Compilation

0368-3133 2014/15a

Lecture 10

Register Allocation II

Noam Rinetzky

Register allocation

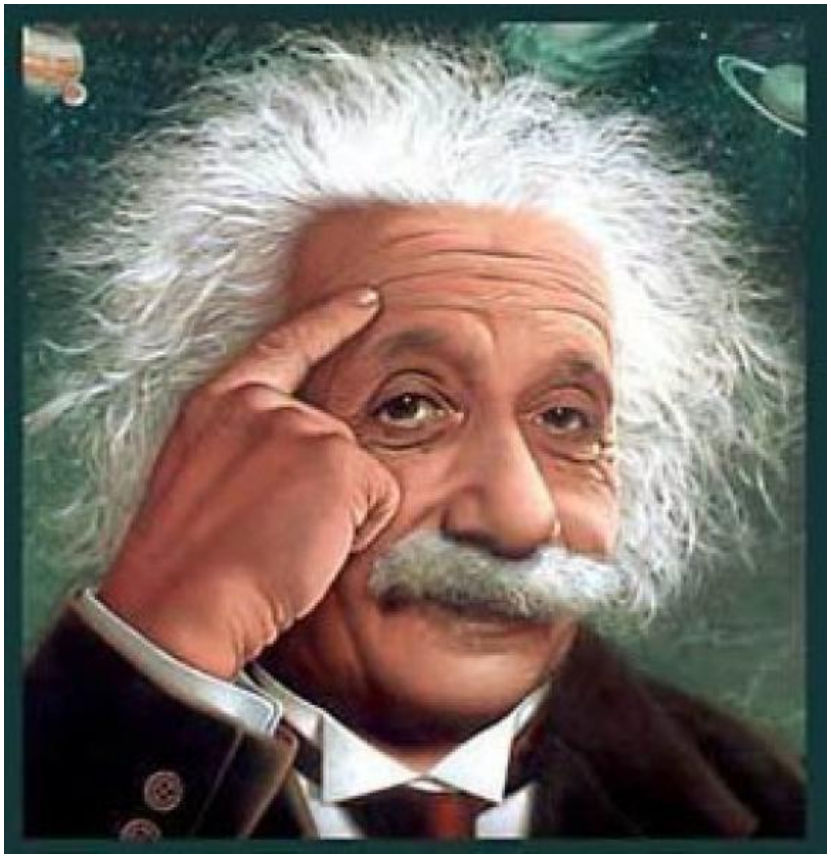
- **Registers**: dedicated memory locations
  - Quick access
  - Used to perform computations
- Number of registers is limited
- Need to allocate them in a clever way
  - RA is a critical step in any compiler
    - Huge effect on performance

(Too) simple approach

- Load operands from main memory
- Compute
- Store result in memory

- Problem: inefficient execution due to moving data to & from memory

Solution: smart allocation



Goal: Reduce number of temporaries*

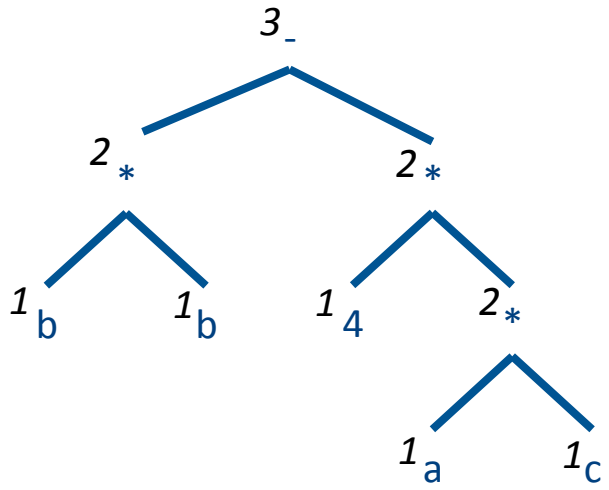
- Single expressions
  - Sethi-Ullman (heavy tree first)
  - Sethi-Ullman + spilling (light sub-tree first)
- Basic blocks
  - AST → Dependency graph → DAG → Code
- Control flow graphs (CFG) of procedures
  - Using liveness analysis

*Temporaries ~ registers

Goal: Reduce number of temporaries*

- Sethi & Ullman RA for single expressions
  - 👍 Optimal number of registers
  - 👎 Unbounded number of registers
  - 👎 Only for side effect-free expressions

b*b-4*a*c



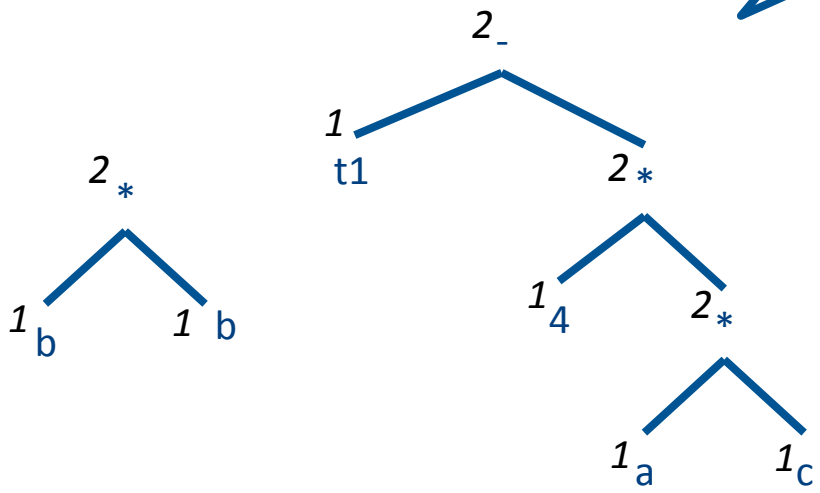
Gen. code for "heavy" tree first

Sethi & Ullman RA + spilling

- Turn a single expression to a sequence of assignments
  - 👎 Optimal number of registers
  - 👍 Bounded number of registers
  - 👎 Only for side effect-free expressions

Gen. code for "light" sub-tree first

b*b-4*a*c



Spilling
- Store registers in stack
- Load when needed

t1 = b*b; t1-4*a*c

Basic Blocks

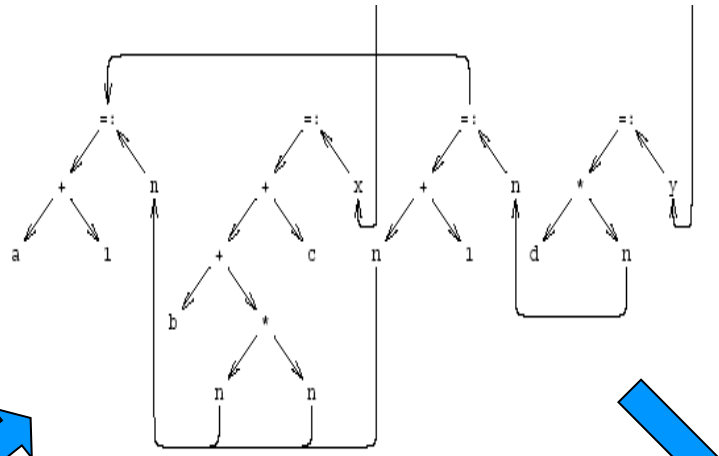
- Given a sequence of instructions

- **Basic block** is a subsequence with
  - **single entry** (to first instruction)
    - No jumps to the middle of the block
  - **single exit** (last instruction)

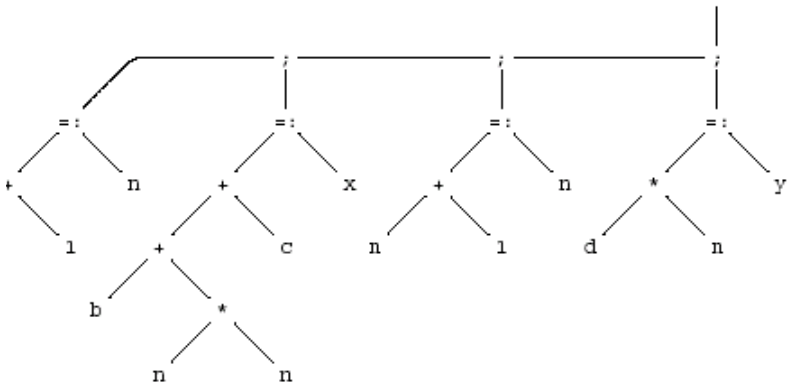
➢ Code execute as a sequence

AST for a Basic Block

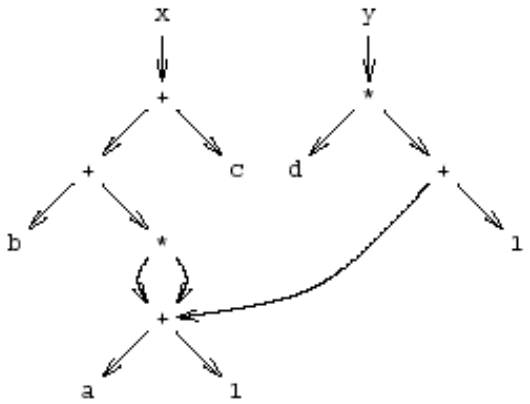
```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```



Dependency graph



AST



Simplified Dependency graph

AST for a Basic Block

```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,X1

Load_Reg    X1,R1
Mult_Reg    X1,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x

Load_Reg    X1,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,R2

Load_Reg    R2,R1
Mult_Reg    R2,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x

Load_Reg    R2,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

Non optimized

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,X1

Load_Reg    X1,R1
Mult_Reg    X1,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x

Load_Reg    X1,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

Optimized

Data dependencies

- Inside expressions
  - Operator depends on operands
  - Assignment depends on assigned expressions
- Between statements
  - From assignments to their use

BB➔Dependency graphs ➔ Code

- Define a partial order on assignments
  - a < b ⇔ a must be executed before b
- BB ➔Dependency graph
  - Represented as a directed graph
    - Nodes are assignments
    - Edges represent dependency
- Simplify Dependency graph
  - Shortcut evaluation
- Serialize dependency graph ➔ Code

Simplified Dependency Graph → Code

- Linearize the dependency graph
  - Instructions must follow dependency
  - Many solutions exist
  - Machine dependent instructions

- Linearize with infinite number of registers
  - “Symbolic” registers
  - Assign physical registers later
    - May need additional spill

“Global” Register Allocation

- Register allocation for multiple basic blocks (CFGs)

Input

- Sequence of instructions using unbounded number of temporary variables
  - aka symbolic registers
- “machine description”
    - # of registers, restrictions

Output

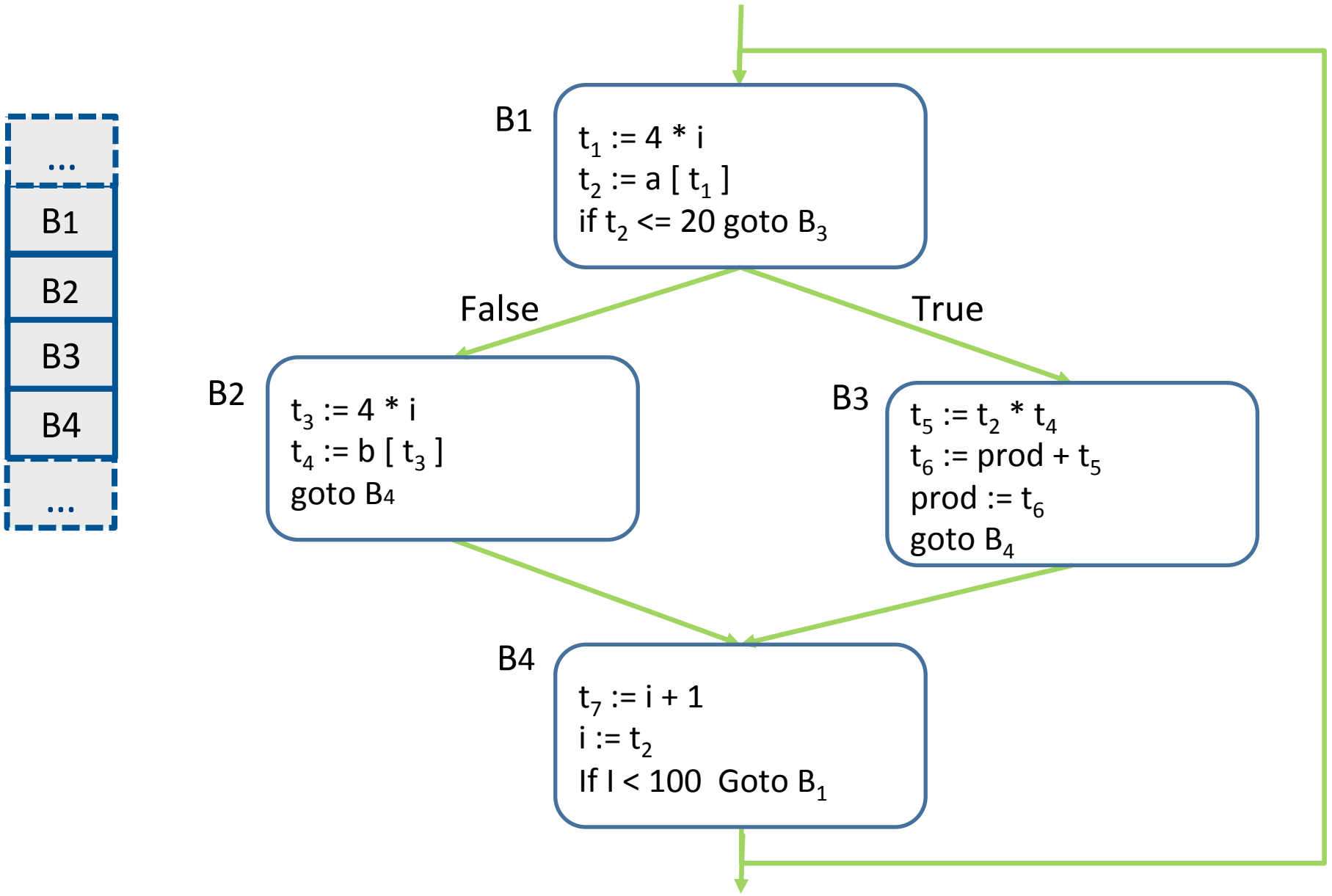
- Sequence of instructions using (bounded number) of physical registers
  - Some MOV instructions removed

Global Register allocation

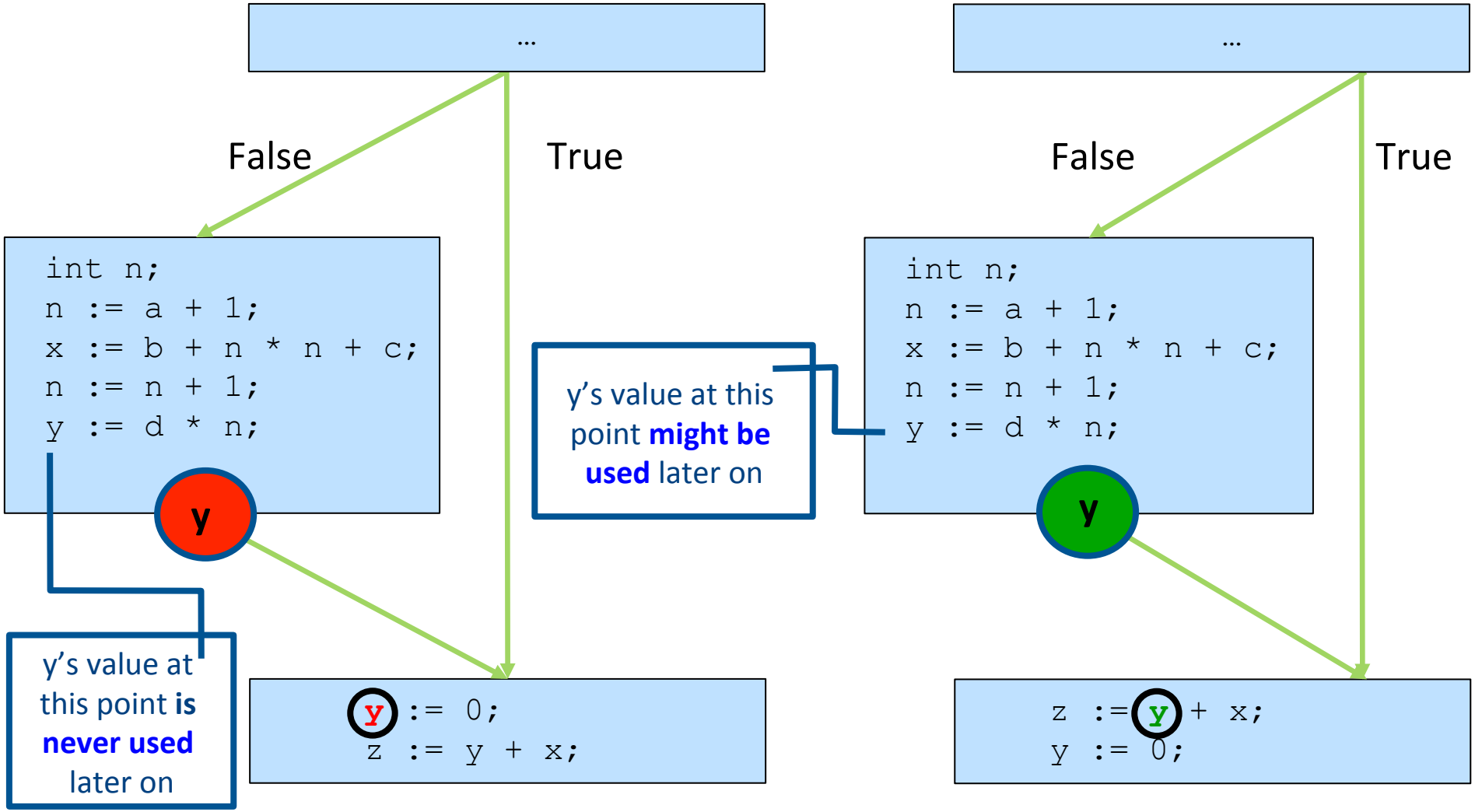
- Use liveness analysis to determine live variables at every program point
- Computes interference graphs
- Allocate same register to non-interfering variables
  - Variable interfere if both are live at a BB exit

Not dependency graphs!

Control Flow Graphs (CFGs)

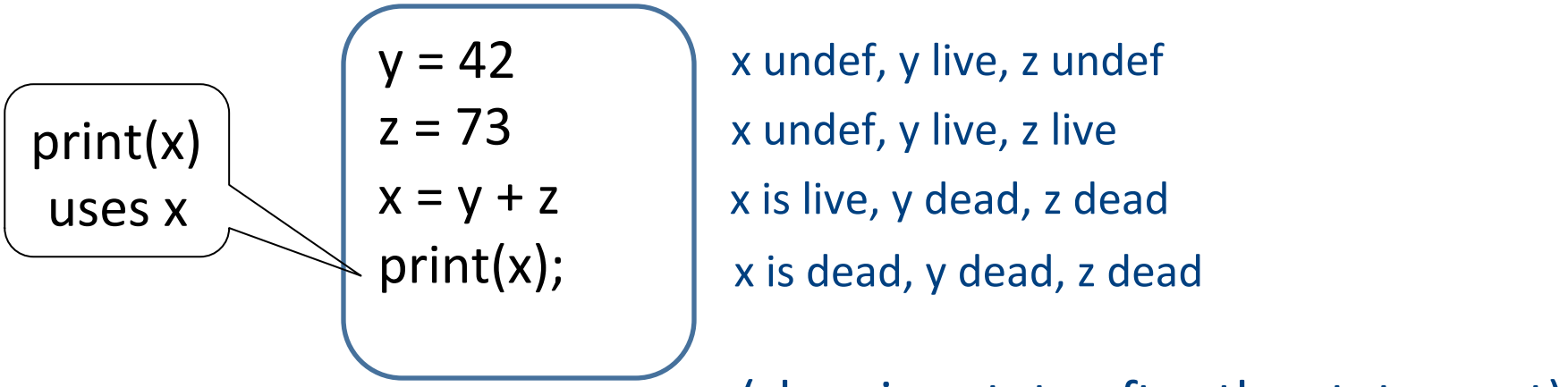


Liveness information



Variable Liveness

- A statement x = y + z
  - **defines** x
  - **uses** y and z
- A variable x is live at a program point if its value (at this point) is used at a later point



Interference

- For every node n in CFG, we have out[n]
  - Set of temporaries live out of n
- Two variables *interfere* if they appear in the same out[n] of any node n
  - **Cannot be allocated to the same register**
- Conversely, if two variables do not interfere with each other, they can be assigned the same register
  - We say they have disjoint live ranges
- How to assign registers to variables?

Interference graph

- Nodes of the graph = variables
- Edges connect variables that interfere with one another
- Nodes will be assigned a color corresponding to the register assigned to the variable
- Two colors can’t be next to one another in the graph

Interference graph construction

```
b = a + 2

c = b * b

b = c + 1

return b * a
```

Interference graph construction

```
b = a + 2

c = b * b

b = c + 1
                    {b, a}
return b * a
```

Interference graph construction

```
b = a + 2

c = b * b
                    {a, c}
b = c + 1
                    {b, a}
return b * a
```

Interference graph construction

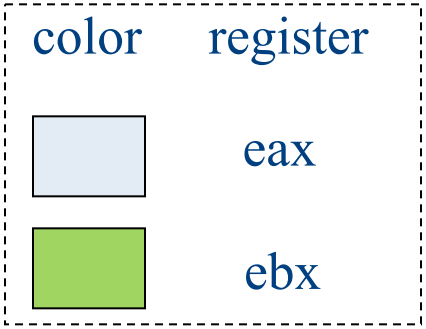
```
b = a + 2
                    {b, a}
c = b * b
                    {a, c}
b = c + 1
                    {b, a}
return b * a
```

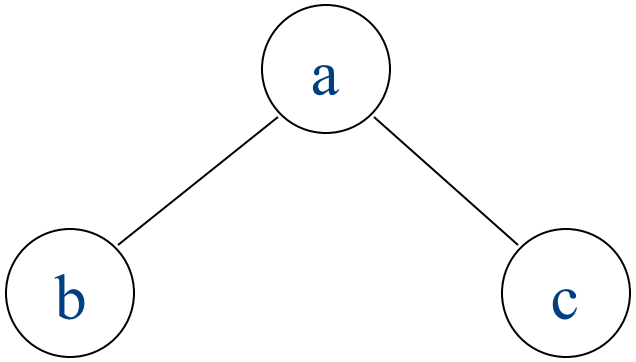
Interference graph construction

```
                    {a}
b = a + 2
                    {b, a}
c = b * b
                    {a, c}
b = c + 1
                    {b, a}
return b * a
```

Interference graph

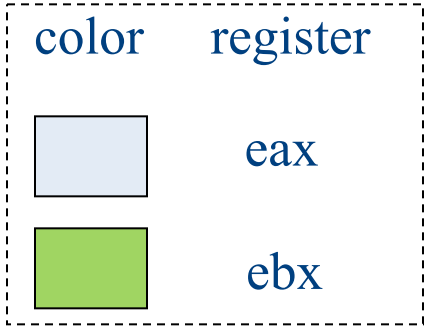
```
                    {a}
b = a + 2
                    {b, a}
c = b * b
                    {a, c}
b = c + 1
                    {b, a}
return b * a
```

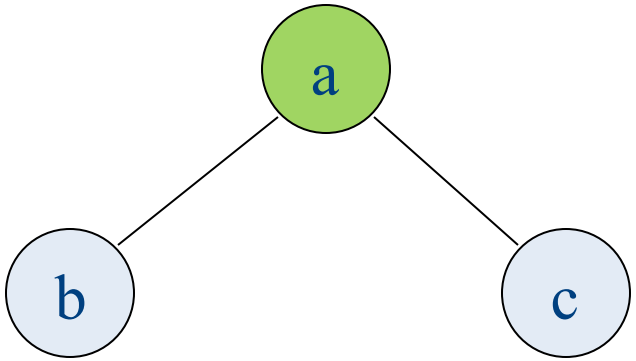




Colored graph

```
                    {a}
b = a + 2
                    {b, a}
c = b * b
                    {a, c}
b = c + 1
                    {b, a}
return b * a
```





Graph coloring

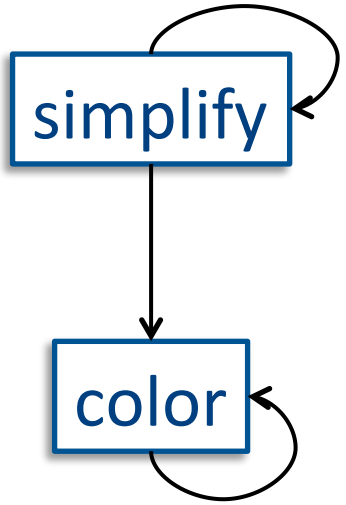
- This problem is equivalent to graph-coloring, which is NP-hard if there are at least three registers
- No good polynomial-time algorithms (or even good approximations!) are known for this problem
  - We have to be content with a heuristic that is good enough for RIGs that arise in practice

Coloring by simplification [Kempe 1879]

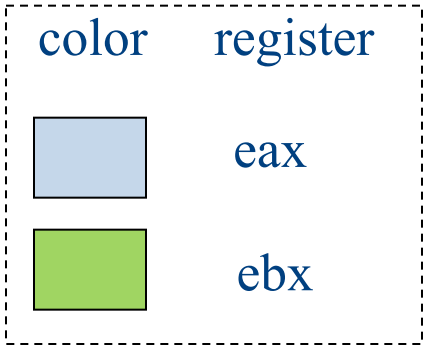
- How to find a **k**-coloring of a graph
- Intuition:
  - Suppose we are trying to ***k****-color a graph and find a node* with fewer than ***k*** *edges*
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in
  - Reason: fewer than *k neighbors* → *some color must be* left over

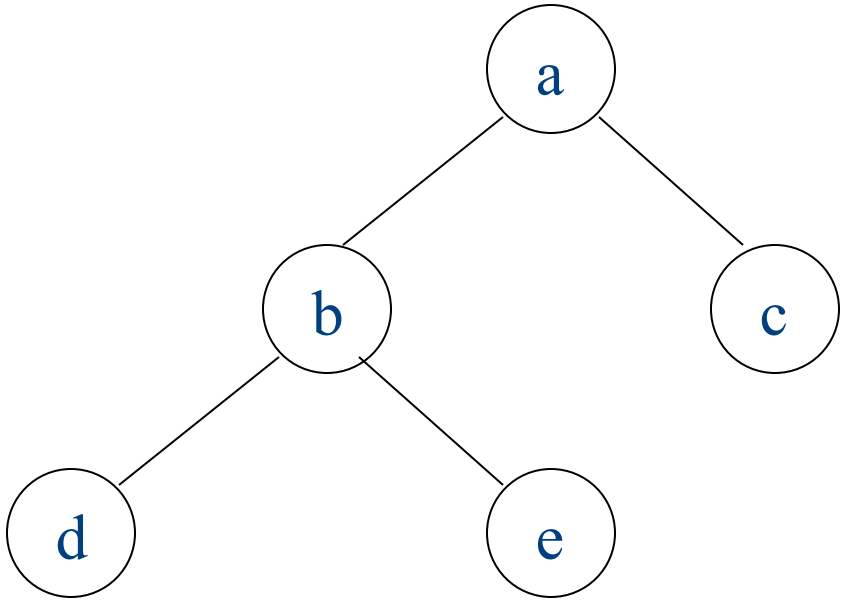
Coloring by simplification [Kempe 1879]

- How to find a k-coloring of a graph
- Phase 1: Simplification
  - Repeatedly simplify graph
  - When a variable (i.e., graph node) is removed, push it on a stack
- Phase 2: Coloring
  - Unwind stack and reconstruct the graph as follows:
  - Pop variable from the stack
  - Add it back to the graph
  - Color the node for that variable with a color that it doesn't interfere with



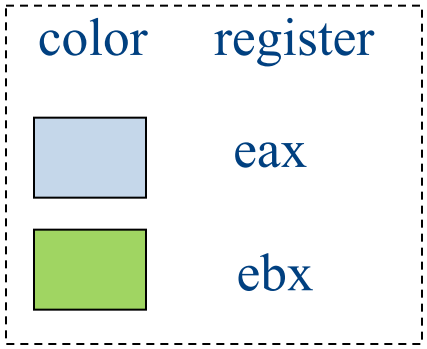
Coloring k=2

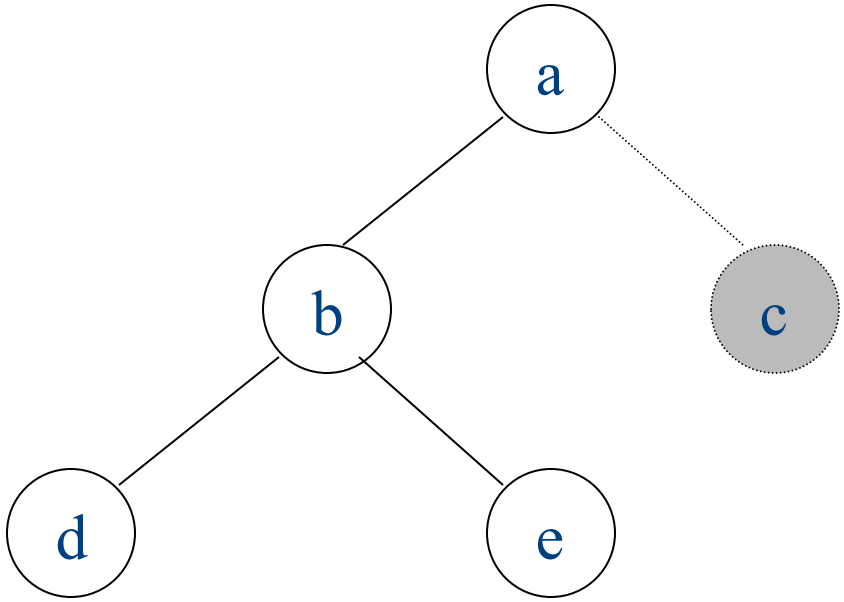




stack:

Coloring k=2

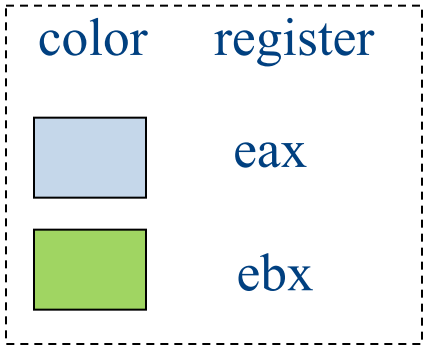


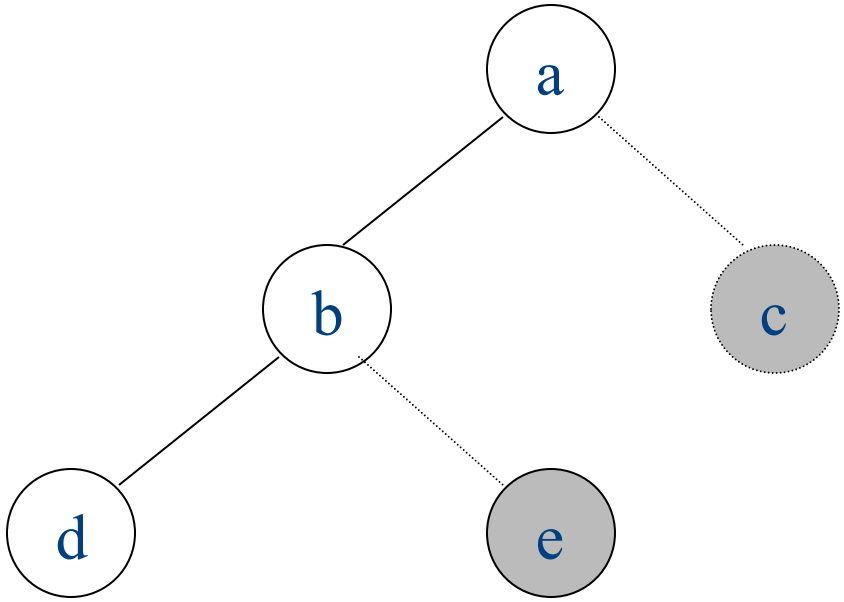


stack:

c

Coloring k=2



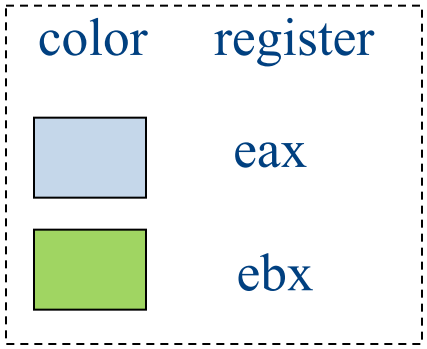


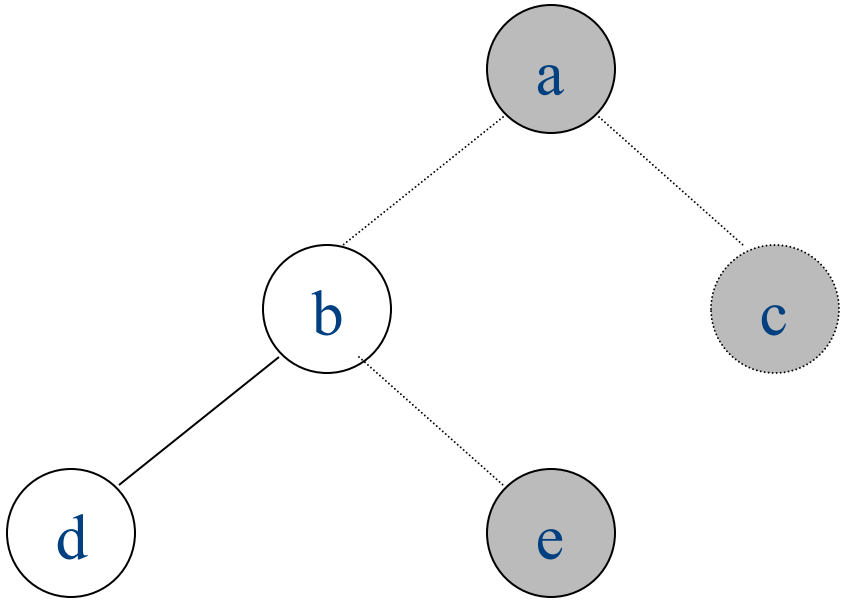
stack:

e

c

Coloring k=2





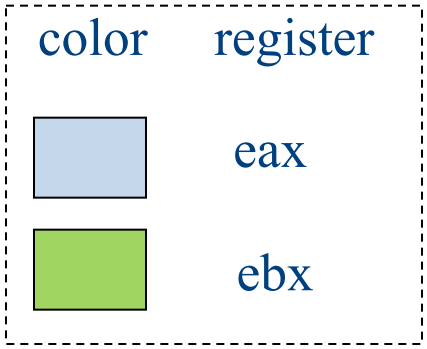
stack:

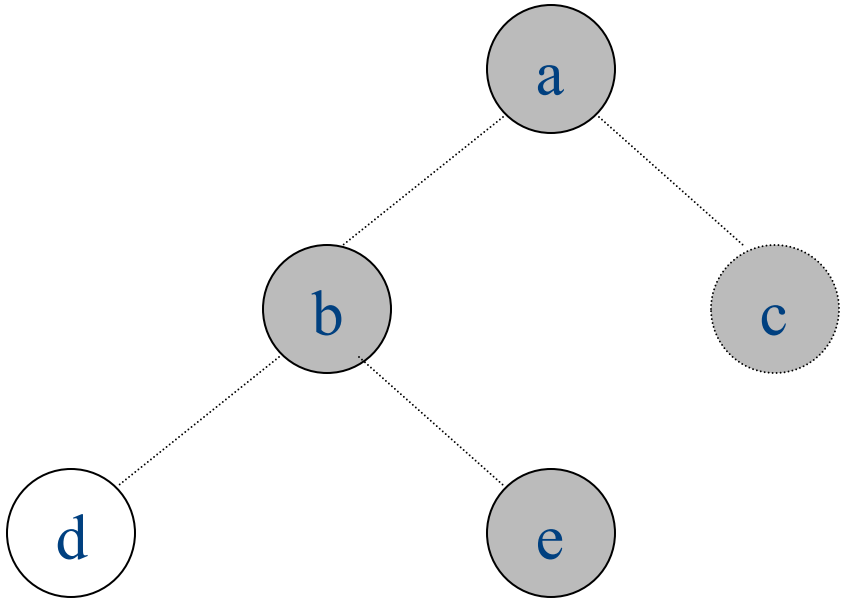
a

e

c

Coloring k=2





stack:

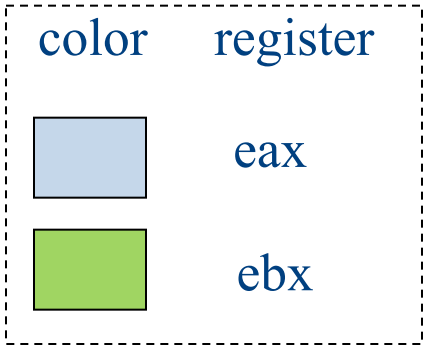
b

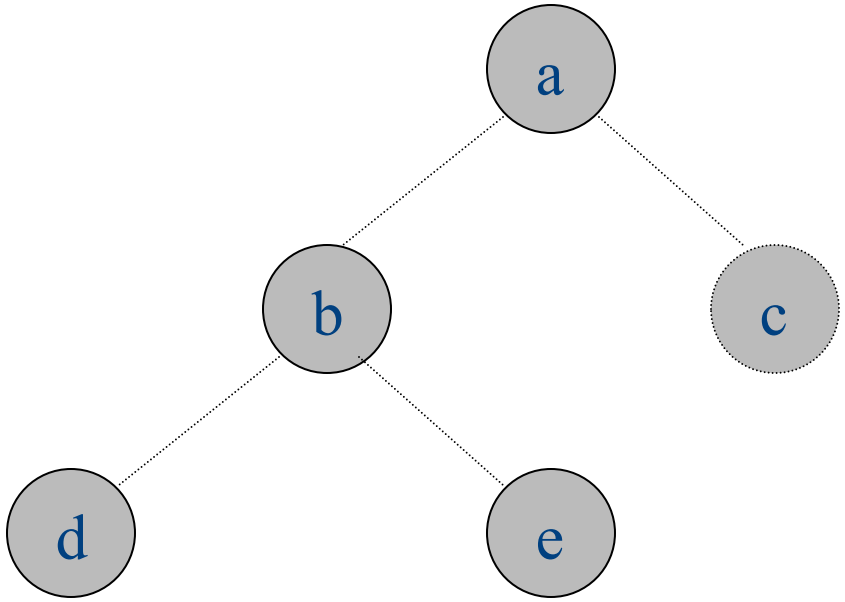
a

e

c

Coloring k=2





stack:

d

b

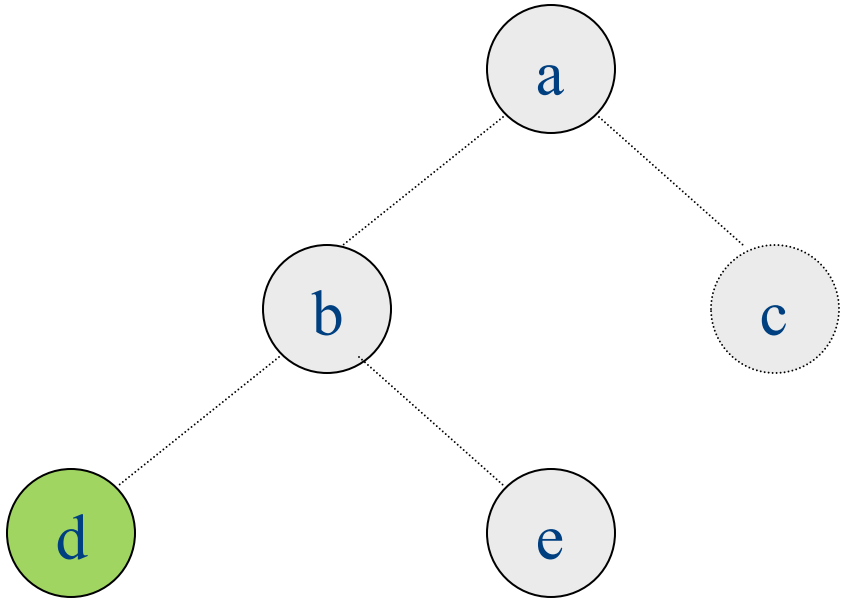
a

e

c

Coloring k=2

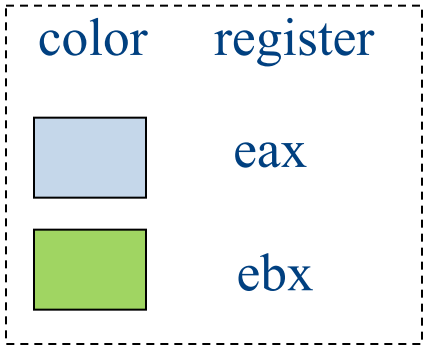
color	register
	eax
	ebx

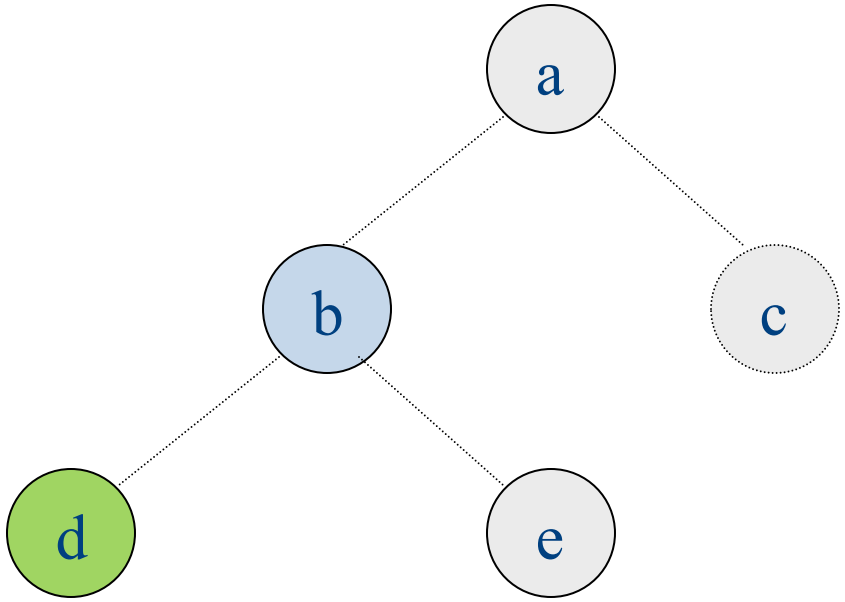


stack:

b
a
e
c

Coloring k=2





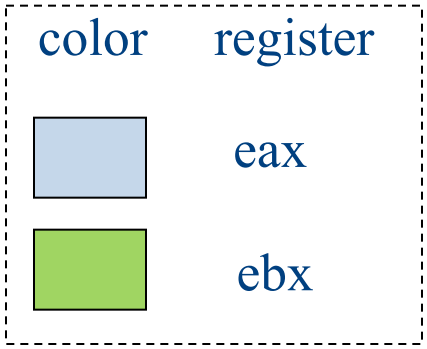
stack:

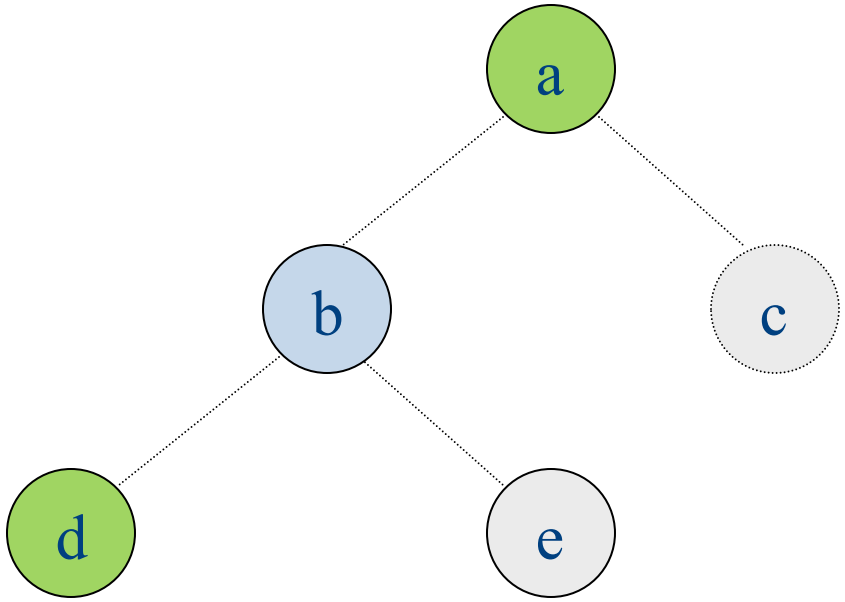
a

e

c

Coloring k=2



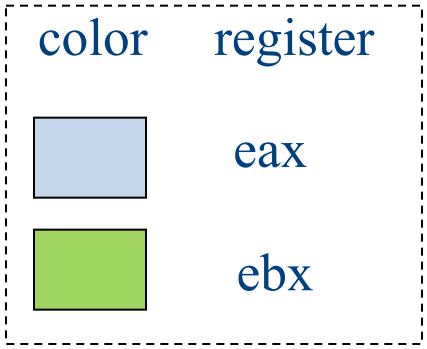


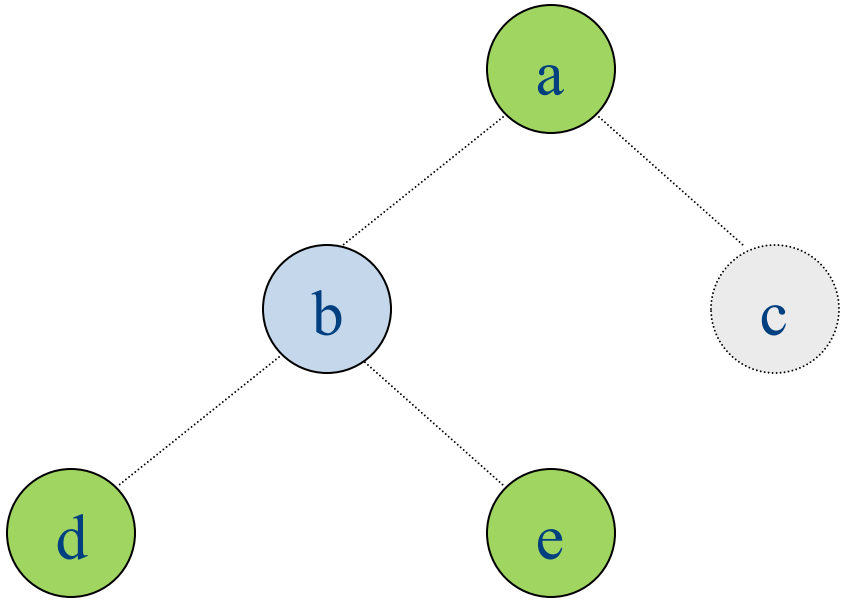
stack:

e

c

Coloring k=2

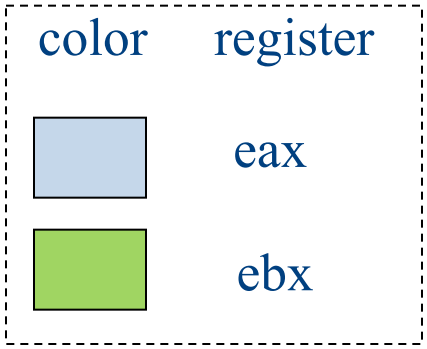


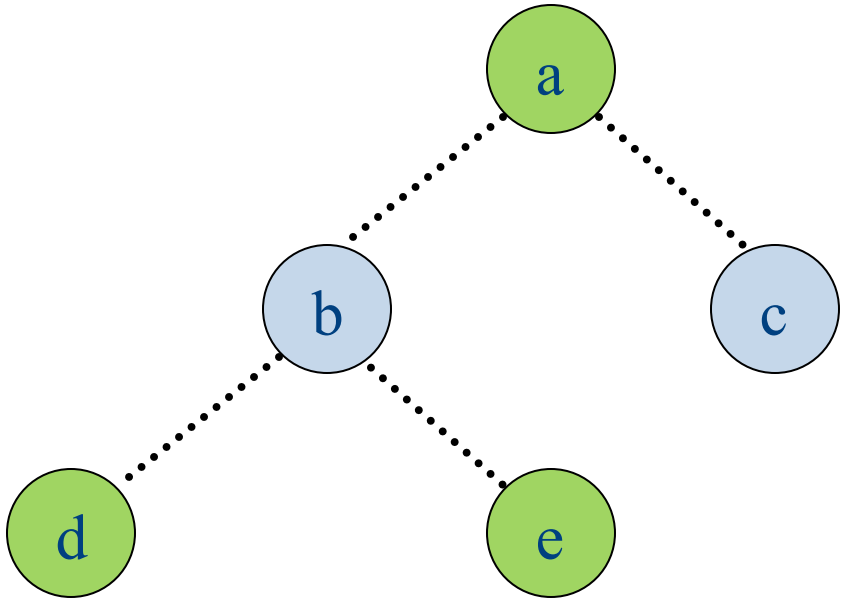


stack:

c

Coloring k=2



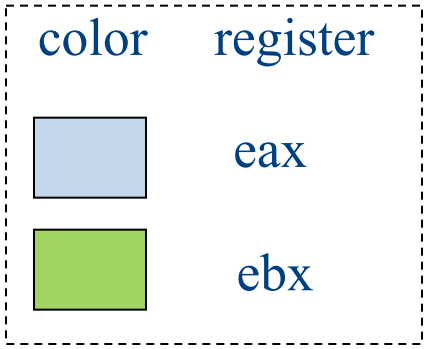


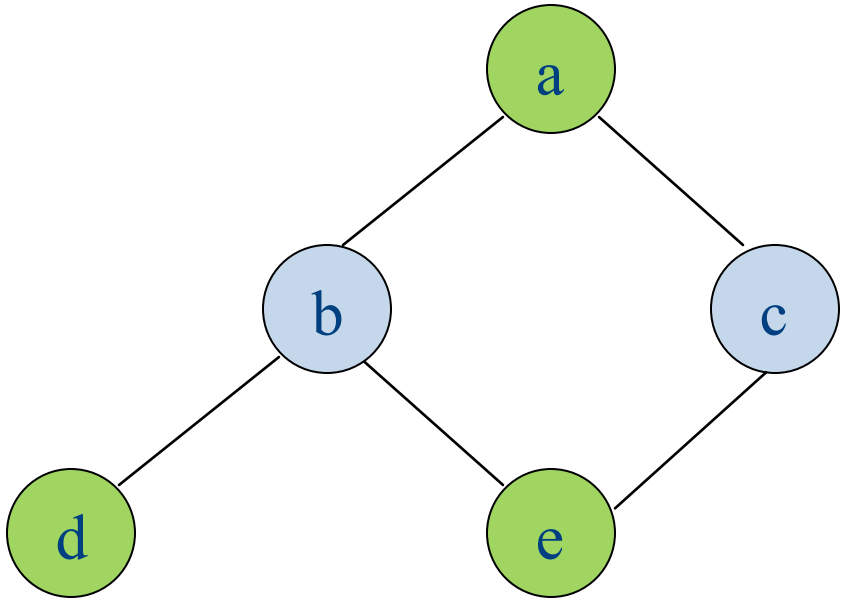
stack:

Failure of heuristic

- If the graph cannot be colored, it will eventually be simplified to graph in which every node has at least K neighbors
- Sometimes, the graph is still K-colorable!
- Finding a K-coloring in all situations is an NP-complete problem
  - We will have to approximate to make register allocators fast enough

Coloring k=2



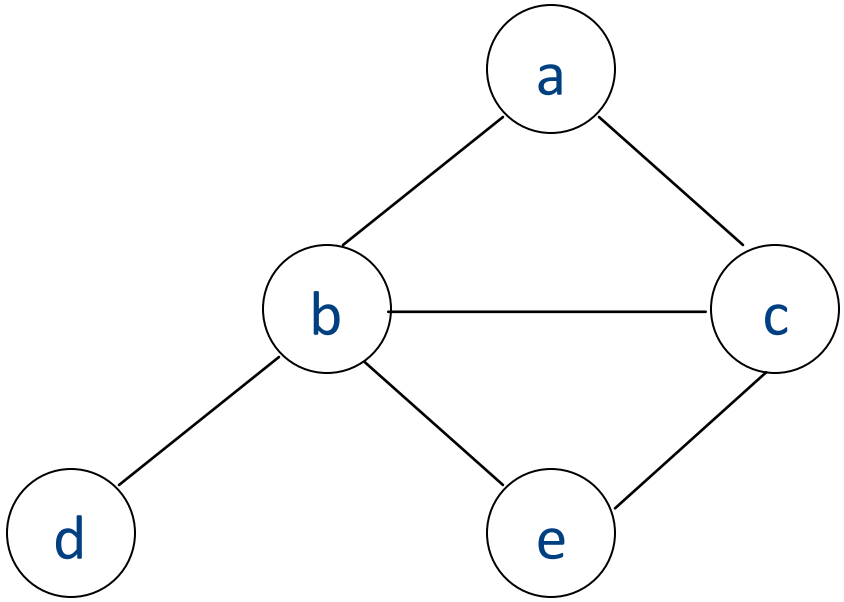


stack:

Coloring k=2

color	register
	eax
	ebx

Some graphs can't be colored in K colors:

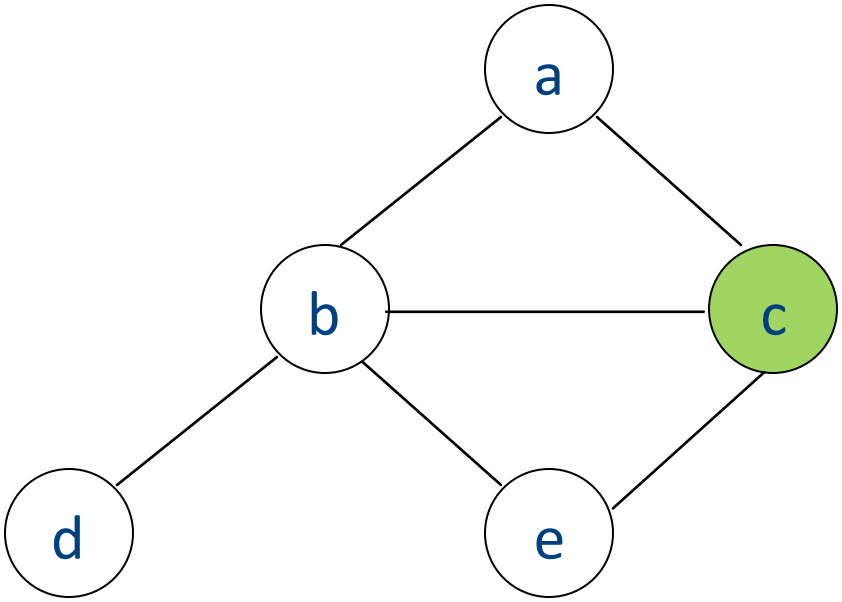


stack:
c
b
e
a
d

Coloring k=2

color	register
	eax
	ebx

Some graphs can't be colored in K colors:

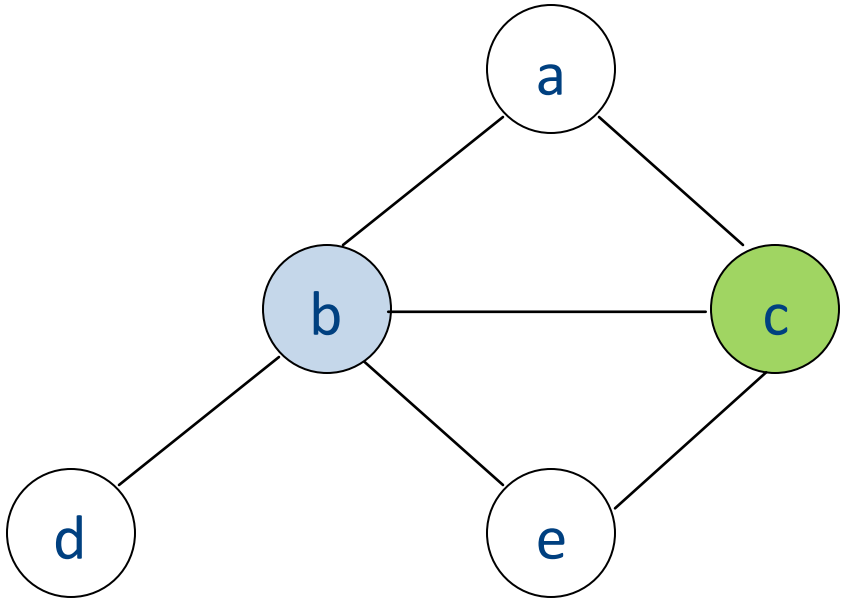


stack:
b
e
a
d

Coloring k=2

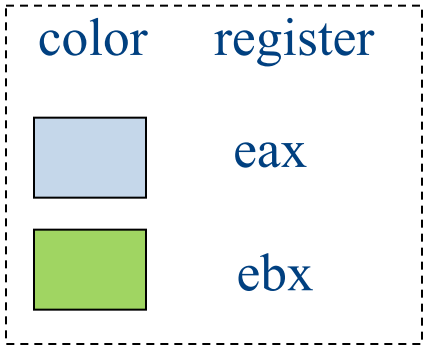
color	register
	eax
	ebx

Some graphs can't be colored in K colors:

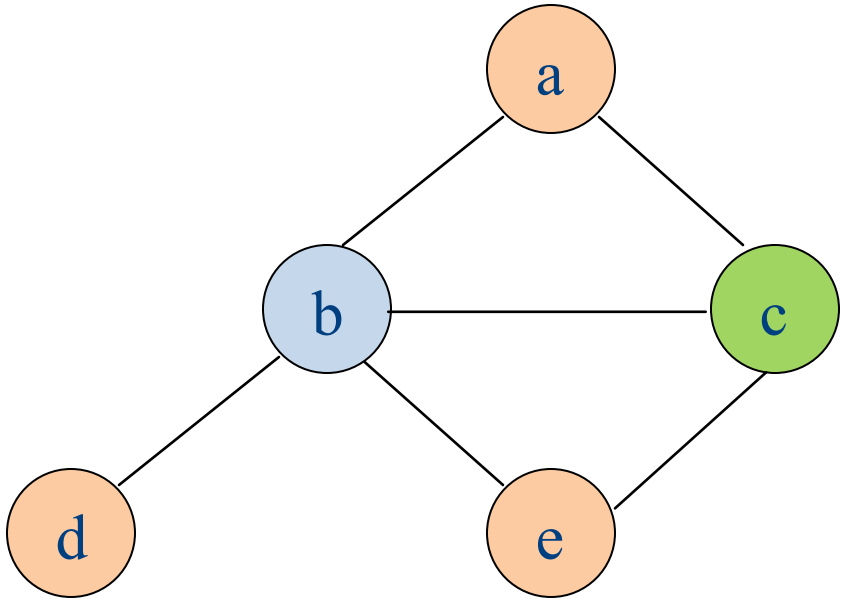


stack:
e
a
d

Coloring k=2



Some graphs can't be colored in K colors:



stack:
e
a
d

no colors left for e!

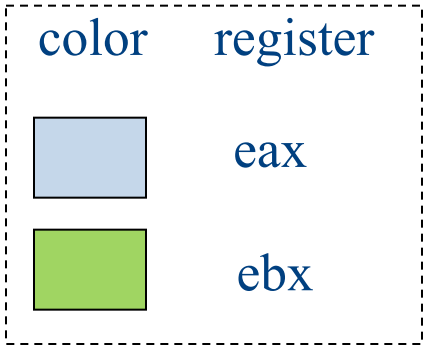
Chaitin's algorithm

- Choose and remove an arbitrary node, marking it "troublesome"
  - Use heuristics to choose which one
  - When adding node back in, it may be possible to find a valid color
  - Otherwise, we have to spill that node

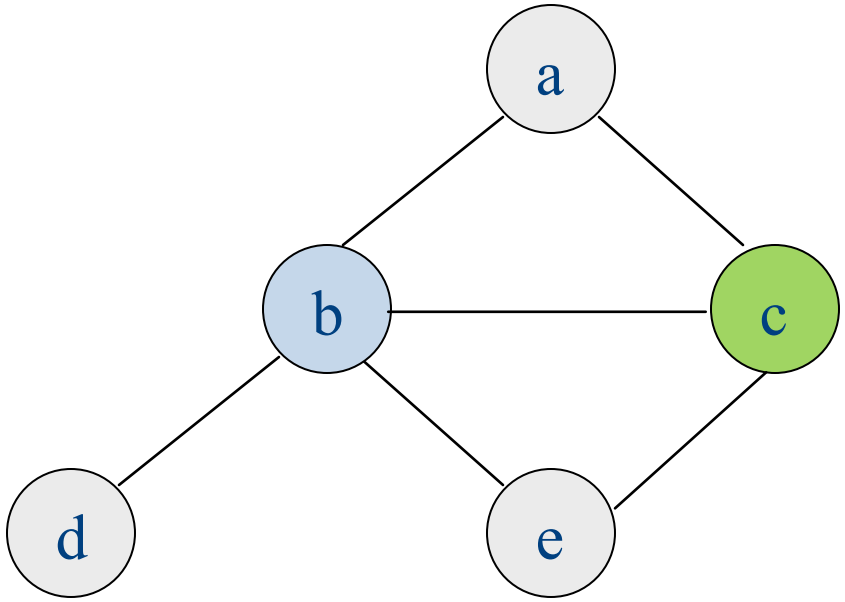
Spilling

- Phase 3: spilling
  - once all nodes have K or more neighbors, pick a node for spilling
    - There are many heuristics that can be used to pick a node
    - Try to pick node not used much, not in inner loop
    - Storage in activation record
  - Remove it from graph
- We can now repeat phases 1-2 without this node
- Better approach – rewrite code to spill variable, recompute liveness information and try to color again

Coloring k=2



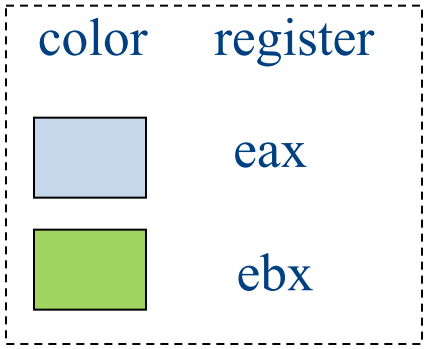
Some graphs can't be colored in K colors:



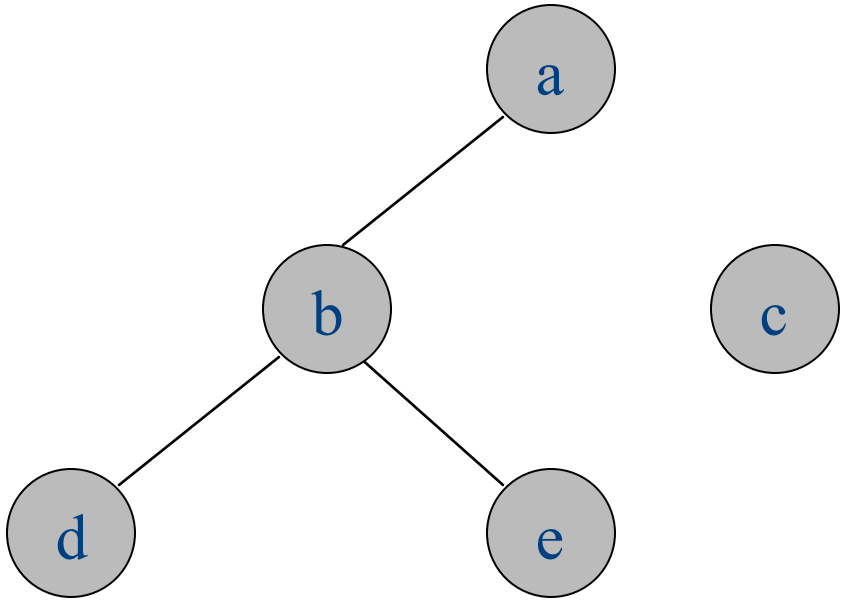
stack:
e
a
d

no colors left for e!

Coloring k=2

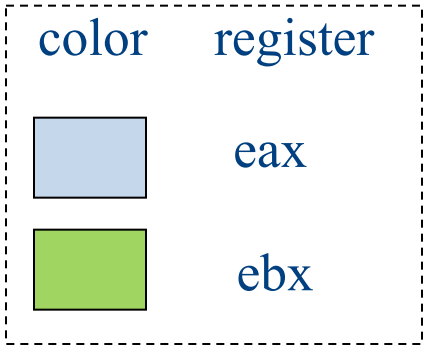


Some graphs can't be colored in K colors:

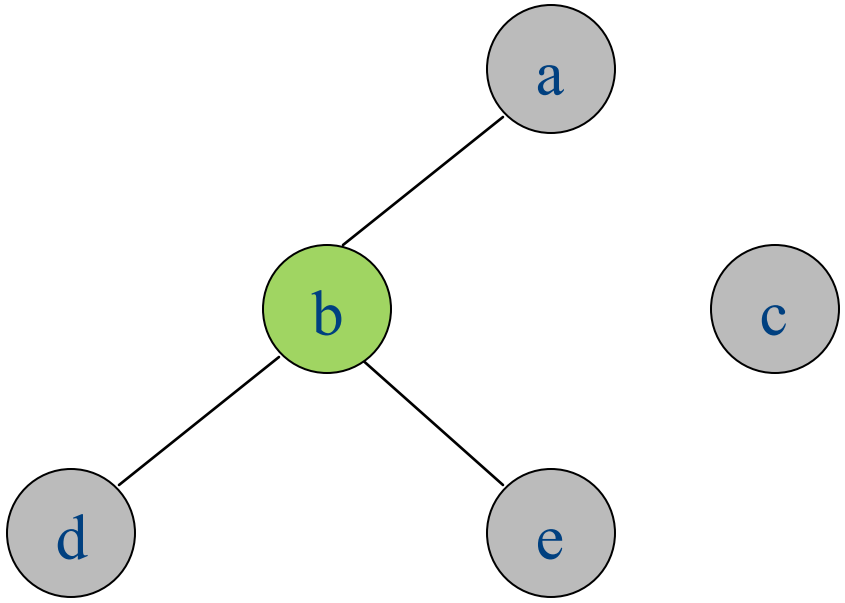


stack:
b
e
a
d

Coloring k=2

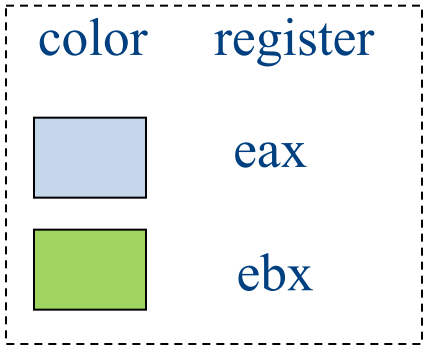


Some graphs can't be colored in K colors:

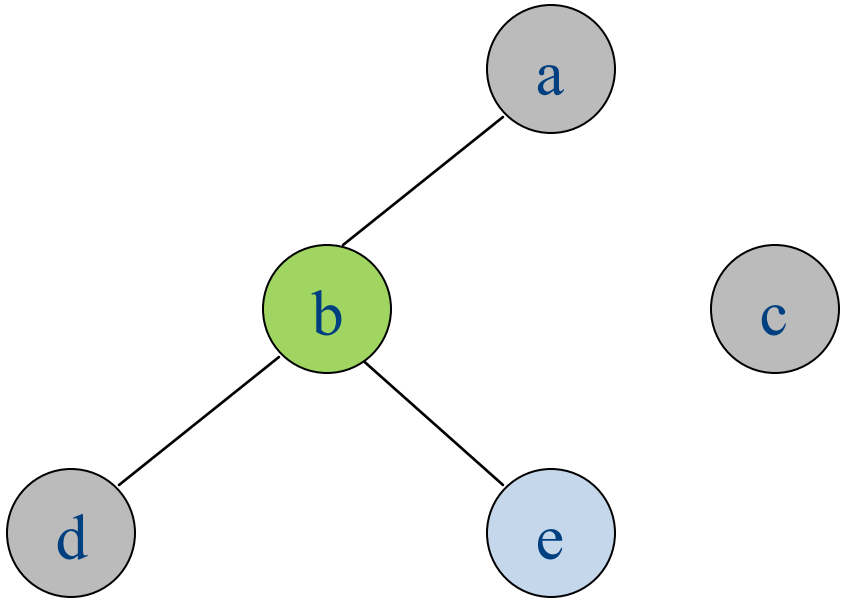


stack:
e
a
d

Coloring k=2

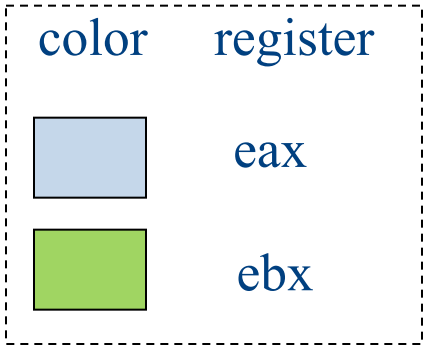


Some graphs can't be colored in K colors:

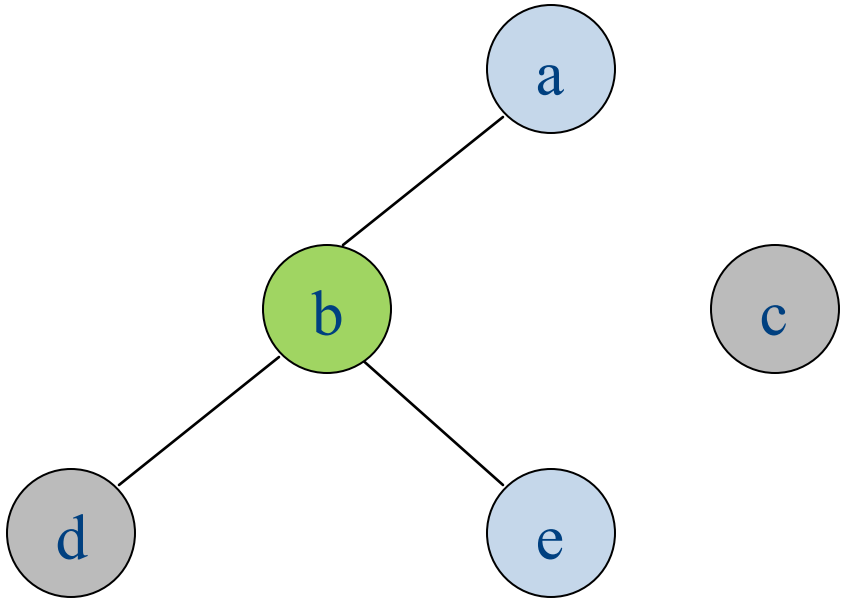


stack:
a
d

Coloring k=2

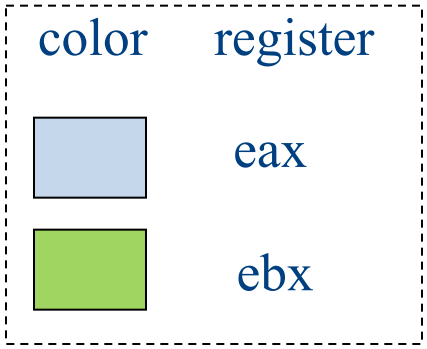


Some graphs can't be colored in K colors:

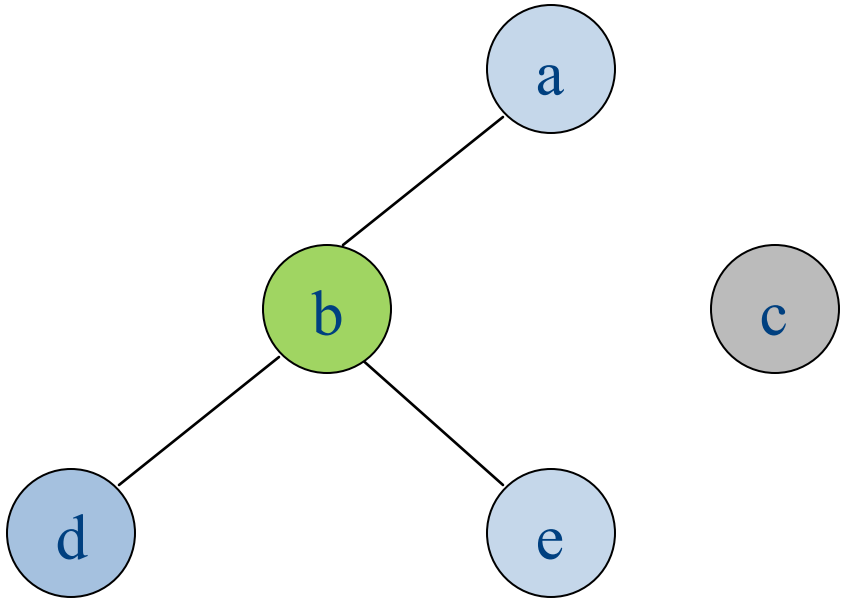


stack:
d

Coloring k=2



Some graphs can't be colored in K colors:



stack:

Handling precolored nodes

- Some variables are pre-assigned to registers
  - Eg: mul on x86/pentium
    - uses eax; defines eax, edx
  - Eg: call on x86/pentium
    - Defines (trashes) caller-save registers eax, ecx, edx
- To properly allocate registers, treat these register uses as special temporary variables and enter into interference graph as precolored nodes

Handling precolored nodes

- Simplify. Never remove a pre-colored node – it already has a color, i.e., it is a given register
- Coloring. Once simplified graph is all colored nodes, add other nodes back in and color them using precolored nodes as starting point

Optimizing move instructions

- Code generation produces a lot of extra mov instructions

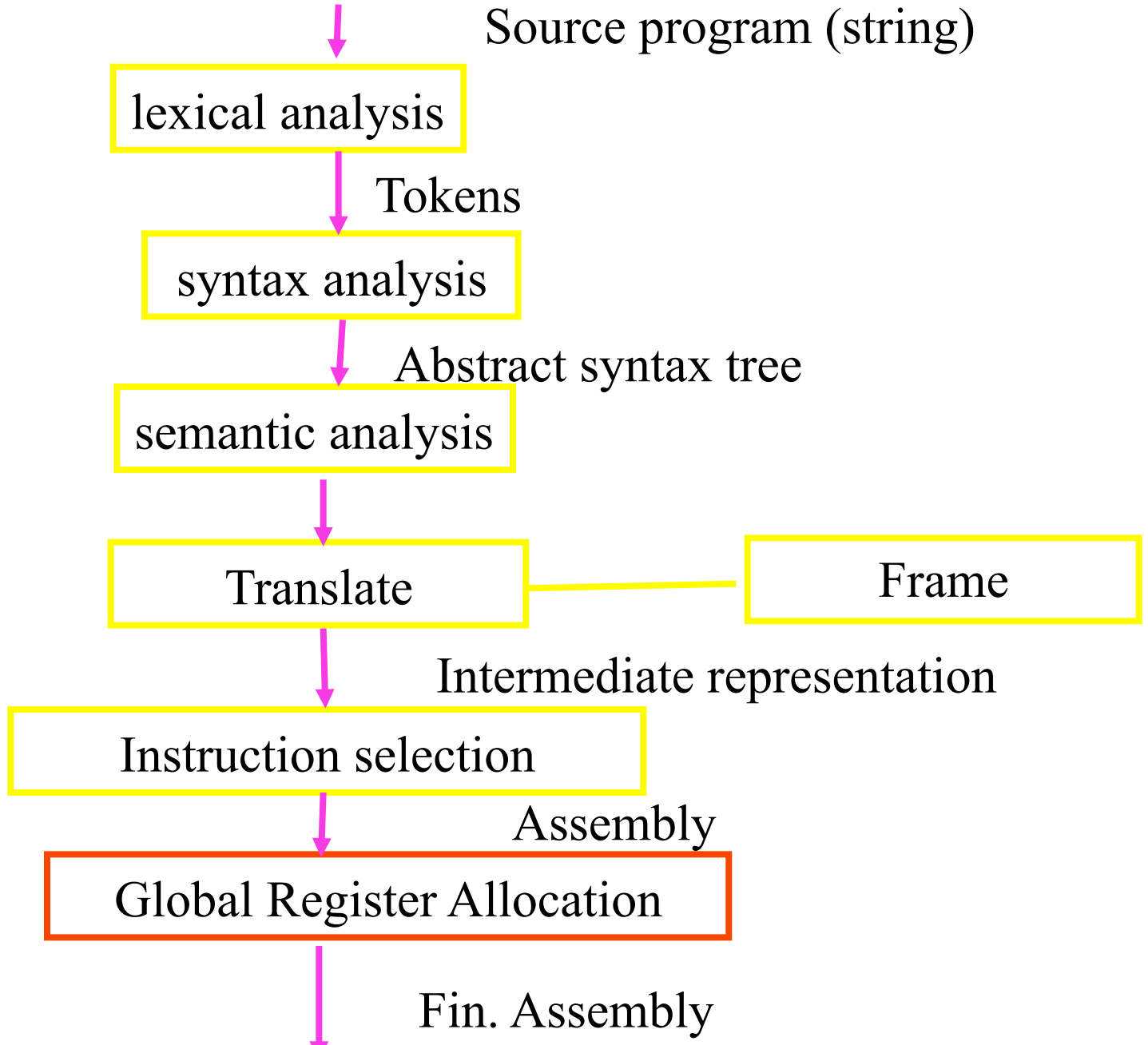
  mov t5, t9

- If we can assign t5 and t9 to same register, we can get rid of the mov
  - effectively, copy elimination at the register allocation level
- Idea: if t5 and t9 are not connected in inference graph, coalesce them into a single variable; the move will be redundant
- Problem: coalescing nodes can make a graph un-colorable
  - Conservative coalescing heuristic

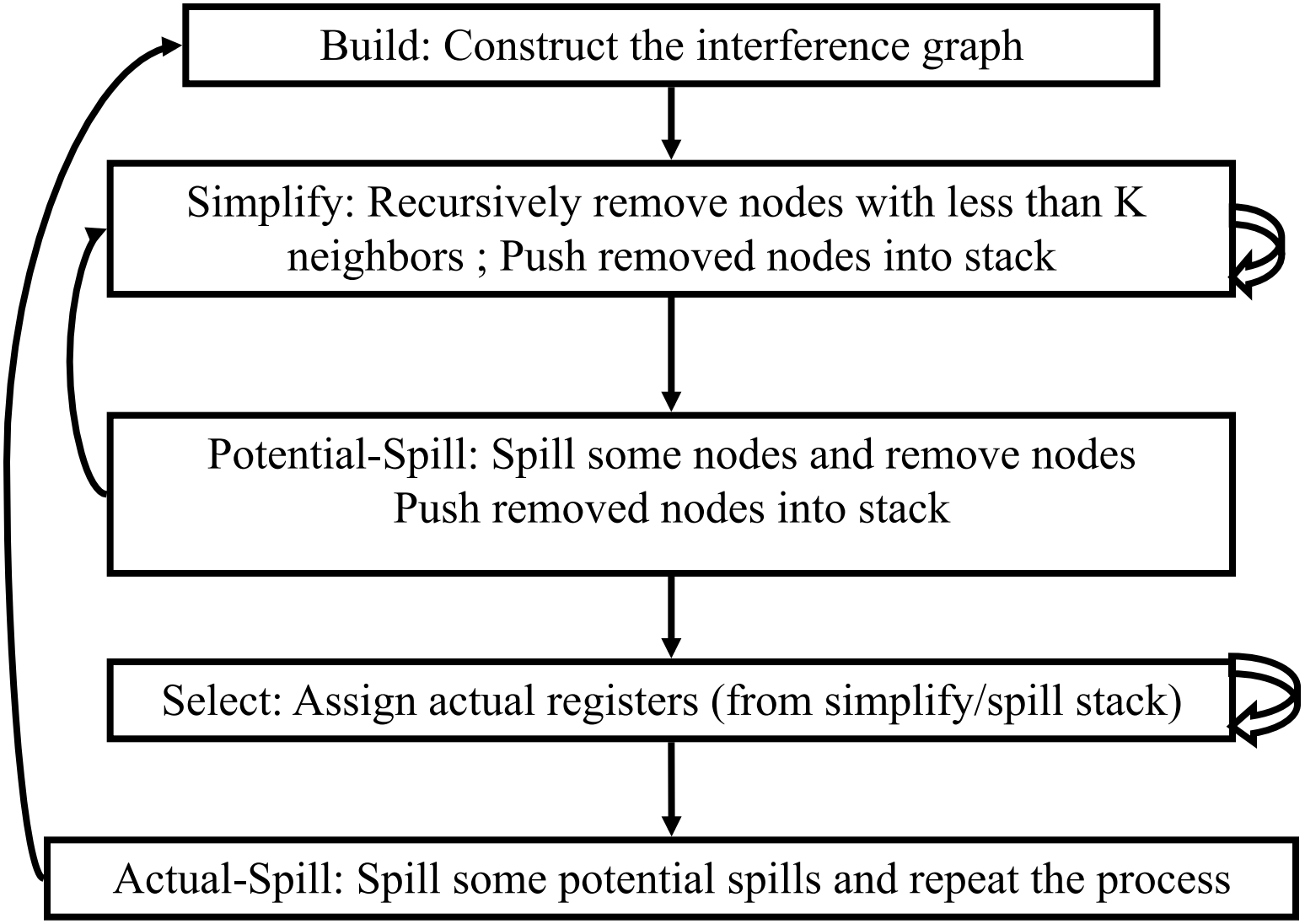
“Global” Register Allocation

- Input:
  - Sequence of machine code instructions (assembly)
    - Unbounded number of temporary registers
- Output
  - Sequence of machine code instructions (assembly)
  - Machine registers
  - Some MOVE instructions removed
  - Missing prologue and epilogue

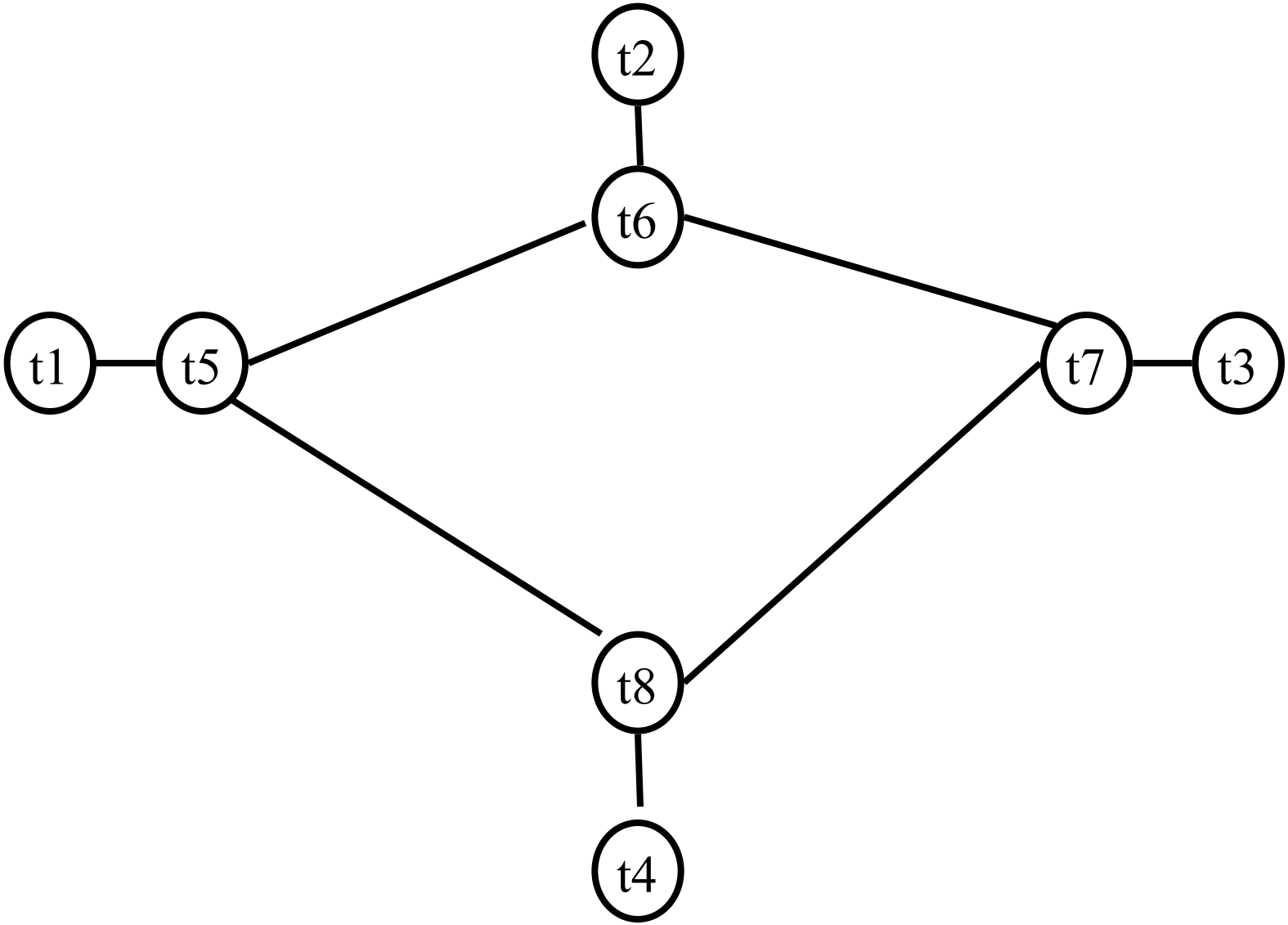
Basic Compiler Phases



Graph Coloring by Simplification



Artificial Example K=2



Coalescing

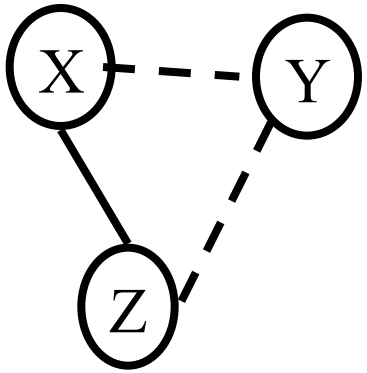
- MOVs can be removed if the source and the target share the same register
- The source and the target of the move can be merged into a single node (unifying the sets of neighbors)
- May require more registers
- Conservative Coalescing
  - Merge nodes only if the resulting node has fewer than K neighbors with degree $\geqslant$ K (in the resulting graph)

Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

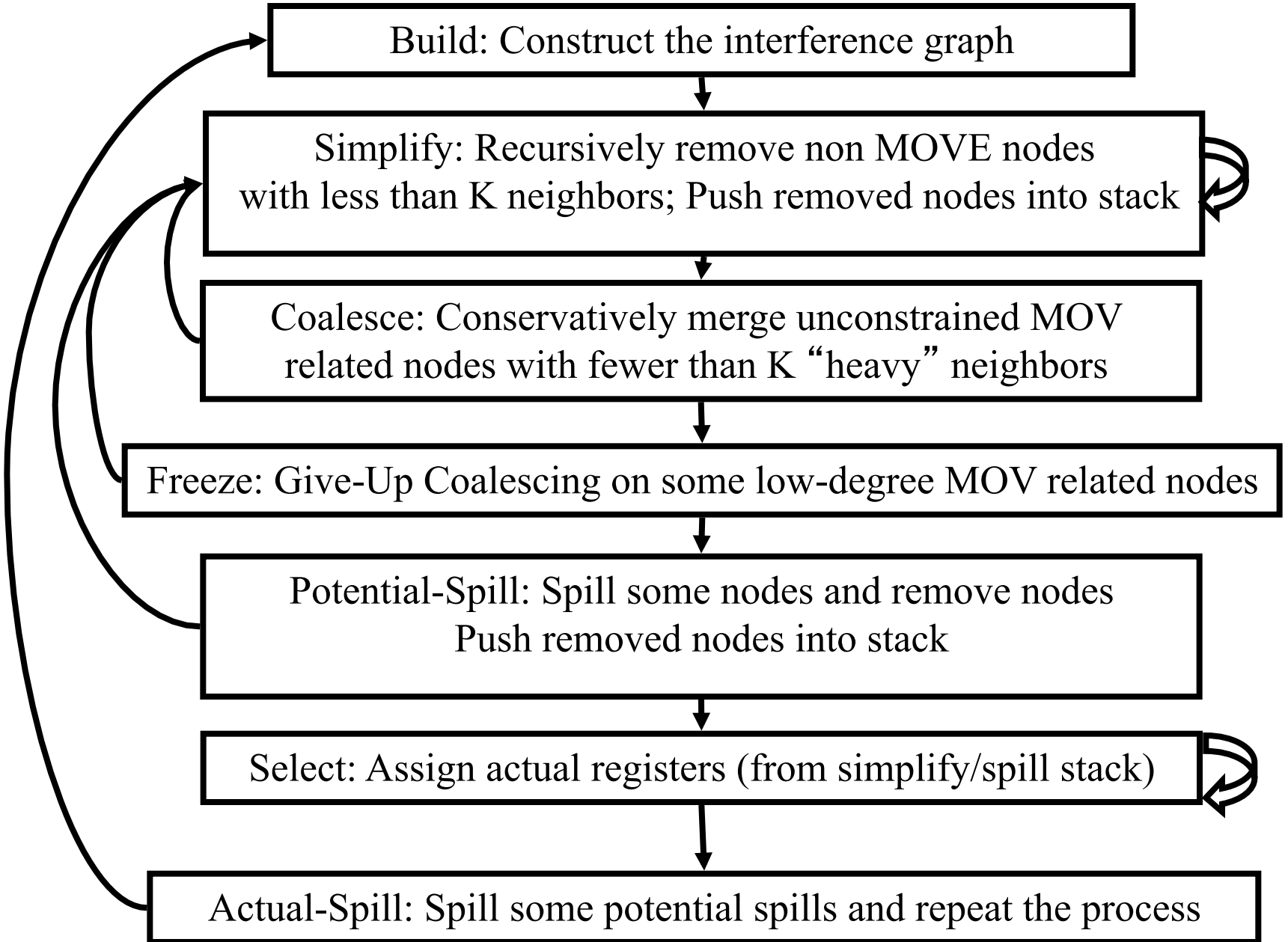
X ←Y /* X, Y, Z */

Y←Z



- Constrained MOVs are not coalesced

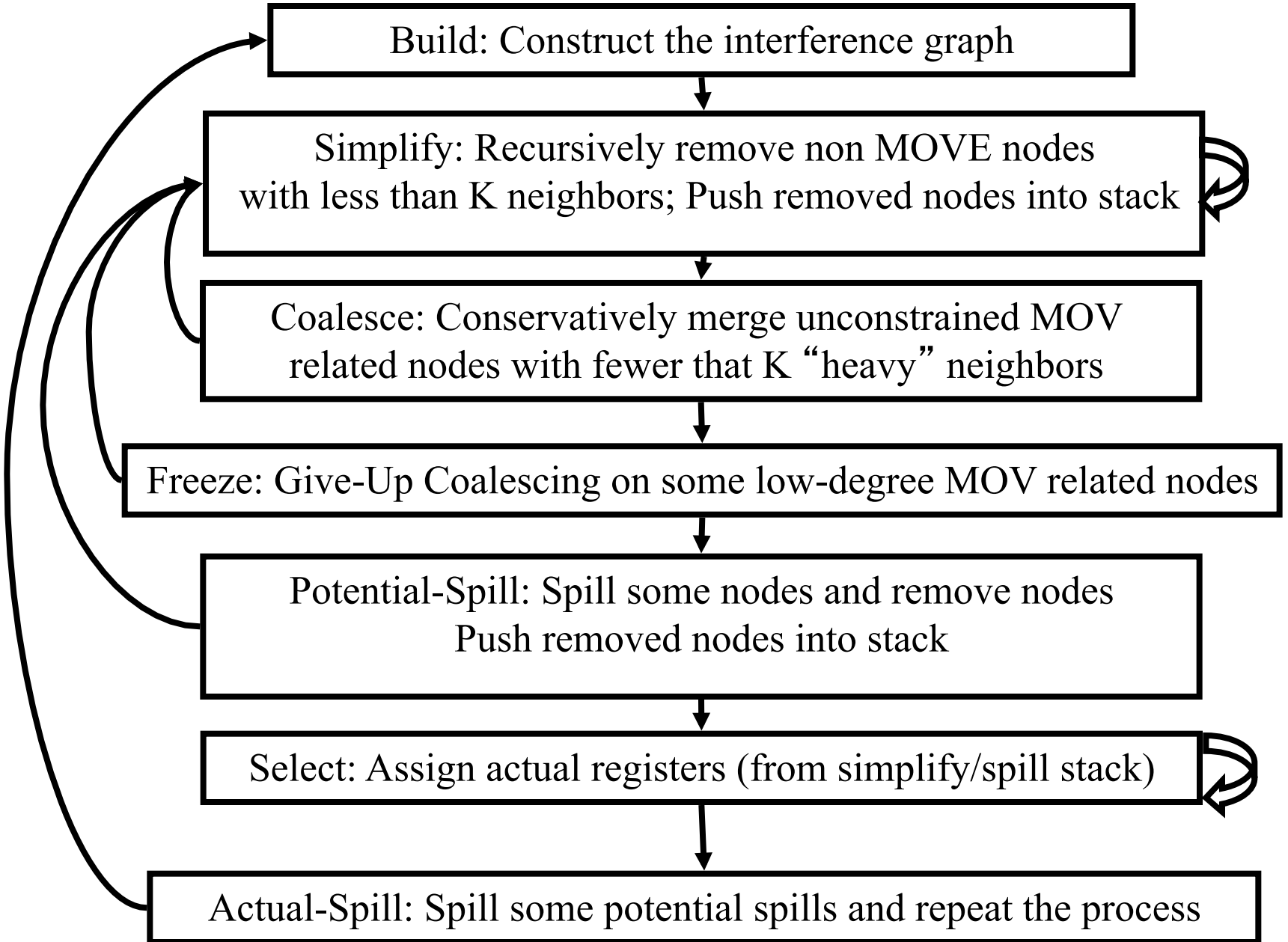
Graph Coloring with Coalescing



Pre-Colored Nodes

- Some registers in the intermediate language are pre-colored:
  - correspond to real registers
    (stack-pointer, frame-pointer, parameters,)
- Cannot be Simplified, Coalesced, or Spilled (infinite degree)
- Interfered with each other
- But normal temporaries can be coalesced into pre-colored registers
- Register allocation is completed when all the nodes are pre-colored

Graph Coloring with Coalescing



Optimizing MOV instructions

- Code generation produces a lot of extra mov instructions

  mov t5, t9

- If we can assign t5 and t9 to same register, we can get rid of the mov
  - effectively, copy elimination at the register allocation level
- Idea: if t5 and t9 are not connected in inference graph, coalesce them into a single variable; the move will be redundant
- Problem: coalescing nodes can make a graph un-colorable
  - Conservative coalescing heuristic

Coalescing

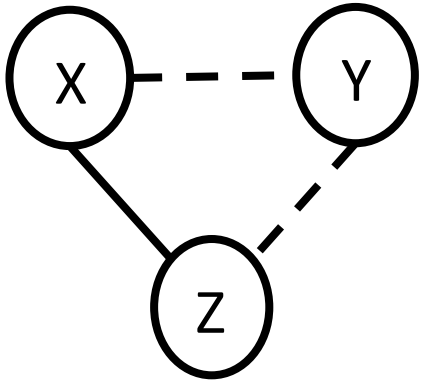
- MOVs can be removed if the source and the target share the same register
- The source and the target of the move can be merged into a single node (unifying the sets of neighbors)
  - May require more registers
  - Conservative Coalescing
    - Merge nodes only if the resulting node has fewer than K neighbors with degree $\geqslant$ K (in the resulting graph)

Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

X ← Y

Y ← Z



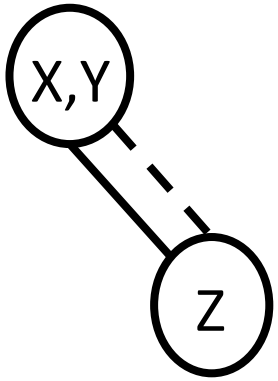
- Constrained MOVs are not coalesced

Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

X ← Y

Y ← Z



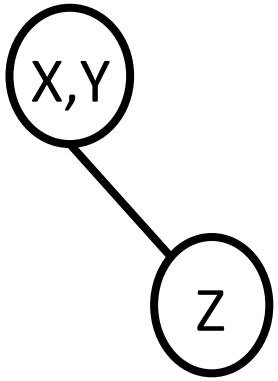
- Constrained MOVs are not coalesced

Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

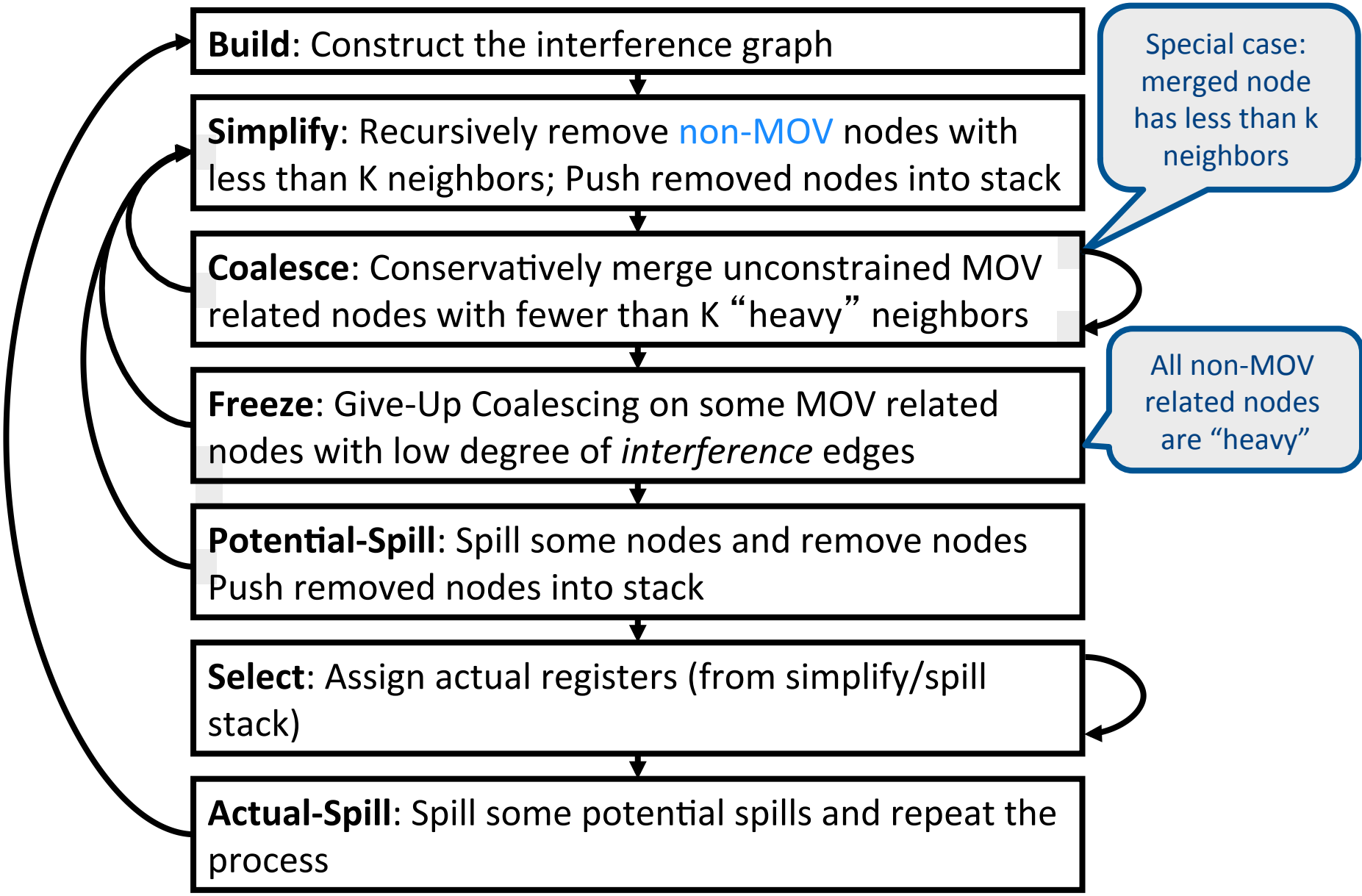
X ← Y

Y ← Z



- Constrained MOVs are not coalesced

Graph Coloring with Coalescing



Pre-Colored Nodes

- Some registers in the intermediate language are pre-colored:
  - correspond to real registers
    (stack-pointer, frame-pointer, parameters,)
- Cannot be Simplified, Coalesced, or Spilled
  - infinite degree
- Interfered with each other
- But normal temporaries can be coalesced into pre-colored registers
- Register allocation is completed when all the nodes are pre-colored

Caller-Save and Callee-Save Registers

- callee-save-registers (MIPS 16-23)
  - Saved by the callee when modified
  - Values are automatically preserved across calls
- caller-save-registers
  - Saved by the caller when needed
  - Values are not automatically preserved
- Usually the architecture defines caller-save and callee-save registers
  - Separate compilation
  - Interoperability between code produced by different compilers/ languages
- But compilers can decide when to use caller/callee registers

Caller-Save vs. Callee-Save Registers

```
int foo(int a)      {
        int b=a+1;
        f1();
        g1(b);
        return(b+2);
 }
```

```
void bar (int y) {
        int x=y+1;
        f2(y);
        g2(2);
}
```

Saving Callee-Save Registers

enter: def(r_7)

…

exit: use(r_7)

enter: def(r_7)

$t_{231} \leftarrow r_7$

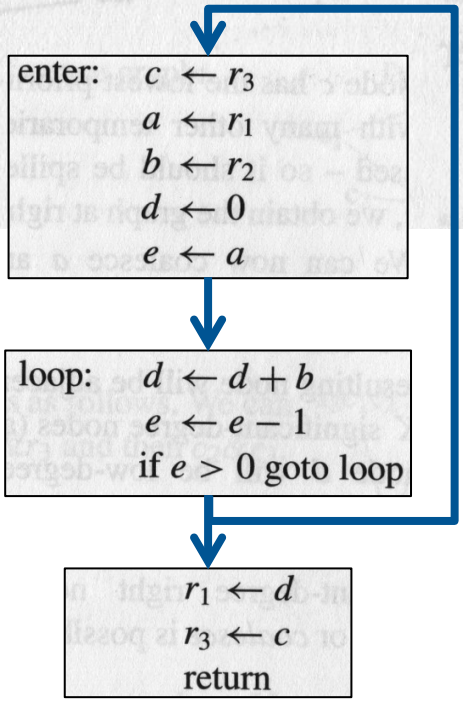
…

$r_7 \leftarrow t_{231}$

exit: use(r_7)

A Complete Example

```
int f(int a, int b) {
   int d=0;
   int e=a;
   do {d = d+b;
       e = e-1;
   } while (e>0);
   return d;
}
```



enter: $c \leftarrow r_3$ — Callee-saved registers
$a \leftarrow r_1$

$b \leftarrow r_2$ — Caller-saved registers

$d \leftarrow 0$

$e \leftarrow a$

loop: $d \leftarrow d + b$

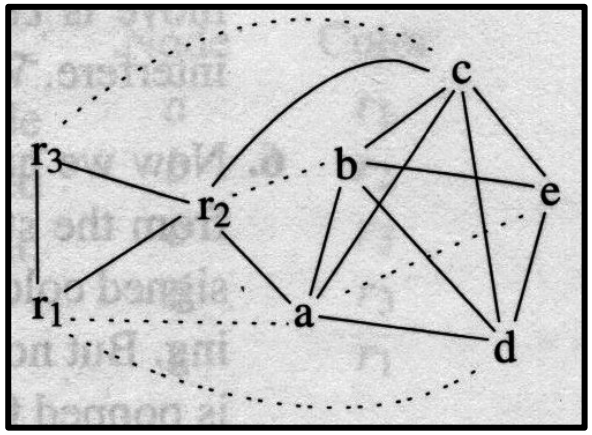
$e \leftarrow e - 1$

if $e > 0$ goto loop

$r_1 \leftarrow d$

$r_3 \leftarrow c$

return (r_1, r_3 *live out*)

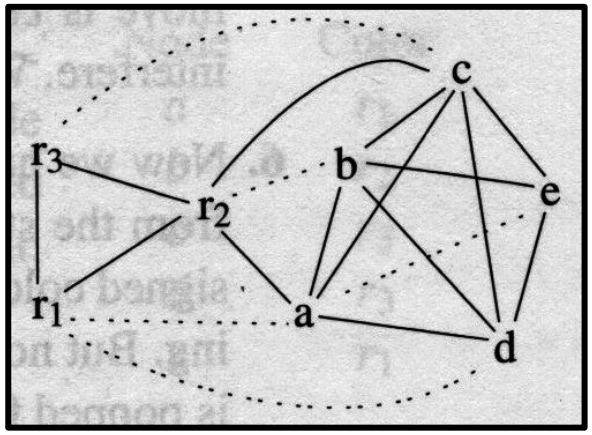


A Complete Example

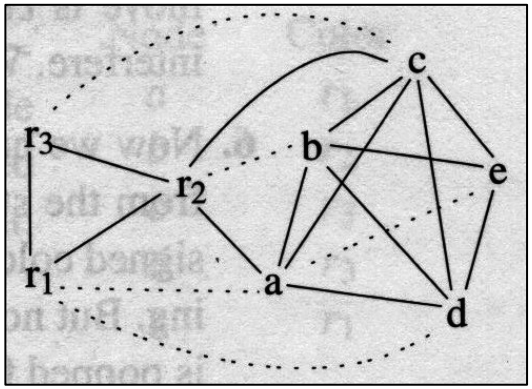
```
int f(int a, int b) {
   int d=0;
   int e=a;
   do {d = d+b;
       e = e-1;
   } while (e>0);
   return d;
}
```

enter: $c \leftarrow r_3$
$a \leftarrow r_1$
$b \leftarrow r_2$
$d \leftarrow 0$
$e \leftarrow a$
loop: $d \leftarrow d + b$
$e \leftarrow e - 1$
if $e > 0$ goto loop
$r_1 \leftarrow d$
$r_3 \leftarrow c$
return (r_1, r_3 *live out*)

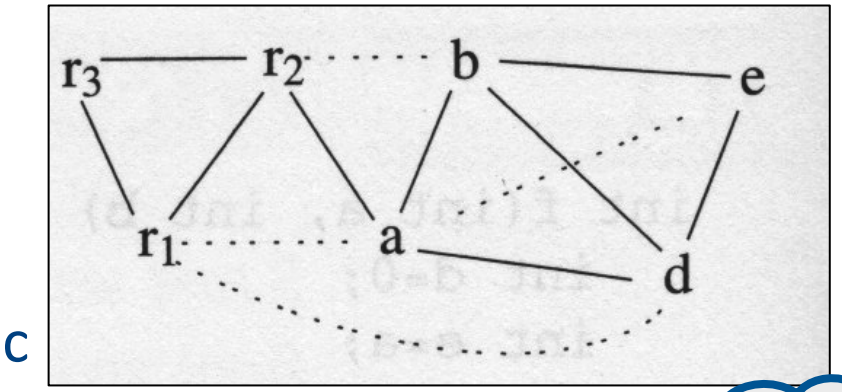
Node	Uses+Defs outside loop		Uses+Defs within loop		Degree		Spill priority
a	(2	+ 10 ×	0	) /	4	=	0.50
b	(1	+ 10 ×	1	) /	4	=	2.75
c	(2	+ 10 ×	0	) /	6	=	0.33
d	(2	+ 10 ×	2	) /	4	=	5.50
e	(1	+ 10 ×	3	) /	3	=	10.33



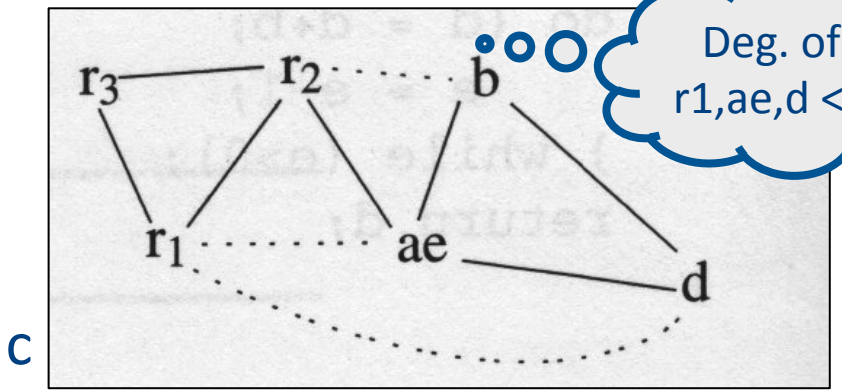
A Complete Example



Spill c



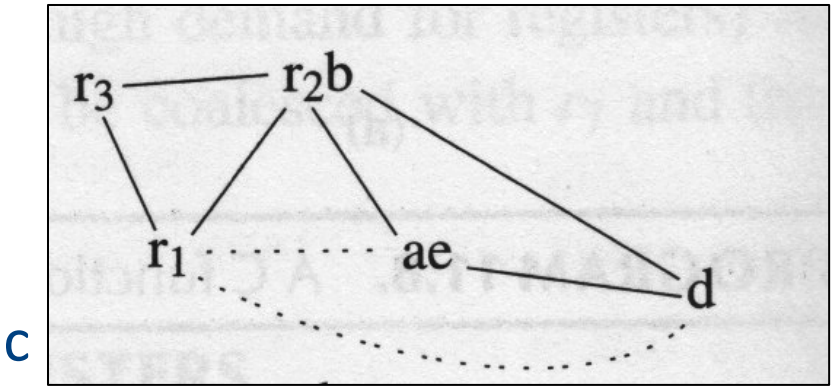
a & e



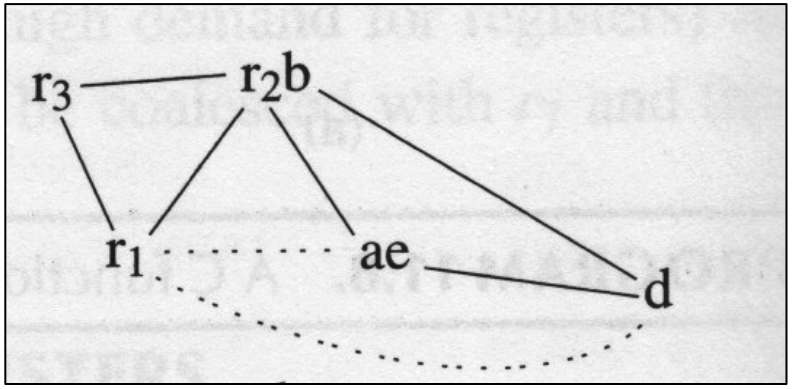
Deg. of r1,ae,d < K

r2 & b

(Alt: ae+r1)

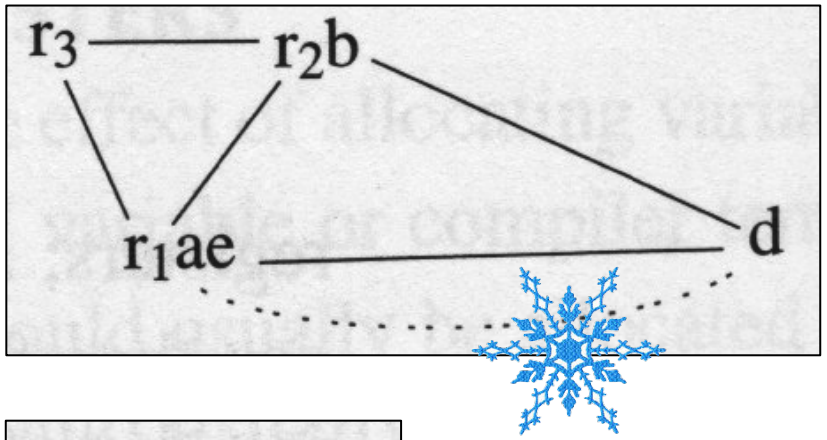


A Complete Example



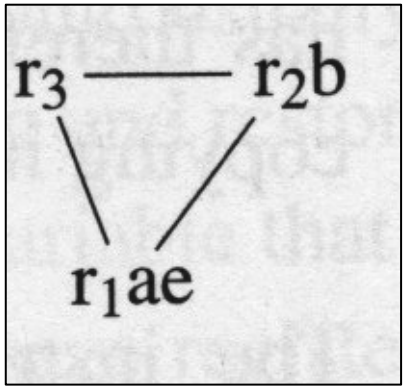
ae & r1

(Alt: …)c



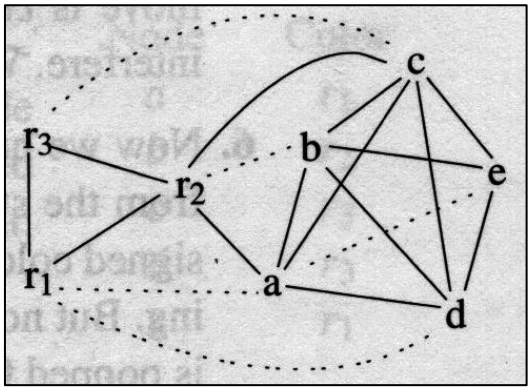
freeze $r_1$ae-d

Simplify d



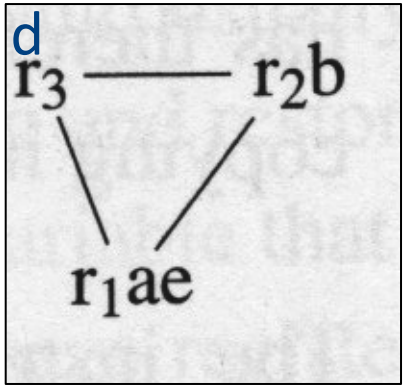
dc

pop c …



(Alt: ae+r1)

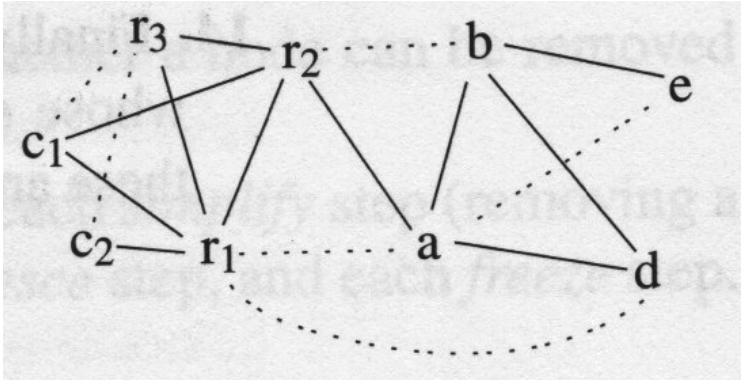
pop d



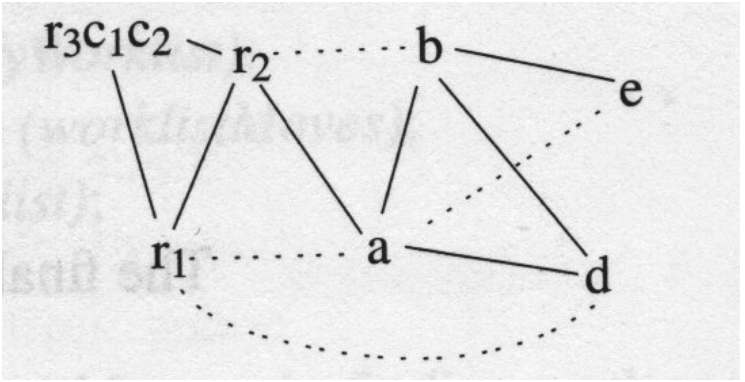
c

A Complete Example

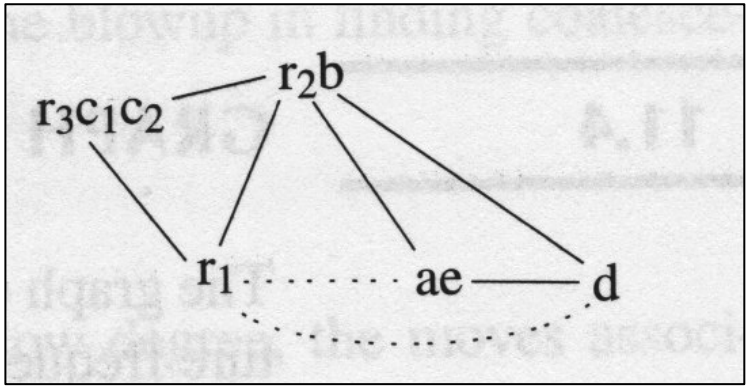
```
enter:  c1 ← r3
        M[cloc] ← c1
        a  ← r1
        b  ← r2
        d  ← 0
        e  ← a
loop:   d  ← d + b
        e  ← e − 1
        if e > 0 goto loop
        r1 ← d
        c2 ← M[cloc]
        r3 ← c2
        return
```



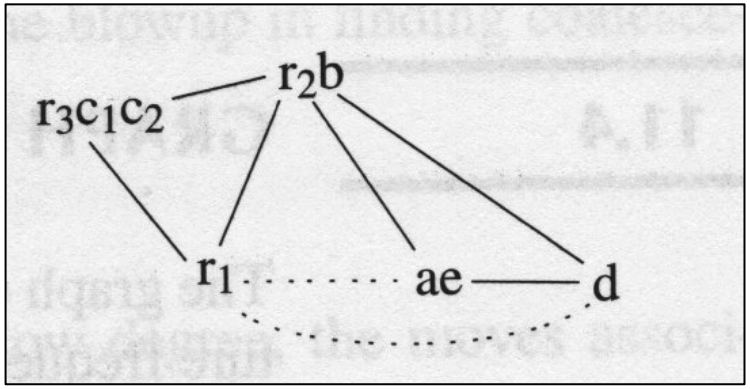
c1&r3, c2 &r3



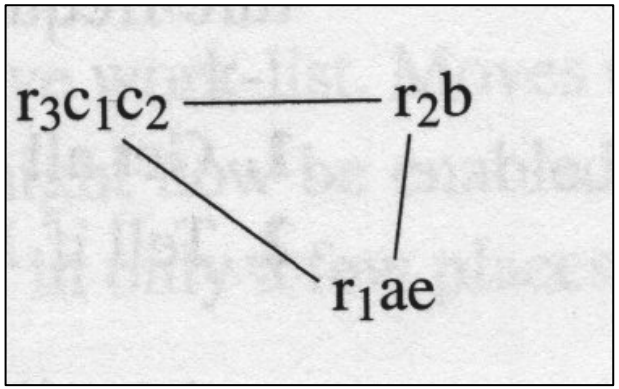
a&e, b&r2



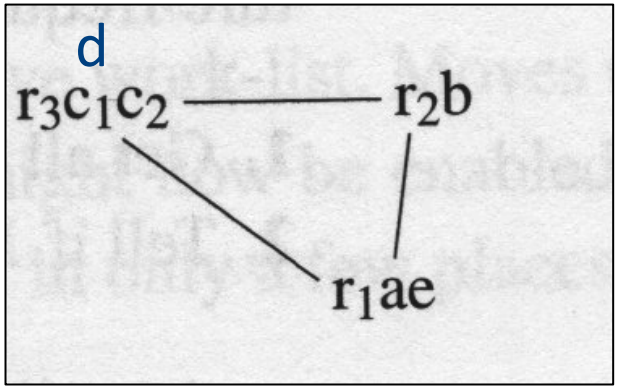
A Complete Example



ae & r1
Simplify d



Pop d



gen code

```
enter:  r3 ← r3
        M[cloc] ← r3
        r1 ← r1
        r2 ← r2
        r3 ← 0
        r1 ← r1
loop:   r3 ← r3 + r2
        r1 ← r1 − 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← M[cloc]
        r3 ← r3
        return
```

"opt"

```
enter:  M[cloc] ← r3
        r3 ← 0
loop:   r3 ← r3 + r2
        r1 ← r1 − 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← M[cloc]
        return
```

Interprocedural Allocation

- Allocate registers to multiple procedures
- Potential saving
  - caller/callee save registers
  - Parameter passing
  - Return values
- But may increase compilation cost
- Function inline can help

Summary

- Two Register Allocation Methods
  - Local of every IR tree
    - Simultaneous instruction selection and register allocation
    - Optimal (under certain conditions)
  - Global of every function
    - Applied after instruction selection
    - Performs well for machines with many registers
    - Can handle instruction level parallelism
- Missing
  - Interprocedural allocation

The End