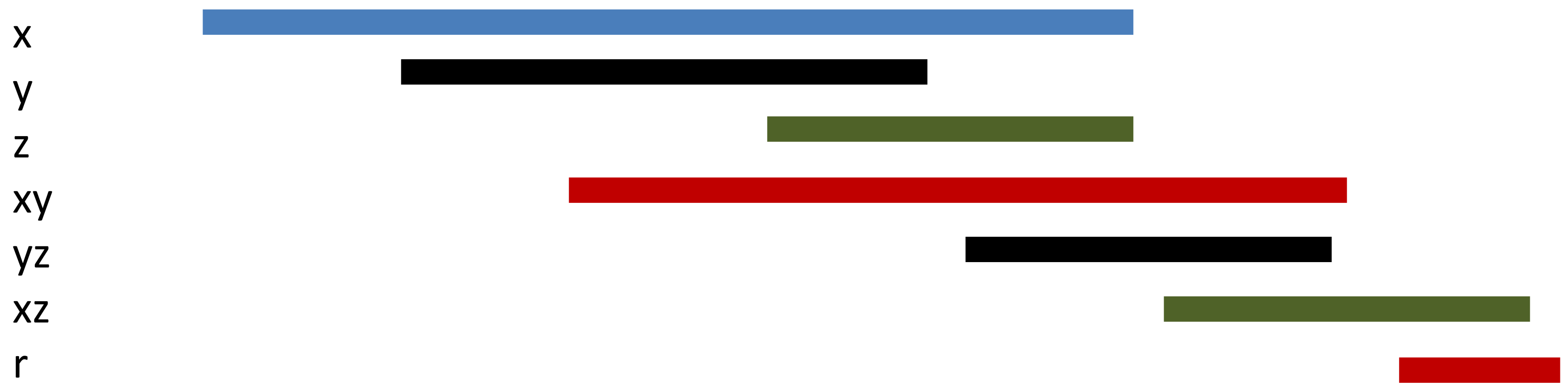
Coloring interference graph

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:

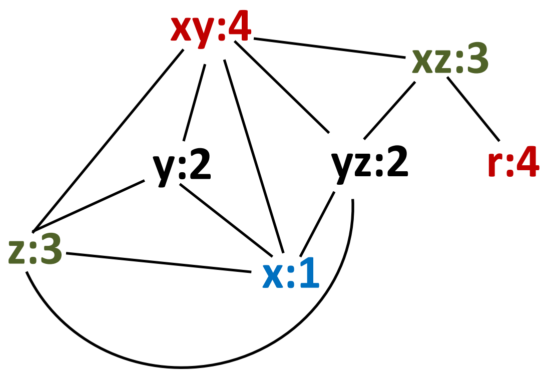
{}   {x}   {x,y}   {y,x,xy}   {y,z,x,xy}   {x,z,xy,yz}   {xy,yz,xz}   {r,xz}   {}



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

if there is an edge between nodes,
then those nodes have different colors.

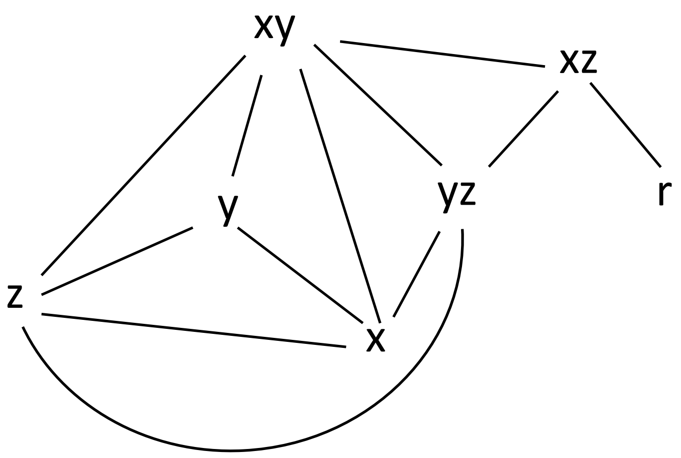
→ standard graph vertex coloring problem



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

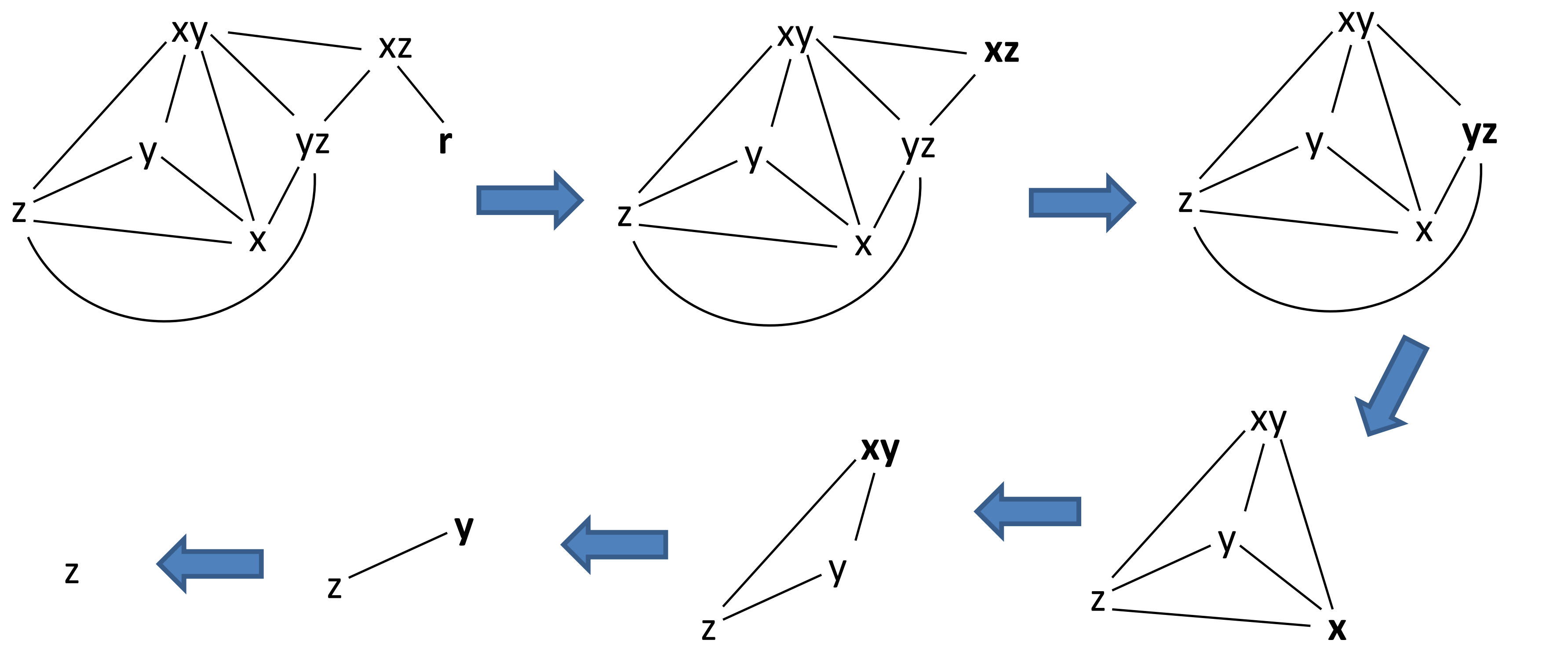


- NP hard
- In practice, we have heuristics that work for typical graphs
- If we cannot fit it all variables into registers, perform a **spill:**

  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)



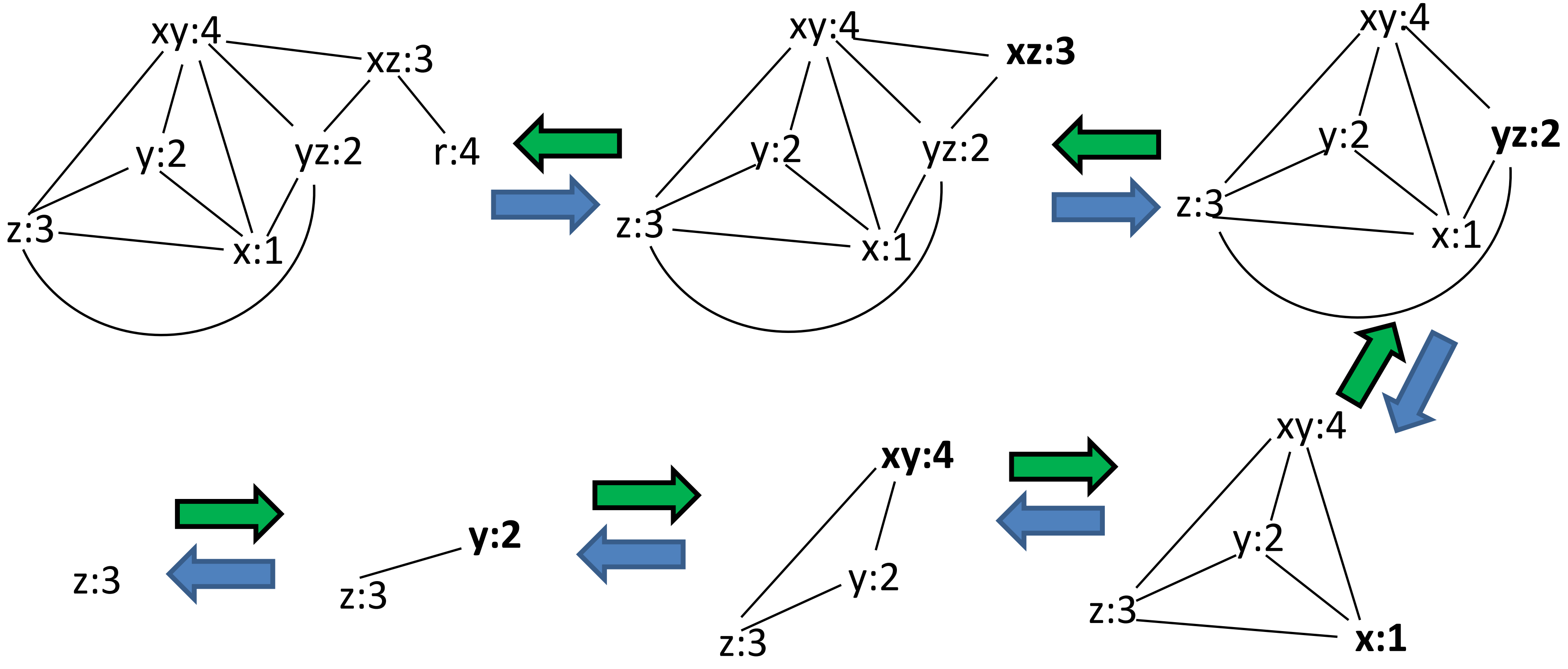
Heuristic for Coloring with K Colors

Select

Assign colors backwards, adding nodes that were removed

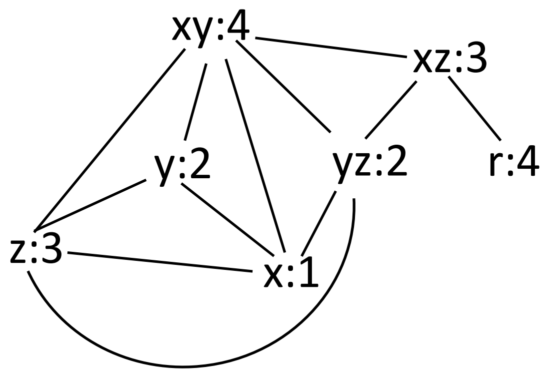
If the node was removed because it had <K neighbors, we will always find a color

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!

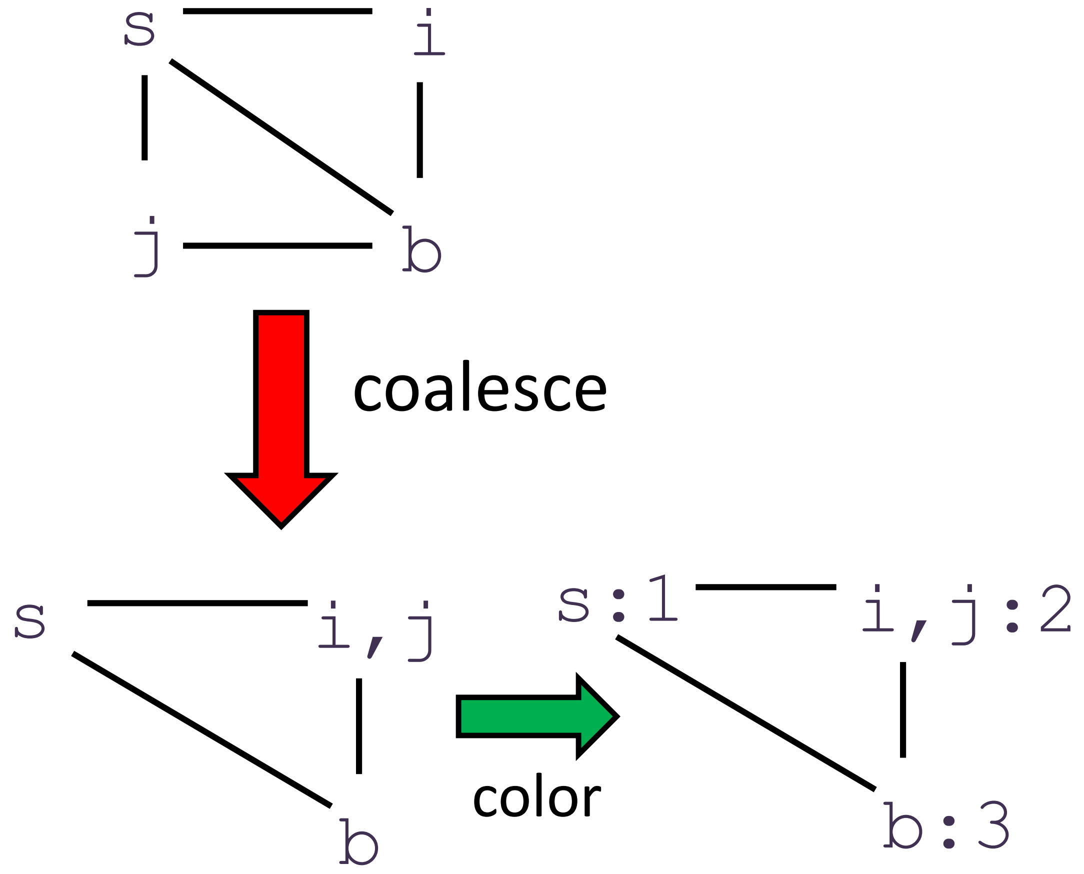
Conservative coalescing: coalesce only if merged node of tmp1 and tmp2 will have a small degree so that we are sure that we will be able to color it (e.g. resulting node has degree < K)

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

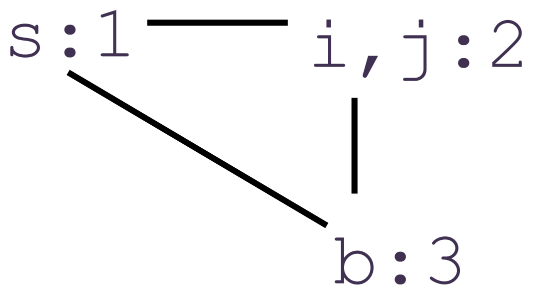
```
i = 0

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
```

Exam preparation: Last year exam

Simple type checking

Suppose that we have

```
class Phone {
def getNumber: Int = { ... }
def call (n: Int) = { ... }
}
class MobilePhone extends Phone {
// more functionality
}
class AntiquePhone(anno: Int) extends Phone {
// more functionality
}
```

Type-check the following:

```
var a: Phone
var b: MobilePhone
a = new AntiquePhone(1981)
b = new MobilePhone
a.call(b.getNumber)
```

$$\dfrac{(a,\texttt{Phone}) \in \Gamma \qquad \dfrac{\dfrac{}{\Gamma\vdash \texttt{new AntiquePhone(1981): AntiquePhone}} \qquad \texttt{AntiquePhone <: Phone}}{\Gamma\vdash \texttt{new AntiquePhone(1981): Phone}}}{\Gamma\vdash \texttt{a = new AntiquePhone(1981): Unit}}$$

$$\dfrac{(b,\texttt{MobilePhone}) \in \Gamma \qquad \Gamma\vdash \texttt{new MobilePhone: MobilePhone}}{\Gamma\vdash \texttt{b = new MobilePhone : Unit}}$$

$$\dfrac{\dfrac{\dfrac{(\texttt{b},\texttt{MobilePhone}) \in \Gamma}{\Gamma \vdash \texttt{b} : \texttt{MobilePhone}} \qquad \Gamma \vdash \texttt{MobilePhone} <: \texttt{Phone}}{\Gamma \vdash \texttt{b} : \texttt{Phone}} \qquad \Gamma^{\texttt{Phone}} \vdash \texttt{getNumber}: \texttt{Int}}{\Gamma \vdash \texttt{b.getNumber}: \texttt{Int}}$$

$$\dfrac{\dfrac{\dfrac{(\texttt{a},\texttt{Phone}) \in \Gamma}{\Gamma \vdash \texttt{a} : \texttt{Phone}} \qquad \Gamma^{\texttt{Phone}} \vdash \texttt{call}: \texttt{Int => Unit}}{\Gamma \vdash \texttt{a.call}: \texttt{Int => Unit}} \qquad \Gamma \vdash \texttt{b.getNumber}: \texttt{Int}}{\Gamma \vdash \texttt{a.call(b.getNumber)}: \texttt{Unit}}$$

Typing object-oriented language

No primitive value (no Int, no Bool)

Expressions:

- `x.methodName(p)` standard method call where x and p are variable names;
- `new T()` class constructor
- `null` special value that can be assigned to any variable

Statements:

- `var x: T = e` var declaration with assignment
- `x = e` assignment

Typing object-oriented language

The goal of our type system is preventing null-pointer exceptions. We introduce for each regular object type T the *null-annotated* types:

- T^+ meaning that a variable of this type may also be null
- T^- meaning that a variable of this type cannot be null

Of course, $T^- <: T^+$

The programmer should still only write type T in the code, but the type checker will choose one of T^+ and T^- for type checking. The type of variable is determined at its declaration and always has the same type as the expression that is assigned to it.

- methods can return null.
- If a method called on a null object, there is a runtime error.

Soundness property: **If program type checks, runtime error from dereferencing a null value cannot occur**

Typing object-oriented language

Give sound type rules for this language.

Expressions:

- `x.methodName(p)` standard method call where x and p are variable names;
- `new T()` class constructor
- `null` special value that can be assigned to any variable

Statements:

- `var x: T = e` var declaration with assignment
- `x = e` assignment

$$\frac{}{\Gamma \vdash \texttt{null} : T^{+}} \qquad \frac{}{\Gamma \vdash \texttt{new T()} : T^{-}}$$

$$\frac{\Gamma \vdash x : T_2{}^{-} \qquad \Gamma^{T2} \vdash m : T_1 \Rightarrow T_3 \qquad \Gamma \vdash p : T_1{}^{+}}{\Gamma \vdash x.m(p) : T_3{}^{+}}$$

$$\frac{\Gamma \vdash x : T_1{}^{+} \qquad \Gamma \vdash e : T_1{}^{+}}{\Gamma \vdash x = e : \texttt{Unit}} \qquad \frac{\Gamma \vdash x : T_1{}^{-} \qquad \Gamma \vdash e : T_1{}^{-}}{\Gamma \vdash x = e : \texttt{Unit}}$$

$$\frac{\Gamma \vdash e : T^{+} \qquad \Gamma + (x, T^{+}) \vdash e2 : T_2}{\Gamma \vdash \texttt{var}\ x : T = e;\ e2 : T_2}$$

$$\frac{\Gamma \vdash e : T^{-} \qquad \Gamma + (x, T^{-}) \vdash e2 : T_2}{\Gamma \vdash \texttt{var}\ x : T = e;\ e2 : T_2}$$

Typing object-oriented language

Induction proof:

P(n): For every tree of height ≤ n, (a) any expression type checking to T^- can never be null. (b) If the expression type checks for any type, then it will not raise nullPointerException.

P(1):

- If the rule is for null, then it type checks to T^+ so (a) is true. (b) is true as well.
- If the rule is for new, then it type checks to T^- and is never null so (a) is true. (b) is true as well.

Typing object-oriented language

Induction proof:

P(n): For every tree of height ≤ n, (a) any expression type checking to T^- can never be null. (b) If the expression type checks for any type, then it will not raise nullPointerException.

Suppose the rule true for P(n), and let D be a derivation tree of size n+1 of an expressions which type checks. Depending on the last rule applied:

- if the rule is for methods, then it type checks to T+ so (a) is true. Because P(n)(a) and x type checks to T-, x cannot be null so (b) is true.
- if the rule is for assignment +, then (a) will be true because we do not change variable which type is T-. (b) is true.

Typing object-oriented language

Induction proof:

P(n): For every tree of height ≤ n, (a) any expression type checking to T^- can never be null. (b) If the expression type checks for any type, then it will not raise nullPointerException.

Suppose the rule true for P(n), and let D be a derivation tree of size n+1 of an expressions which type checks. Depending on the last rule applied:

- if the rule is for assignment -, then e will not be null according to P(n) so after assignment x will not be null so (a) is true. (b) is also true.
- similar proof for assignment with simultaneous declaration.

Therefore, P(n+1) and we conclude the proof.

Live variable analysis

```
1. x = input()
                  – {x}
2. w = x * 100
                  – {w, x}
3. y = w + x
                  – {w, x, y}
4. z = y * x
                  – {w, y, z}
5. u = y - z
                  – {w, z, u}
6. v = z - w
                  – {w, z, u, v}
7. x = z + w
                  – {x, u, v}
8. w = x * v
                  – {x, w, u, v}
9. y = u - v
                  – {x, w, y}
10.v = x * w
                  – {y, v}
11.x = y + v
                  – {x}
```

How many registers do we need?

Liveness table

```
      1 2 3 4 5 6 7 8 9 0 1
x :   WX|X|XR | | WX|X|XR W
y :   | | WX|XR | | | WX|XR
z :   | | | WX|X|XR | | | |
w :   | WX|X|X|X|XR WX|XR |
u :   | | | | WX|X|X|XR | |
v :   | | | | | WX|X|XR WXR
```

Liveness table

```
         1 2 3 4 5 6 7 8 9 0 1

x :      WX|X|XR | | WX|X|XR W
y :      | | WX|XR | | | WX|XR
z :      | | | WX|X|XR | | | |
w :      | WX|X|X|X|XR WX|XR |
u :      | | | | WX|X|X|XR | |
v :      | | | | | WX|X|XR WXR
```

Answer is : 4