# Compilation

## 0368-3133 (Semester A, 2013/14)

## Lecture 10: Register Allocation

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv and Eran Yahav

# Promo: (Re)thinking Software Design

- Daniel Jackson (MIT)
- This Wednesday 12:00
- Gilman 223

What is the essence of software design? Researchers and practitioners have for many years quoted Fred Brooks's assertions that "conceptual integrity is the most important consideration in system design" and is "central to product quality".
But what exactly is conceptual integrity? In this talk, I'll report on progress in a new research project that explores this question by attempting to develop a theory of conceptual design, and applying it experimentally in a series of redesigns of common applications (such as Git, Gmail and Dropbox)

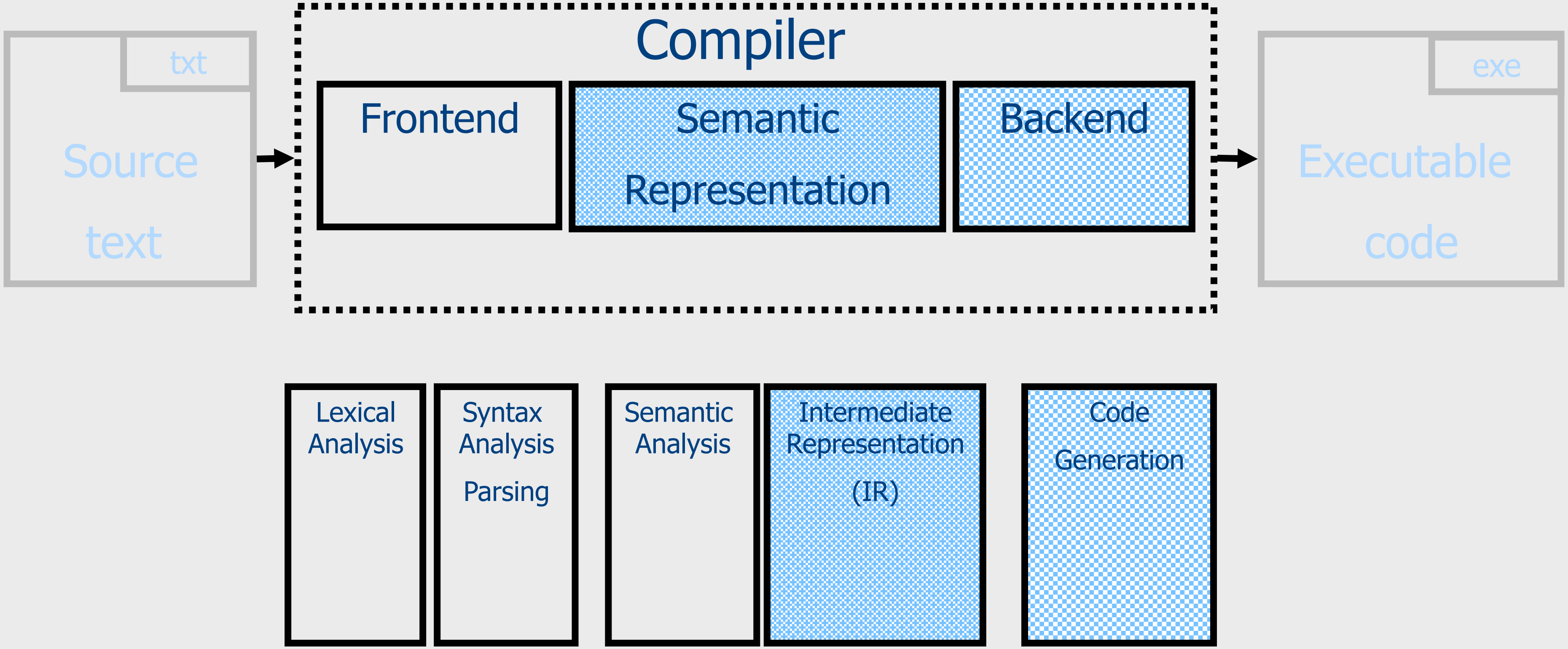
# What is a Compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

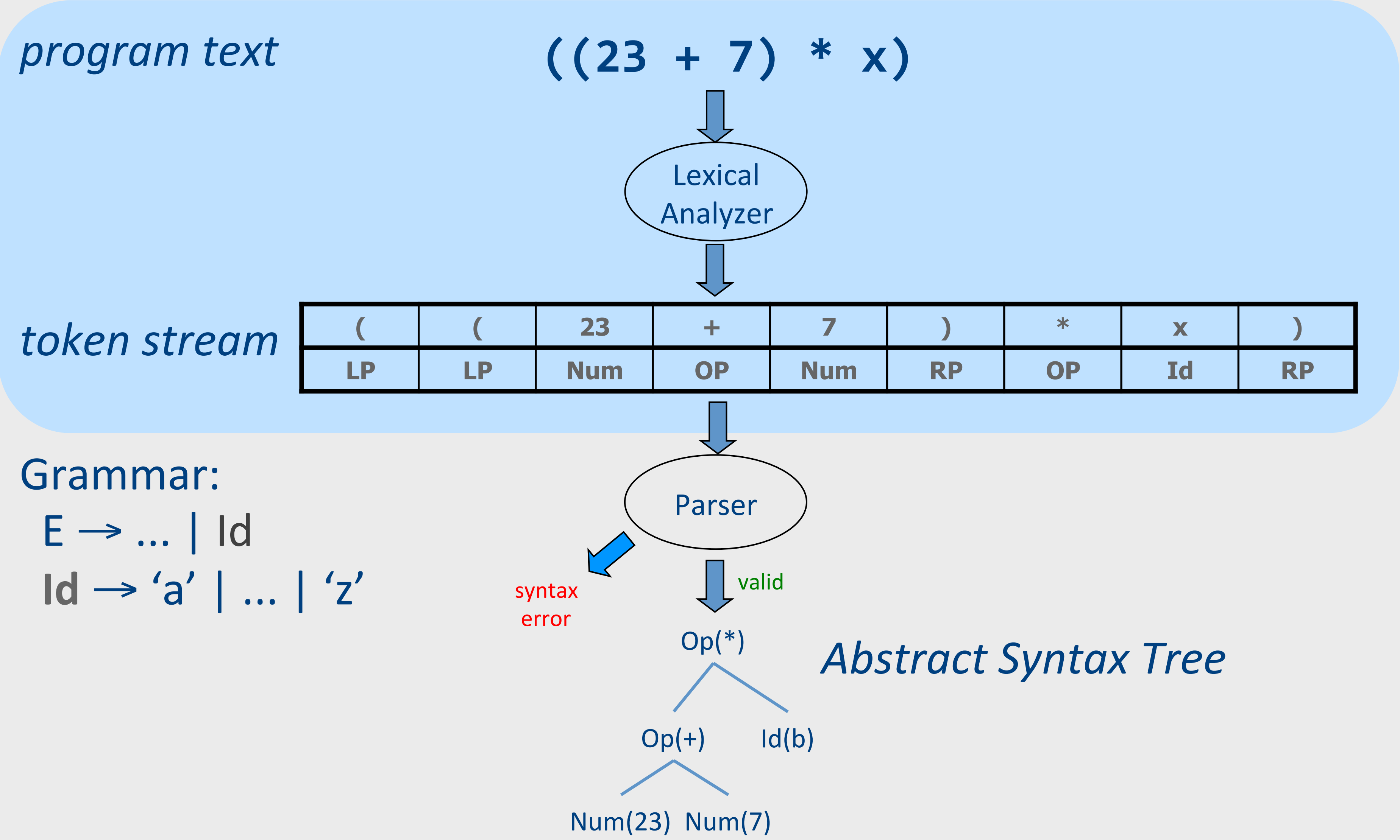
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

# Conceptual Structure of a Compiler



# From scanning to parsing



*program text*

**((23 + 7) * x)**

*token stream*

| ( | ( | 23 | + | 7 | ) | * | x | ) |
|---|---|---|---|---|---|---|---|---|
| LP | LP | Num | OP | Num | RP | OP | Id | RP |

Grammar:

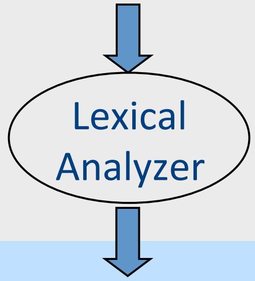
E → … | Id

Id → ‘a’ | … | ‘z’

*Abstract Syntax Tree*

# From scanning to parsing

*program text* **((23 + 7) * x)**



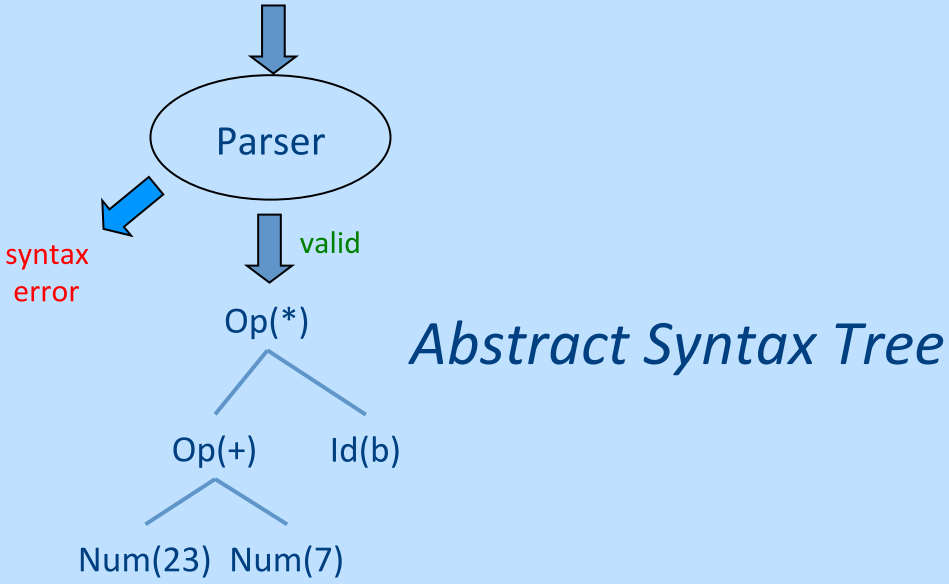
*token stream*

| ( | ( | 23 | + | 7 | ) | * | x | ) |
|---|---|---|---|---|---|---|---|---|
| LP | LP | Num | OP | Num | RP | OP | Id | RP |

Grammar:

E → ... | Id

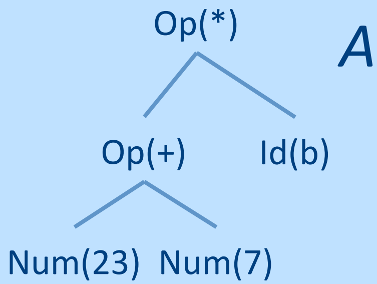
**Id** → ‘a’ | ... | ‘z’



# Context Analysis

Type rules

$$\frac{E1 : int \qquad E2 : int}{E1 + E2 : int}$$

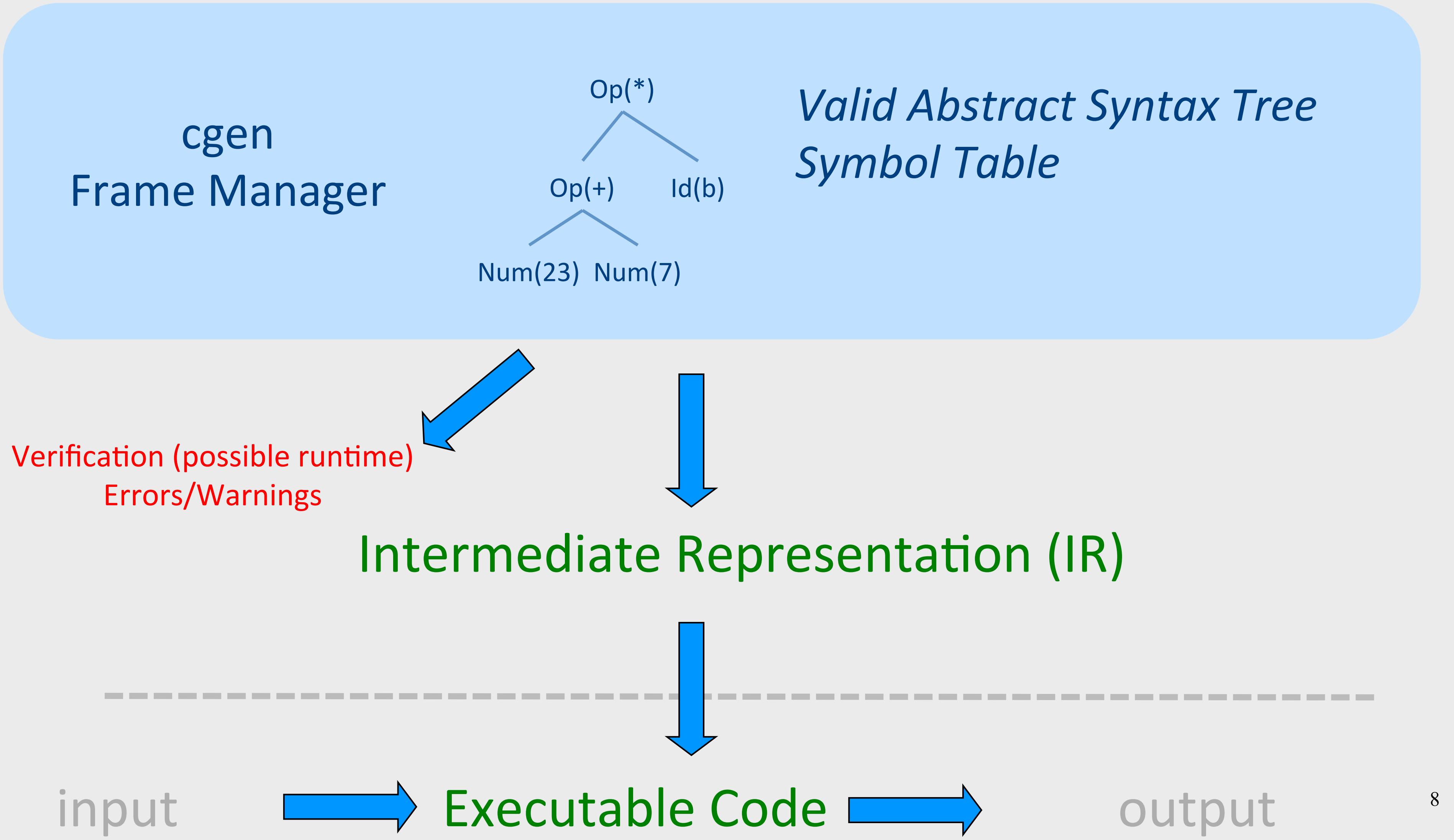


*Abstract Syntax Tree*

Semantic Error

Valid + Symbol Table

# Code Generation



cgen
Frame Manager

*Valid Abstract Syntax Tree*
*Symbol Table*

Verification (possible runtime)
Errors/Warnings

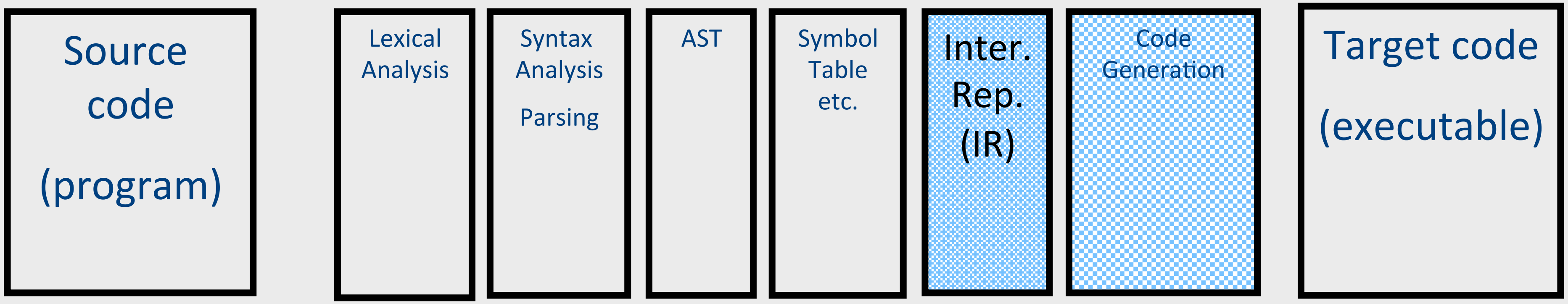
Intermediate Representation (IR)

input

Executable Code

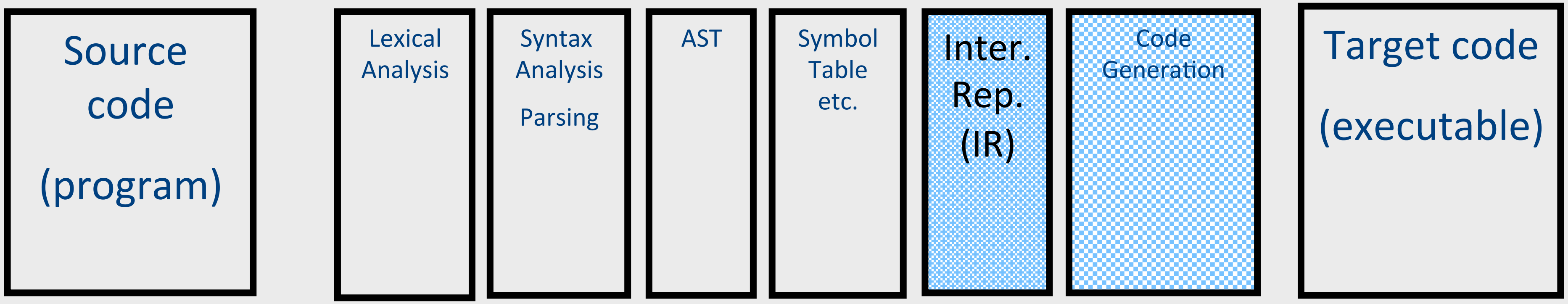
output

# Register Allocation



- The process of assigning variables to registers and managing data transfer in and out of registers
- Using registers intelligently is a critical step in any compiler
  - A good register allocator can generate code orders of magnitude better than a bad register allocator

# Register Allocation: Goals



Source code

(program)

Lexical Analysis

Syntax Analysis

Parsing

AST

Symbol Table etc.

Inter. Rep. (IR)

Code Generation

Target code

(executable)

- Reduce number of temporaries (registers)
  - Machine has at most K registers
  - Some registers have special purpose
    - E.g., pass parameters
- Reduce the number of move instructions
  - `MOVE R1,R2` // R1 ← R2

# Registers

- Most machines have a set of registers, dedicated memory locations that
  - can be accessed quickly,
  - can have computations performed on them, and
  - are used for special purposes (e.g., parameter passing)
- Usages
  - Operands of instructions
  - Store temporary results
  - Can (should) be used as loop indexes due to frequent arithmetic operation
  - Used to manage administrative info
    - e.g., runtime stack

# Register allocation

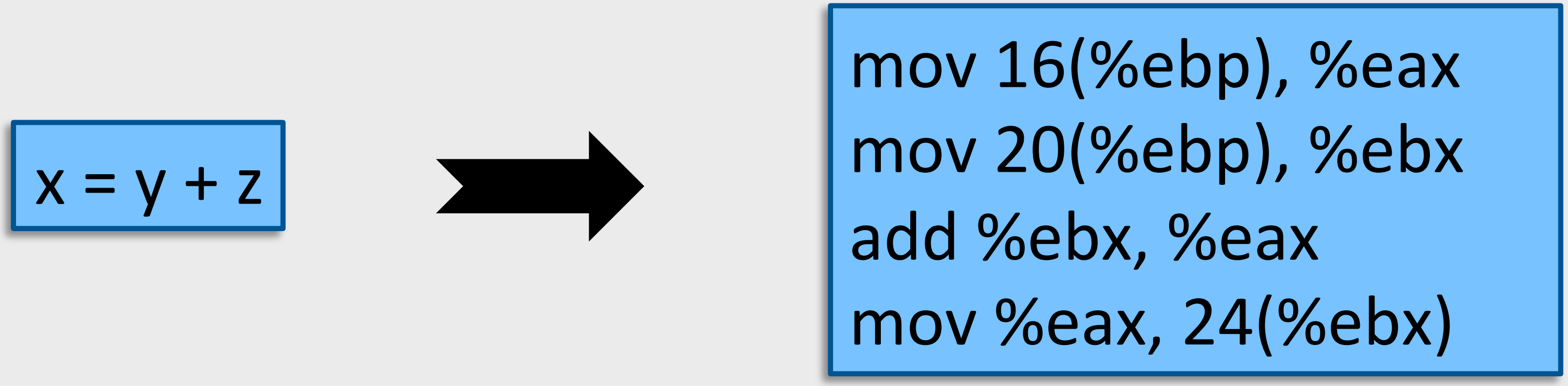
- In TAC, there are an unlimited number of variables
- On a physical machine there are a small number of registers:
  - x86 has four general-purpose registers and a number of specialized registers
  - MIPS has twenty-four general-purpose registers and eight special-purpose registers

# Spilling

- Even an optimal register allocator can require more registers than available
- Need to generate code for every correct program
- The compiler can save temporary results
  - Spill registers into temporaries
  - Load when needed
- Many heuristics exist

# Simple approach

- Straightforward solution:
  - Allocate each variable in activation record
  - At each instruction, bring values needed into registers, perform operation, then store result to memory



```
x = y + z
```

```
mov 16(%ebp), %eax
mov 20(%ebp), %ebx
add %ebx, %eax
mov %eax, 24(%ebx)
```

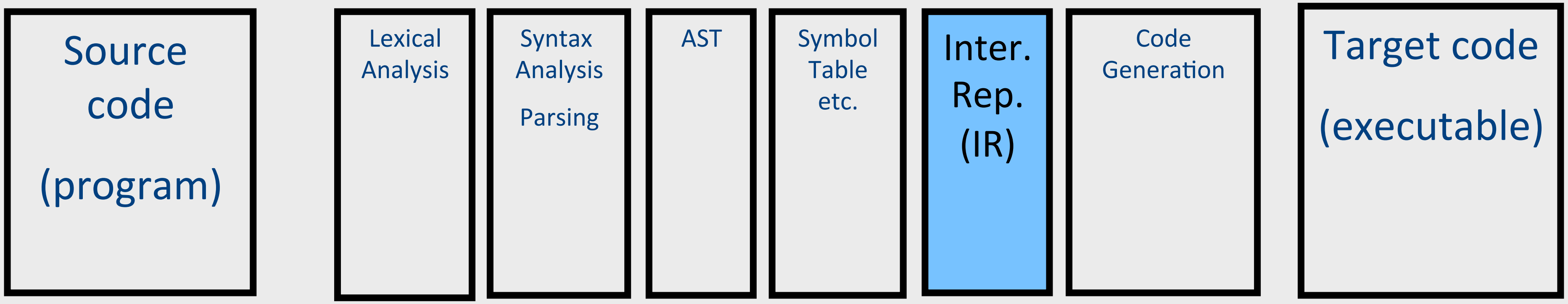
- Problem: program execution very inefficient–moving data back and forth between memory and registers

# Register Allocation

- Machine-agnostic optimizations
    - Assume unbounded number of registers
  - Expression trees (tree-local)
  - Basic blocks (block-local)

- Machine-dependent optimization
    - K registers
    - Some have special purposes
  - Control flow graphs (global register allocation)

# Register Allocation: IR



Source code

(program)

Lexical Analysis

Syntax Analysis

Parsing

AST

Symbol Table etc.

Inter. Rep. (IR)

Code Generation

Target code

(executable)

# Register Allocation

- Machine-agnostic optimizations
    - Assume unbounded number of registers
  - Expression trees
  - Basic blocks

- Machine-dependent optimization
    - K registers
    - Some have special purposes
  - Control flow graphs (whole program)

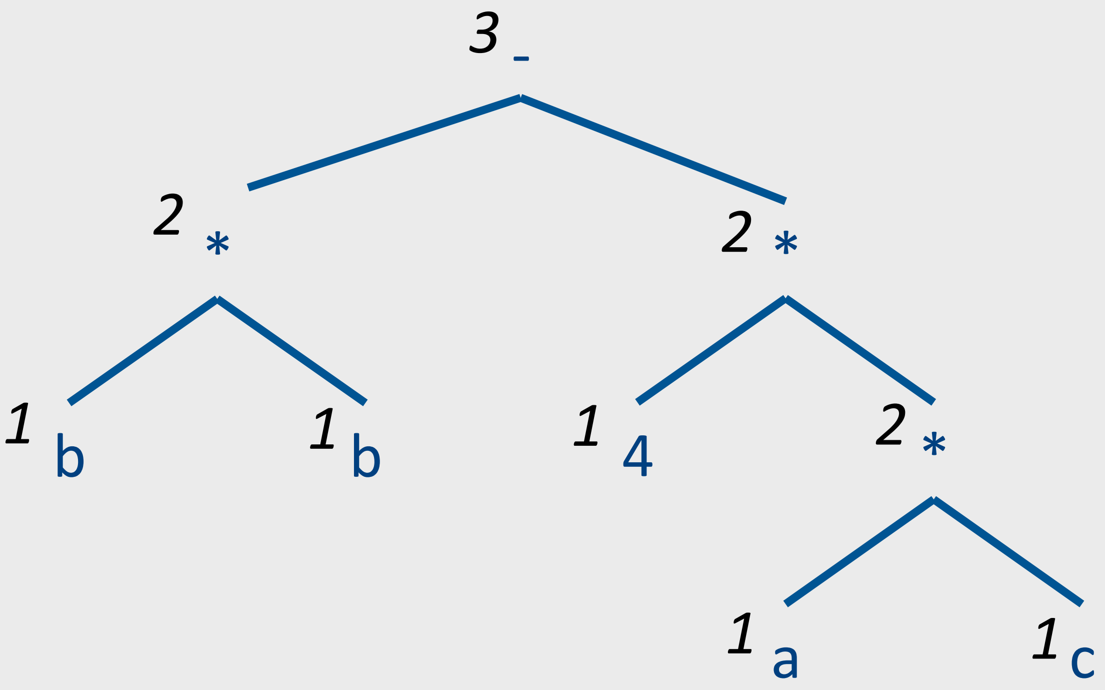
# Sethi-Ullman translation

- Algorithm by Ravi Sethi and Jeffrey D. Ullman to emit optimal TAC
  - Minimizes number of temporaries for a **single expression**

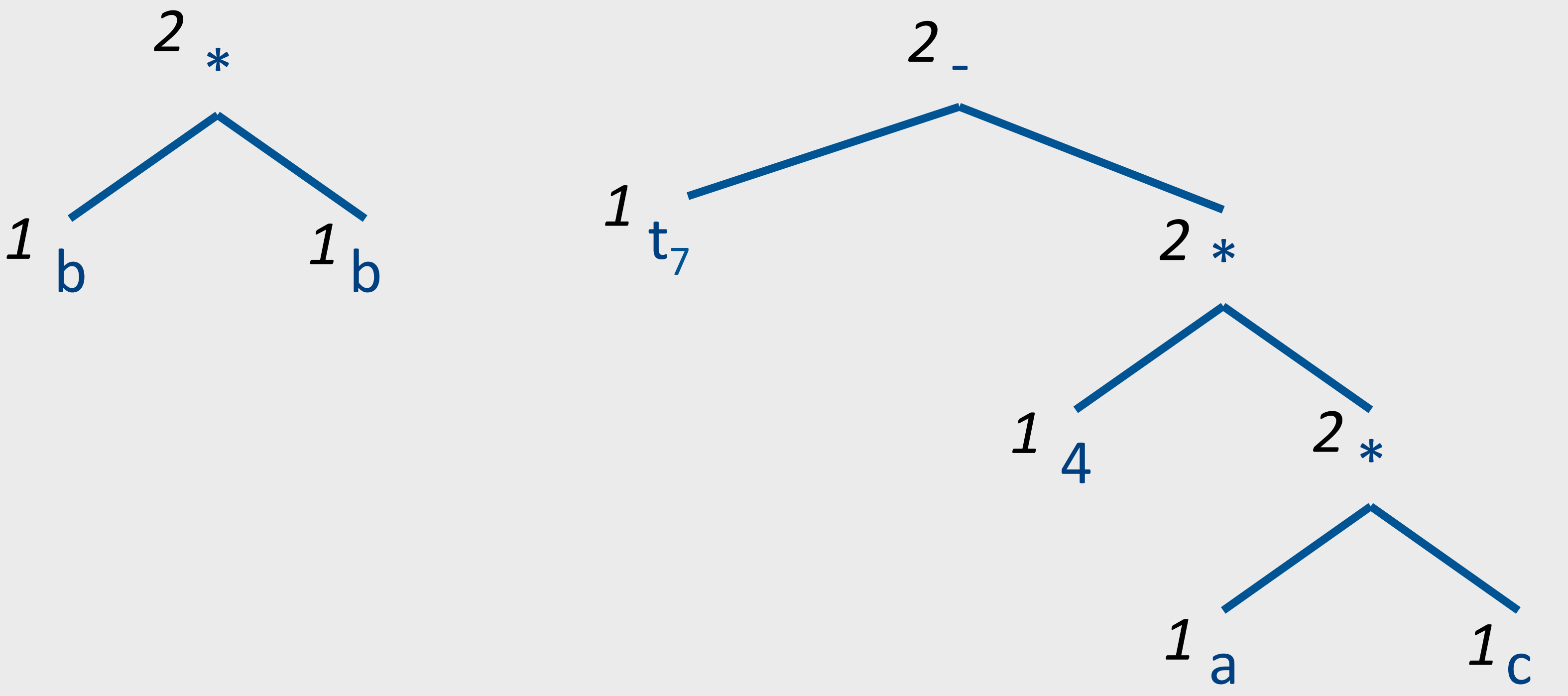
# Simple Spilling Method

- Heavy tree – Needs more registers than available
- A “heavy” tree contains a “heavy” subtree whose dependents are “light”
- Simple spilling
    - Generate code for the light tree
    - Spill the content into memory and replace subtree by temporary
    - Generate code for the resultant tree

# Example (optimized): b*b-4*a*c



# Example (spilled): x := b*b-4*a*c



```
t7 := b * b          x := t7 - 4 * a * c
```

# Register Allocation

- Machine-agnostic optimizations
    - Assume unbounded number of registers
  - Expression trees
  - Basic blocks

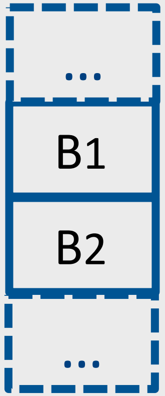
- Machine-dependent optimization
    - K registers
    - Some have special purposes
  - Control flow graphs (whole program)

# Basic Blocks

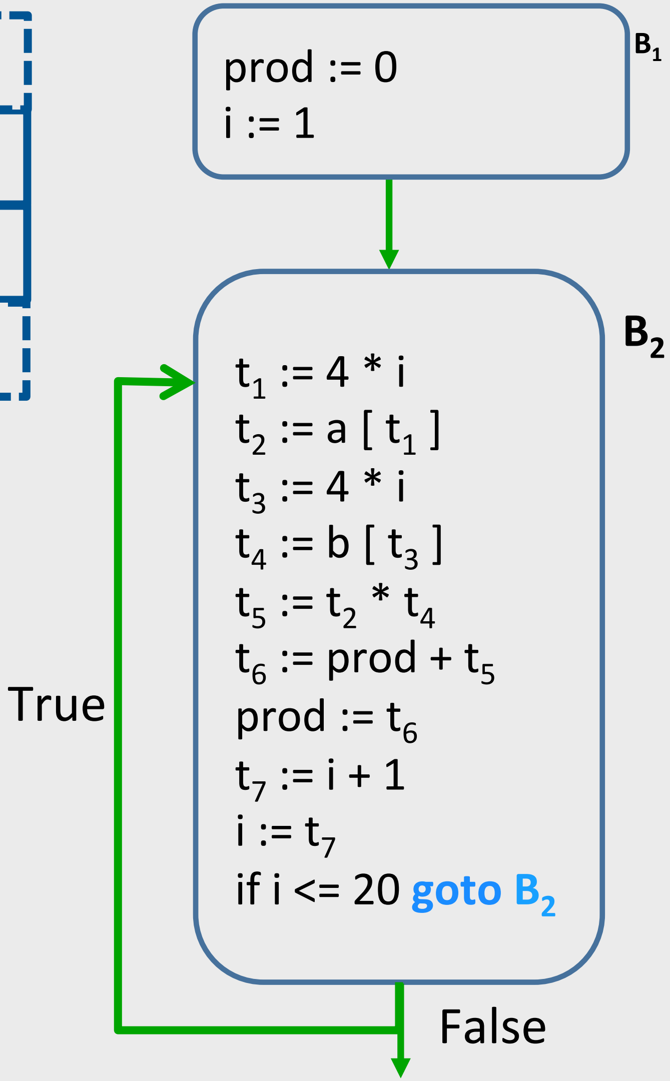
- **basic block** is a sequence of instructions with
  - **single entry** (to first instruction), no jumps to the middle of the block
  - **single exit** (last instruction)
  - code execute as a sequence from first instruction to last instruction without any jumps
- edge from one basic block B1 to another block B2 when the last statement of B1 may jump to B2

# control flow graph

- A directed graph G=(V,E)
- nodes V = basic blocks
- edges E = control flow
  - (B1,B2) ∈ E when control from B1 flows to B2



- Leaders-based construction
  - Target of jump instructions
  - Instructions following jumps



```
prod := 0
i := 1
```

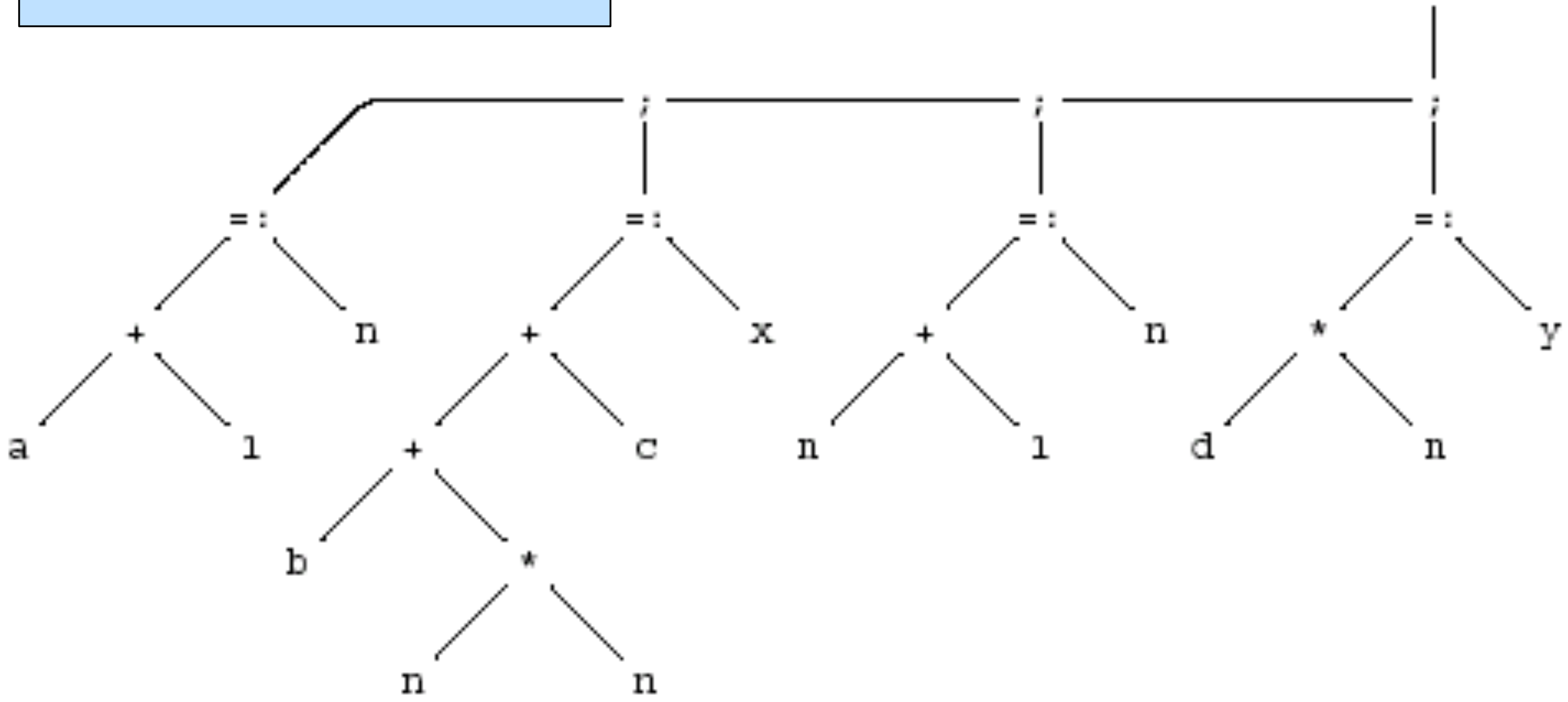
```
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
t5 := t2 * t4
t6 := prod + t5
prod := t6
t7 := i + 1
i := t7
if i <= 20 goto B2
```

# Register Allocation for B.B.

- Dependency graphs for basic blocks
- Transformations on dependency graphs
- From dependency graphs into code
    - Instruction selection
        - linearizations of dependency graphs
    - Register allocation
        - At the basic block level

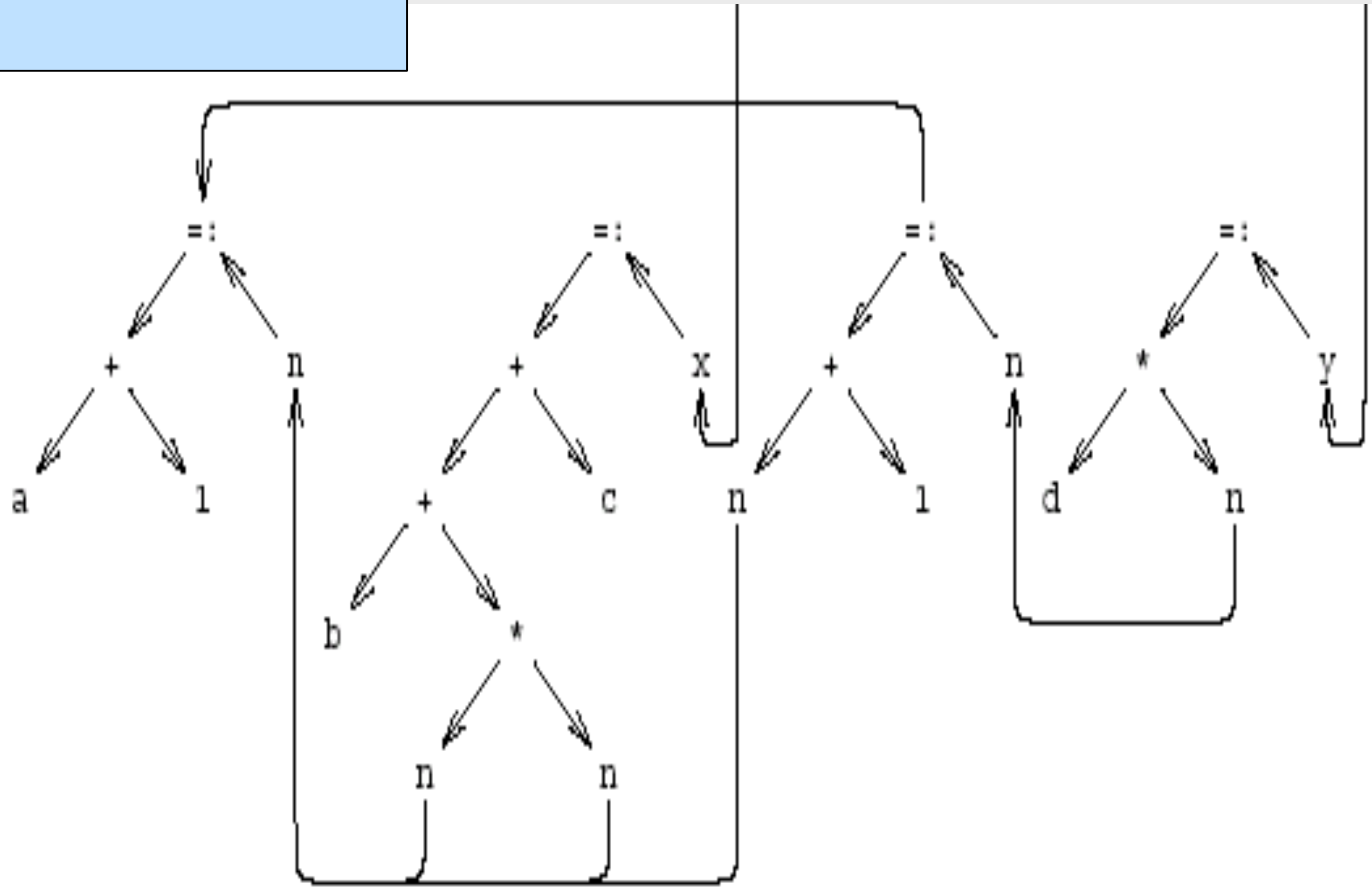
# AST for a Basic Block

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



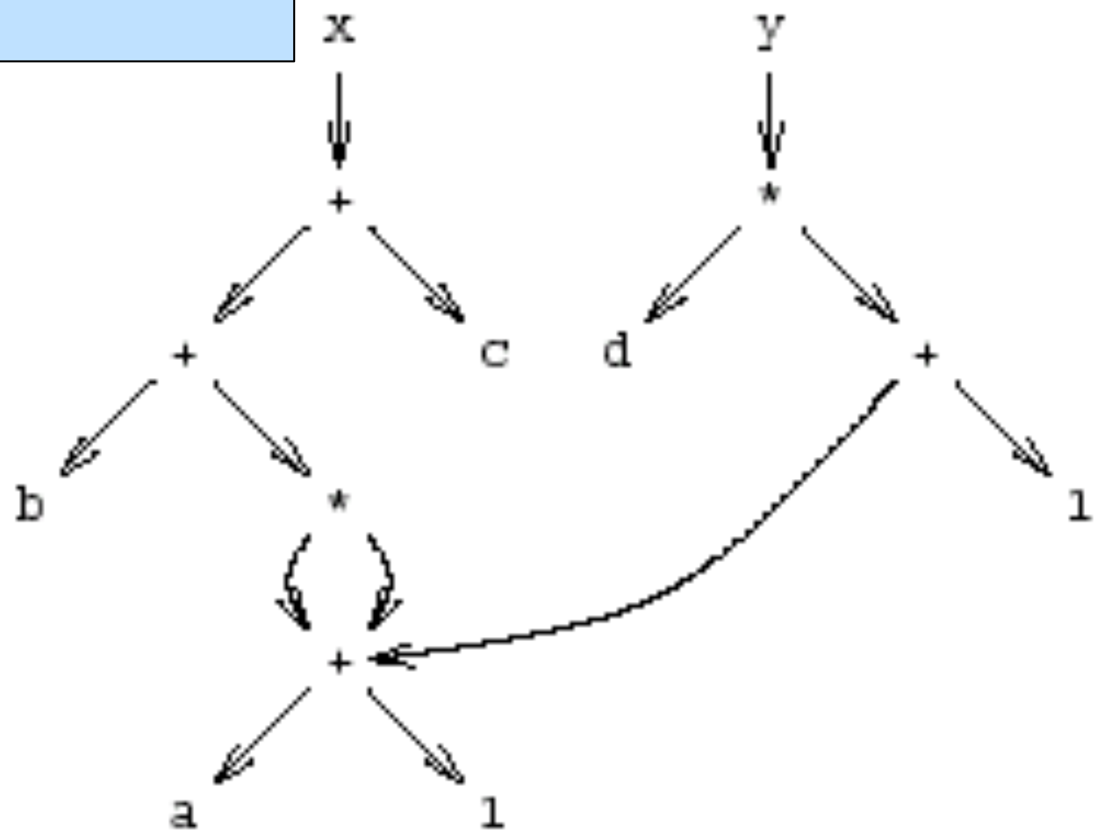
```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

# Dependency graph

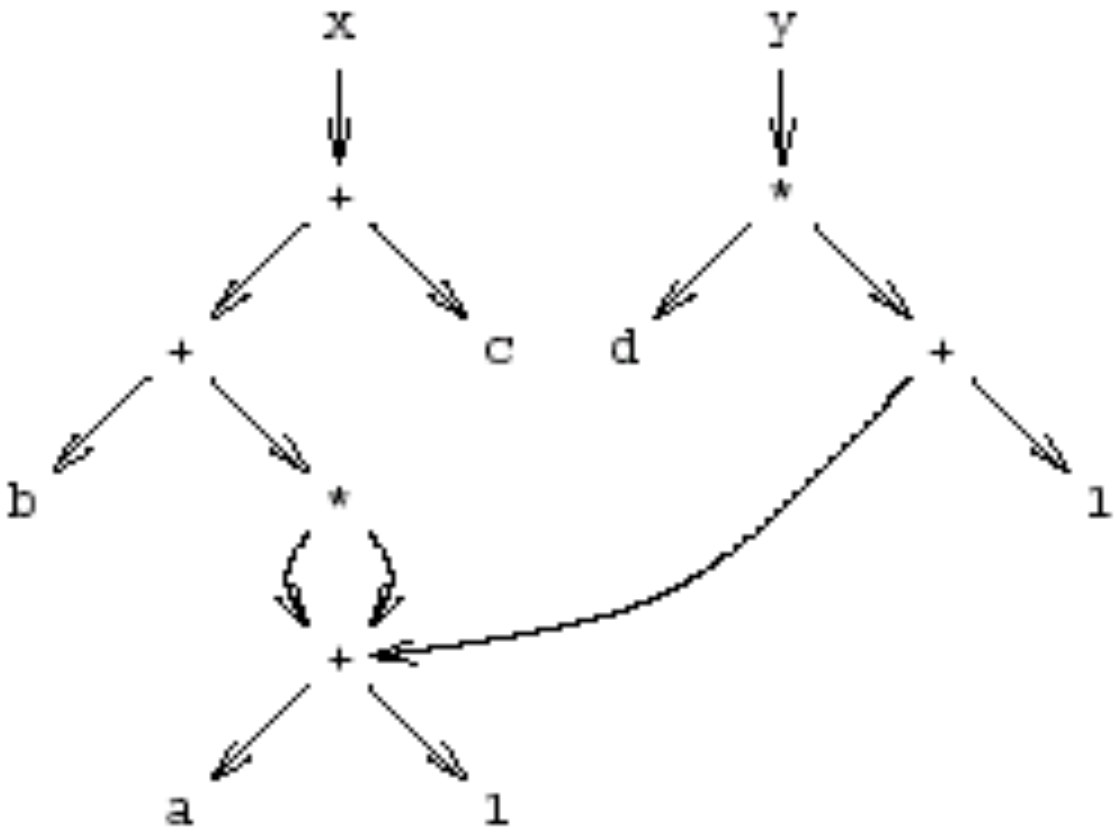


```
{
    int n;
    n := a + 1;
    x := b + n * n + c;
    n := n + 1;
    y := d * n;
}
```

# Simplified Data Dependency Graph



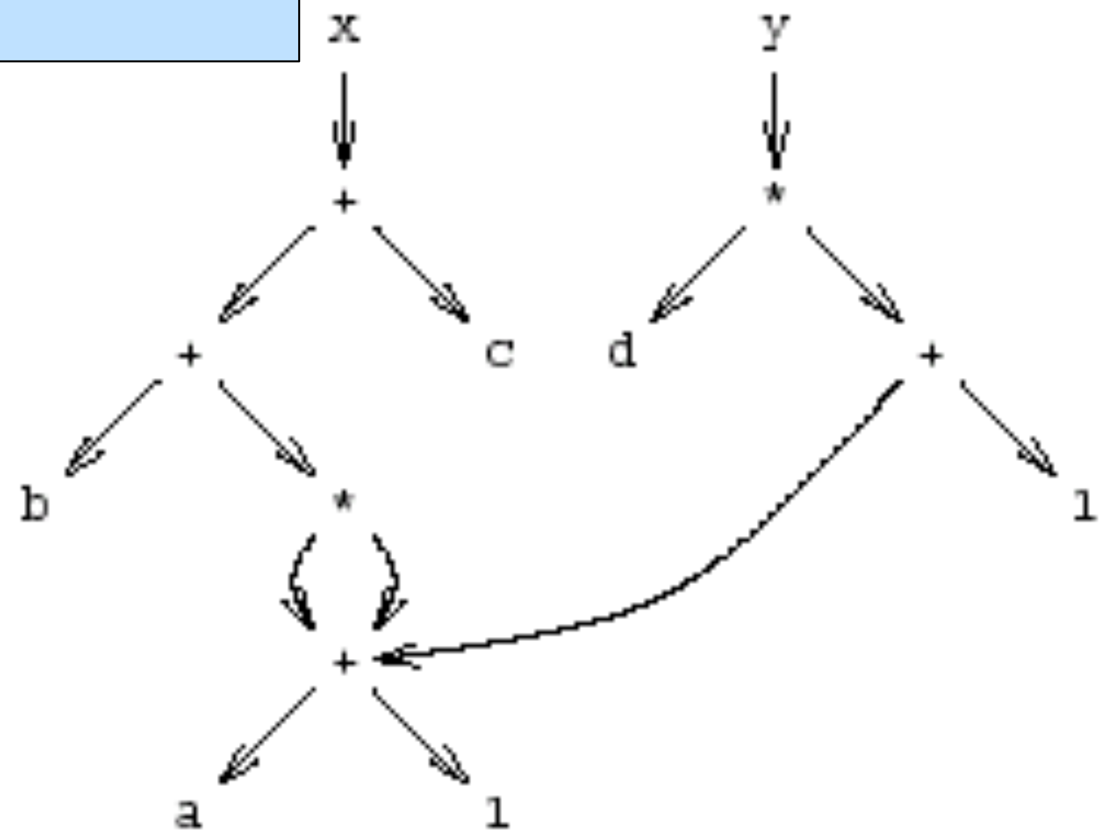
# Pseudo Register Target Code



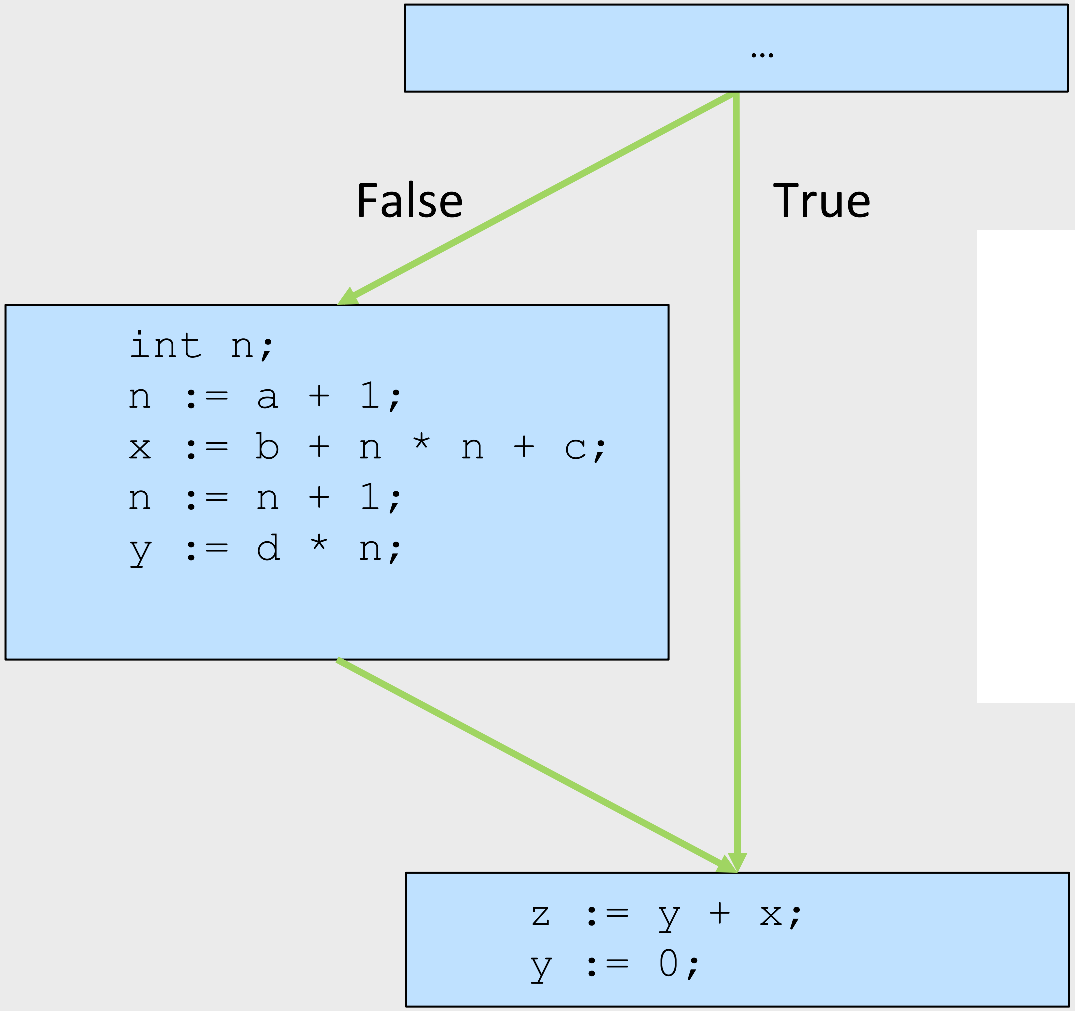
```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg     R1,X1

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
```

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

# Question: Why “`y`”?



# Question: Why "y"?



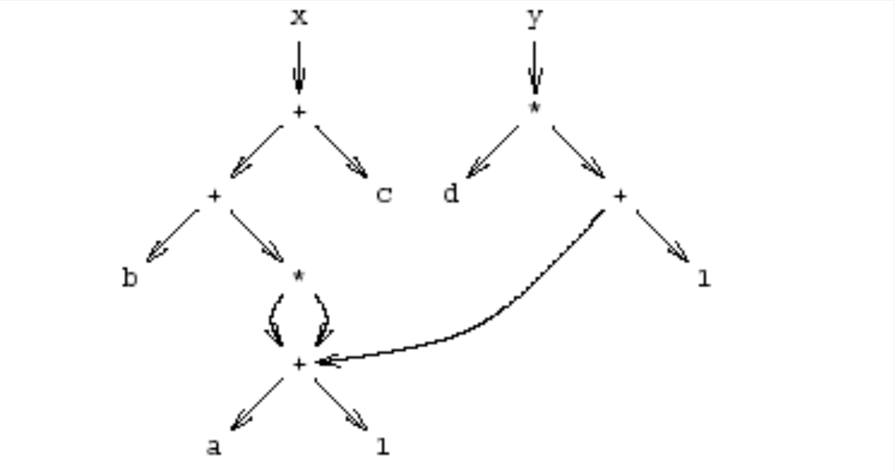
…

False

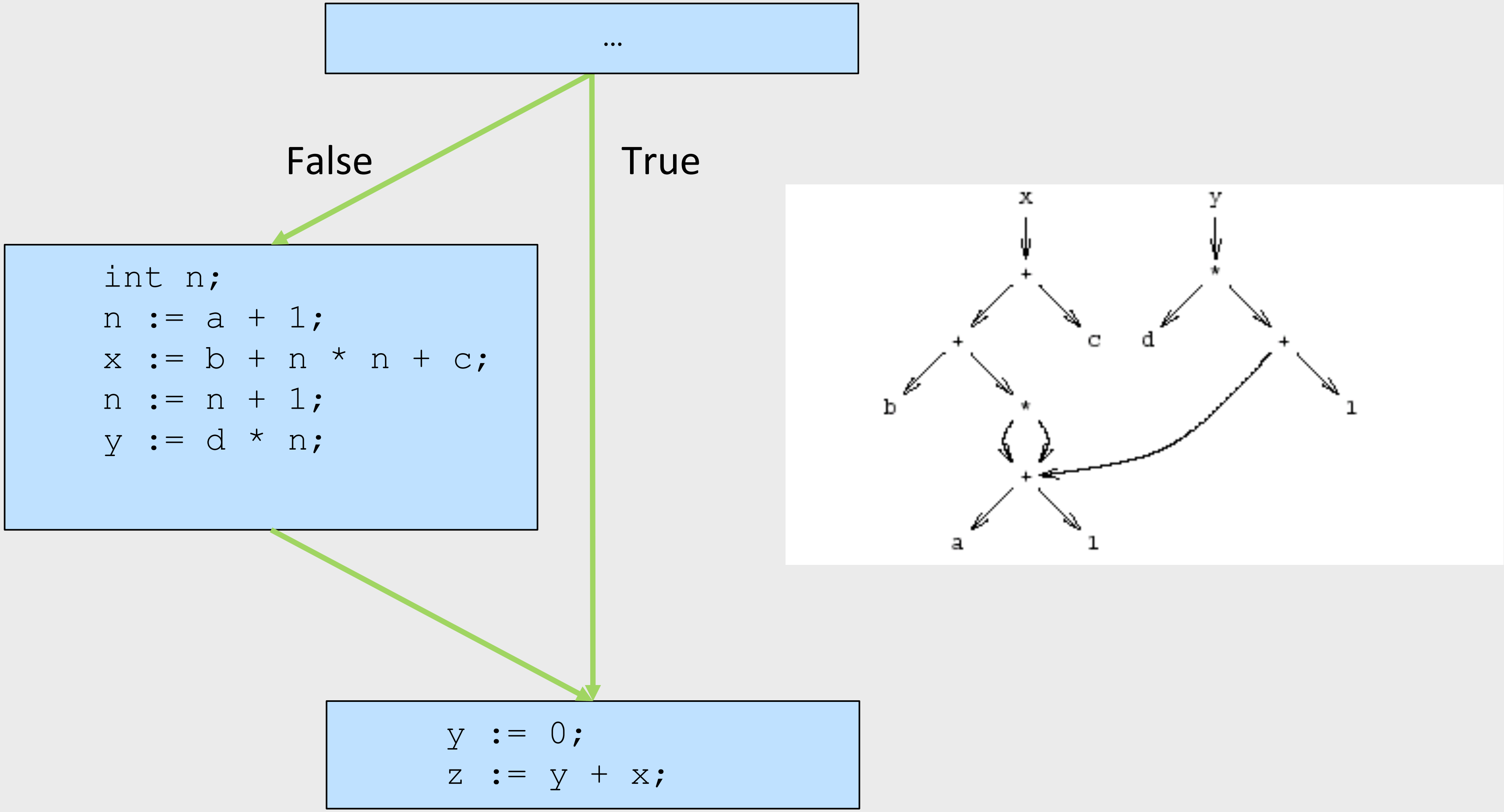
True

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

```
z := y + x;
y := 0;
```



# Question: Why "y"?



…

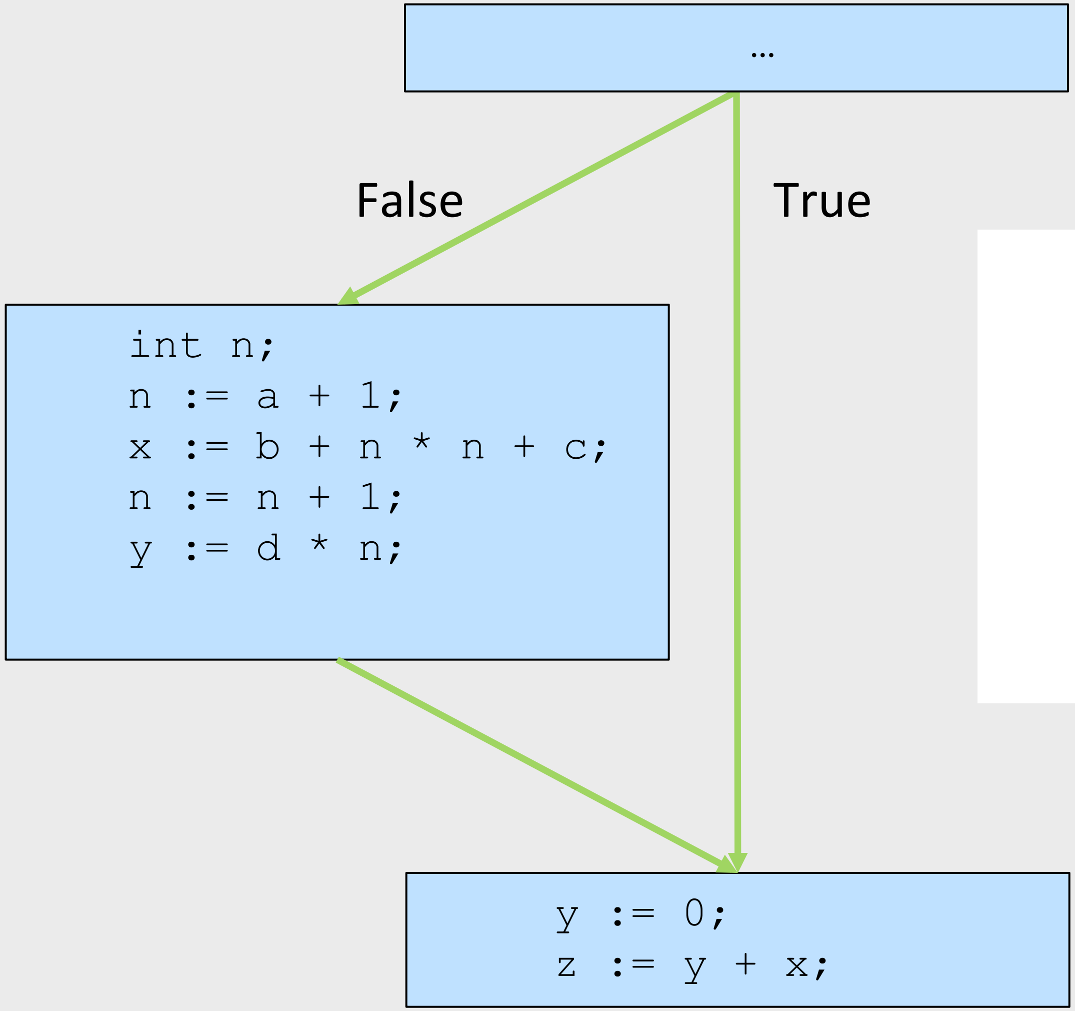
False

True

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

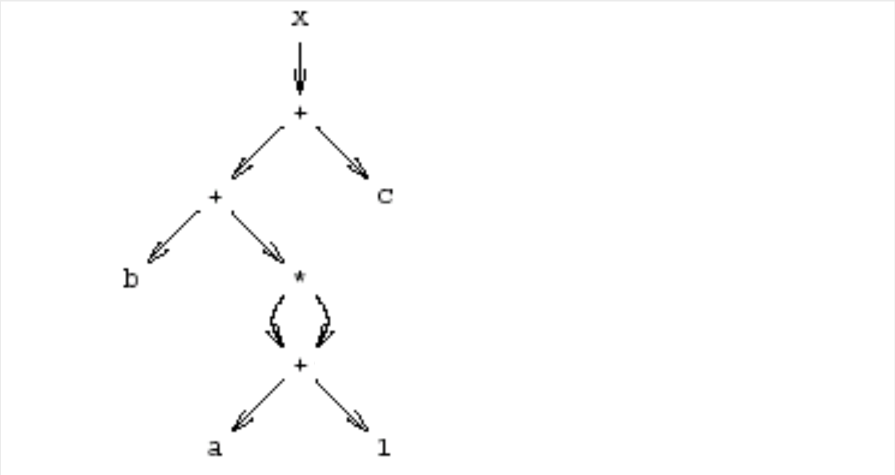
```
y := 0;
z := y + x;
```

# Question: Why "y"?

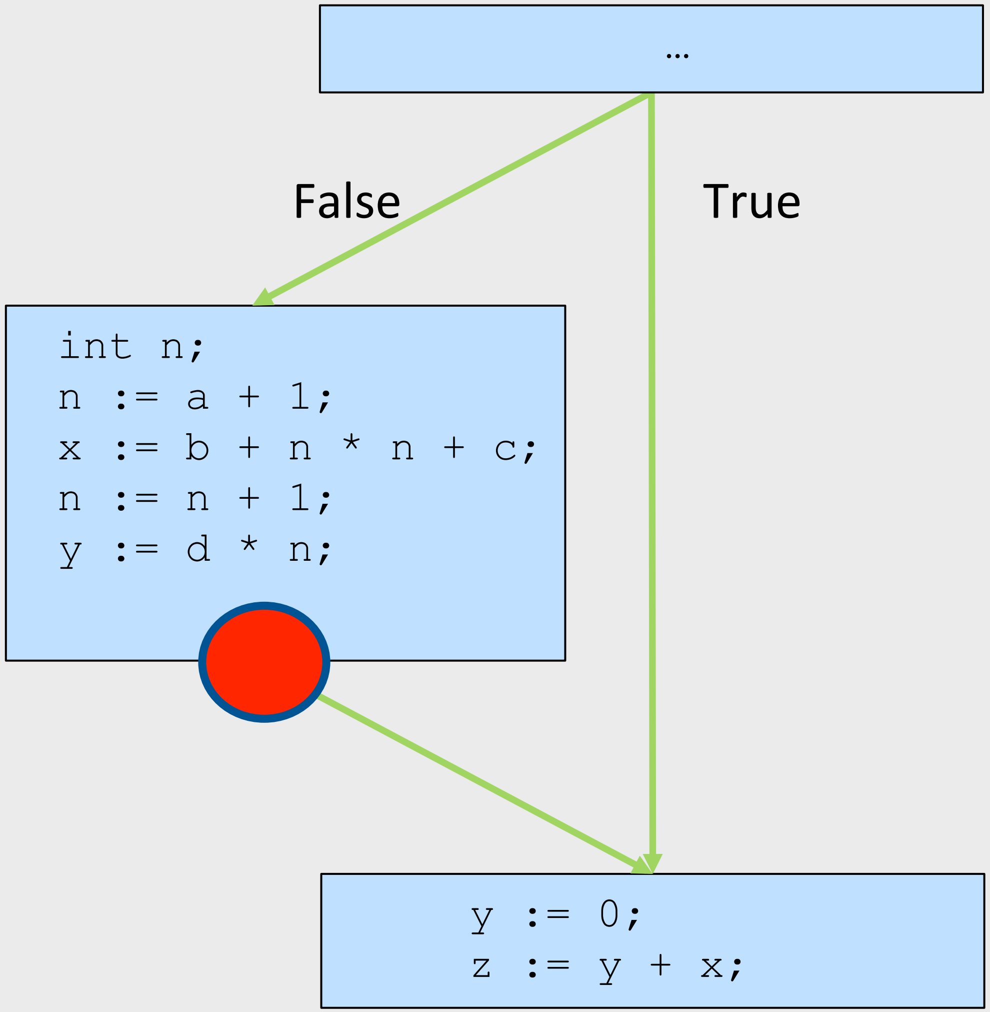


```
int n;
n := a + 1;
x := b + n * n + c;
n := n + 1;
y := d * n;
```

```
y := 0;
z := y + x;
```

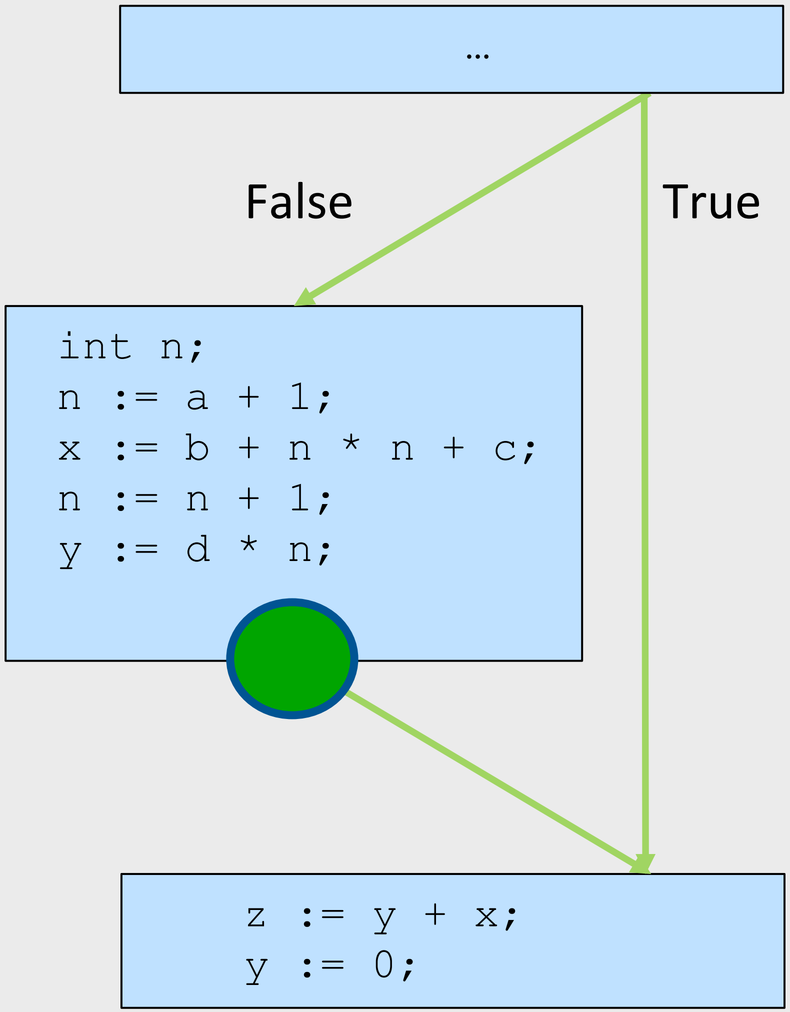


# y, dead or alive?



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

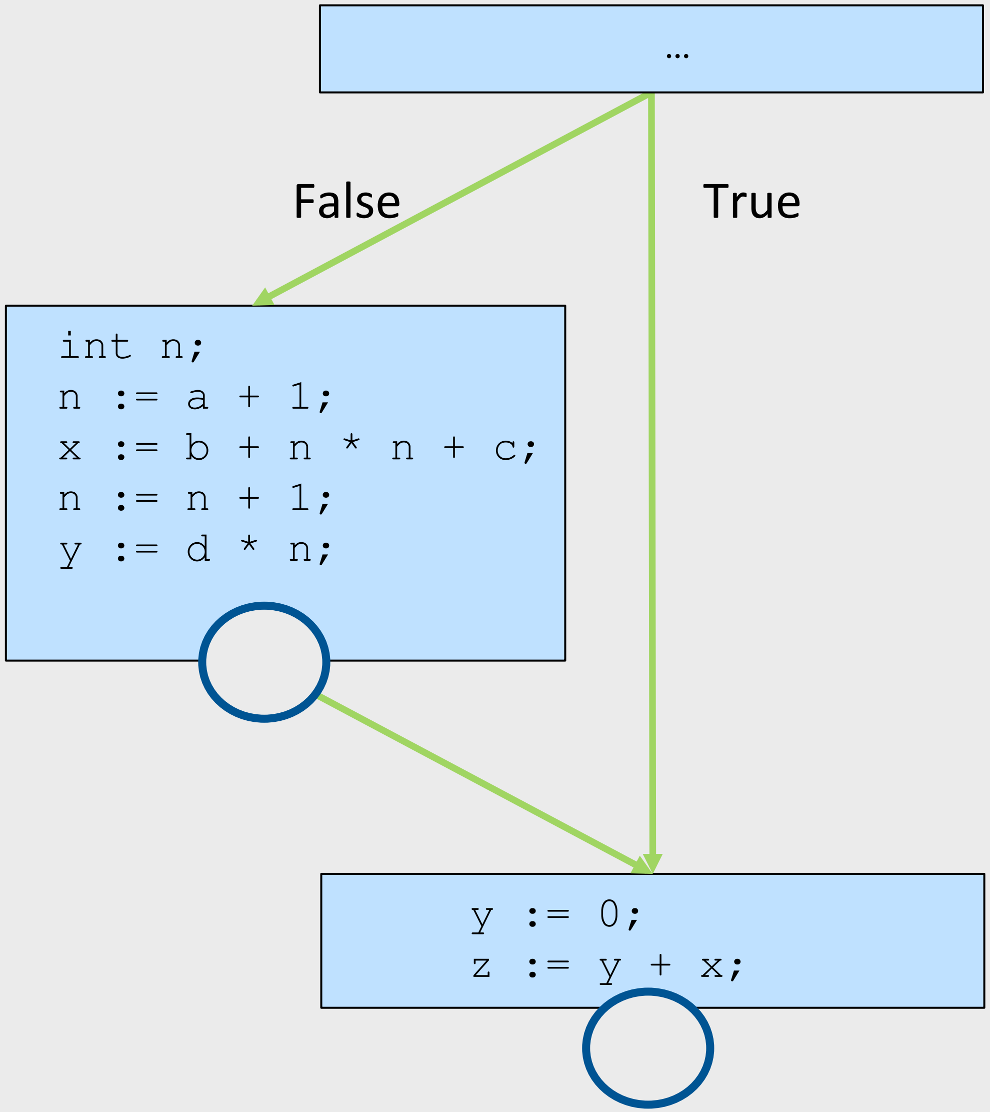
```
y := 0;
z := y + x;
```



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

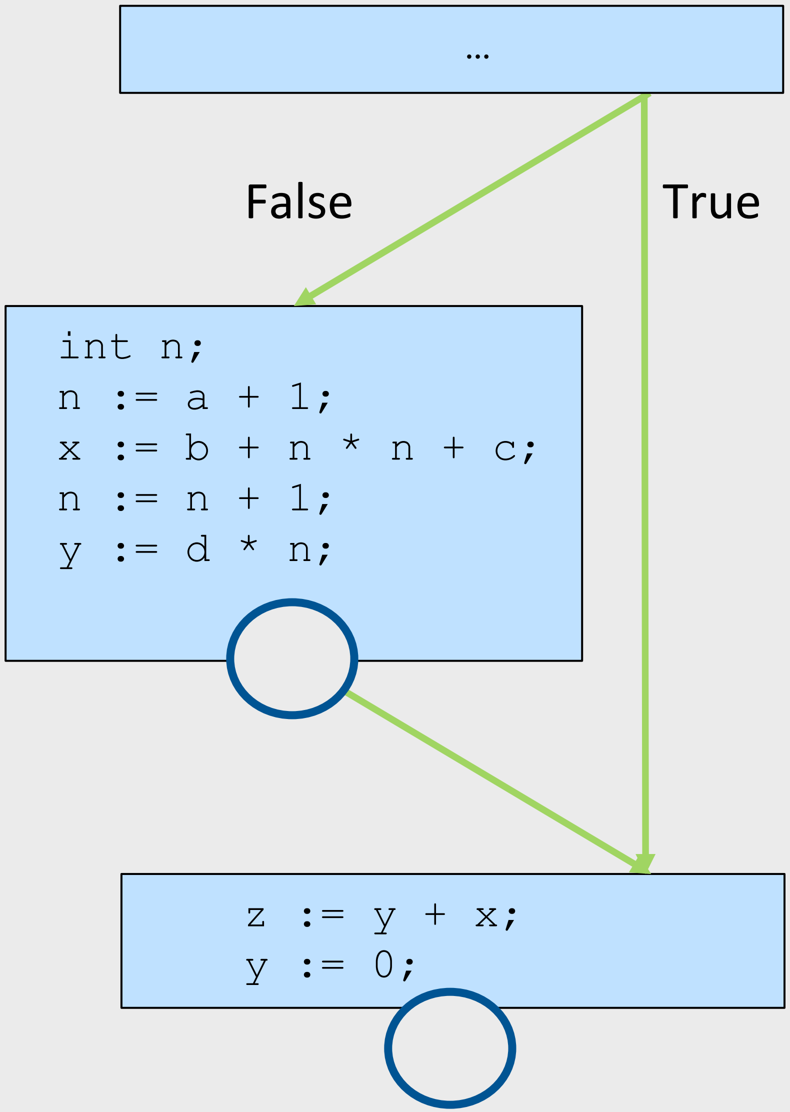
```
z := y + x;
y := 0;
```

# x, dead or alive?



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

```
y := 0;
z := y + x;
```



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

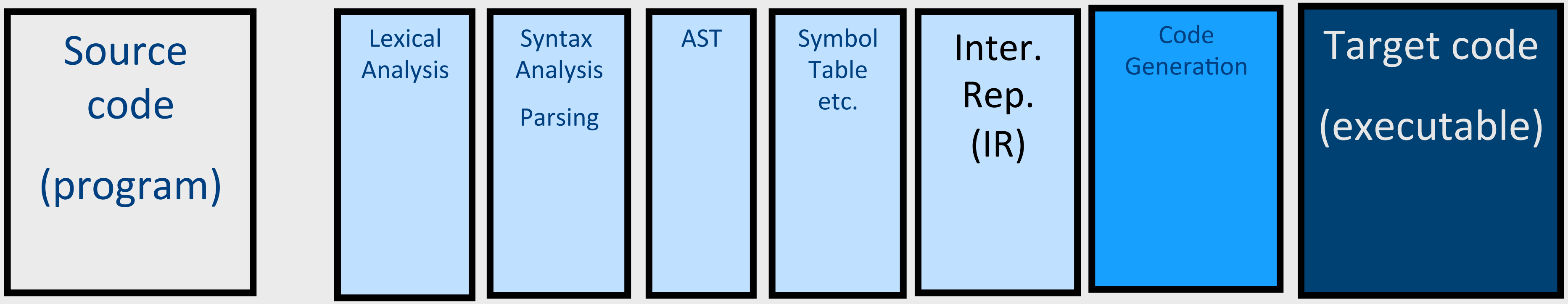
```
z := y + x;
y := 0;
```

# Register Allocation

- Machine-agnostic optimizations
    - Assume unbounded number of registers
  - Expression trees
  - Basic blocks

- Machine-dependent optimization
    - K registers
    - Some have special purposes
  - Control flow graphs (global register allocation)

# Register Allocation: Assembly



Source code

(program)

Lexical Analysis

Syntax Analysis

Parsing

AST

Symbol Table etc.

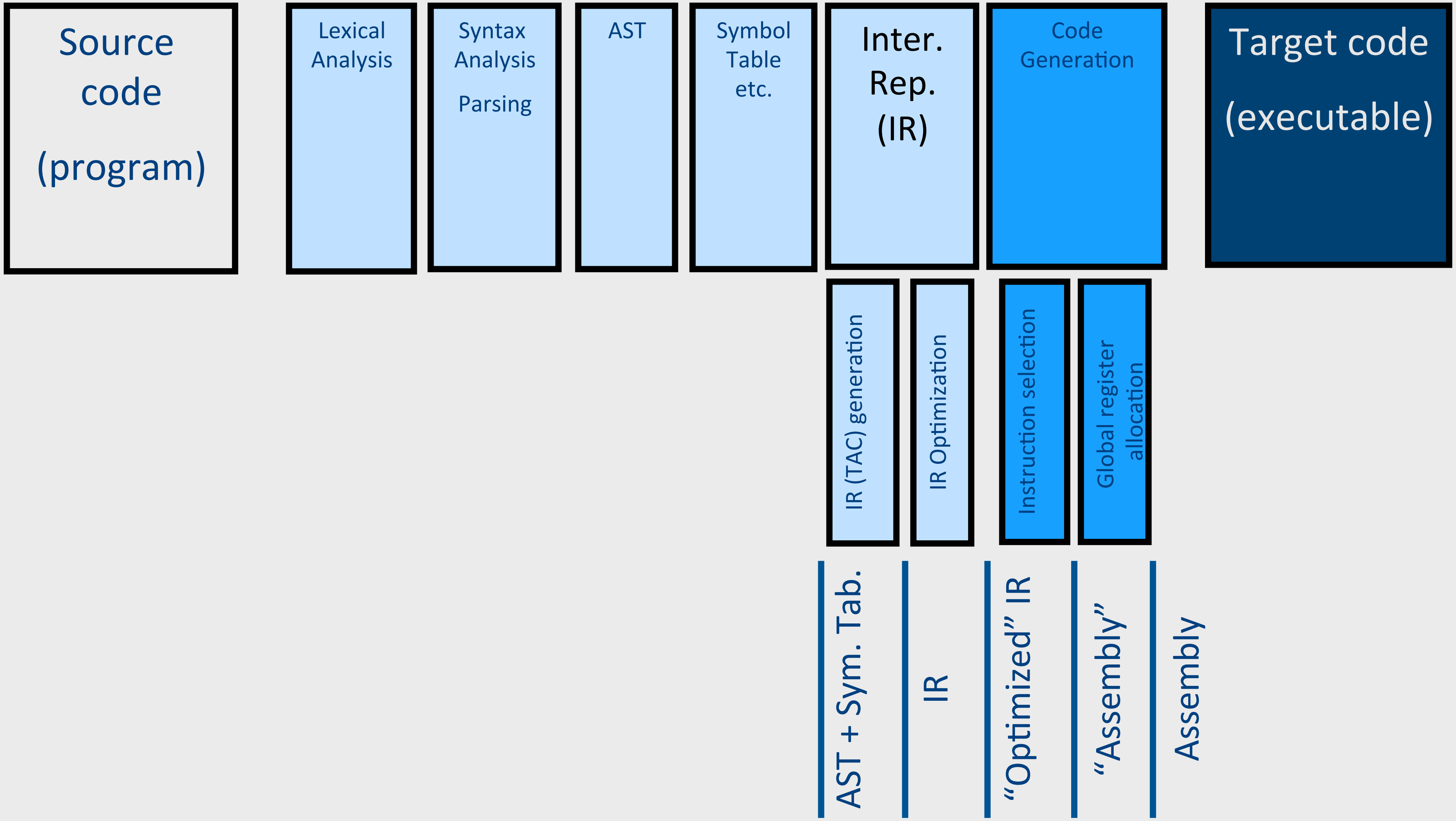
Inter. Rep. (IR)

Code Generation

Target code

(executable)

# Register Allocation: Assembly



Source code

(program)

Lexical Analysis

Syntax Analysis

Parsing

AST

Symbol Table etc.

Inter. Rep. (IR)

Code Generation

Target code

(executable)

IR (TAC) generation

IR Optimization

Instruction selection

Global register allocation

AST + Sym. Tab.

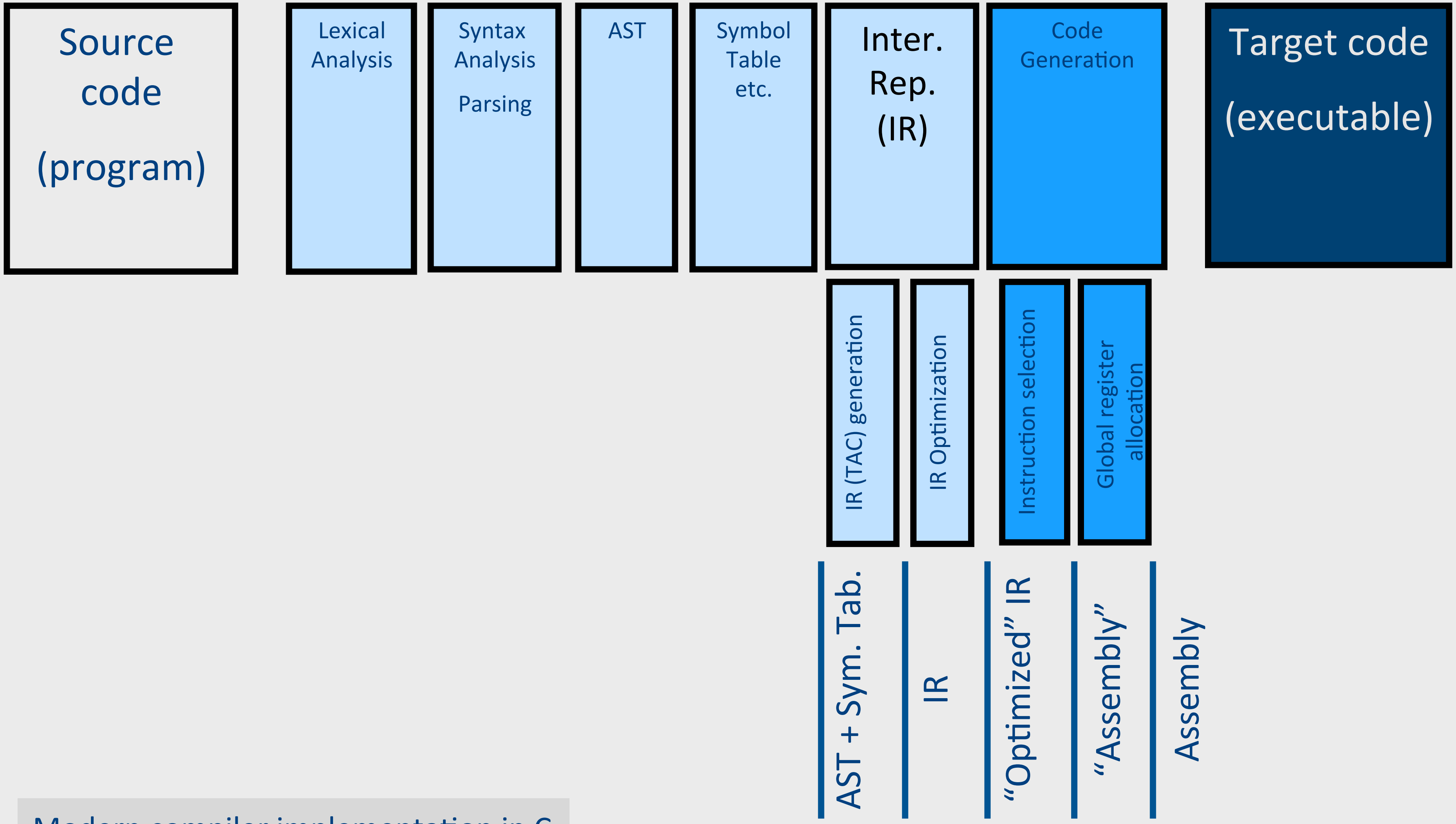
IR

“Optimized” IR

“Assembly”

Assembly

# Register Allocation: Assembly



Modern compiler implementation in C
Andrew A. Appel

# “Global” Register Allocation

- Input:
  - Sequence of machine instructions (“assembly”)
    - Unbounded number of temporary variables
      - aka symbolic registers
  - “machine description”
    - # of registers, restrictions
- Output
  - Sequence of machine instructions using machine registers (assembly)
  - Some MOV instructions removed

# 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

| | |
|---|---|
| `y = 42` | x undef, y live, z undef |
| `z = 73` | x undef, y live, z live |
| `x = y + z` | x is live, y dead, z dead |
| `print(x);` | x is dead, y dead, z dead |

(showing state after the statement)

# Find a register allocation

| variable | register |
|---|---|
| a | ? |
| b | ? |
| c | ? |

register

eax

ebx

```
b = a + 2

c = b * b

b = c + 1

return b * c
```

# Is this a valid allocation?

| variable | register |
|---|---|
| a | eax |
| b | ebx |
| c | eax |

register

eax

ebx

```
b = a + 2

c = b * b

b = c + 1

return b * c
```

```
ebx = eax + 2

eax = ebx * ebx

ebx = eax + 1

return ebx * eax
```

Overwrites previous value of ‘a’ also stored in eax

# Is this a valid allocation?

| variable | register |
|---|---|
| a | eax |
| b | ebx |
| c | eax |

register

eax

ebx

```
b = a + 2

c = b * b

b = c + 1

return b * c
```

```
ebx = eax + 2

eax = ebx * ebx

ebx = eax + 1

return ebx * eax
```

Value of ‘a’ stored in eax is not needed anymore so reuse it for ‘b’

# Is this a valid allocation?

| variable | register |
|---|---|
| a | eax |
| b | ebx |
| c | eax |

register

eax

ebx

```
b = a + 2

c = b * b

b = c + 1

return b * a
```

```
ebx = eax + 2

eax = ebx * ebx

ebx = eax + 1

return ebx * eax
```

Value of ‘a’ stored in eax is not needed anymore so reuse it for ‘b’

# Main idea

- 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
```

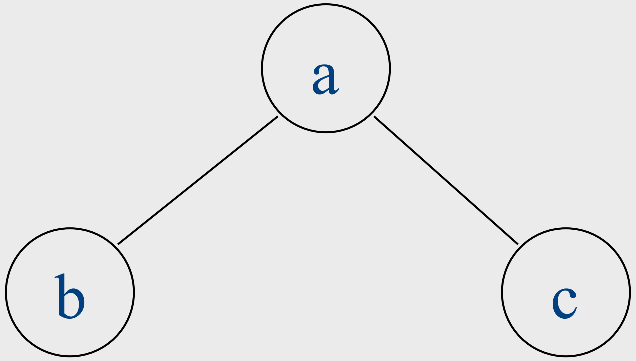
# Interference graph construction

```
                    {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
```

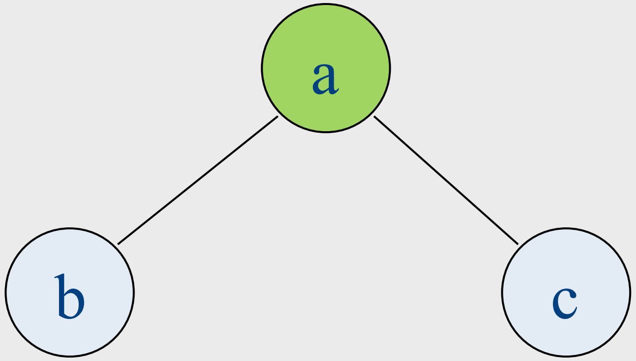
| color | register |
|---|---|
| ⬜ | eax |
| 🟩 | ebx |



# Colored graph

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

| color | register |
|---|---|
| ⬜ | eax |
| 🟩 | ebx |



# Graph coloring

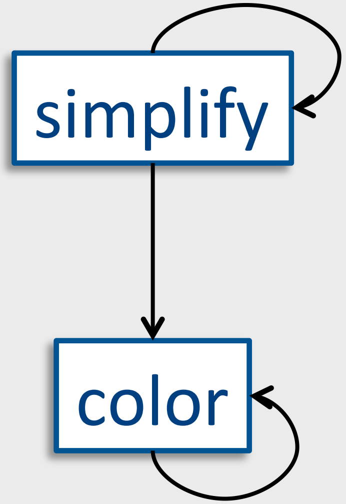
- This problem is equivalent to graph-coloring, which is NP-hard if there are at least three registers
- No good polynomial-time algorithms (or even good approximations!) are known for this problem
  - We have to be content with a heuristic that is good enough for RIGs that arise in practice

# 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

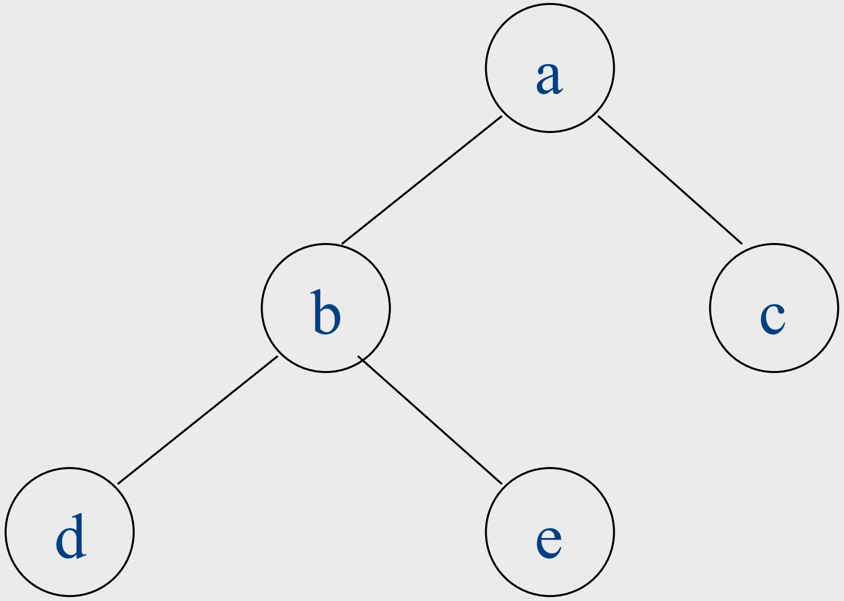
# Coloring by simplification [Kempe 1879]

- 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

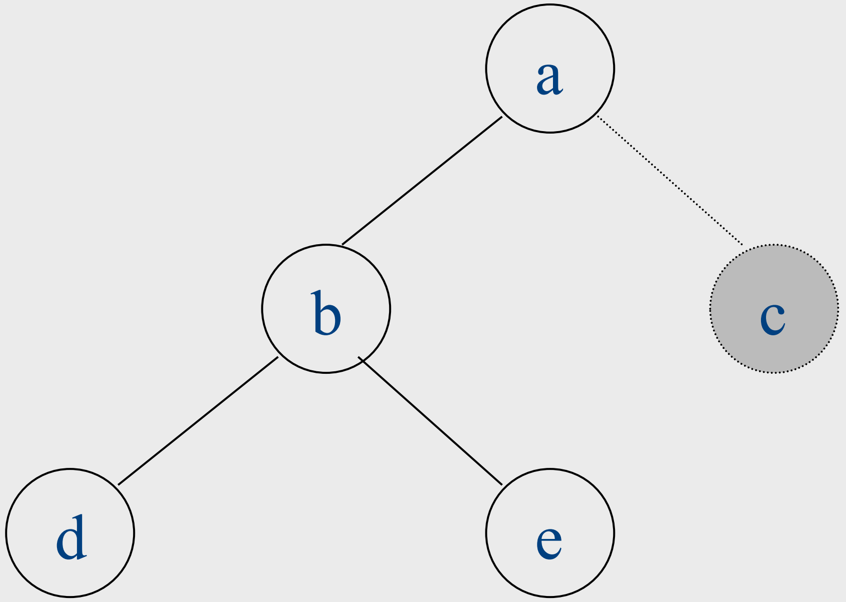
| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |



stack:

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

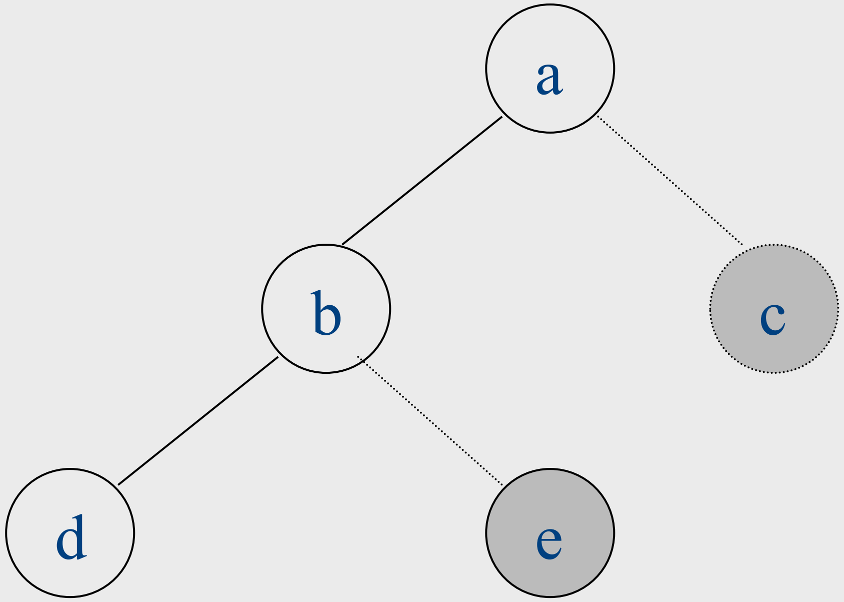


stack:

c

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

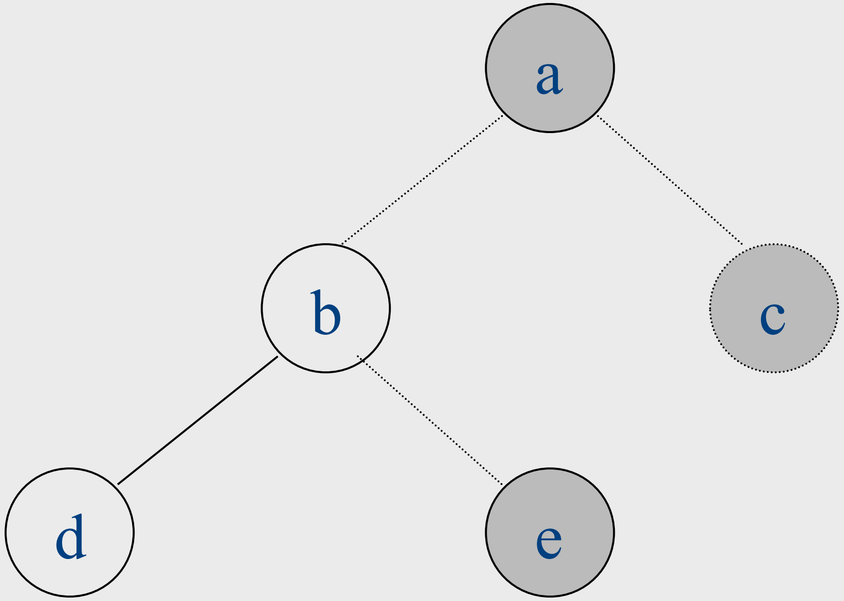


stack:

e
c

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

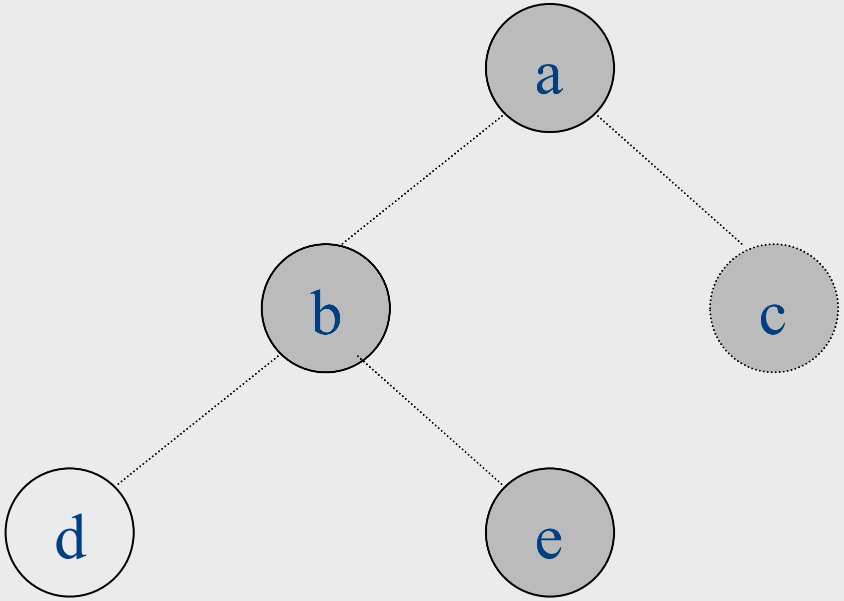


stack:

a
e
c

# Coloring k=2

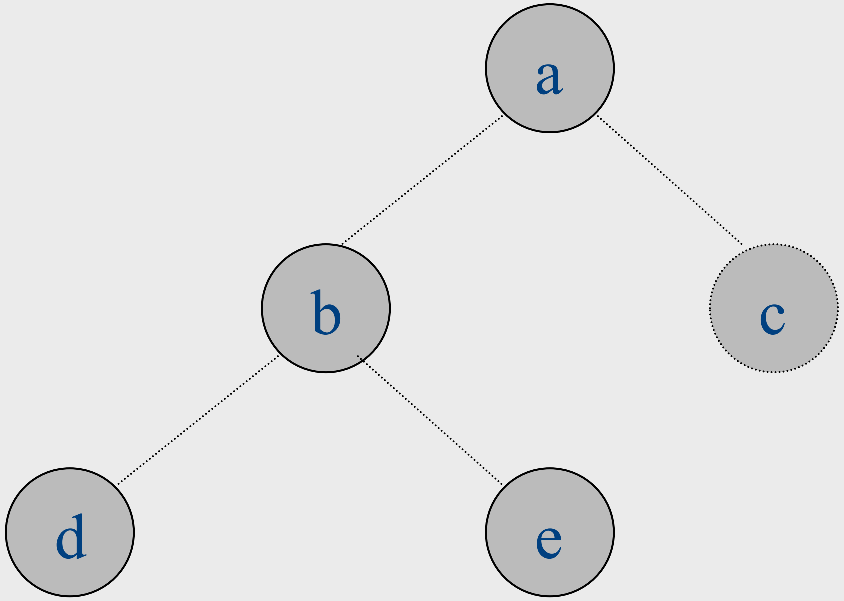
| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |



stack:
b
a
e
c

# Coloring k=2

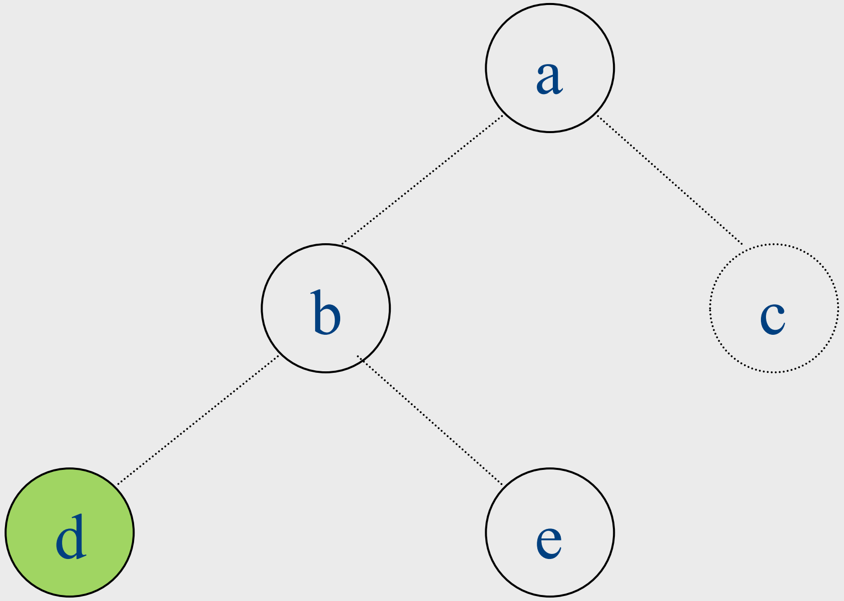
| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |



stack:
d
b
a
e
c

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

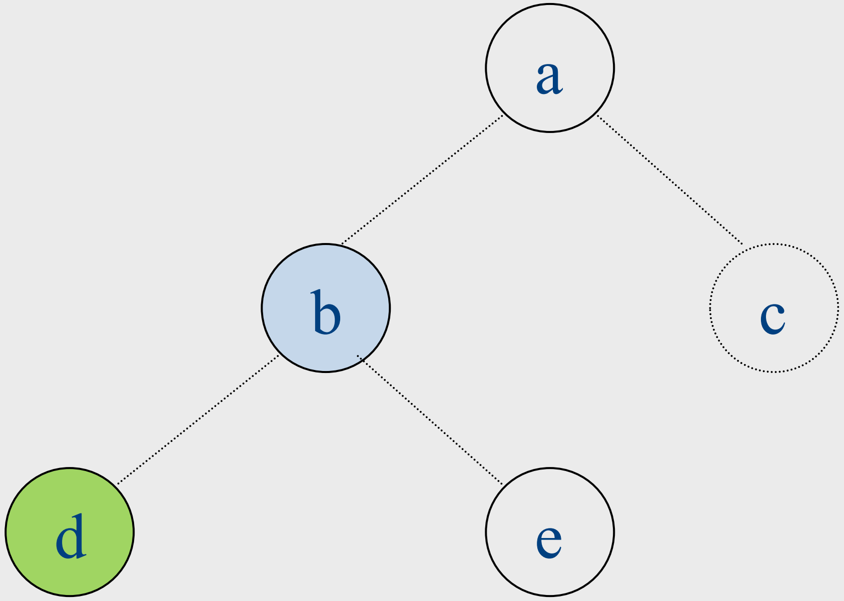


stack:

b
a
e
c

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

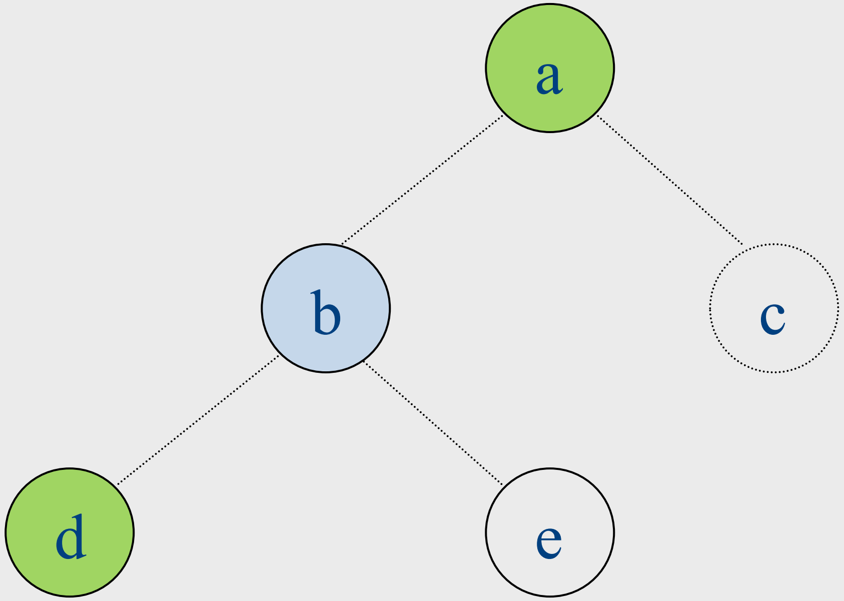


stack:

a
e
c

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

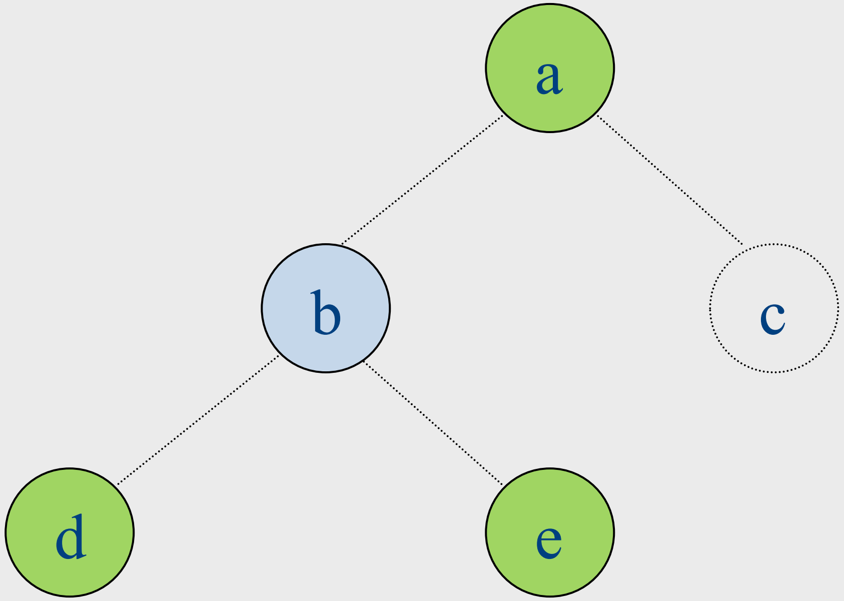


stack:

e
c

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

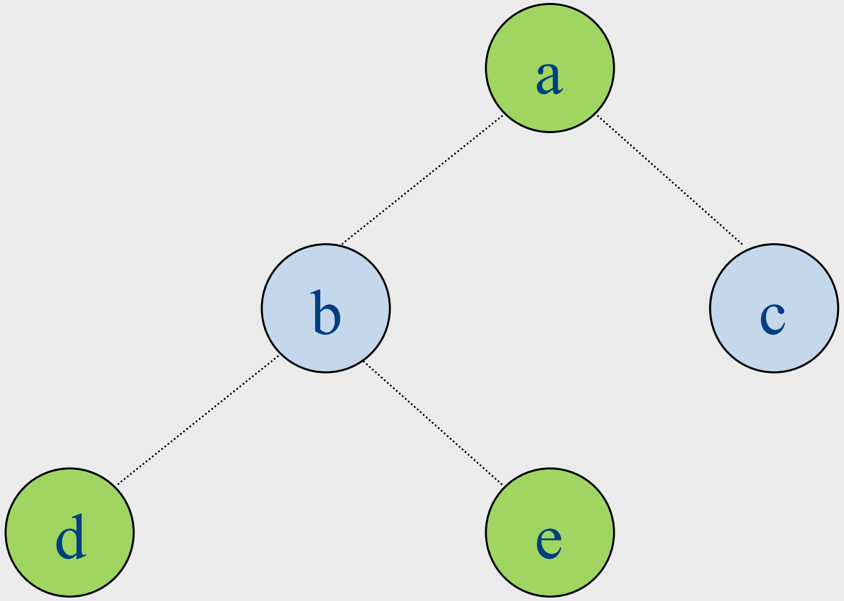


stack:

c

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |



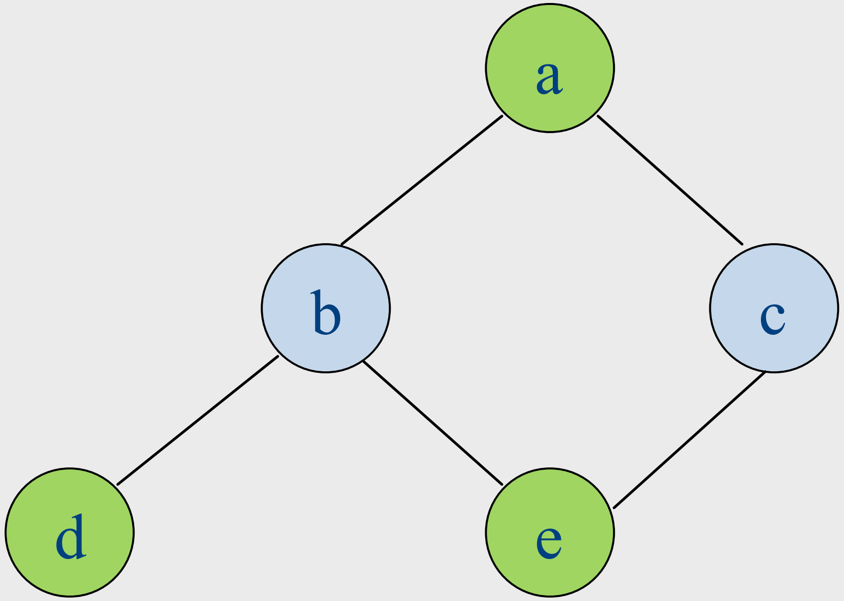
stack:

# 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

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

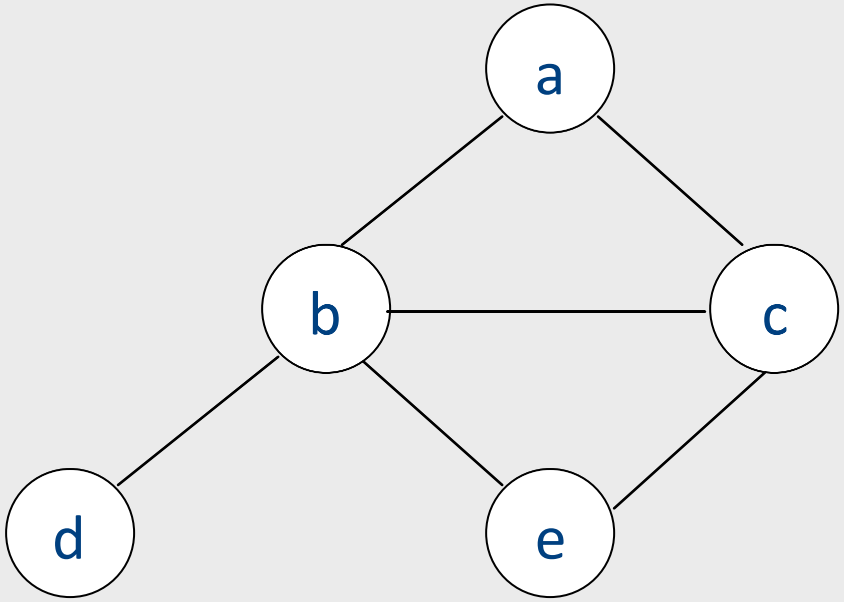


stack:

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can’t be colored in K colors:

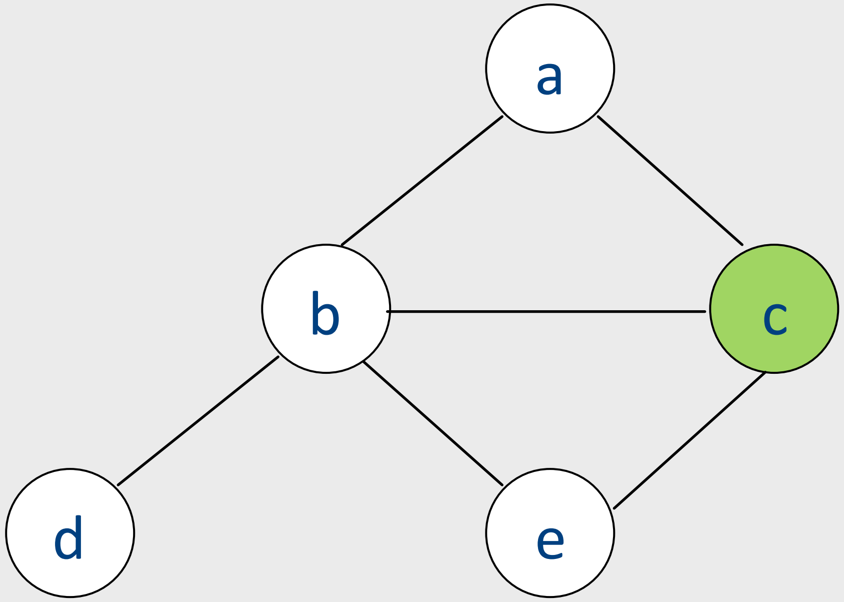


stack:
c
b
e
a
d

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can’t be colored in K colors:

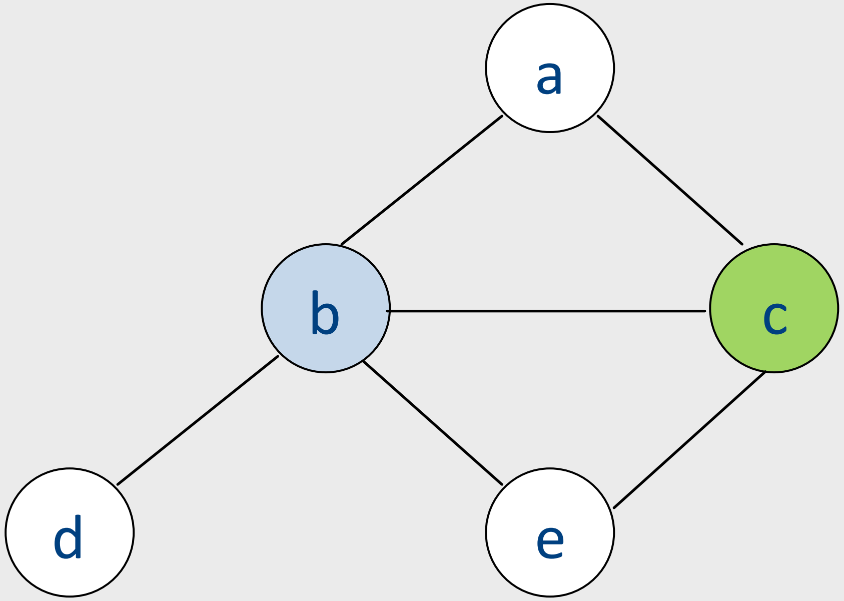


stack:
b
e
a
d

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can’t be colored in K colors:

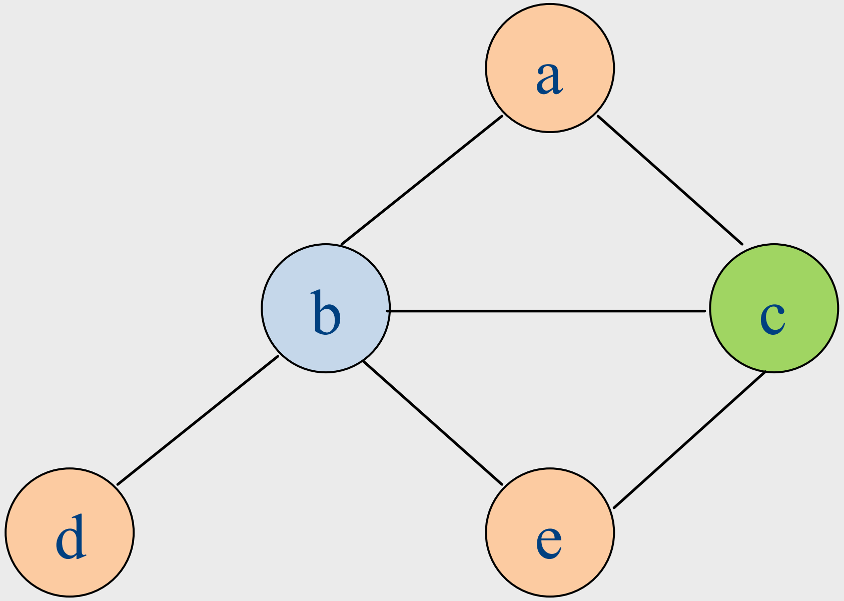


stack:
e
a
d

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

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

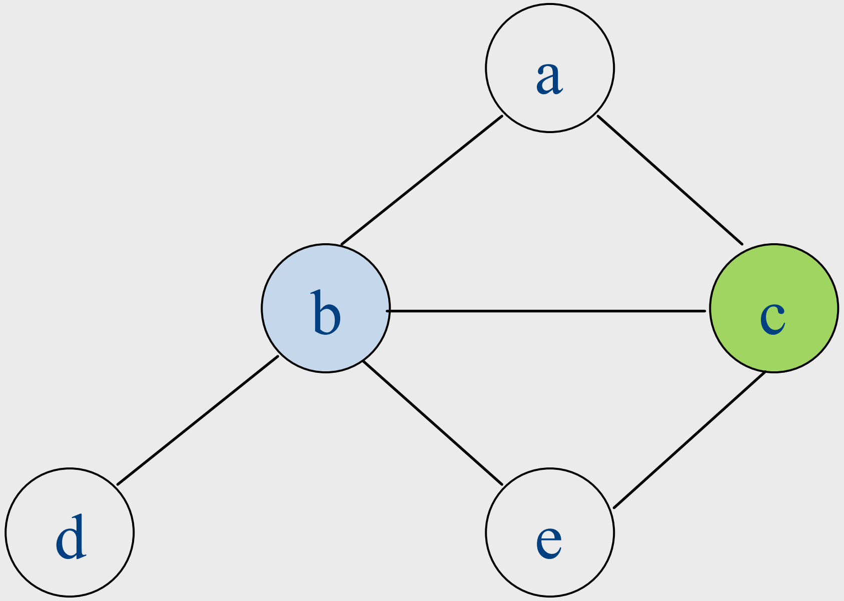
# Spilling

- 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

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can't be colored in K colors:



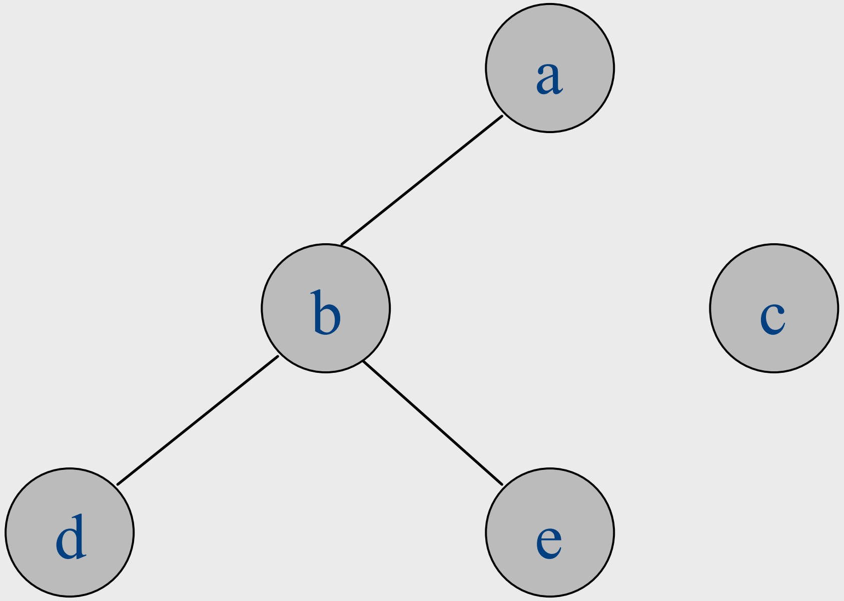
stack:
e
a
d

**no colors left for e!**

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can’t be colored in K colors:

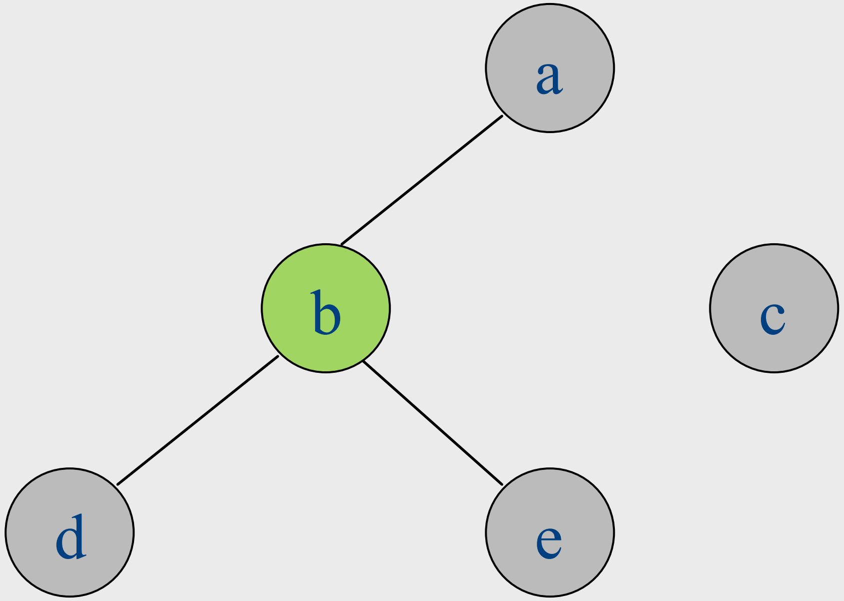


stack:
b
e
a
d

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can't be colored in K colors:

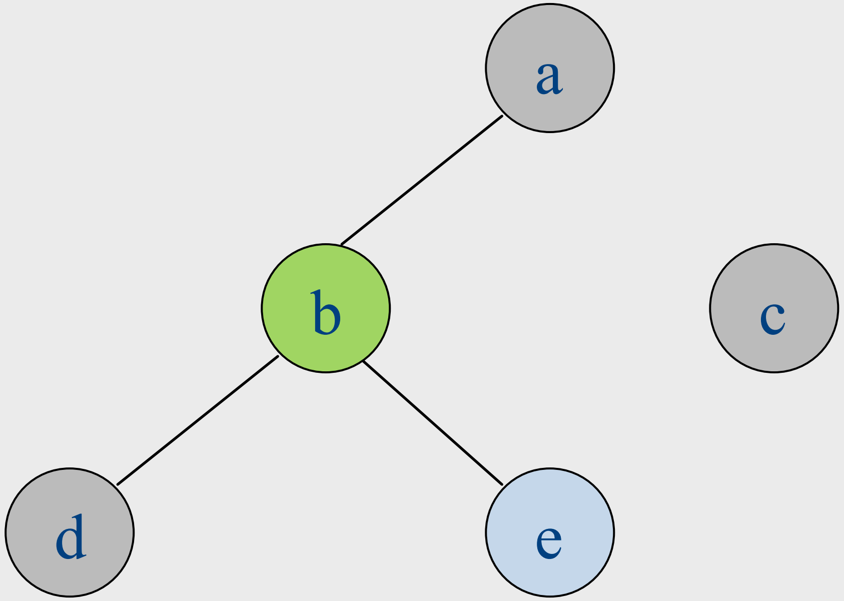


stack:
e
a
d

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can’t be colored in K colors:

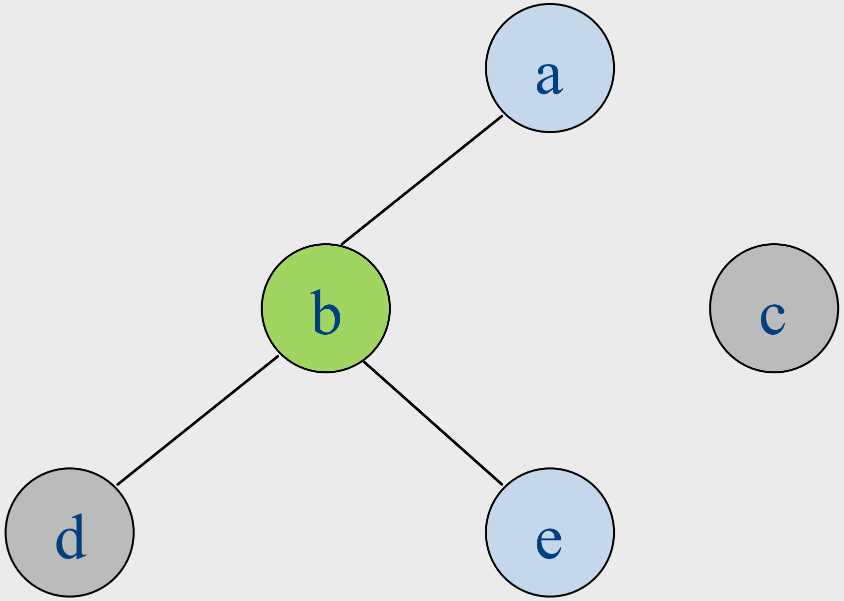


stack:
a
d

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can’t be colored in K colors:

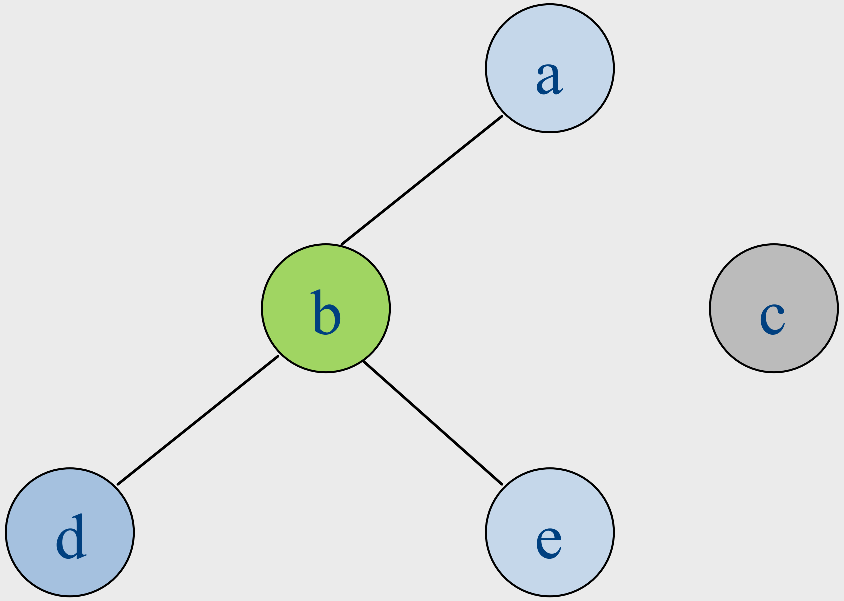


stack:
d

# Coloring k=2

| color | register |
|---|---|
| 🟦 | eax |
| 🟩 | ebx |

Some graphs can't be colored in K colors:



stack:

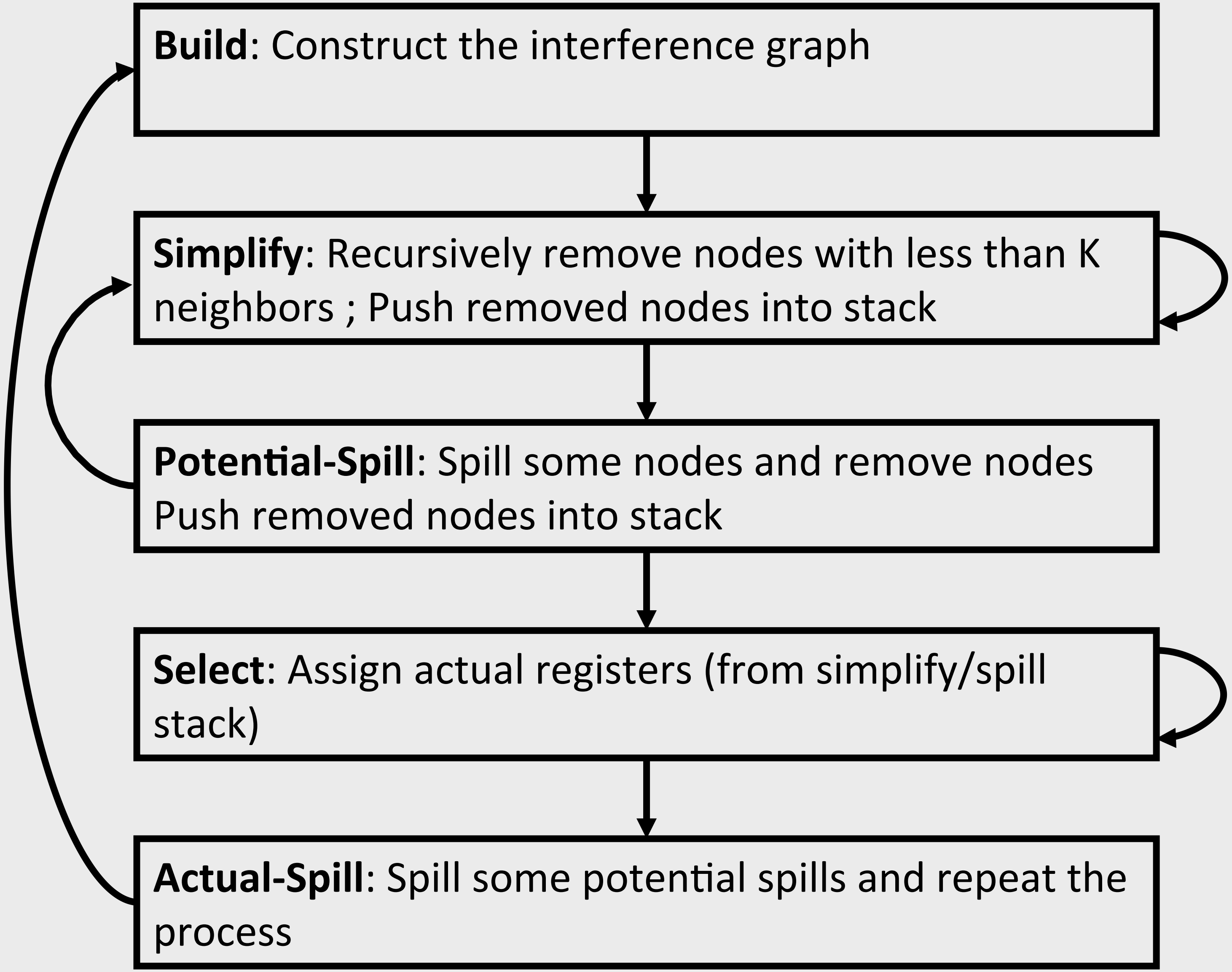
# Handling precolored nodes

- 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

# Graph Coloring by Simplification



**Build**: Construct the interference graph

**Simplify**: Recursively remove nodes with less than K neighbors ; Push removed nodes into stack

**Potential-Spill**: Spill some nodes and remove nodes Push removed nodes into stack

**Select**: Assign actual registers (from simplify/spill stack)

**Actual-Spill**: Spill some potential spills and repeat the process

# 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

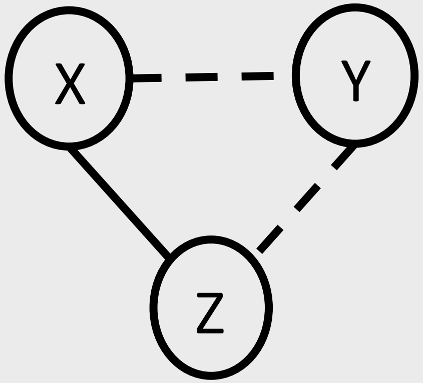
- MOVs can be removed if the source and the target share the same register
- The source and the target of the move can be merged into a single node (unifying the sets of neighbors)
  - May require more registers
  - Conservative Coalescing
    - Merge nodes only if the resulting node has fewer than K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



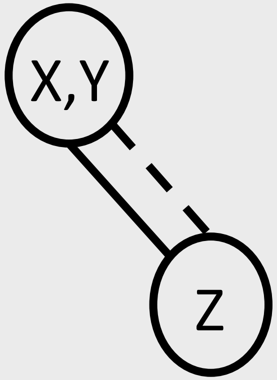
- Constrained MOVs are not coalesced

# Constrained Moves

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

X ← Y

Y ← Z



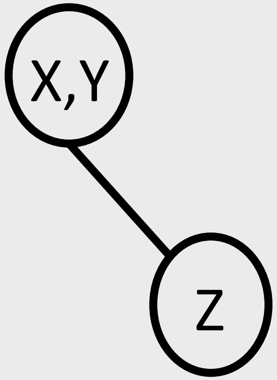
- Constrained MOVs are not coalesced

# Constrained Moves

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

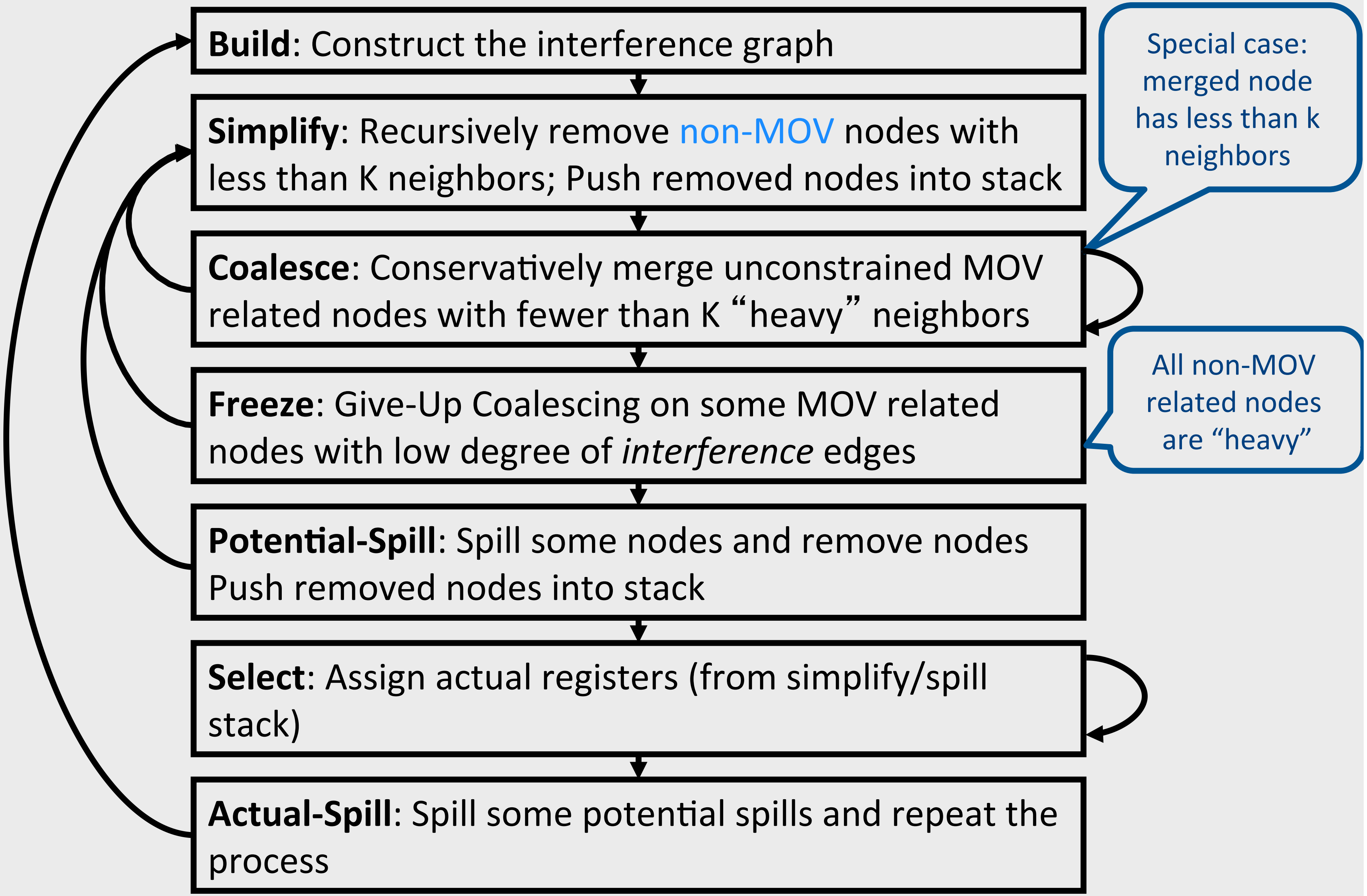
X ← Y

Y ← Z



- Constrained MOVs are not coalesced

# Graph Coloring with Coalescing



**Build**: Construct the interference graph

**Simplify**: Recursively remove non-MOV nodes with less than K neighbors; Push removed nodes into stack

**Coalesce**: Conservatively merge unconstrained MOV related nodes with fewer than K "heavy" neighbors

**Freeze**: Give-Up Coalescing on some MOV related nodes with low degree of *interference* edges

**Potential-Spill**: Spill some nodes and remove nodes Push removed nodes into stack

**Select**: Assign actual registers (from simplify/spill stack)

**Actual-Spill**: Spill some potential spills and repeat the process

Special case: merged node has less than k neighbors

All non-MOV related nodes are "heavy"

# Spilling

- Many heuristics exist
  - Maximal degree
  - Live-ranges
  - Number of uses in loops
- The whole process need to be repeated after an actual spill

# 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($r_7$)

$t_{231} \leftarrow r_7$

…

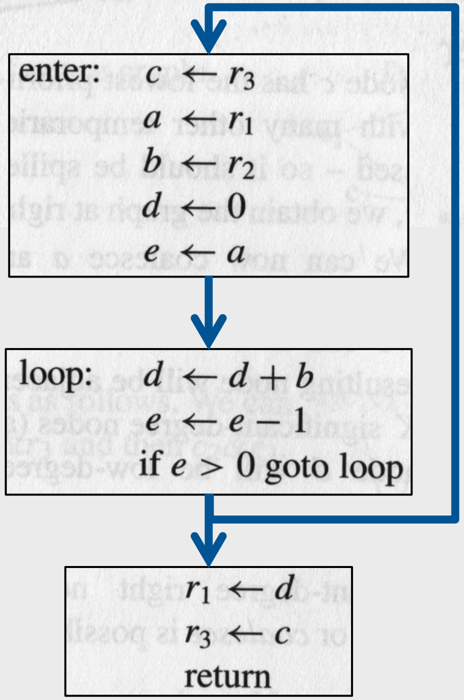
$r_7 \leftarrow t_{231}$

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;
}
```

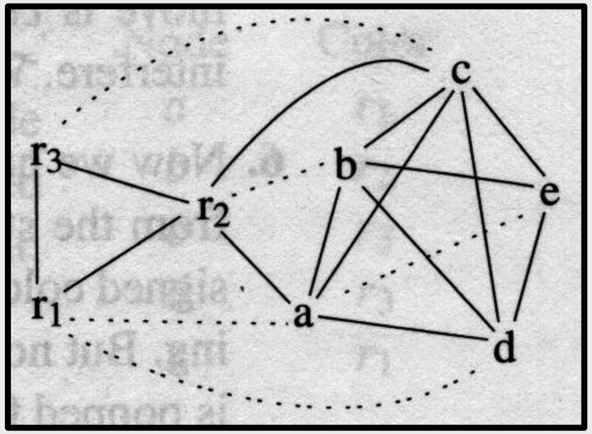
```
enter:  $c \leftarrow r_3$      Callee-saved registers
        $a \leftarrow r_1$
        $b \leftarrow r_2$      Caller-saved registers
        $d \leftarrow 0$
        $e \leftarrow a$
loop:   $d \leftarrow d + b$
        $e \leftarrow e - 1$
        if $e > 0$ goto loop
        $r_1 \leftarrow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



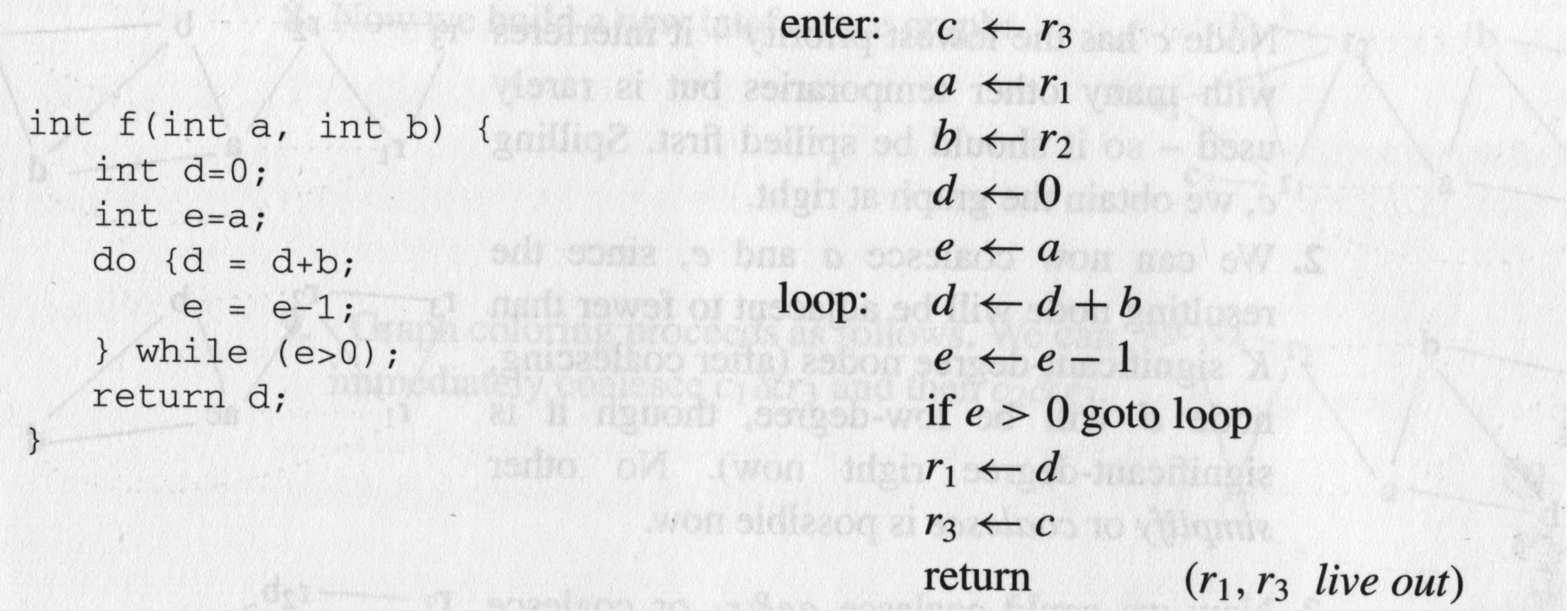
enter: $c \leftarrow r_3$
$a \leftarrow r_1$
$b \leftarrow r_2$
$d \leftarrow 0$
$e \leftarrow a$

loop: $d \leftarrow d + b$
$e \leftarrow e - 1$
if $e > 0$ goto loop

$r_1 \leftarrow d$
$r_3 \leftarrow c$
return



# 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: $d \leftarrow d + b$

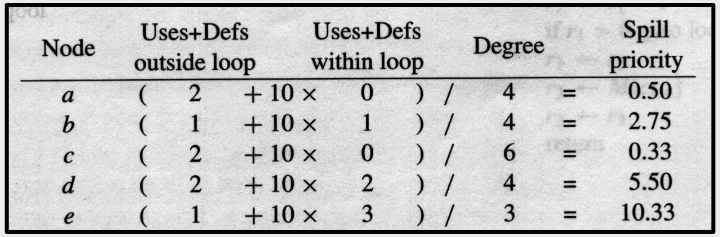
$e \leftarrow e - 1$

if $e > 0$ goto loop

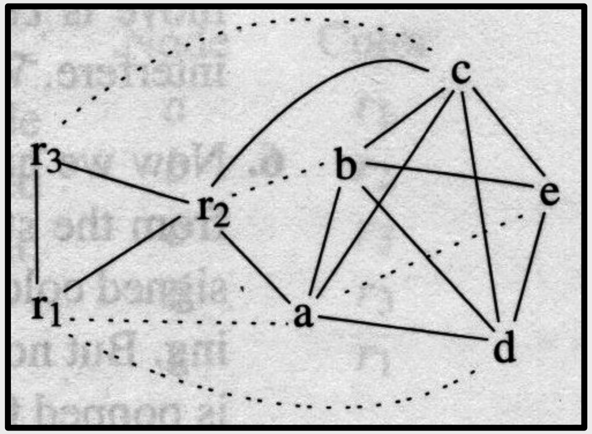
$r_1 \leftarrow d$

$r_3 \leftarrow c$

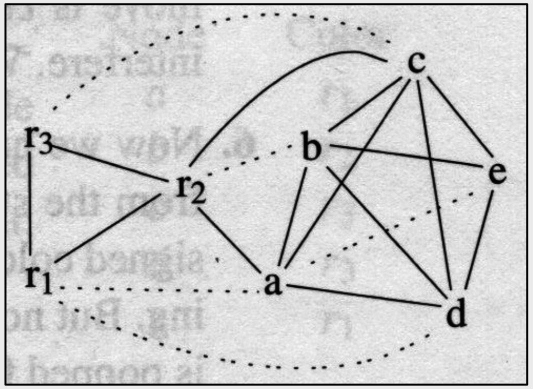
return $(r_1, r_3$ *live out*$)$



| 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 |

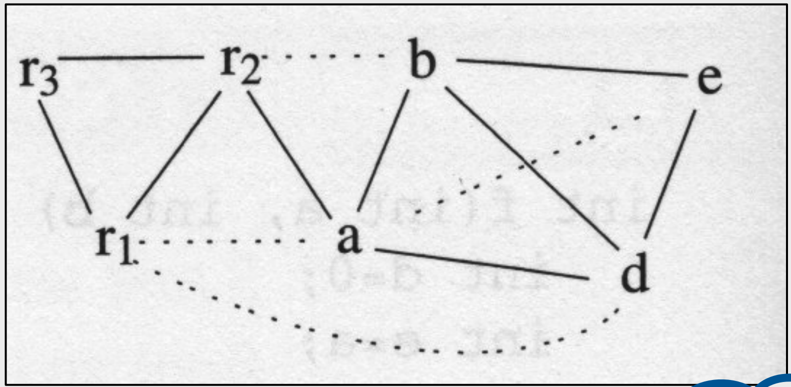


# A Complete Example



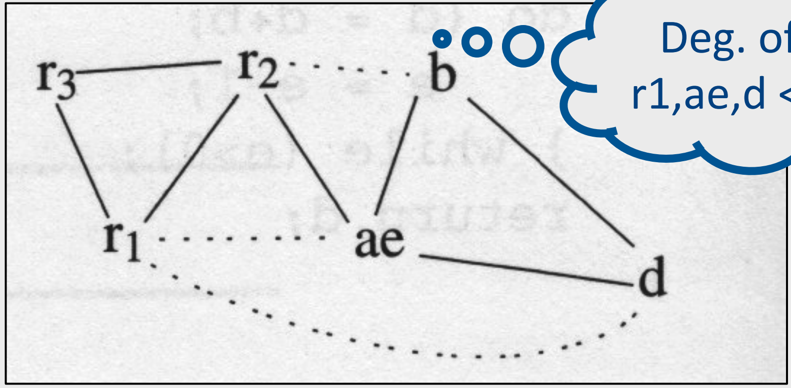
Spill c

c



a & e

c

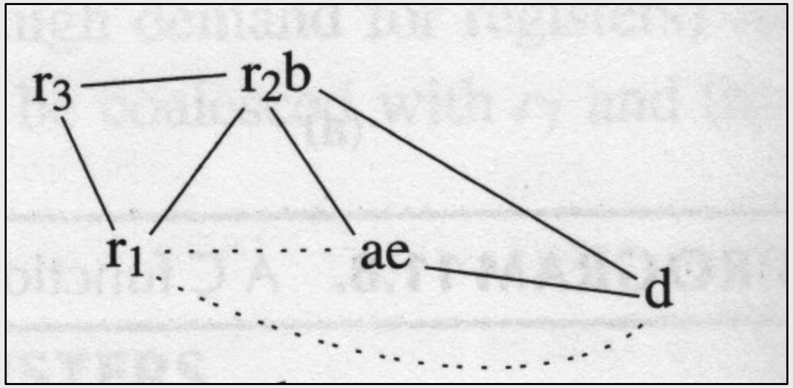


Deg. of r1,ae,d < K

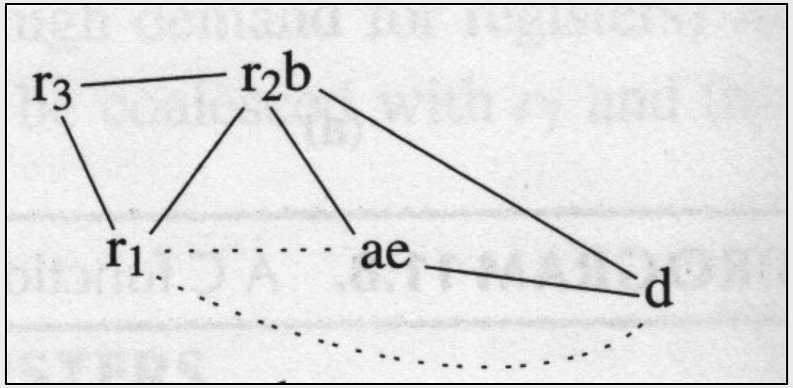
r2 & b

(Alt: ae+r1)

c

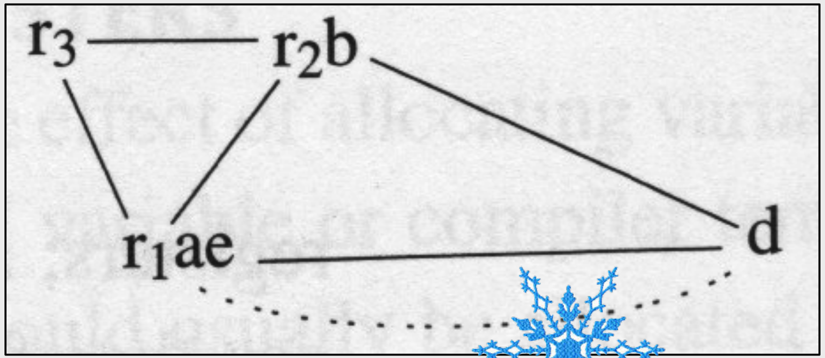


# A Complete Example



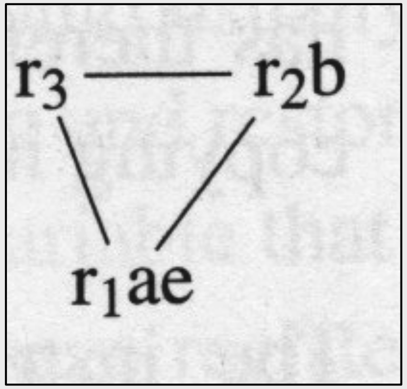
ae & r1

(Alt: …)$_c$



freeze $r_1$ae-d

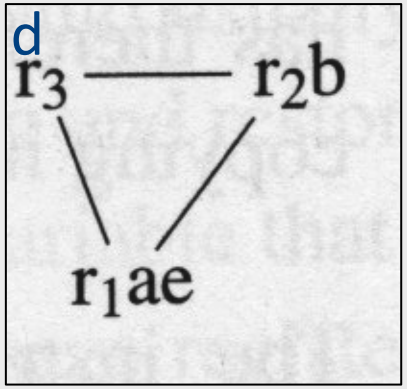
Simplify d



dc

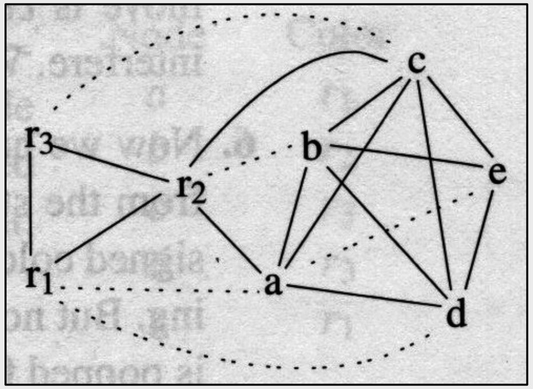
d

pop d



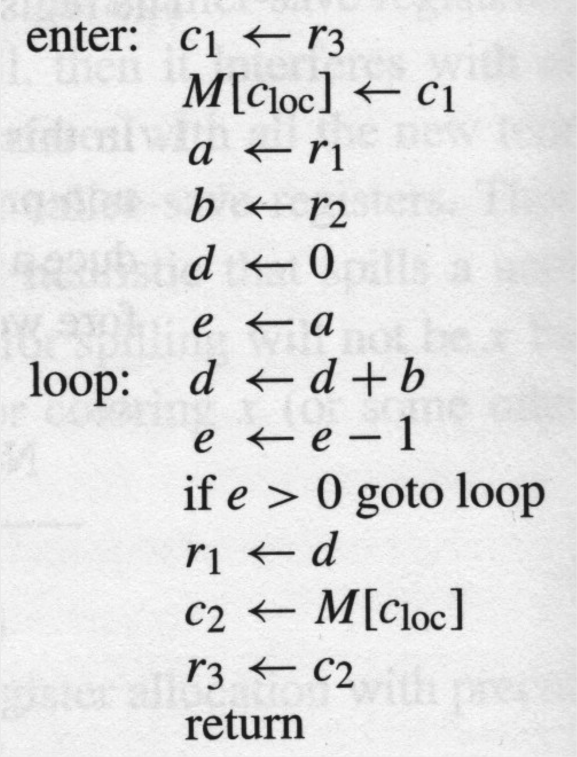
c

pop c …

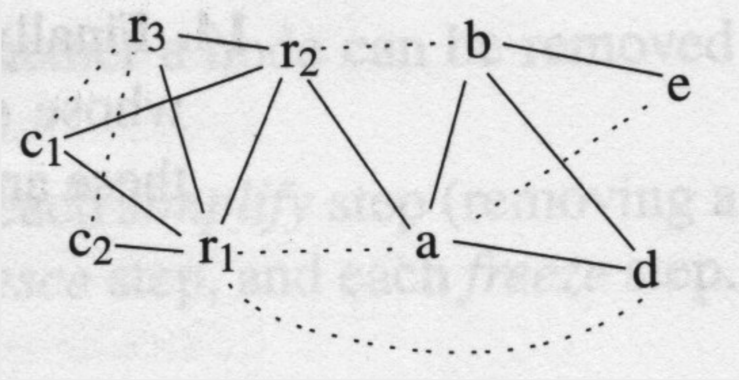


(Alt: ae+r1)

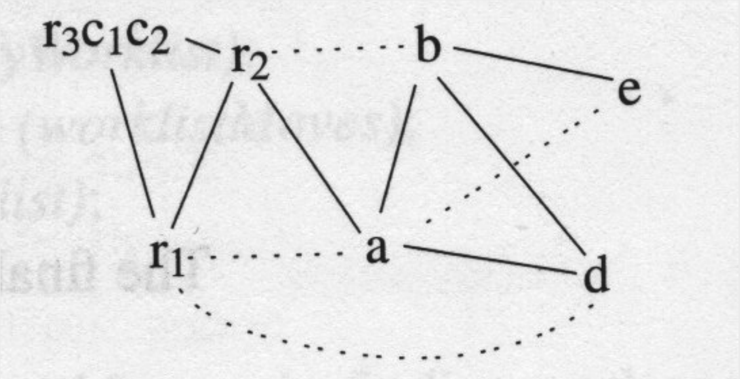
# A Complete Example



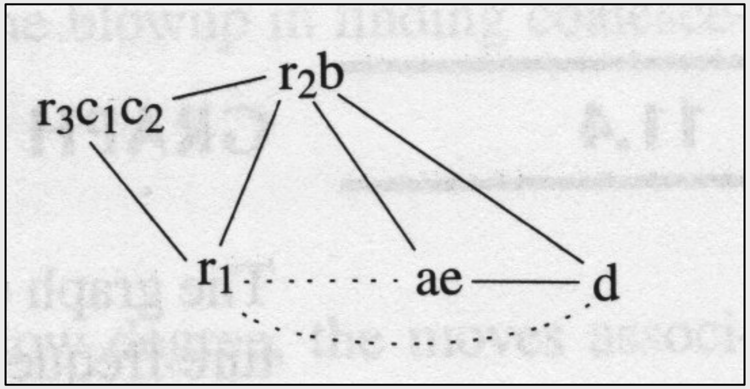
```
enter:  $c_1 \leftarrow r_3$
        $M[c_{loc}] \leftarrow c_1$
        $a \leftarrow r_1$
        $b \leftarrow r_2$
        $d \leftarrow 0$
        $e \leftarrow a$
loop:   $d \leftarrow d + b$
        $e \leftarrow e - 1$
        if $e > 0$ goto loop
        $r_1 \leftarrow d$
        $c_2 \leftarrow M[c_{loc}]$
        $r_3 \leftarrow c_2$
        return
```



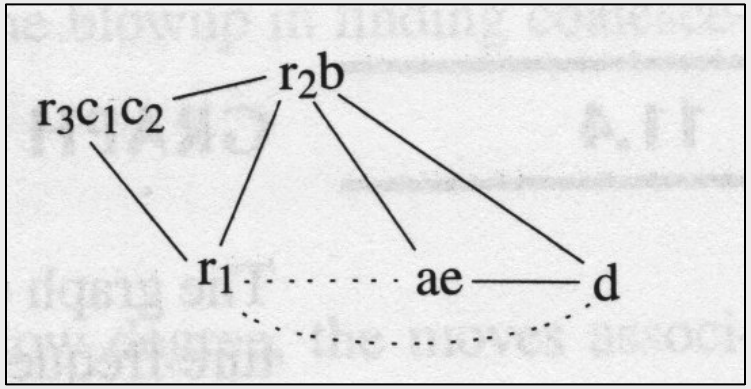
c1&r3, c2 &r3



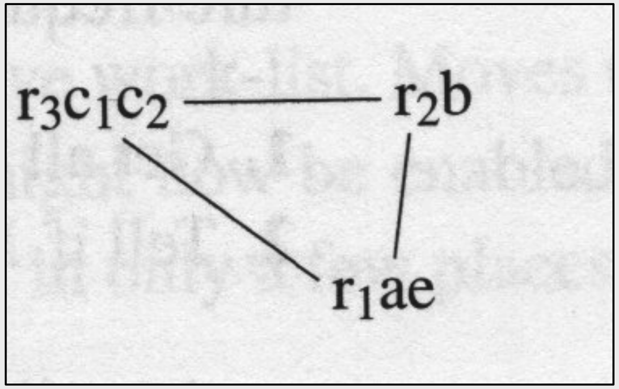
a&e, b&r2



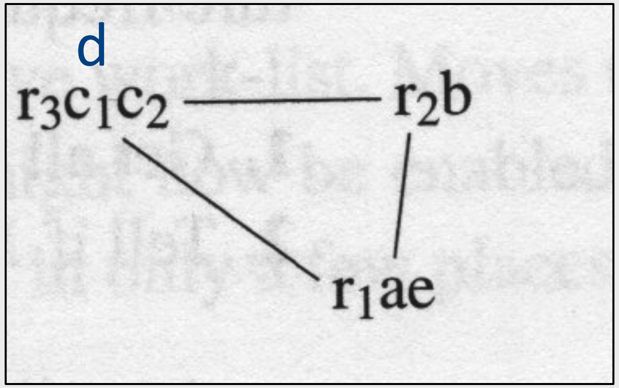
# A Complete Example



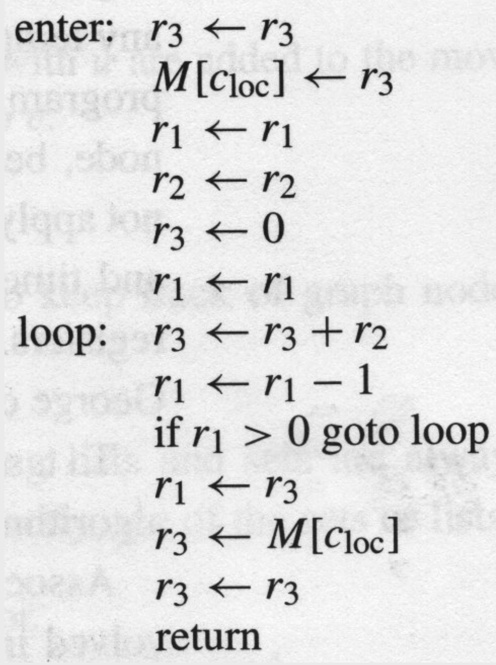
ae & r1
Simplify d



Pop d

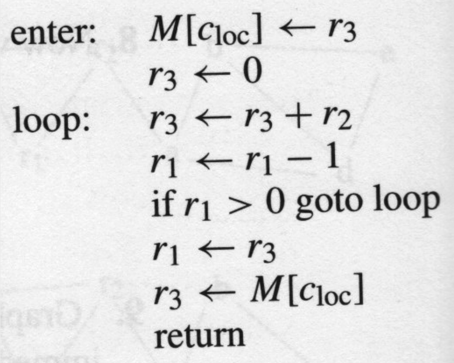


gen code



```
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