Translating control flow structures more efficiently

Introduce an imaginary large instruction **`branch`**`(c,nThen,nElse)`.

Here `c` is a potentially complex boolean expression (the main reason why **`branch`** is not a built-in bytecode instruction),
whereas `nTrue` and `nFalse` are the labels we jump to depending on the boolean value of `c`.

We will show how to

- use **`branch`** to compile **`if`** and short-circuiting operators,
- by expanding **`branch`** recursively into concrete bytecode instructions.

Translating control flow structures more efficiently

```
$[\mathtt{if}\ (e_{cond})\ e_{then}\ \mathtt{else}\ e_{else}] :=$

  block nAfter
    block nElse
      block nThen
        branch($e_{cond}$, nThen, nElse)
      end //nThen:
      $[e_{then}]$
      br nAfter
    end //nElse:
    $[e_{else}]$
  end //nAfter:
  $[e_{rest}]$
```

Decomposing conditions in branch

```
branch(!$e$,nThen,nElse) :=
  branch($e$,nElse,nThen)

branch($e_1$ && $e_2$,nThen,nElse) :=
  block nLong
    branch($e_1$,nLong,nElse)
  end //nLong:
  branch($e_2$,nThen,nElse)

branch($e_1$ || $e_2$,nThen,nElse) :=
  block nLong
    branch($e_1$,nThen,nLong)
  end //nLong:
  branch($e_2$,nThen,nElse)
```

Decomposing conditions in `branch`

```
branch($true$,nThen,nElse) :=
  br nThen

branch($false$,nThen,nElse) :=
  br nElse

branch($b$,nThen,nElse) :=  (where $b$ is a local var)
  get_local #b
  br_if nThen
  br nElse
```

Decomposing conditions in `branch`

```
branch($e_1 == e_2$,nThen,nElse) :=   (where $e_1, e_2$ are of type $int$)
  [$e_1$]
  [$e_2$]
  i32.eq
  br_if nThen
  br nElse
```

… *analogously for other relations*

Returning the result from `branch`

Consider storing $x = c$
where x, c are boolean and c contains $\&\&$ or $\|$.

How do we put the result of c on the stack so it can be stored in x?

```
[x = c] :=
  block nAfter
    block nElse
      block nThen
        branch(c,nThen,nElse)
      end //nThen:
      i32.const 1
      br nAfter
    end //nElse:
    i32.const 0
  end //nAfter:
  set_local #x
```

Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the "rest" of the program.

Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the "rest" of the program.

$\Rightarrow$ We can generalize our translation function $[\![\cdot]\!]$ to take a destination label designating the "rest" in the surrounding code.

Destination label parameters

Recall that in **`branch`**`(c,nThen,nElse)` we had two arguments `nThen` and `nElse`, which told us where to jump to execute code of the corresponding branches.

Similarly, up until now we explicitly enclosed our translated program fragments in an `nAfter` block, so we could jump to the "rest" of the program.

$\Rightarrow$ We can generalize our translation function $[\,\cdot\,]$ to take a destination label designating the "rest" in the surrounding code.

$$[\,\cdot\,] \Rightarrow [\,\cdot\,]\,\texttt{nAfter}$$

$\Rightarrow$ The caller of the translation function determines where to continue!

Translations with an nAfter label parameter (1)

```
$[x = e]$ nAfter :=
  block nSet
    $[e]$ nSet
    // note that the rest of this block is never reached!
  end //nSet:
  set_local #x
  br nAfter

$[s_1; s_2]$ nAfter :=
  block nSecond
    $[s_1]$ nSecond
  end //nSecond:
  $[s_2]$ nAfter
```

Translations with an nAfter label parameter (2)

```
[if ($e_{cond}$) $e_{then}$ else $e_{else}$] nAfter :=
  block nElse
    block nThen
      branch($e_{cond}$,nThen,nElse)
    end //nThen:
    [$e_{then}$] nAfter
  end //nElse:
  [$e_{else}$] nAfter

[return $e$] nAfter :=
  block nRet
    [$e$] nRet
  end //nRet:
  return
```

Switch statements

Let us assume our language had a switch statement (like C and Java do, for instance):

```
switch ($e_{scrutinee}$) {
  case $c_1$: $e_1$
  ...
  case $c_n$: $e_n$
  default: $e_{default}$
}
```

▷ How can we compile such switch statements?

Compiling switch statements

```
[$s_{switch}$] nAfter :=
  block nDefault
    block nCase$_n$
      ...
        block nCase$_1$
          block nTest
            [$e_{scrutinee}$] nTest
          end //nTest:
          tee_local #s   (where s is some fresh local of type i32)
          i32.const $c_1$; i32.eq; br_if nCase$_1$
          get_local #s
          i32.const $c_2$; i32.eq; br_if nCase$_2$
          ...
          br nDefault
        end //nCase$_1$:
        [$e_1$] nCase$_2$
      ...
    end //nCase$_n$:
    [$e_n$] nDefault
  end //nDefault:
  [$e_{default}$] nAfter
```

Compiling switch statements

```
[$s_{switch}$] nAfter :=
 block nDefault
  block nCase$_n$
   ...
    block nCase$_1$
     block nTest
      [$e_{scrutinee}$] nTest
     end //nTest:
     tee_local #s   (where s is some fresh local of type i32)
     i32.const $c_1$; i32.eq; br_if nCase$_1$
     get_local #s
     i32.const $c_2$; i32.eq; br_if nCase$_2$
     ...
     br nDefault
    end //nCase$_1$:
    [$e_1$] nCase$_2$
   ...
  end //nCase$_n$:
  [$e_n$] nDefault
 end //nDefault:
 [$e_{default}$] nAfter
```

▷ How do we translate break?

Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

$\Rightarrow$ Let us extend the translation function by another label parameter.

Compiling switch statements

At any point during the translation of **`switch`** we want to keep track not only where to jump *after*, but also where to jump on a `break`!

⇒ Let us extend the translation function by another label parameter.

$$[\,\cdot\,]\ \texttt{nAfter} \Rightarrow [\,\cdot\,]\ \texttt{nAfter}\ \texttt{nBreak}$$

⇒ The caller of the translation function determines where to continue in the "normal" case, but also when `break` is called!

Compiling switch statements

Translating `break` then is straightforward: One simply ignores `nAfter` and follows `nBreak` instead.

```
[break] nAfter nBreak :=
  br nBreak
```

▷ What do we have change in our translation of `switch` statements?

Compiling switch statements with breaks

```
[$s_{switch}$] nAfter nBreak :=
 block nDefault
  block nCase$_n$
   ...
    block nCase$_1$
     block nTest
      [$e_{scrutinee}$] nTest nBreak
     end //nTest:
     tee_local #s   (where s is some fresh local of type i32)
     i32.const $c_1$; i32.eq; br_if nCase$_1$
     get_local #s
     i32.const $c_2$; i32.eq; br_if nCase$_2$
     ...
     br nDefault
    end //nCase$_1$:
    [$e_1$] nCase$_2$ nAfter
   ...
  end //nCase$_n$:
  [$e_n$] nDefault nAfter
 end //nDefault:
 [$e_{default}$] nAfter nAfter
```

Translating While Statement

Consider translation of the **while** statement, which gets 'nextLabel' destination, specifying where to jump when exiting the loop.
We assume that the instructions emitted are inside the block that introduced nextLabel.

What is the translation schema?

```
[ while (cond) stmt ] nextLabel =
```

Translating While Statement

Consider translation of the **while** statement, which gets 'nextLabel' destination, specifying where to jump when exiting the loop.
We assume that the instructions emitted are inside the block that introduced nextLabel.

What is the translation schema?

```
[ while (cond) stmt ] nextLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel
  end
```

break Statement

In many languages, a break statement can be used to exit from the loop. For example, it is possible to write code such as this:

```
while (cond1) {
  code1
  if (cond2) break;
  code2
}
```

Loop executes `code1` and checks the condition `cond2`. If condition holds, it exists. Otherwise, it continues and executes `code2` and then goes to the beginning of the loop, repeating the process.

Give translation scheme for this loop construct and explain how the translation of other constructs needs to change.

break Statement - Propagating Exit Label

For a **break** statement to know where to jump, it needs to be given a label indicating the exit of the loop. When we translate a statement (such as **if**) potentially containing **break**, the translation of this statement needs both the parameter to pass on to **break** as well as the parameter to jump to during normal execution. Therefore, each statement needs two destination parameters: the 'nextLabel' and the 'loopExit' label. For example,

```
[ if (cond) thenC else elseC ] nextL loopExitL =
```

break Statement - Propagating Exit Label

For a **break** statement to know where to jump, it needs to be given a label indicating the exit of the loop. When we translate a statement (such as **if**) potentially containing **break**, the translation of this statement needs both the parameter to pass on to **break** as well as the parameter to jump to during normal execution. Therefore, each statement needs two destination parameters: the 'nextLabel' and the 'loopExit' label. For example,

```
[ if (cond) thenC else elseC ] nextL loopExitL =
  block elseL
    block thenL
      branch(cond, thenL, elseL)
    end // thenL
    [thenC] nextL loopExitL
  end // elseL
  [elseC] nextL loopExitL
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ]
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel nextLabel
  end
```

break Statement - Using and Setting Labels

Translating **break**:

```
[ break ] nextLabel loopExitLabel =
  br loopExitLabel
```

Translating while:

```
[ while (cond) stmt ] nextLabel loopExitLabel =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextLabel)
    end // bodyLabel
    [ stmt ] startLabel nextLabel
  end
```

What if we want to have **continue** that goes to beginning of the loop?

Loops with break and continue

Translating **break**:

```
[ break ] nextL loopExitL loopStartL =
  br loopExitL
```

Translating **continue**:

```
[ continue ] nextL loopExitL loopStartL =
  br loopStartL
```

Translating while:

```
[ while (cond) stmt ] nextL loopExitL loopStartL =
  loop startLabel
    block bodyLabel
      branch(cond, bodyLabel, nextL)
    end // bodyLabel
    [ stmt ] startLabel nextL startLabel
  end
```

Explain difference between labels `loopStartL` and `startLabel`

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 - slow even with cache

A **few fast** registers

R0,R1,…,R31

Basic Instructions of Register Machines

$R_i \leftarrow \mathrm{Mem}[R_j]$load

$\mathrm{Mem}[R_j] \leftarrow R_i$ 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

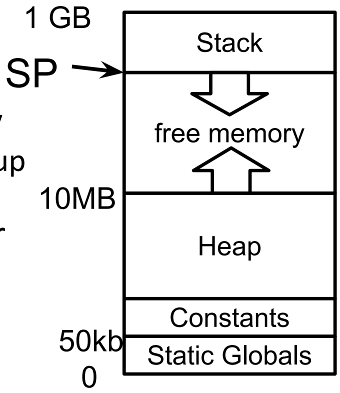
Both dynamically allocated heap and stack expand

Heap is **more general**:

- Can allocate, read/write, 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 efficient: top of stack pointer (SP) is a register

- allocation is simple: increment, decrement
- to allocate N bytes on stack (**push**): **SP := SP - N**
- to deallocate N bytes on stack (**pop**): **SP := SP + N**



Exact picture varies depend on hardware, OS, language runtime

WASM vs General Register Machine Code
Naïve Correct Translation

WASM:

imul.32

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 7 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
```

How many variables?

x,y,z,xy,xz,res1

Do we need 7 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
```

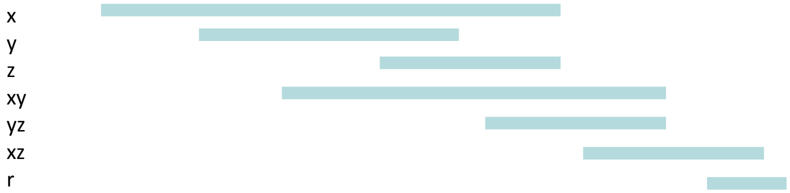
Idea of Register Allocation

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}   {}



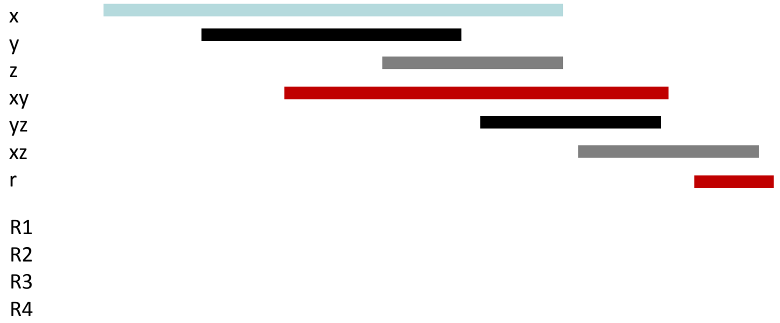
Color Variables
Avoid Overlap of Same Colors

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

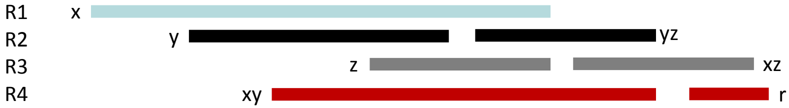
program:

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



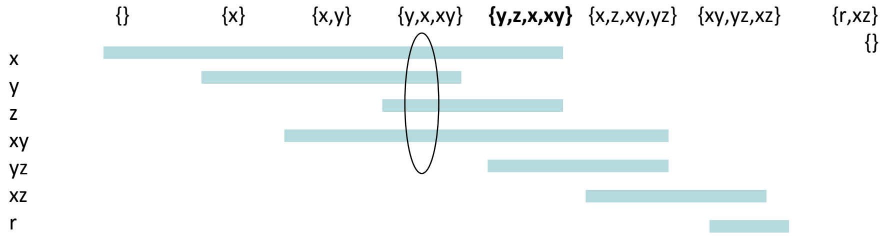
Each color denotes a register
4 registers are enough for this 7-variable program

How to assign colors to variables?

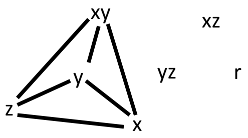
program:

```
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:



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



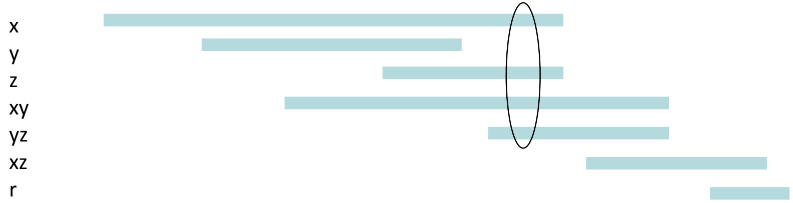
Edges between members of each set

program:

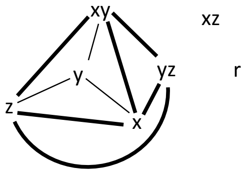
x = m[0]; y = m[1]; xy = x*y; z = m[2]; yz = y*z; xz = x*z; r = xy + yz; m[3] = r + xz

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**



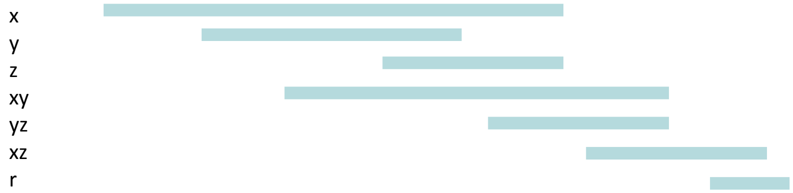
Final interference graph

program:

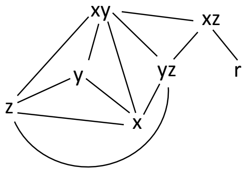
```
x = m[0];   y = m[1];   xy = x*y;   z = m[2];   yz = y*z;   xz = x*z;   r = xy + yz;   m[3] = r + xz
```

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**



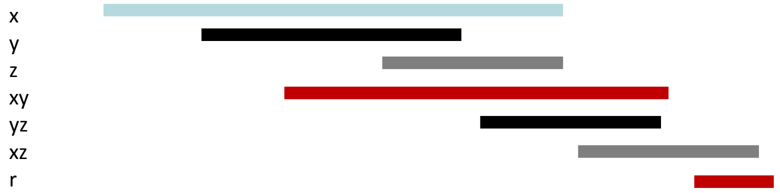
Coloring interference graph

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}   {}
```

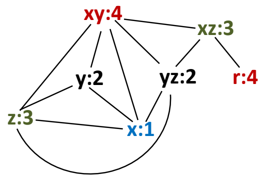


Need to assign colors (register numbers) to nodes such that:

if there is an edge between nodes,

then those nodes have different colors.

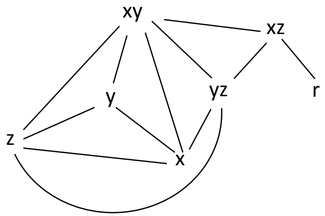
→ standard graph vertex coloring problem



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

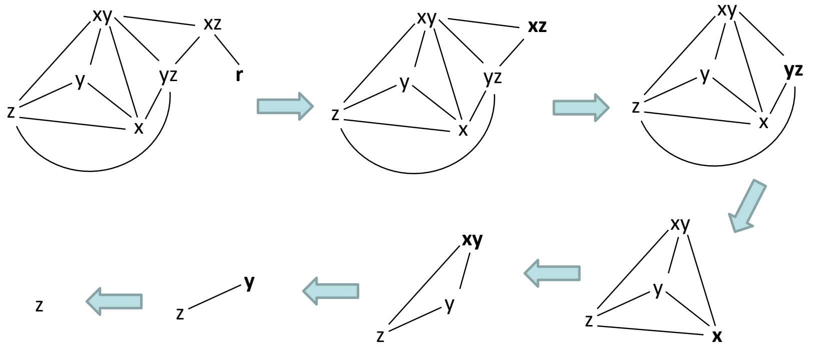


- NP hard
- In practice, we have heuristics that work for typical graphs
- If we cannot fit it all variables into registers,
  perform a **spill:**

  store variable into memory and load again before using

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)



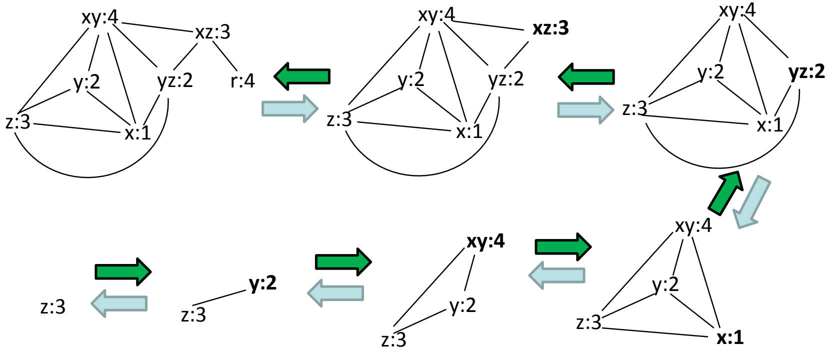
Heuristic for Coloring with K Colors

Select

Assign colors backwards, adding nodes that were removed

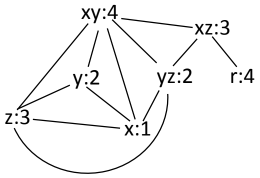
If the node was removed because it had <K neighbors, we will always find a color

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!

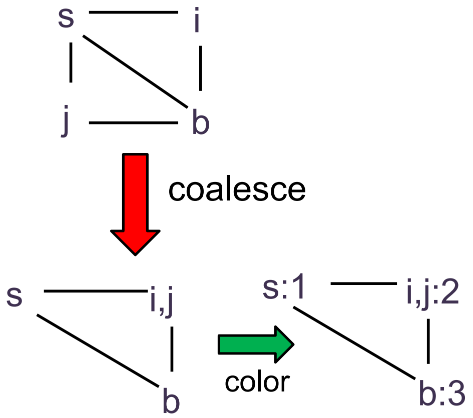
Conservative coalescing: coalesce only if merged node of tmp1 and tmp2 will have a small degree so that we are sure that we will be able to color it (e.g. resulting node has degree < K)

Run Register Allocation
use 3 registers, coalesce j=i

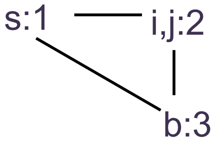
```
i = 0

s = s + i

i = i + b

j = i

s = s + j + b

j = j + 1
```

Run Register Allocation
use 3 registers, coalesce 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 use 3 registers, coalesce 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
```