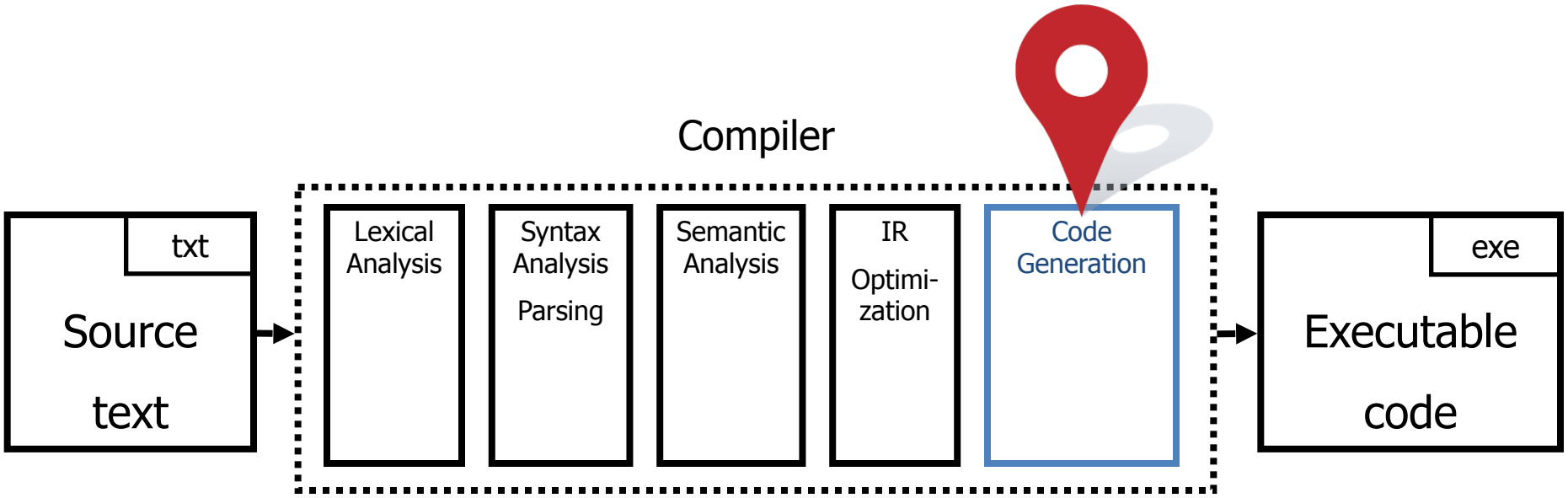
Theory of Compilation

Lecture 06

You are here


Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code Generation
exe
Executable code

LLVM IR

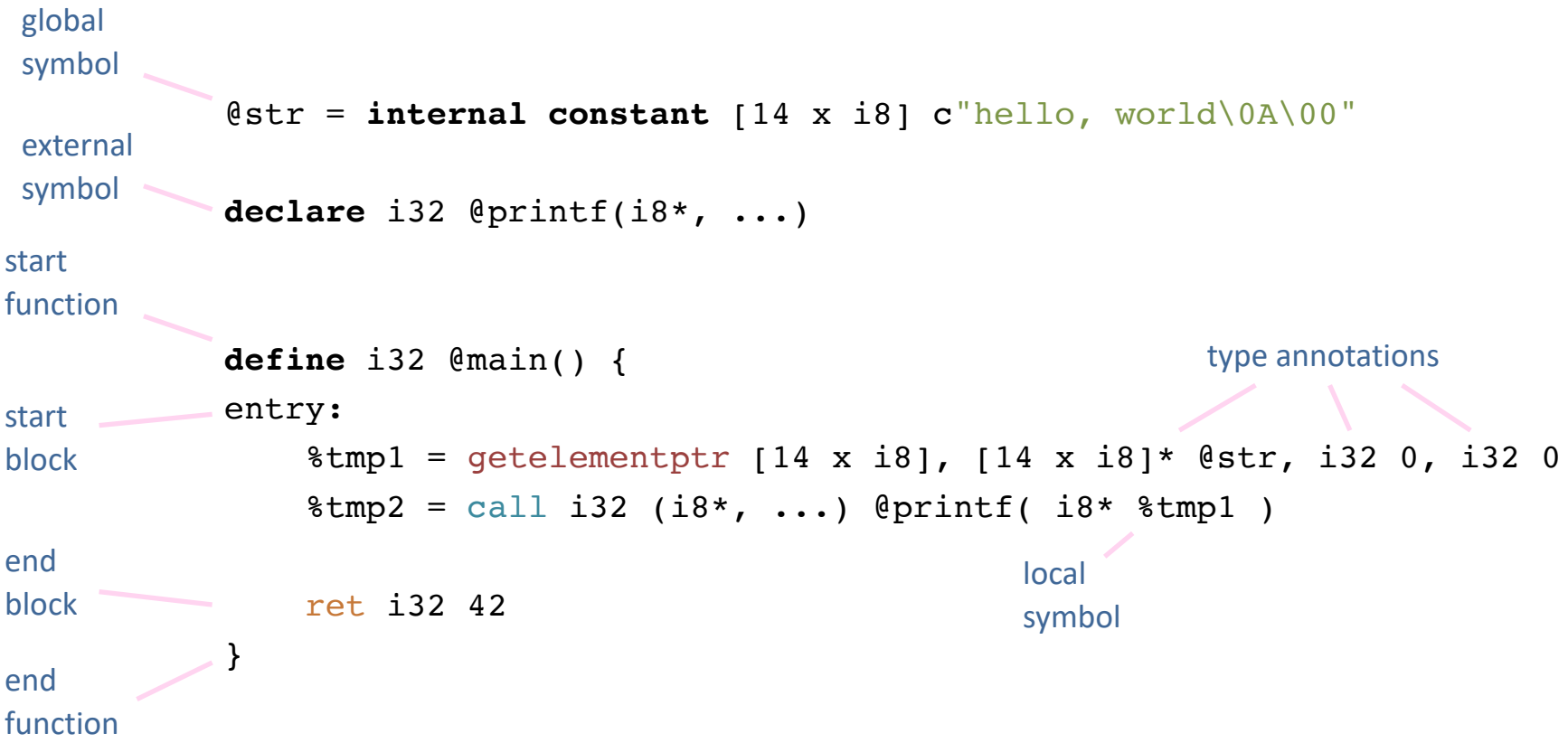
- LLVM = Low-Level Virtual Machine
  - A well-known misnomer: it is (mostly) a **compiler** framework
- Flat IR, similar to 3-address code in nature
- All values are ***typed*** (unlike assembly)

```
%num = add i32 %inp, 48
```

target opcode type operands

https://llvm.org/docs/LangRef.html

LLVM IR — Hello World




```
@str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)


define i32 @main() {
entry:
    %tmp1 = getelementptr [14 x i8], [14 x i8]* @str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 )

    ret i32 42
}
```

LLVM IR — Types

- Numeric
  - Integers — any bit width!
  - Floating point: 16, 32, 64 bit
- Pointer
- Label (= code address)
- Aggregate
  - Array (fixed dimensions)
  - Struct

```
i1
i32
i1942652
half, float, double
i8 *
label
[40 x i32]
[12 x [10 x float]]
{ float, [ 4 x i32 ] }
```

https://llvm.org/docs/LangRef.html

LLVM IR — Memory

```
%a = alloca i32
store i32 5, i32* %a
%rd = load i32, i32* %a
```

allocate on stack
return type is a pointer

write to memory address *a = 5

read from memory address rd = *a

```
%a = alloca [4 x i32]
%el = getelementptr [4 x i32],
          [4 x i32]* %a, i32 0, i32 1
store i32 5, i32* %el
%rd = load i32, i32* %el
```

aggregate type

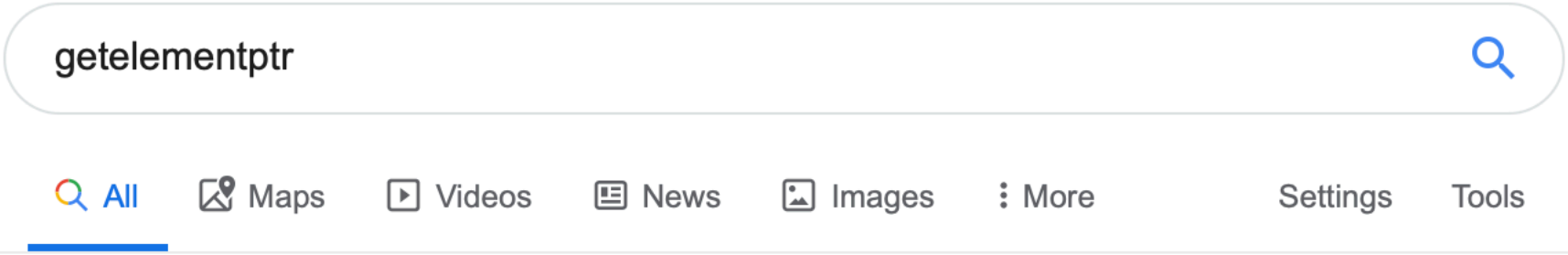
base pointer

idx_1

idx_2

compute address of element in aggregate
return type is a pointer

LLVM IR — Memory



LLVM — Gotchas

- LLVM blocks must start with a label and end with a *Terminator instruction*

`ret` `br` `switch` `indirectbr` `invoke` `callbr` `resume`
`catchswitch` `catchret` `cleanupret` `unreachable`

- LLVM variables are *Single Static Assignment* (SSA)
  - Wait, what?

Single Static Assignment

- Variables can only appear on the left-hand side of ***one*** assignment.

```
cond:
   %b = icmp slt i32 %i, %j
   br i1 %b, label %then,
             label %else
then:
   %max = or i32 0, %j
   br label %exit

else:
   %max = or i32 0, %i
   br label %exit

exit:
   ret i32 %max
```

✗

```
cond:
   %b = icmp slt i32 %i, %j
   br i1 %b, label %then,
             label %else
then:
   %max1 = or i32 0, %j
   br label %exit

else:
   %max2 = or i32 0, %i
   br label %exit

exit:
   %max = phi i32 [ %max1, %then ],
                  [ %max2, %else ]
   ret i32 %max
```

✓

Single Static Assignment

- Memory store operations are **not** subject to single static assignment rules.

Possible compilation strategy:

Allocate memory for local variables as an array

```
%a = alloca [4 x i32]
%lvar =
  getelementptr [4 x i32],
   [4 x i32]* %a, i32 0, i32 1
```

variable offset

```
cond:
   %b = icmp slt i32 %i, %j
   br i1 %b, label %then,
             label %else
then:
   store i32 %j, i32* %lvar
   br label %exit

else:
   store i32 %j, i32* %lvar
   br label %exit

exit:
   %max = load i32, i32* %lvar
   ret i32 %max
```

variable address

Allocating Memory

```
int count;  float money;  int[42] balances;
```

- Type checking helped us guarantee correctness
- Now we want to know:
  - How much memory to allocate on the heap/stack for variables
  - Where to find variables (based on offsets)
  - How to compute address of an element inside array (size of stride based on type of element)

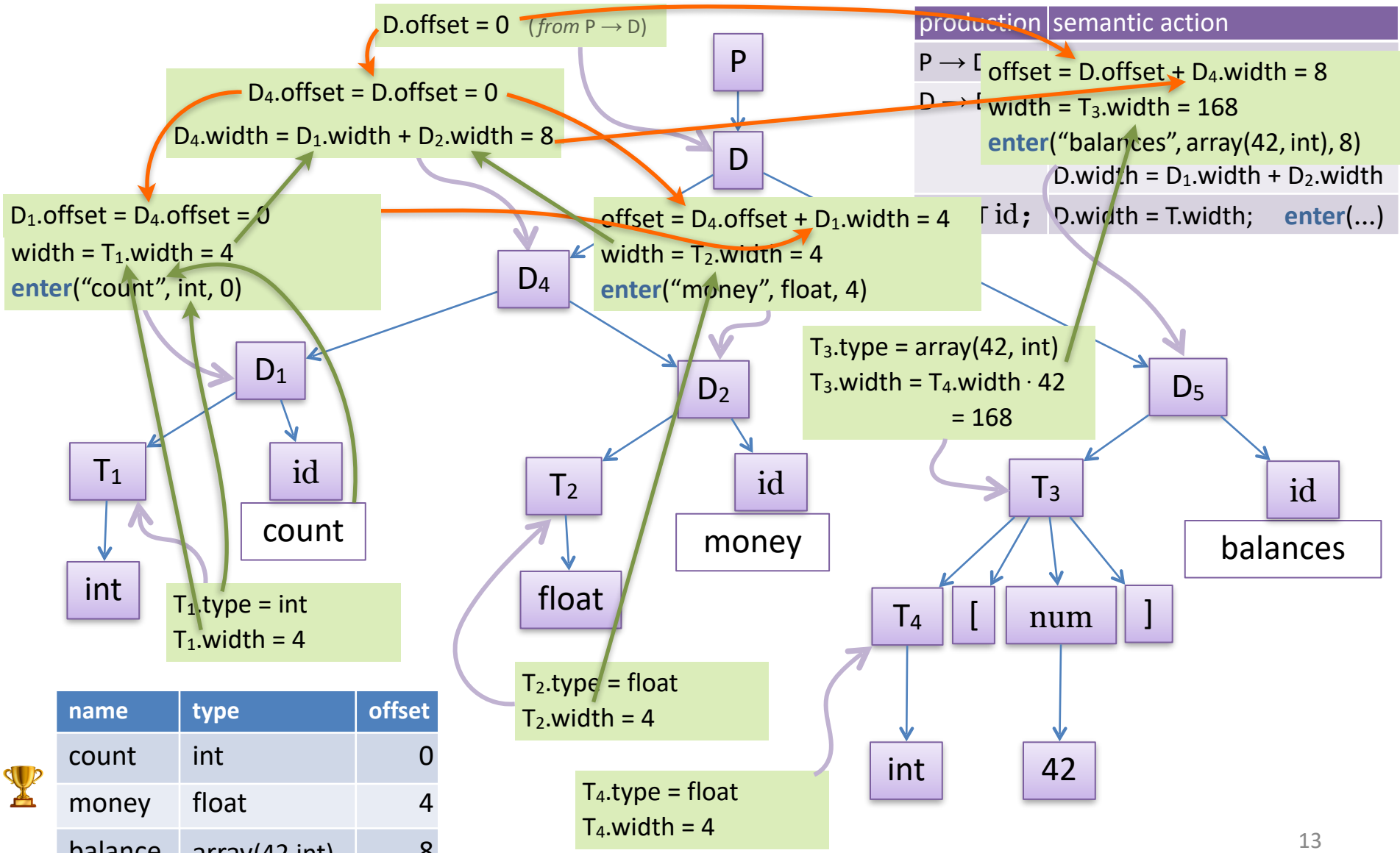
Allocating Memory

- Attribute "offset" with amount of memory allocated *before* each declaration

production	semantic action
$P \rightarrow D$	D.offset = 0
$D \rightarrow D_1\ D_2$	D_1.offset = D.offset; D_2.offset = D.offset + D_1.width; D.width = D_1.width + D_2.width
$D \rightarrow T\ \mathrm{id};$	D.width = T.width; **enter**(id.name, T.type, D.offset)
$T \rightarrow \mathtt{int}$	T.type = int; T.width = 4
$T \rightarrow \mathtt{long}$	T.type = long; T.width = 8
$T \rightarrow \mathtt{float}$	T.type = float; T.width = 4
$T \rightarrow T_1[\mathrm{num}]$	T.type = array(num.val, T_1.Type); T.width = num.val · T_1.width
$T \rightarrow *T_1$	T.type = pointer(T_1.type); T.width = 4

insert into symbol table with given type and offset

Allocating Memory



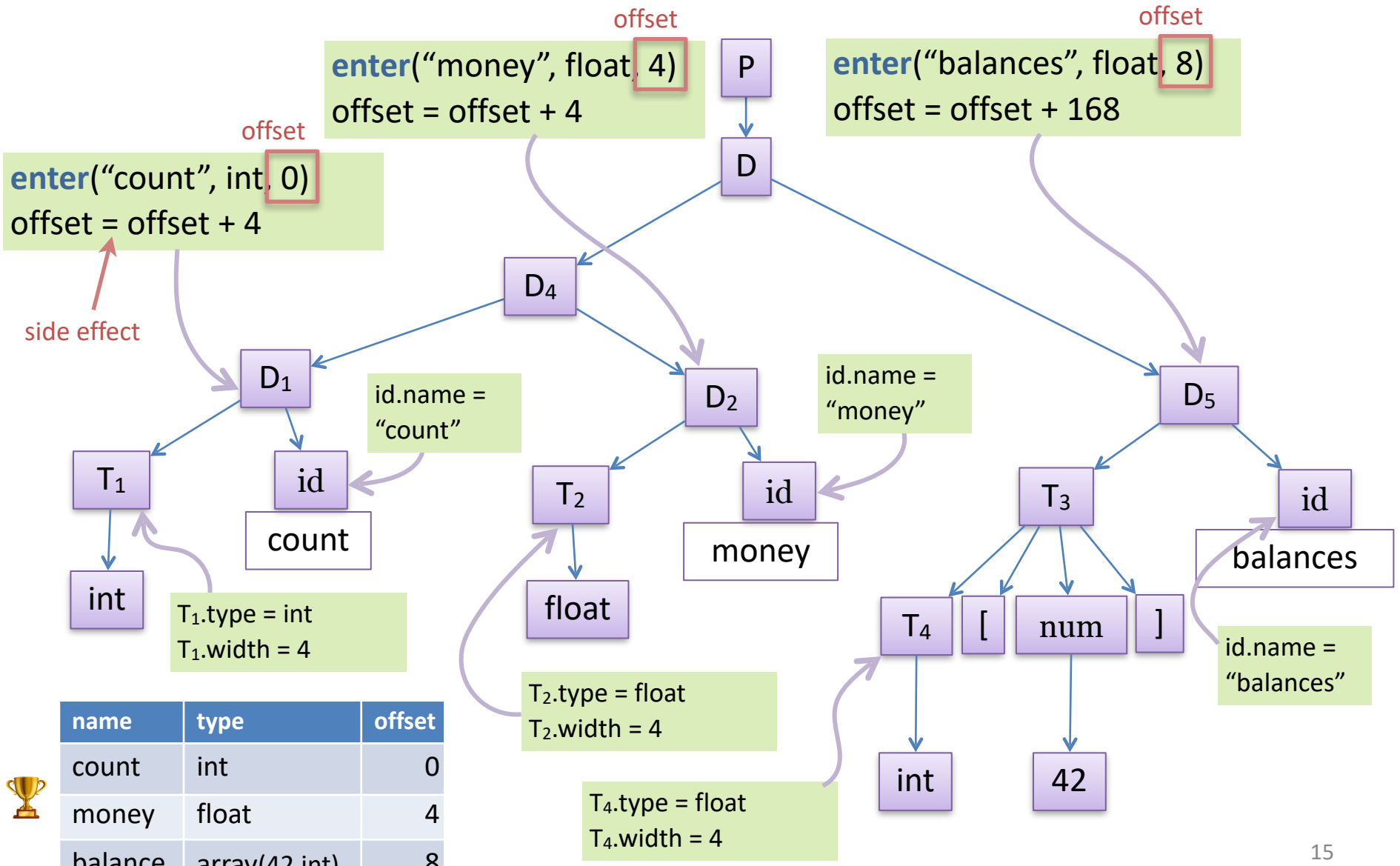
name	type	offset
count	int	0
money	float	4
balance	array(42,int)	8

A Less "Pure" But Simpler Way

- Global variable "offset" with amount of memory allocated so far

production	semantic action
P → **M** D	
M → ε	**offset = 0**
D → D D	
D → T id ;	**enter**(id.name, T.type, **offset**); **offset +=** T.width
T → `int`	T.type = int; T.width = 4
T → `long`	T.type = long; T.width = 8
T → `float`	T.type = float; T.width = 4
T → T_1[num]	T.type = array(num.val, T_1.Type); T.width = num.val · T_1.width
T → *T_1	T.type = pointer(T_1.type); T.width = 4

A Less "Pure" But Simpler Way



name	type	offset
count	int	0
money	float	4
balance	array(42,int)	8

From IR to ASM: Challenges

- Register allocation
- Mapping IR to ASM operations
  - what instruction(s) should be used to implement an IR operation?
  - how do we translate code sequences?
- Dynamic memory allocation
- Call/return of routines
  - managing activation records
- Architecture-specific optimizations

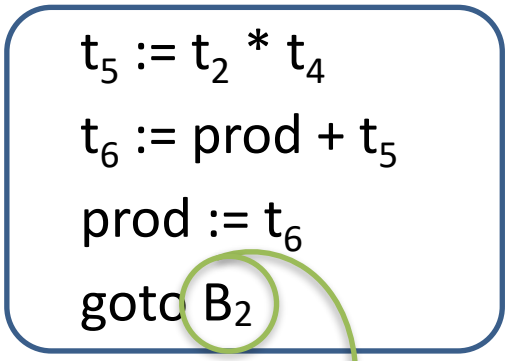
this lecture

≈ week 10

way too complicated

Basic Blocks

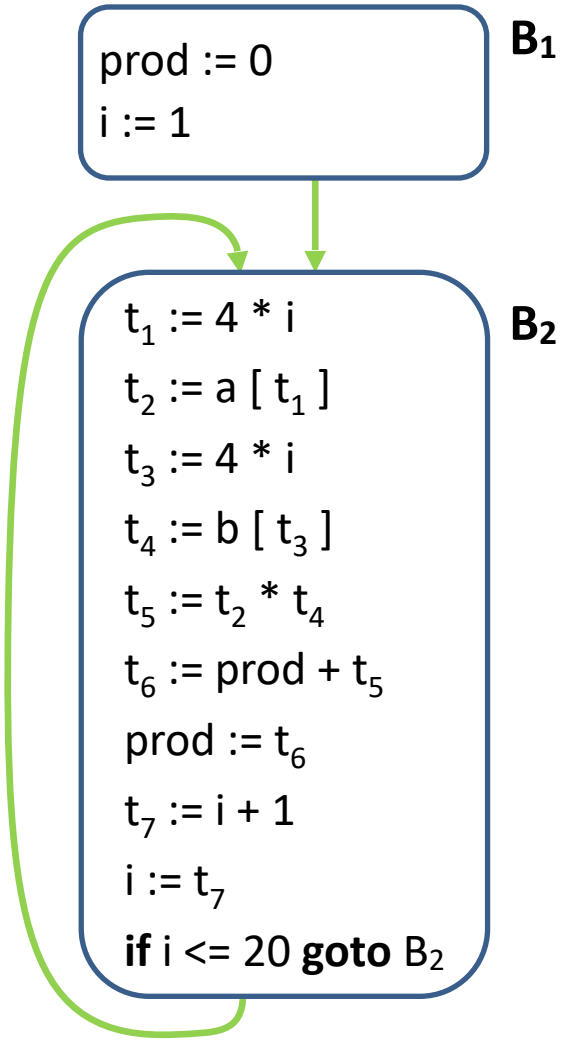
- A **basic block** is a sequence of instructions with
  - a **single entry** (to first instruction), no jumps to the middle of the block
  - a **single exit** (last instruction) code executes as a sequence from first instruction to last instruction without any jumps



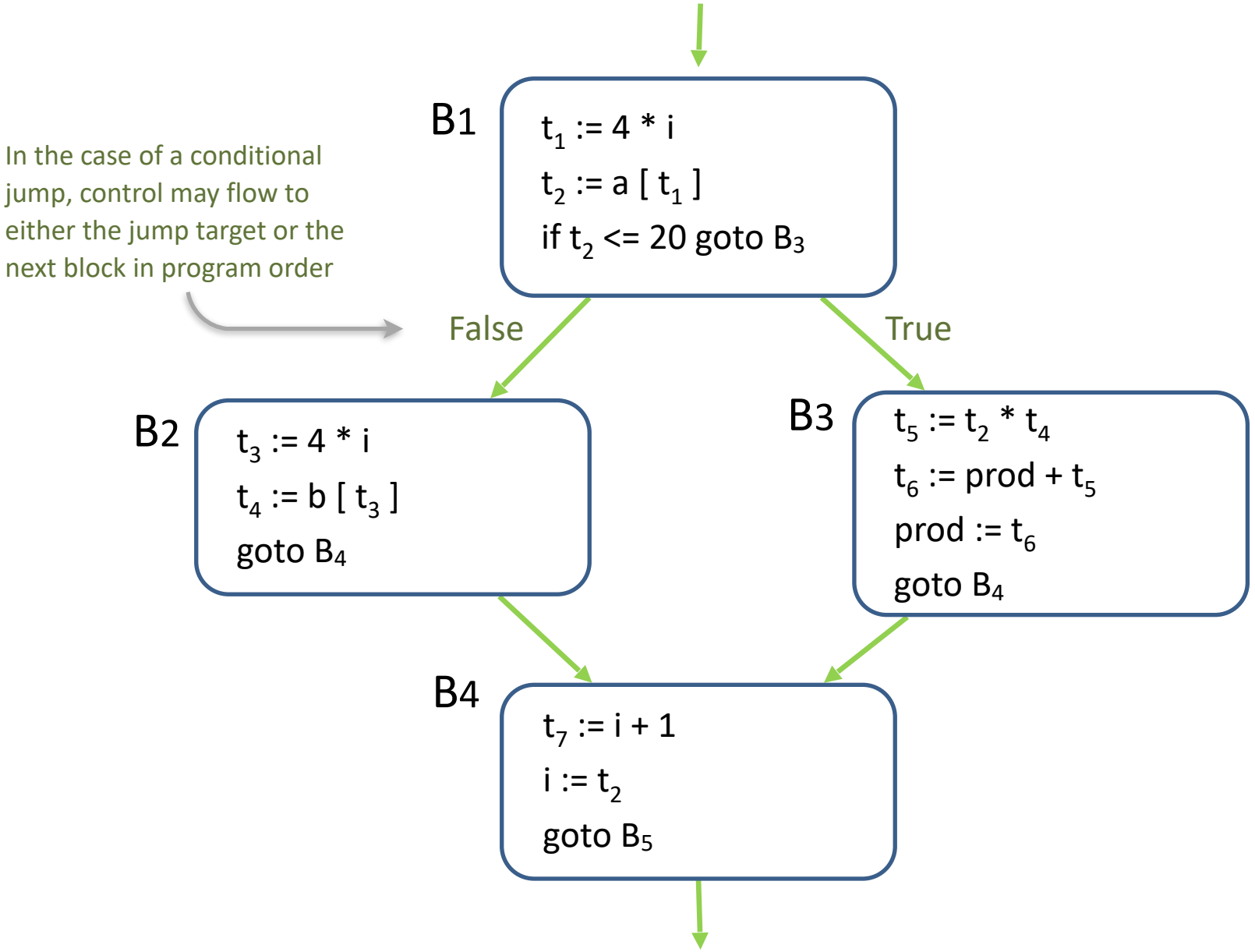
- The last instruction of B_1 may be a (conditional or unconditional) jump to another block B_2

Control Flow Graph: Definition

- A directed graph G = ⟨V, E⟩
  - V = basic blocks
  - E = control flow
    - $(B_1,B_2) \in E$ when control from B_1 *can flow* to B_2



Control Flow Graph: Example



Splitting 3AC Into Basic Blocks

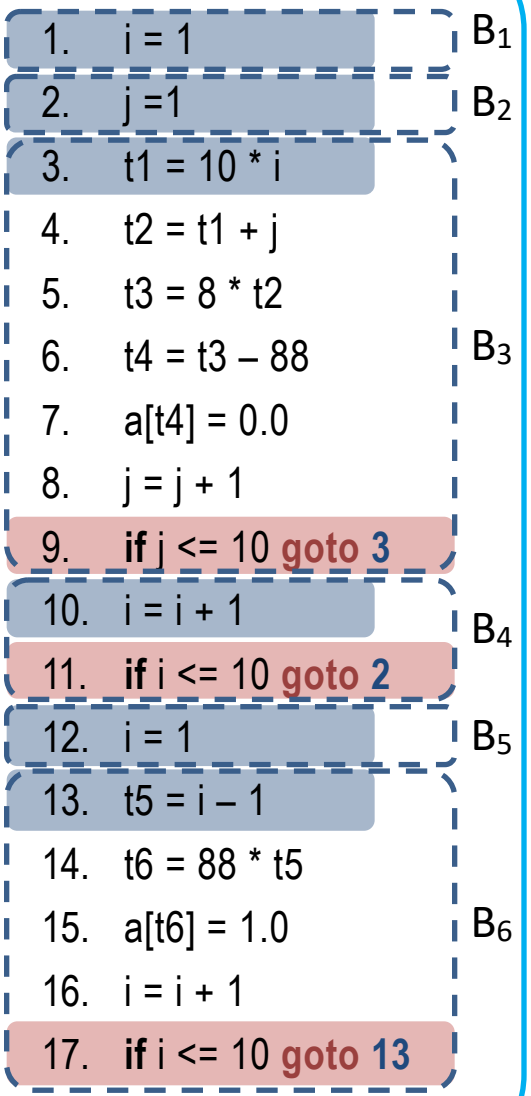
- **Input**: A sequence of 3AC instructions
- **Output**: A list of basic blocks where each instruction is contained in exactly one block
- **Method**
  - Determine the set of **leaders** (first instruction of a block)
    - The *first* instruction is a leader
    - Any instruction that is the *target* of a **jump** is a leader
    - Any instruction that immediately *follows* a **jump** is a leader
  - Each basic block consists of a **leader** and all following instructions up to but not including the next **leader** or the end of the program

Example

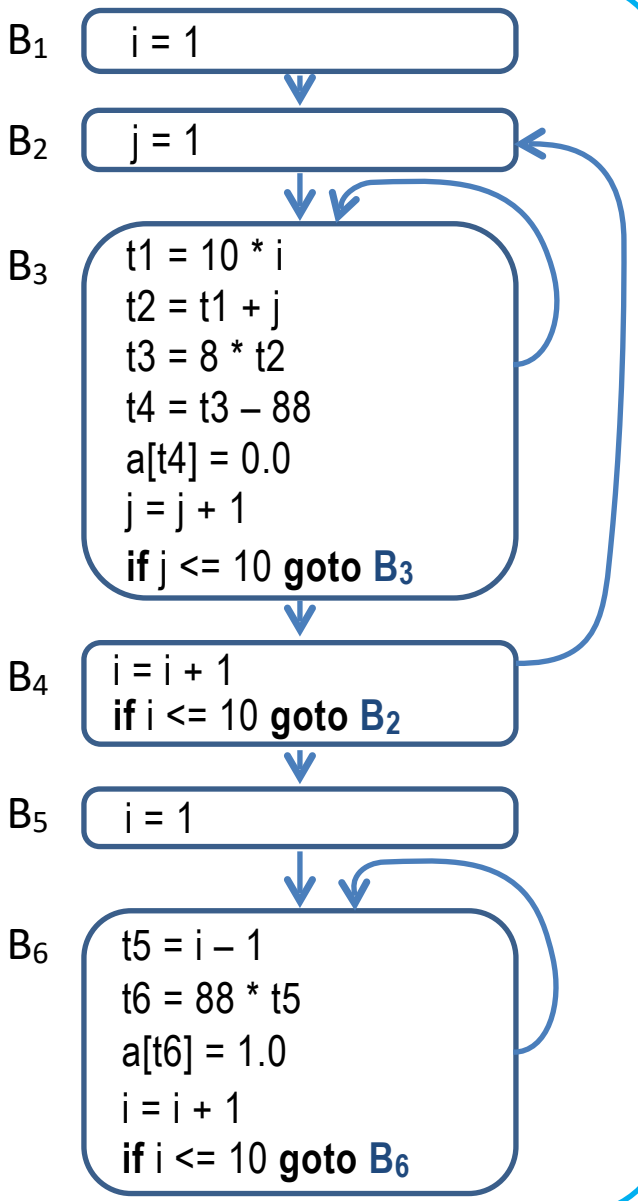
source

```
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i, j] = 0.0;
for i from 1 to 10 do
  a[i, i] = 1.0;
```

IR



CFG



Simple Code Generation

- Assume machine instructions of the form
  - `LD reg, mem`
  - `ST mem, reg`
  - `OP reg, reg, reg`

    e.g. ADD, MUL, XOR
- Registers
  - must used for operands of instructions
  - can be used to store temporary results
  - number of registers is ***limited***

Simple Code Generation

- Number of registers is ***limited***

  ⇒ Need to allocate them **judiciously**

- Straw-man Solution
  - Read from memory before use, write after update

```
x := y ◇ z
```

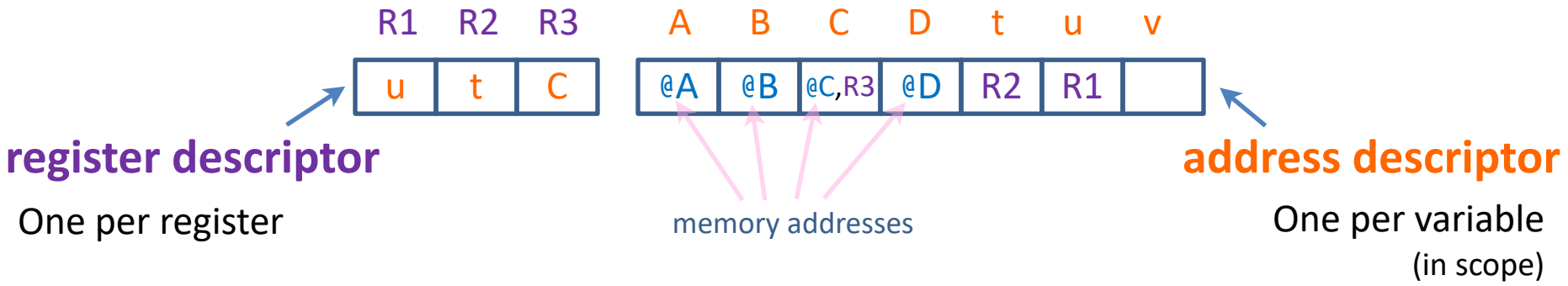
```
LD R1, @y
LD R2, @z
OP R3, R1, R2
ST @x, R3
```

memory addresses

Remember: memory access is **orders of magnitude** slower than register operations!

Simple Code Generation

Descriptor Table



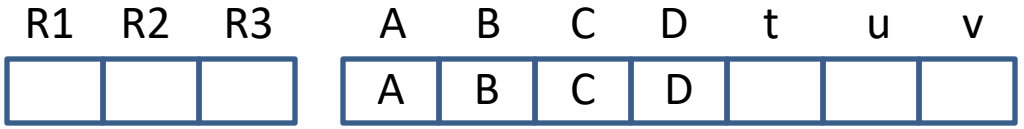
- A **register descriptor** records variable name(s) whose current value is in that register
  - a register may be uninitialized or hold the value of one or more variables
- An **address descriptor** records the location(s) at which the current value of the variable can be found
  - The location may be a register, a memory address, or some set of these

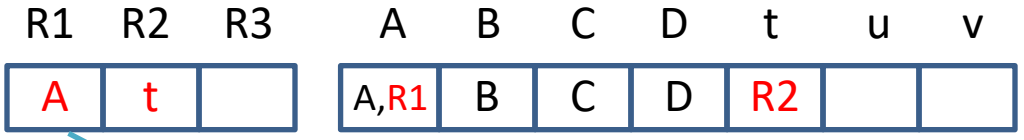
Example

t := A – B
u := A – C
v := t + u
A := D
D := v + u

t := A – B

```
LD R1, @A
LD R2, @B
SUB R2, R1, R2
```



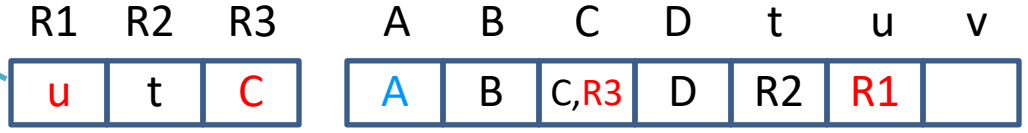


A is already in a register. Nice.

u := A – C

```
LD R3, @C
SUB R1, R1, R3
```

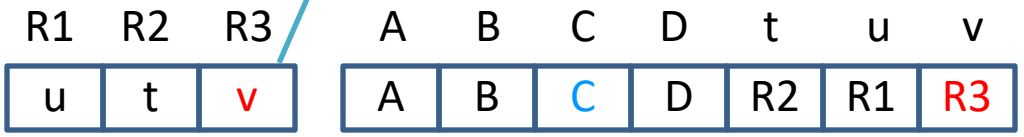
Need a register for u.
But all are taken…
Must *evict* one of the variables



v := t + u

```
ADD R3, R2, R1
```

Same here for v.

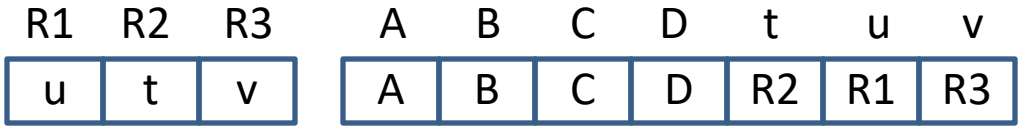


A B C D = variables in function scope
t u v = temporaries in block

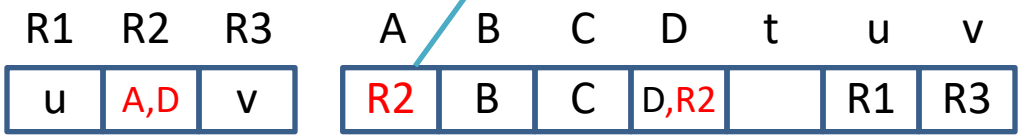
Example

```
t := A – B
u := A – C
v := t + u
A := D
D := v + u
```

```
A := D
  LD R2, @D
```



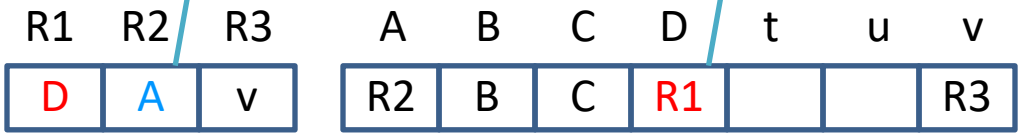
A is **no longer** in memory!




```
D := v + u
  ADD R1, R3, R1
```

D is **no longer** in R2

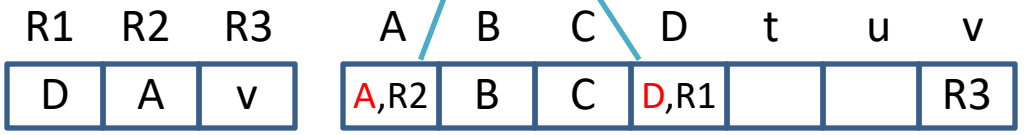
D is **no longer** in memory



block exit

```
  ST @A, R2
  ST @D, R1
```

write back to memory

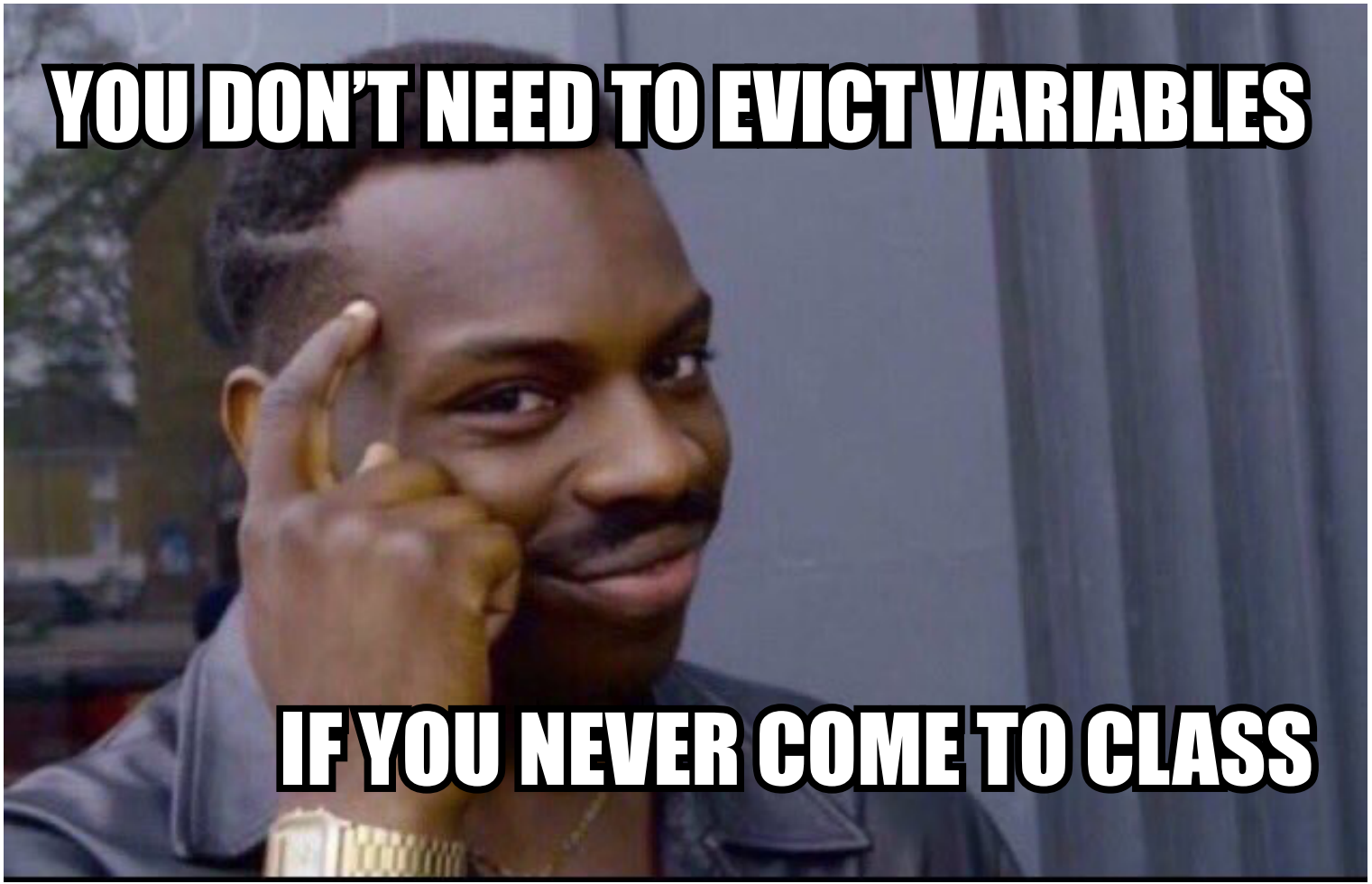


A B C D = variables in function scope

t u v = temporaries in block

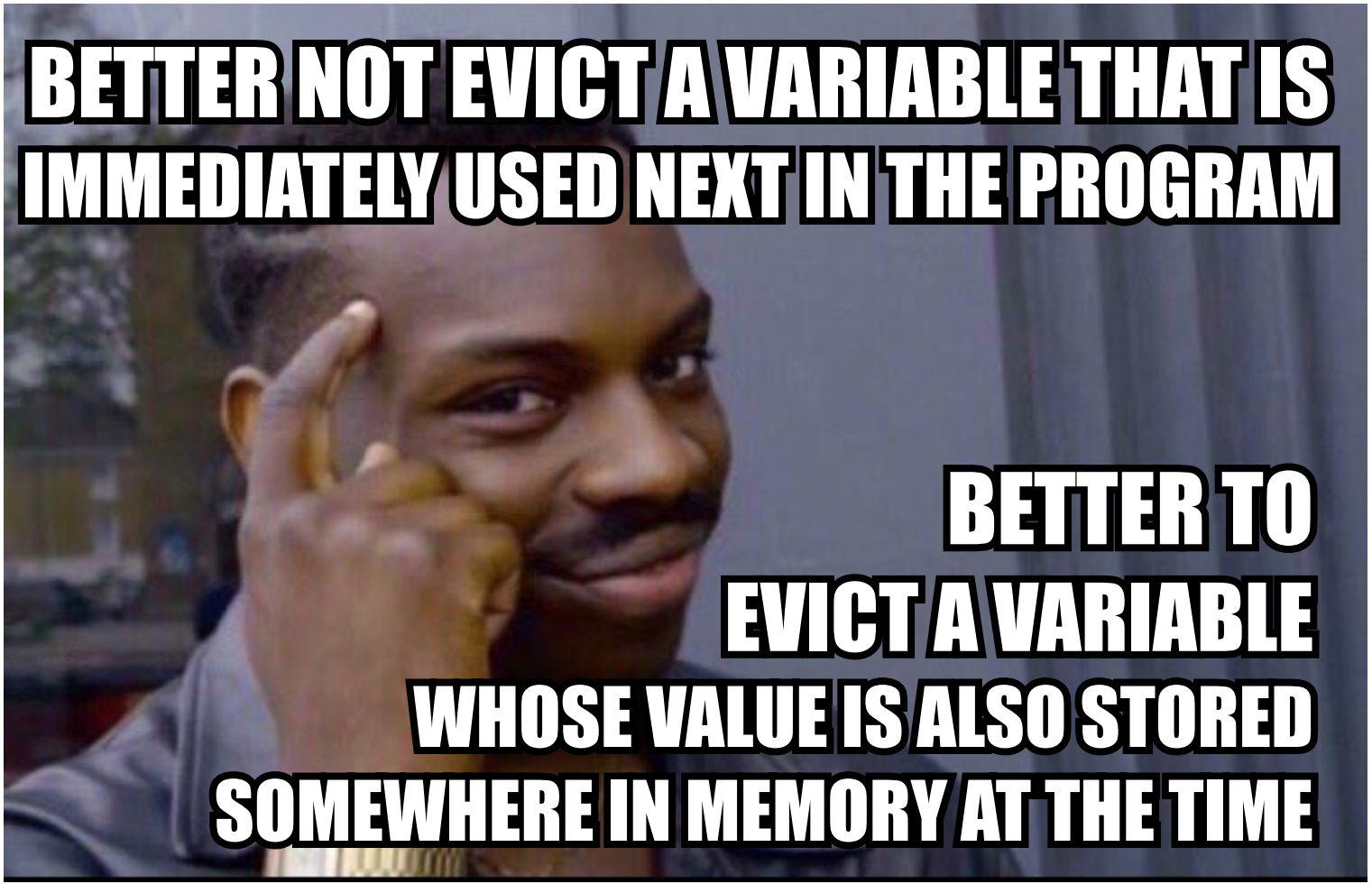
But… How Do We Know..?

- How do we choose which variable to evict..?



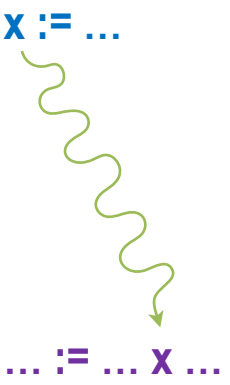
But… How Do We Know..?

- How do we choose which variable to evict..?

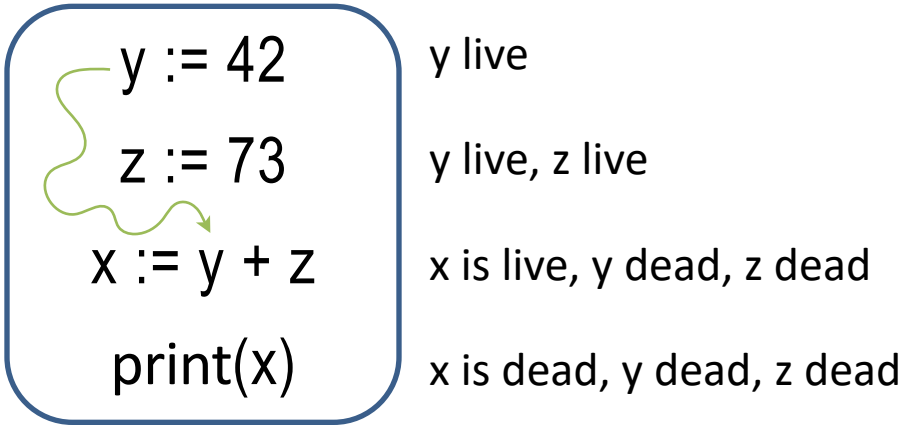


Variable Liveness

- A statement x := y + z :
  - **defines** x
  - **uses** y and z



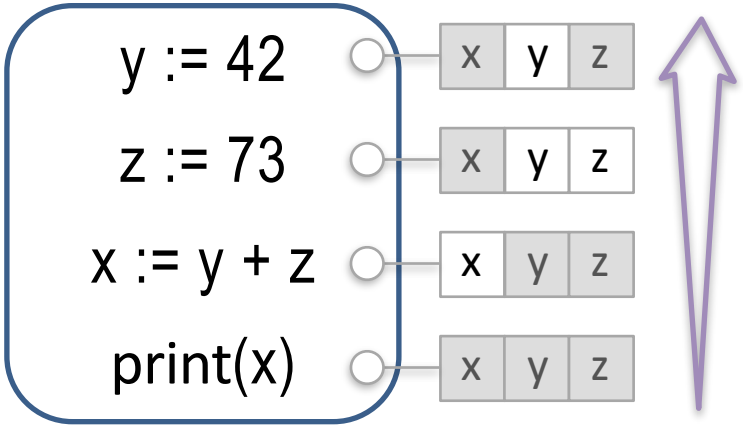
- A variable x is live at a program point if its value is used at a later point, prior to any (re-)definition of x



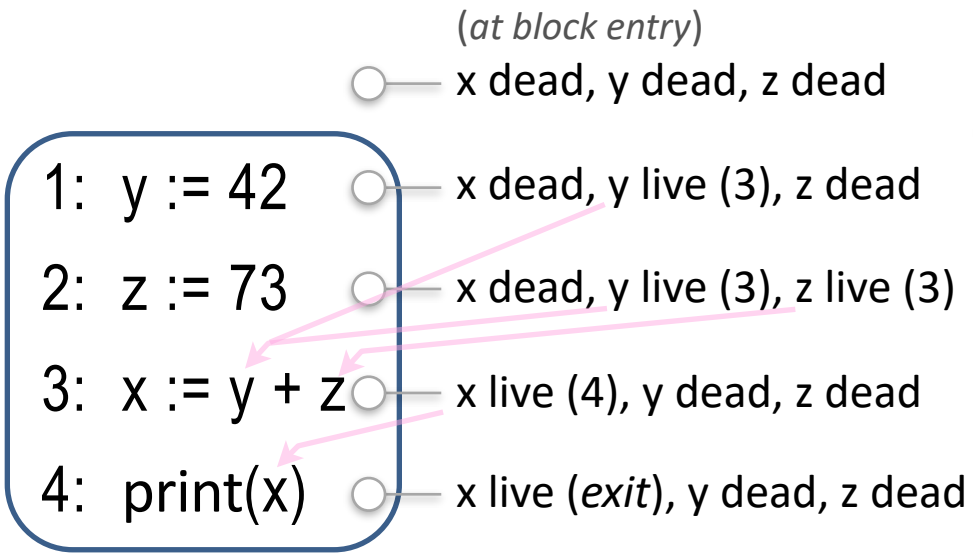
(showing state *after* the statement)

Computing Liveness Information

- Within a single basic block?
- Idea
  - Record next-use information for every variable
  - Scan basic block instructions backwards
  - When variable is read, update its next-use
  - When a variable is assigned to, clear its next-use



Computing Liveness Information



assume: x is a program variable,
y, z temporaries

Computing Liveness Information

```
x := 1
y := x + 3
z := x * 3
x := x * z
```

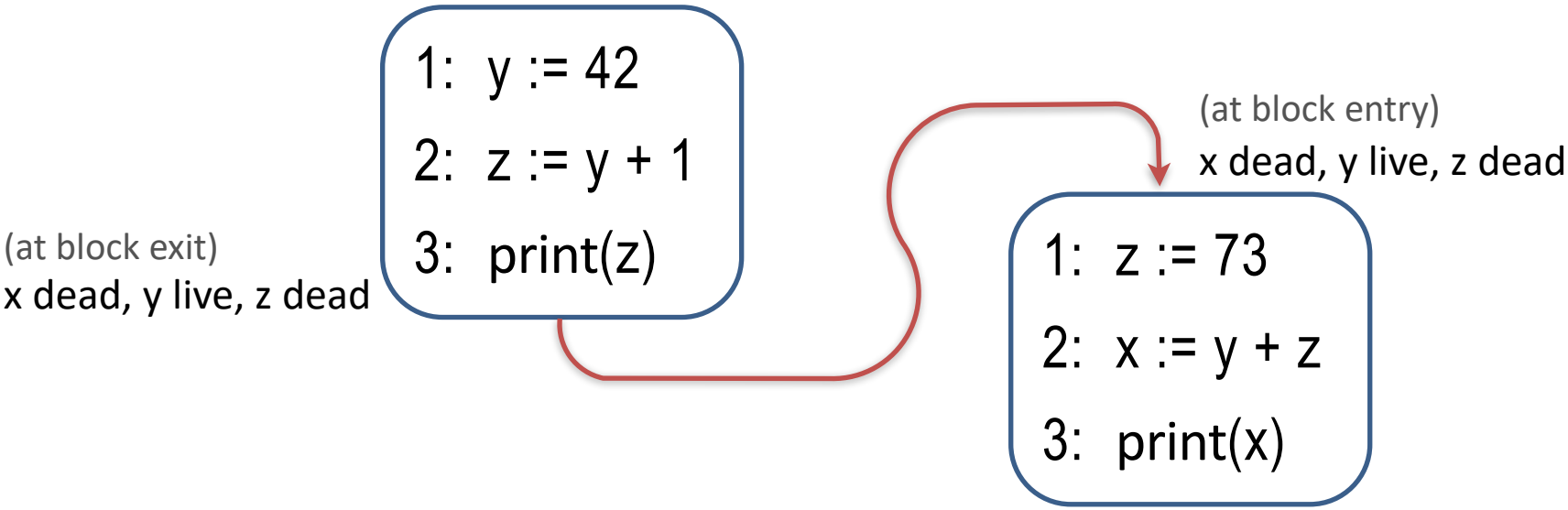
- INPUT: A basic block B of 3AC statements.
  assume all non-temporary variables in B are live on exit.
- Start at the last statement in B and scan backwards
  - At each statement i: x = y + z in B, we do the following:
    1. Attach to i the current liveness and next-use information for x, y, and z.
    2. In the current information record:
       set x to "not live";
       set y and z to "live" and their next use to i.

Is the order between these steps important?

- OUTPUT: For each statement i: x := y + z in B, liveness and next-use information of x, y, and z at (*just after*) i.

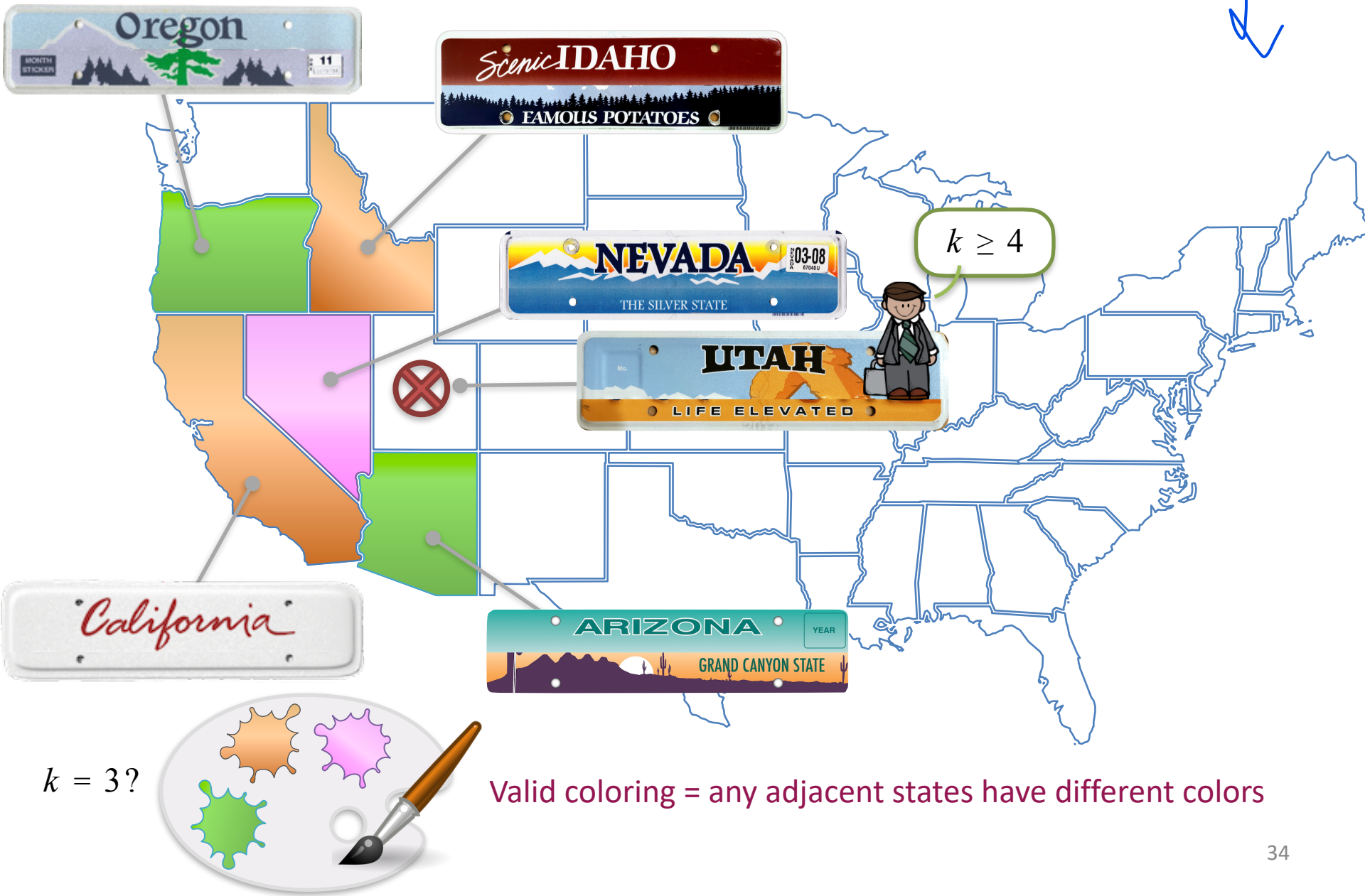
Computing Liveness Information

- Between basic blocks?
  - Continue backward along CFG edges



- Branches?
  - Consider all execution paths
- Loops?
  - Data-flow analysis (next lecture)

Interlude



$k = 3?$

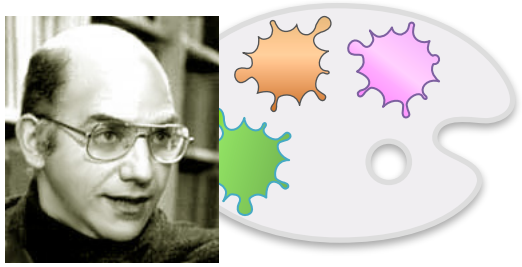
Valid coloring = any adjacent states have different colors

Register Allocation by Graph Coloring

Chaitin & Briggs

(1981) (1992)

- Address register allocation by
  - Liveness analysis
  - Reduction to graph coloring

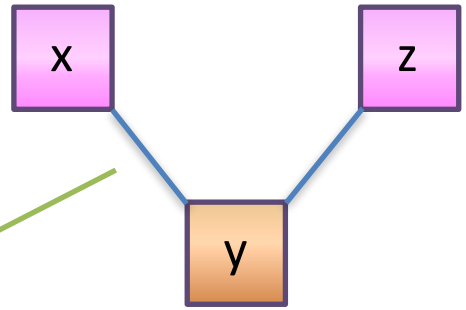


- Main idea
  - Register allocation = coloring of an **interference graph**

vertex = variable
(incl. temporaries)

edge between variables
that "interfere"

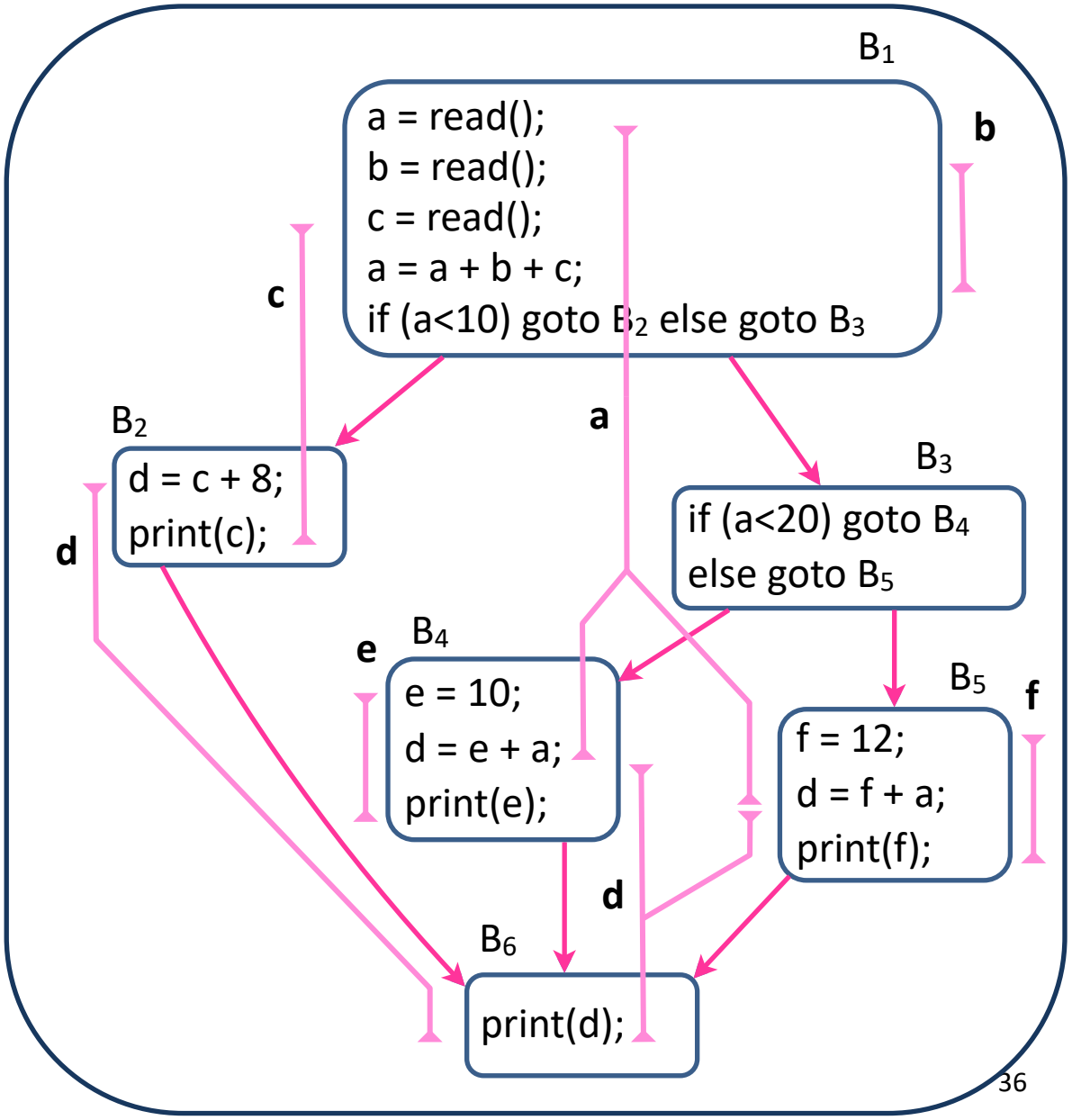
= alive at the same time



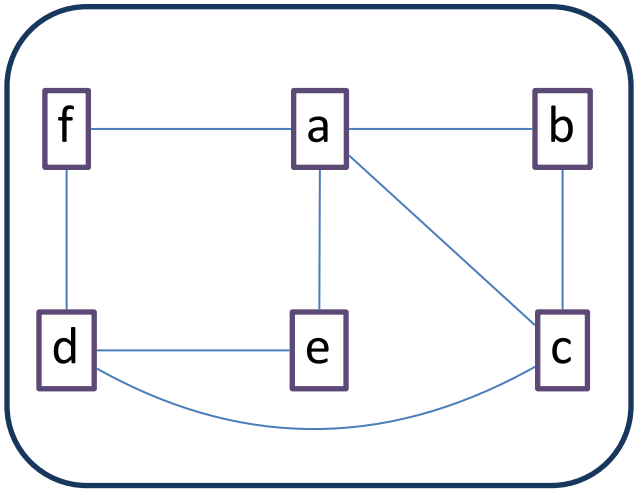
number of colors
=
number of registers

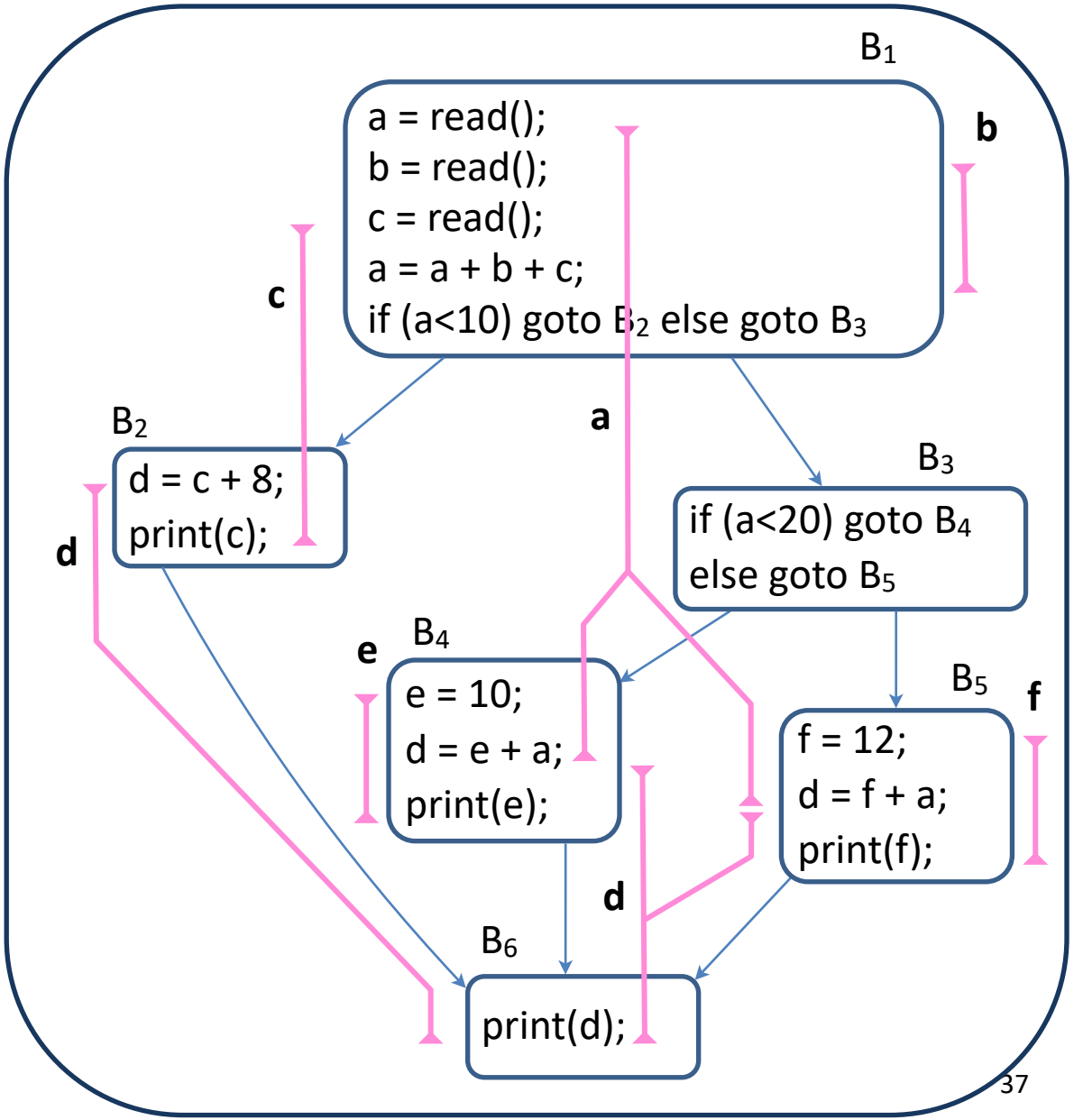
Example: Interference Graph

```
a = read();
b = read();
c = read();
a = a + b + c;
if (a<10) {
    d = c + 8;
    print(c);
} else if (a<20) {
   e = 10;
   d = e + a;
   print(e);
} else {
    f = 12;
    d = f + a;
    print(f);
}
print(d);
```



Example


f
a
b
d
e
c


B1
a = read();
b = read();
c = read();
a = a + b + c;
if (a<10) goto B2 else goto B3
b
c
a
B2
d = c + 8;
print(c);
d
B3
if (a<20) goto B4
else goto B5
e
B4
e = 10;
d = e + a;
print(e);
B5
f = 12;
d = f + a;
print(f);
f
d
B6
print(d);

Register Allocation by Graph Coloring

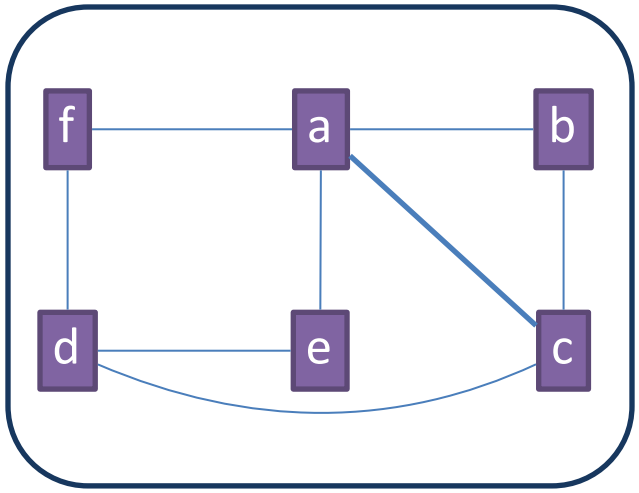
- **Variables that interfere with each other cannot be allocated the same register**

- Graph coloring
  - Classic problem: how to color the nodes of a graph with the lowest possible number of colors such that no two adjacent nodes have the same color
  - Bad news: problem is NP-complete
  - Good news: there are pretty good heuristic approaches

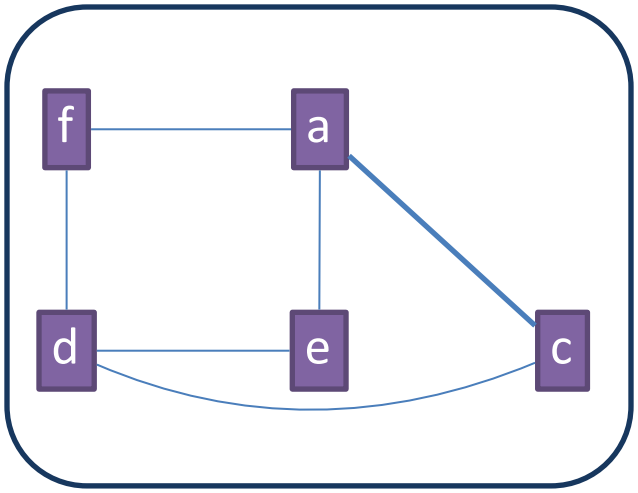
Heuristic Graph Coloring

- Idea: color nodes one by one, coloring the "easiest" node last
- "Easiest nodes" are ones that have lowest degree
  - fewer conflicts
- Algorithm at high-level
  - find a lowest-degree node
  - remove lowest-degree node from the graph
  - color the reduced graph **recursively**
  - re-attach the that node

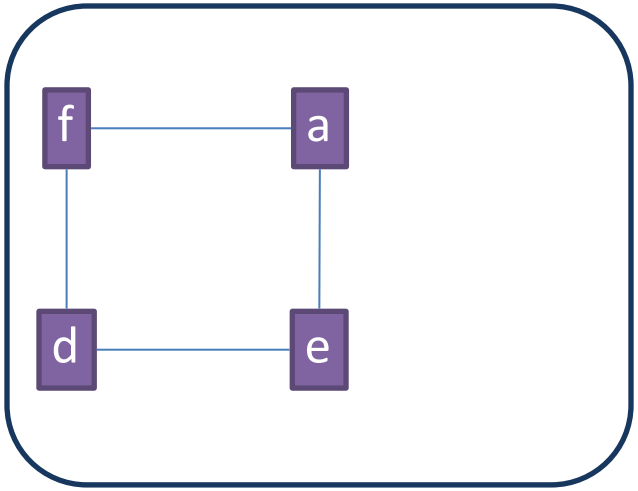
Heuristic Graph Coloring



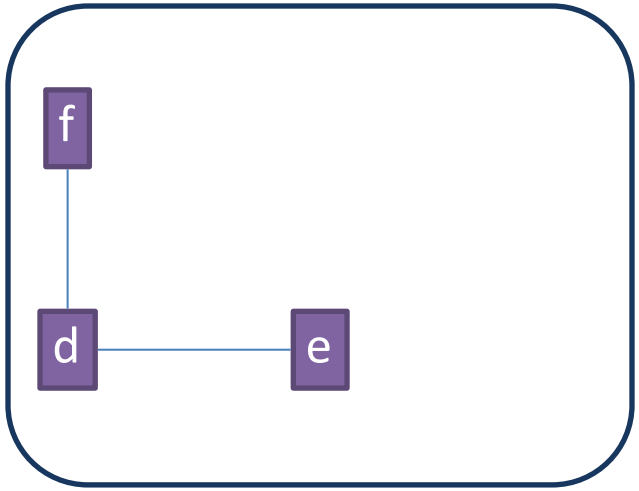
stack: ε



stack: b

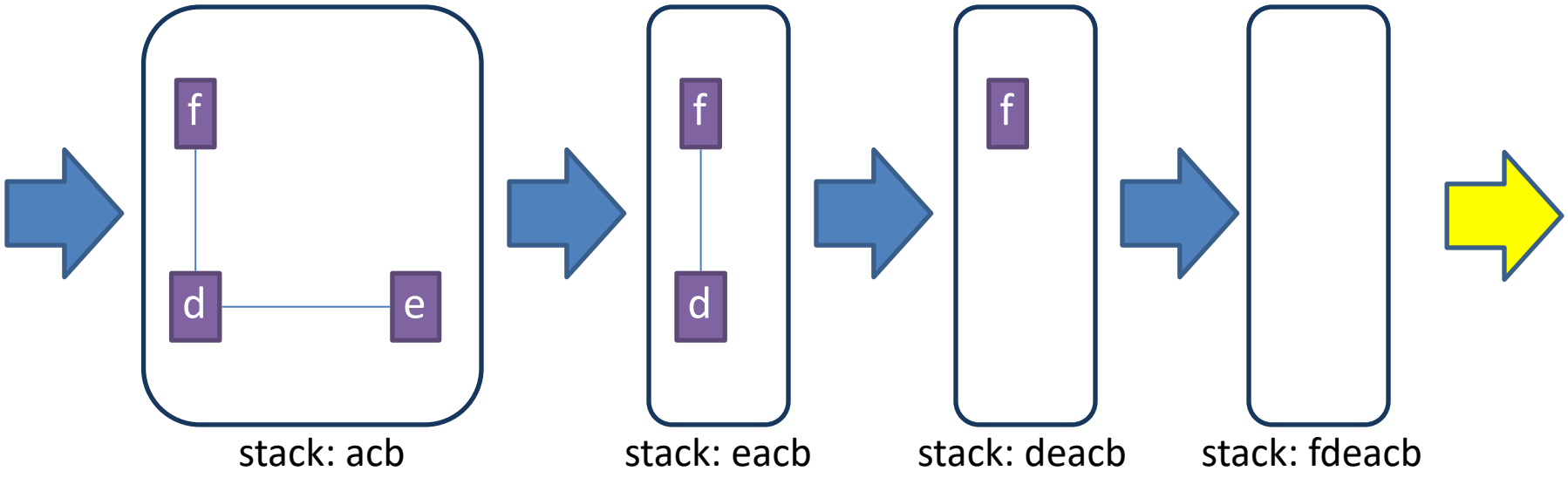


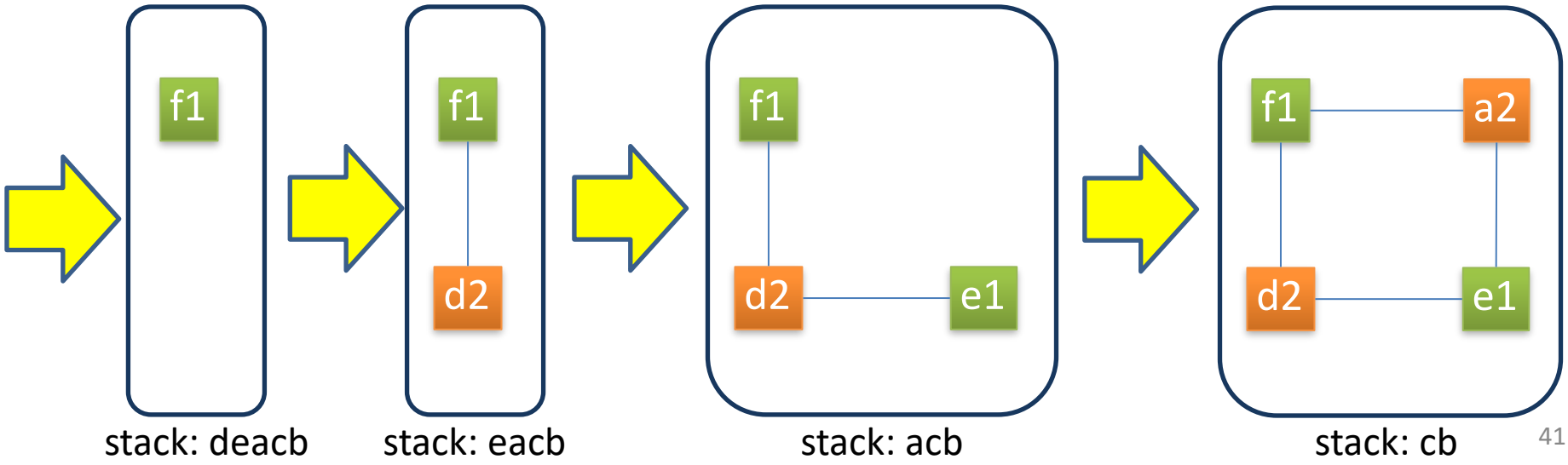
stack: cb



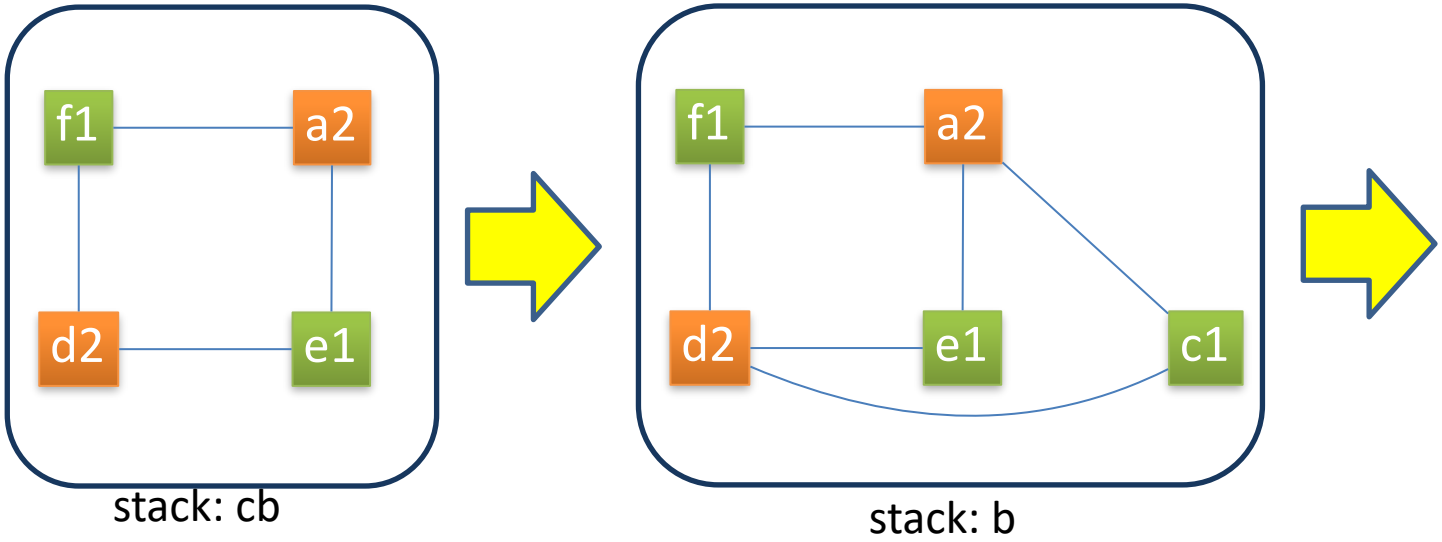
stack: acb

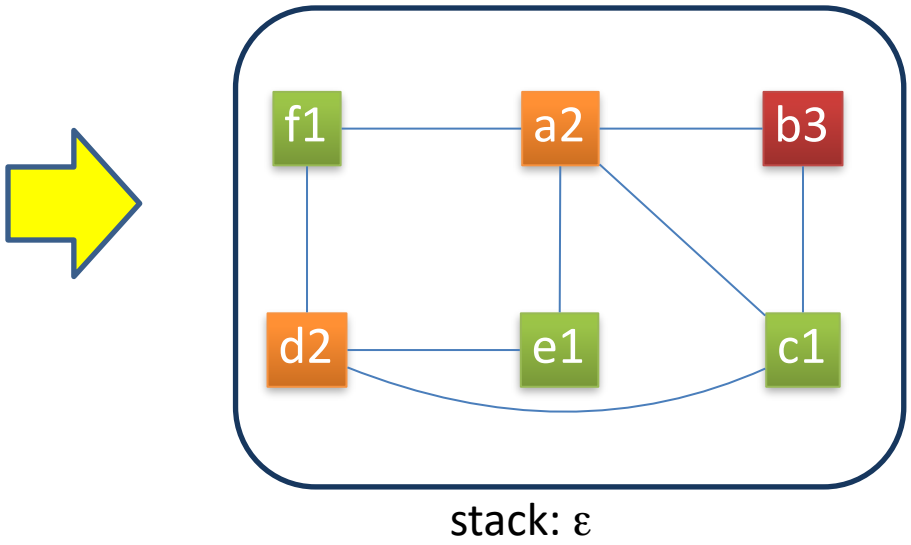
Heuristic Graph Coloring





Heuristic Graph Coloring





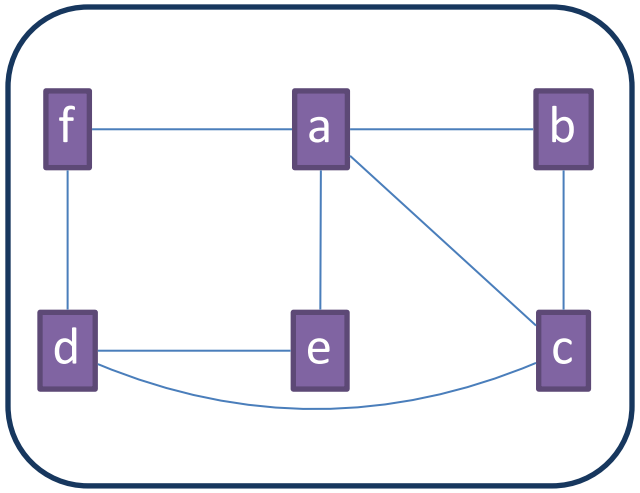
Result:
3 registers for 6 variables

Can we do with 2 registers?

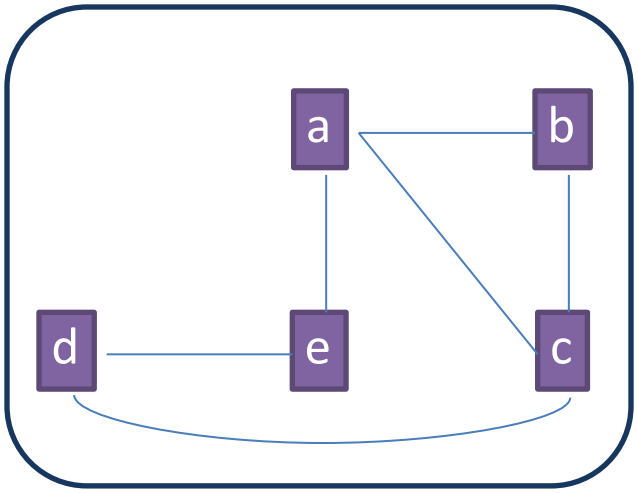
Heuristic Graph Coloring

- Two sources of non-determinism in the algorithm
  - choosing which of the (possibly many) nodes of lowest degree should be detached
  - choosing a free color from the available colors

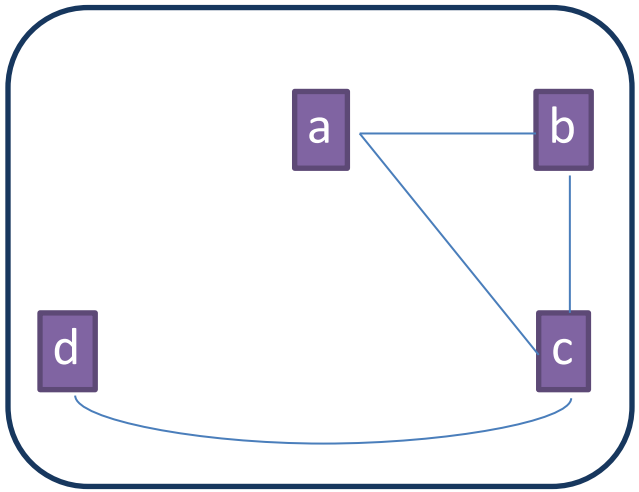
Heuristic Graph Coloring



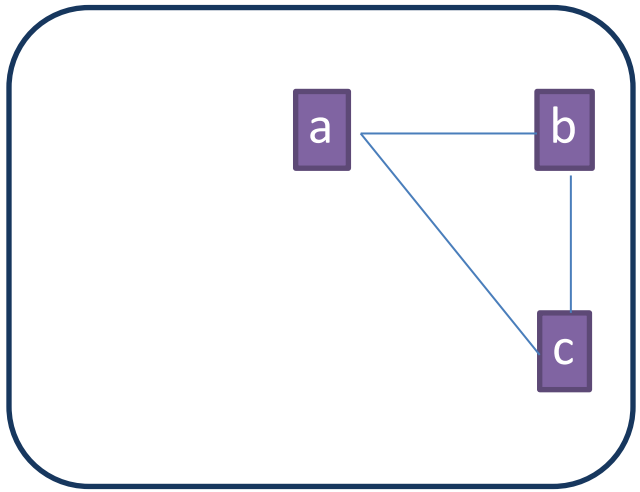
stack: ε



stack: f

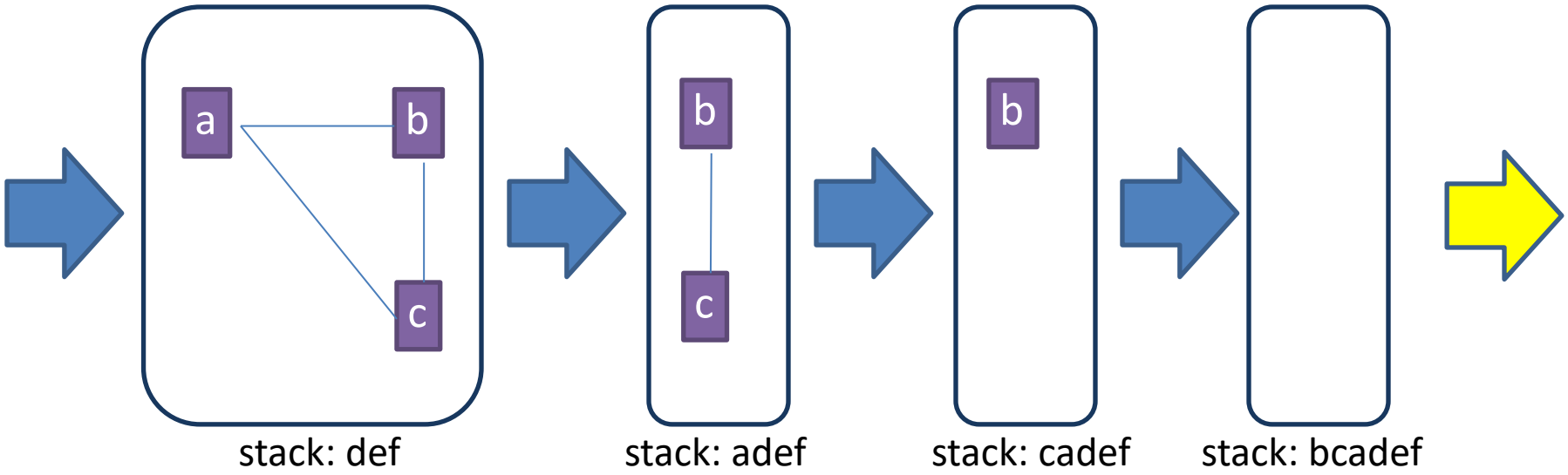


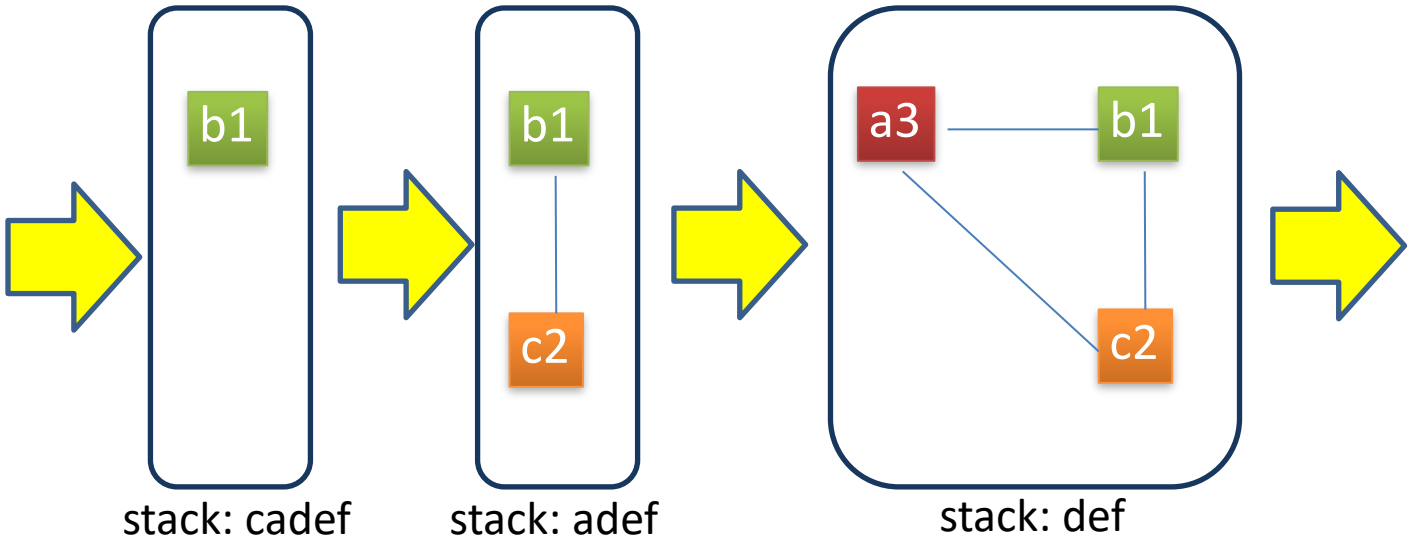
stack: ef



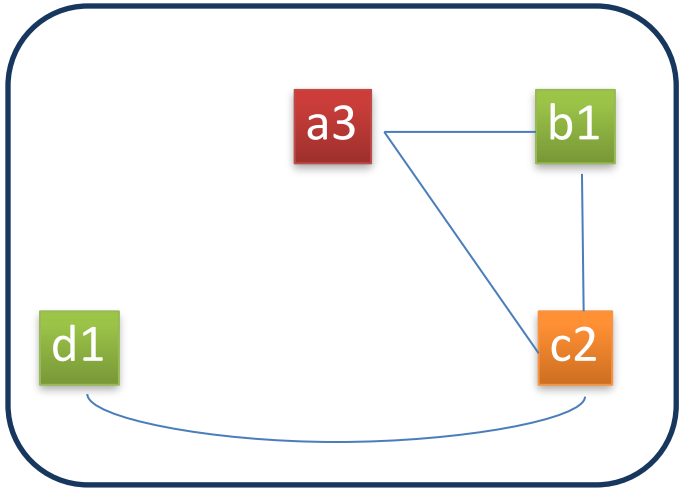
stack: def

Heuristic Graph Coloring

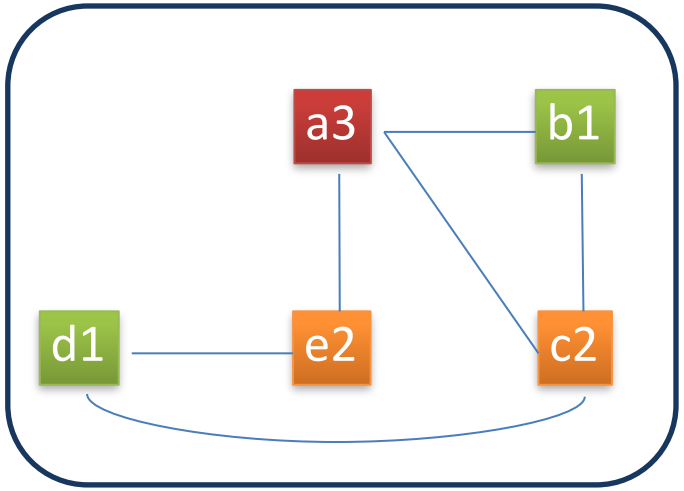




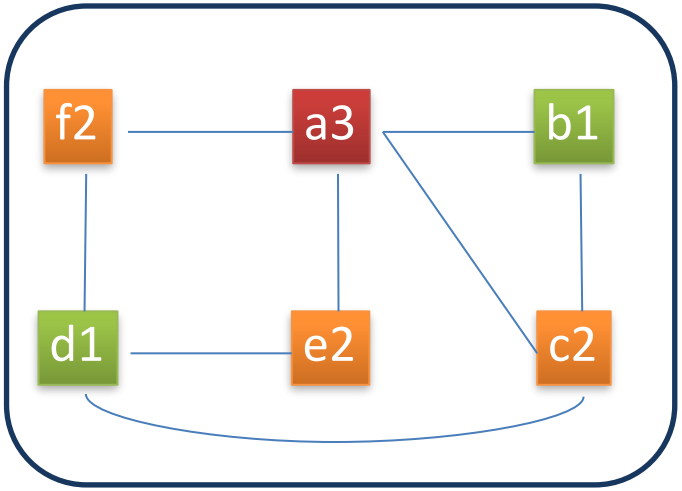
Heuristic Graph Coloring



stack: ef



stack: f



stack: ε

Result:
3 registers for 6 variables

(as before)

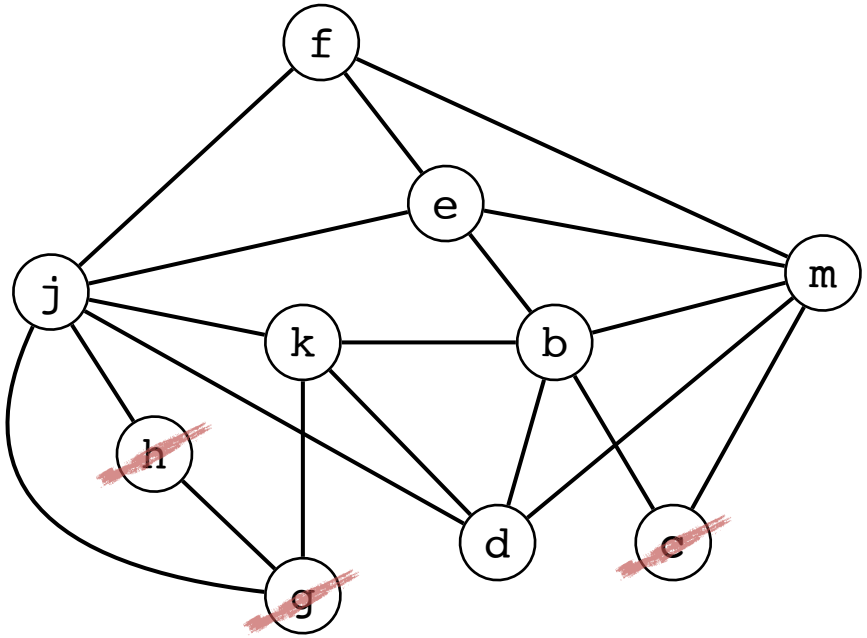
When You Do Not Have Enough Registers

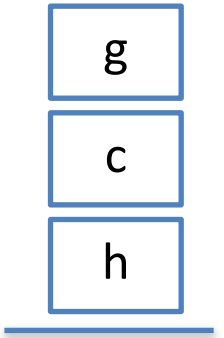
j k

```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
```

live on exit: d j k

$k = 3$





From Appel

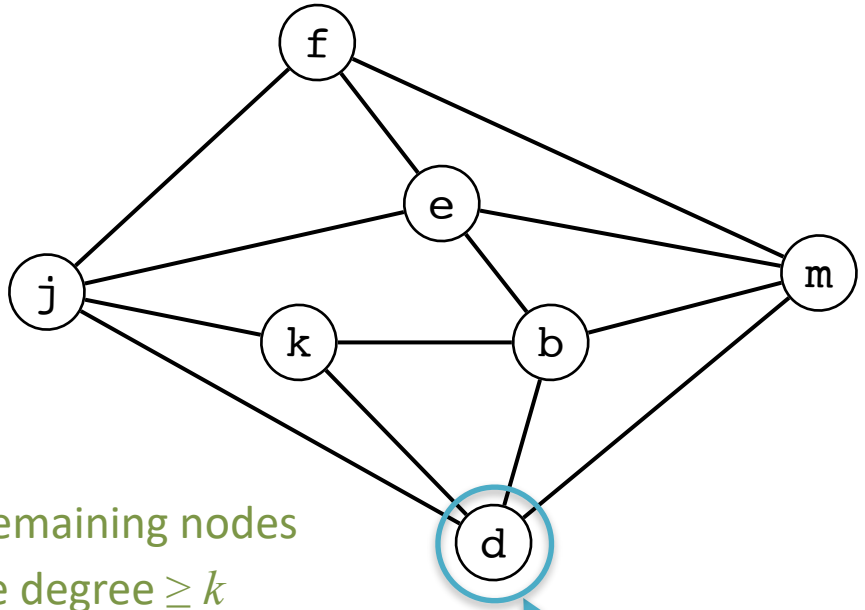
When You Do Not Have Enough Registers

$k = 3$

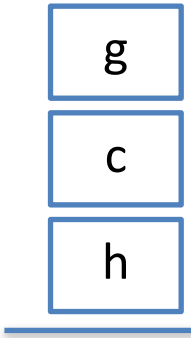
j k

```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
```

d j k

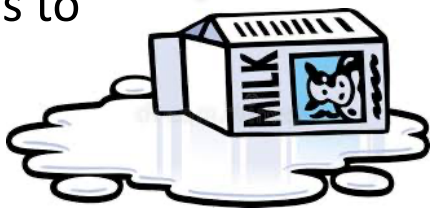


all remaining nodes have degree $\geq k$



- Give up? Hell no!

  Pick one of the variables to *potentially spill*



From Appel

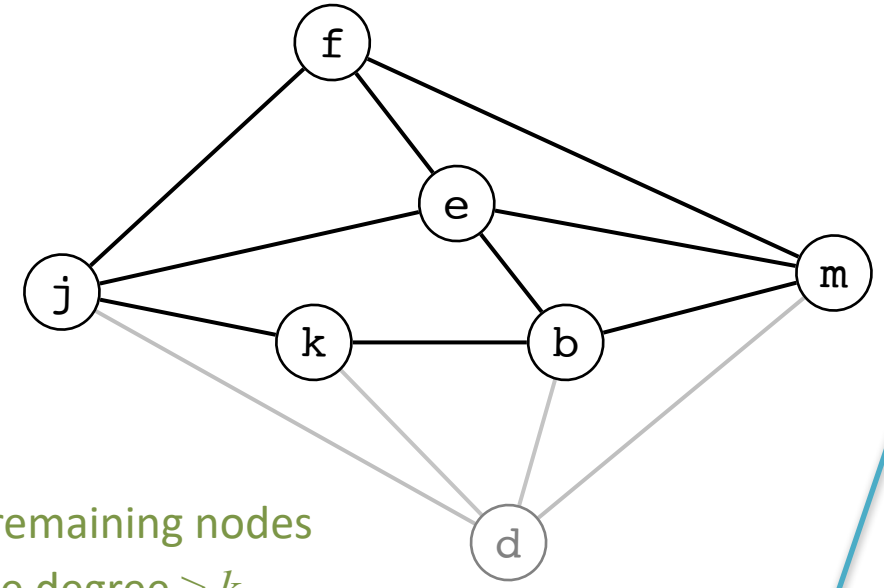
When You Do Not Have Enough Registers

$k = 3$

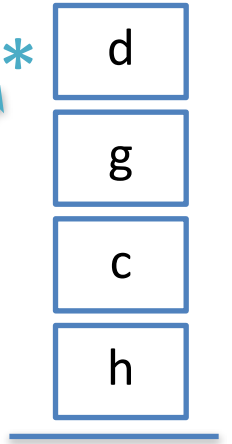
j k

```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
```

d j k

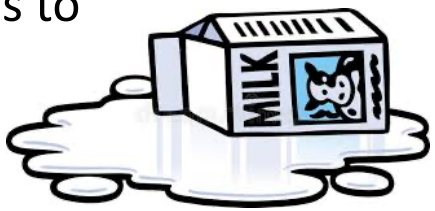


all remaining nodes have degree $\geq k$



- Give up? Hell no!

  Pick one of the variables to *potentially spill*



From Appel

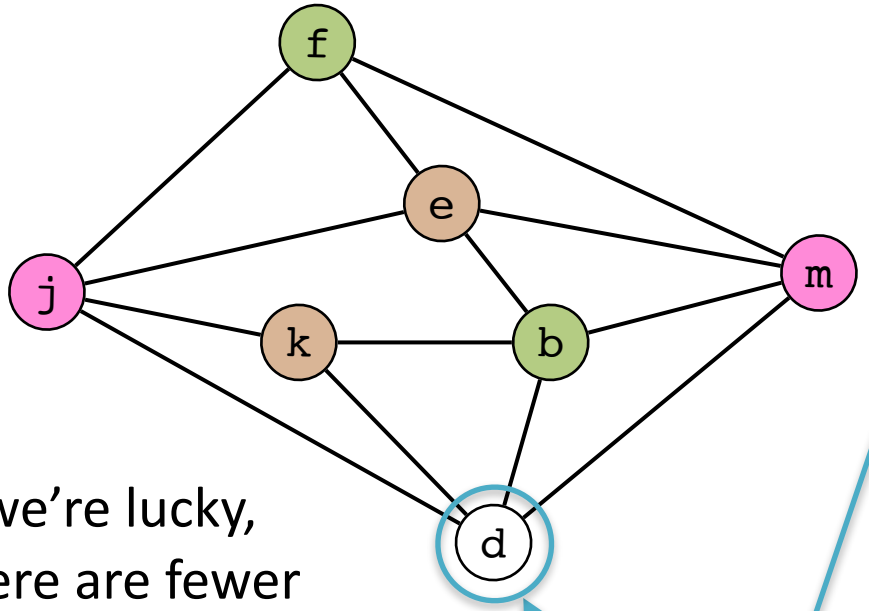
When You Do Not Have Enough Registers

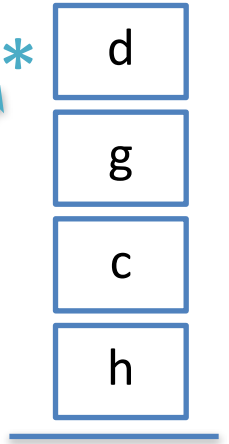
j k

```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d := c
k := m + 4
j := b
```

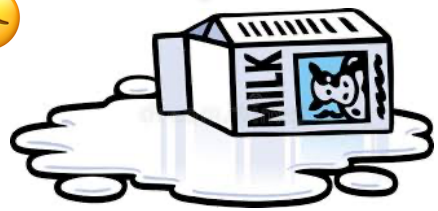
d j k

$k = 3$





- If we're lucky, there are fewer than k colors among its neighbors.
- We got unlucky 😞



From Appel

When You Do Not Have Enough Registers

$k = 3$

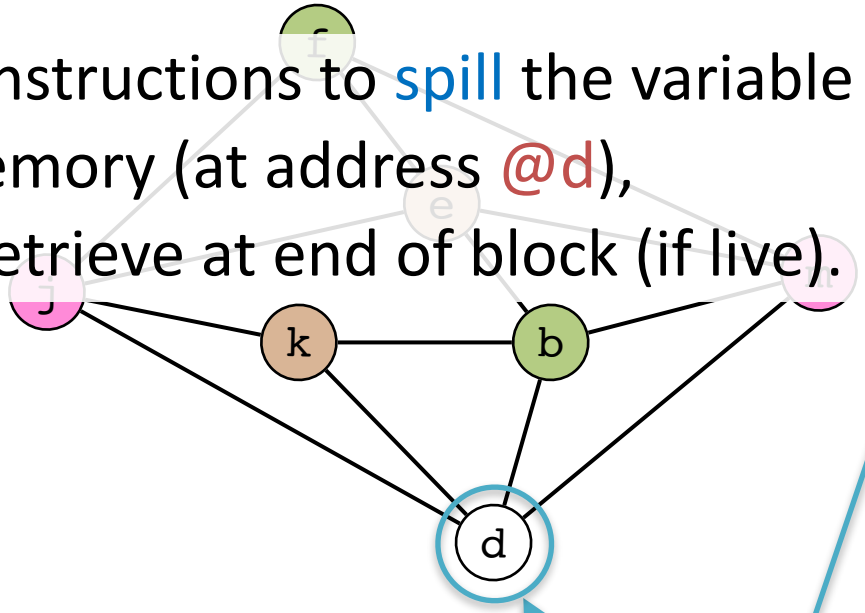
j k

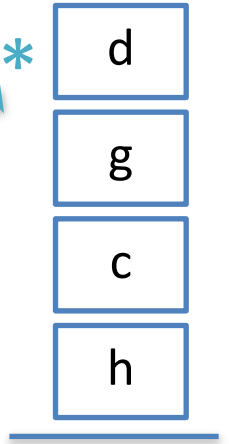
```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d1 := c
*(@d) := d1
k := m + 4
j := b
d2 := *(@d)
```

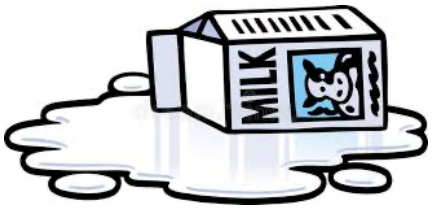
d1 is dead after this point

d2 j k

- Add instructions to spill the variable to memory (at address @d), and retrieve at end of block (if live).







From Appel

When You Do Not Have Enough Registers

$k = 3$

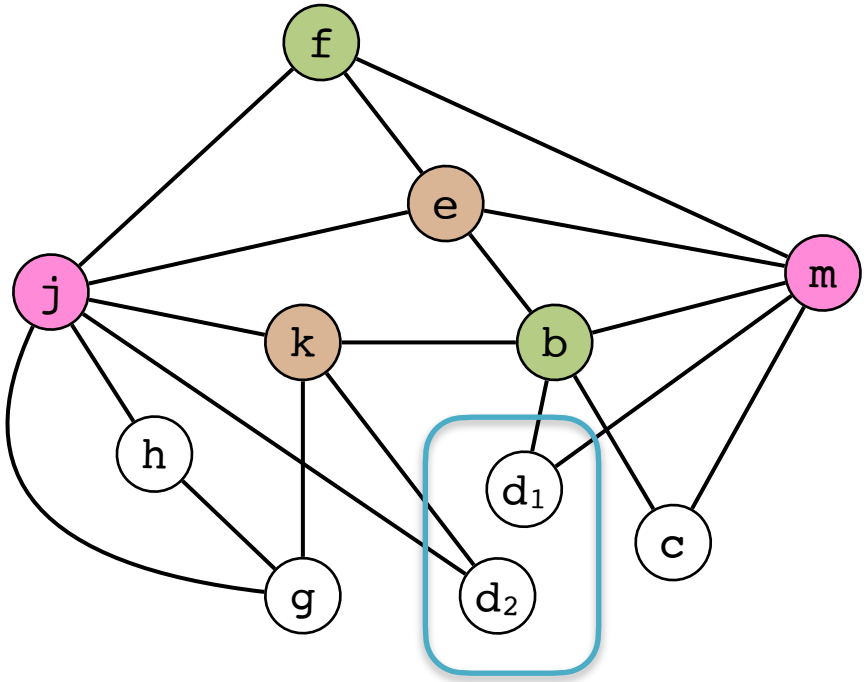
j k

```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d1 := c
*(@d) := d1
k := m + 4
j := b
d2 := *(@d)
```

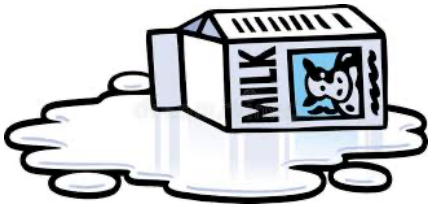
†d_1

d_1 is dead after this point

d_2 j k



- Rinse and repeat…



From Appel

When You Do Not Have Enough Registers

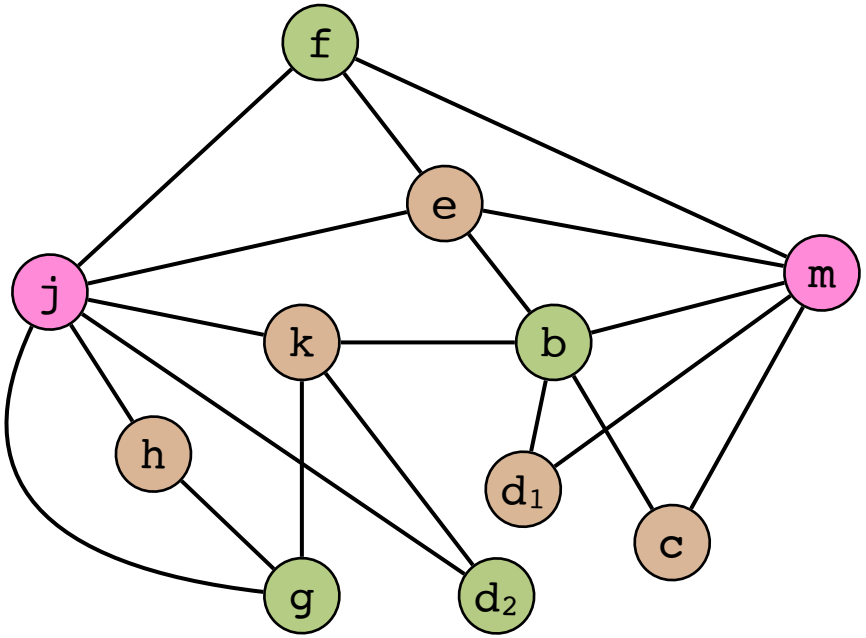
$k = 3$

j k

```
g := *(j+12)
h := k - 1
f := g * h
e := *(j+8)
m := *(j+16)
b := *(f+0)
c := e + 8
d1 := c
*(@d) := d1
k := m + 4
j := b
d2 := *(@d)
```

d_1 is dead after this point

d_2 j k



From Appel

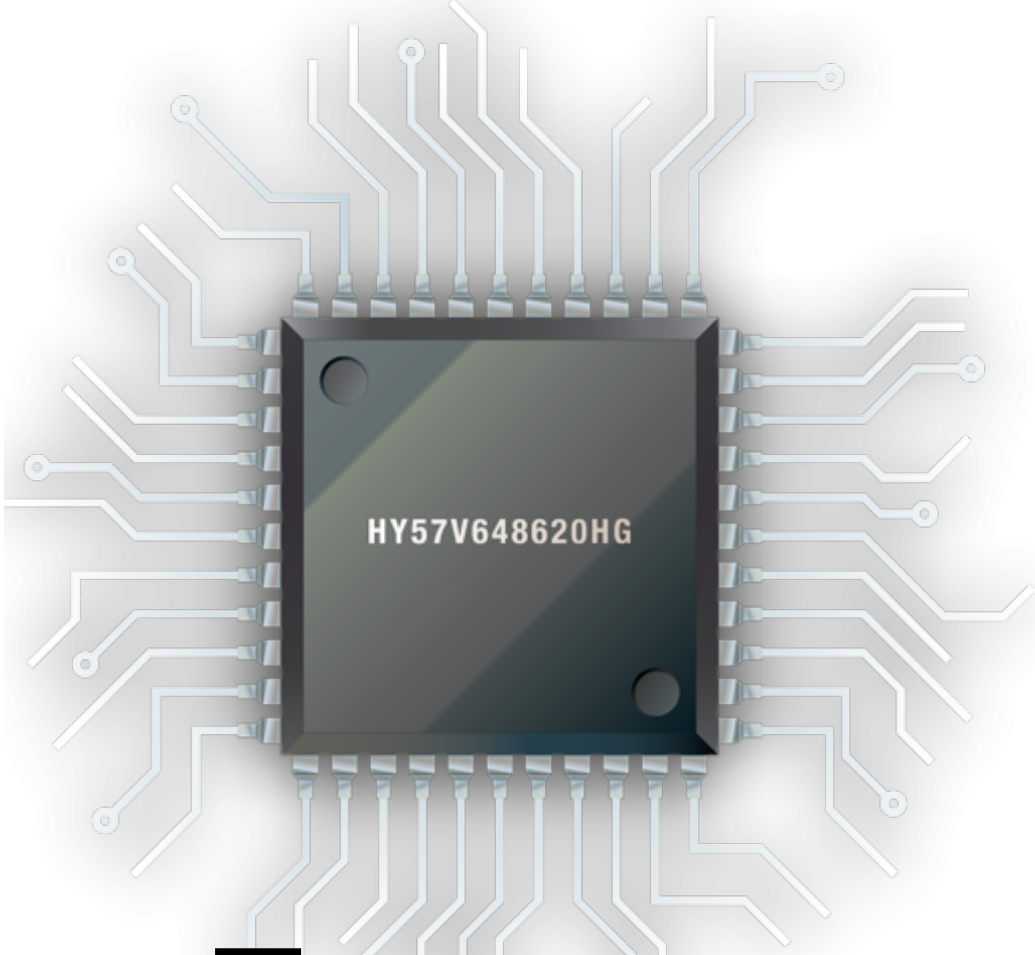
Spill Heuristic

- How to choose a "victim" for (potential) spill? There are many heuristics, *e.g.*:
  - Choose vertex with highest degree
  - Use some cost metric:

$$\text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B)$$

or:

$$\text{cost}(v) = \frac{(\#\text{defs} + \#\text{uses}) \cdot 10^{\text{loop nesting}}}{\text{degree}}$$



Now for some

Assembly Code

Intel IA-32 Assembly

- Going from Assembly to Binary…
  - Assembling
  - Linking

- AT&T syntax vs. Intel syntax
- We will use AT&T syntax
  - matches GNU assembler (GAS)

AT&T vs. Intel Syntax

Attribute	**AT&T** (GAS)	**Intel**
Parameter order	Source, destination	Destination, source
Parameter width	Mnemonics are suffixed with a letter indicating the size of the operands; “b” – byte, “w” – word (16 bit), “l” – double-word (32 bit), “q” – quad-word (64 bit).	Derived from the name of the register that is used
Symbols	Immediate values prefixed by “$”, registers prefixed by “%”	The assembler automatically detects the type of symbols; *i.e.*, if they are registers, immediates, or constants.
Effective addresses	Special syntax: *DISP(BASE,INDEX,SCALE)*	Use +,*, and enclose in square brackets; additionally, size keywords like *byte*, *word*, or *dword* have to be used.

```
movl $64, %ebx                  mov ebx, 64

movl 4(%ebx,%ecx,8), %eax       mov eax, dword [ebx + ecx*8 + 4]
```

IA-32 Registers

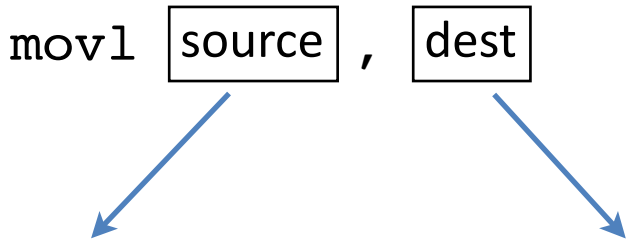
- Eight 32-bit general-purpose registers
  - EAX – accumulator for operands and result data.
    Used to return value from function calls.
  - EBX – pointer to data. Often used as array-base address
  - ECX – counter for string and loop operations
  - EDX – I/O pointer (GP for us)
  - ESI – GP and source pointer for string operations
  - EDI – GP and destination pointer for string operations
  - EBP – stack frame (base) pointer
  - ESP – stack pointer
- EFLAGS register
- EIP (instruction pointer) register
- Six 16-bit segment registers
- … (**ignore the rest for our purposes**)

Not all registers are born equal

- EAX
  - Required operand of MUL, IMUL, DIV and IDIV instructions
  - Contains the result of these operations
- EDX
  - Stores remainder of a DIV or IDIV instruction (EAX stores quotient)
- ESI, EDI
  - ESI – required source pointer for string instructions
  - EDI – required destination pointer for string instructions
- Destination Registers of Arithmetic operations
  - EAX, EBX, ECX, EDX
- ESP – stack pointer for push and pop instructions

IA-32 Addressing Modes

- Machine-instructions take zero or more operands



- Source operand
  - Immediate
  - Register
  - Memory location
- Destination operand
  - Register
  - Memory location

Immediate and Register Operands

- Immediate
  - Value specified in the instruction itself
  - GAS syntax – immediate values preceded by $
  - add **$4**, %esp
- Register
  - Register name is used
  - GAS syntax – register names preceded with %
  - mov **%esp**, %ebp

Memory and Base Displacement Operands

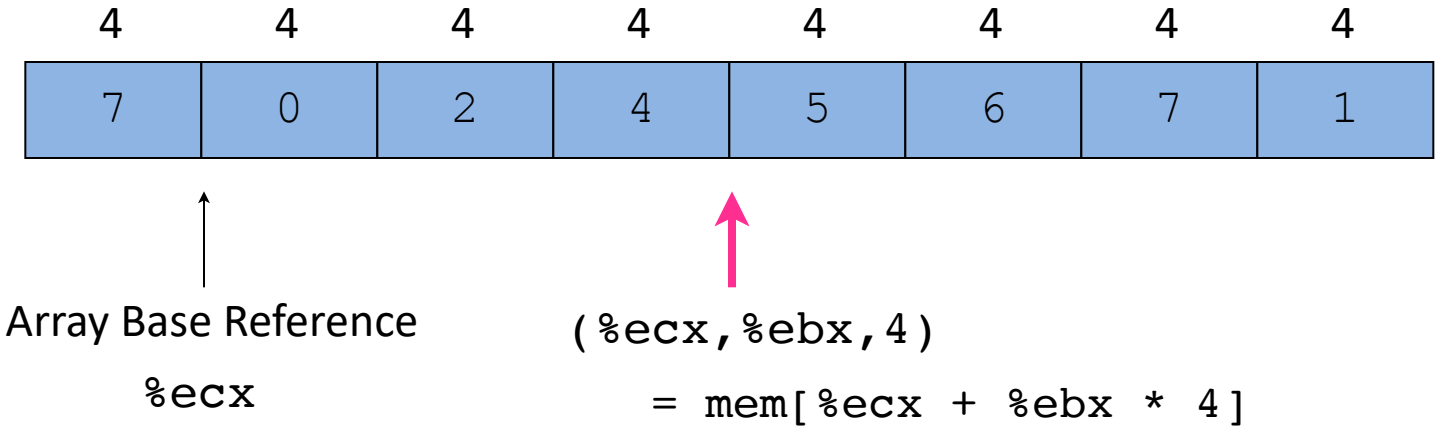
- Memory operands
  - Value at given address
  - GAS syntax — parentheses
  - `mov (%eax), %eax`

(addr)

- Base displacement
  - Value at computed address
  - Address computed out of
    - base register, offset register, scale factor, displacement
  - address = base + (offset*scale) + disp
  - `movl $42, $2(%eax)` ---------- %eax + 2
  - `movl $42, $1(%eax,%ecx,4)` ------ %eax + 4 * %ecx + 2

disp(base, offset, scale)

Base Displacement Addressing




```
mov (%ecx,%ebx,4), %eax
```

address = base + (index*scale) + displacement

address = %ecx + (3*4) + 0 = %ecx + 12

```
base   = %ecx
offset = %ebx
scale  = 4
```

Base Displacement Addressing

- Address computation

`lea` (load effective address)

```
lea (%ebx,%esi,8), %edi
```

%ebx + 8 * %esi

in contrast:

```
mov (%ebx,%esi,8), %edi
```

%ebx + 8 * %esi mem[ ]

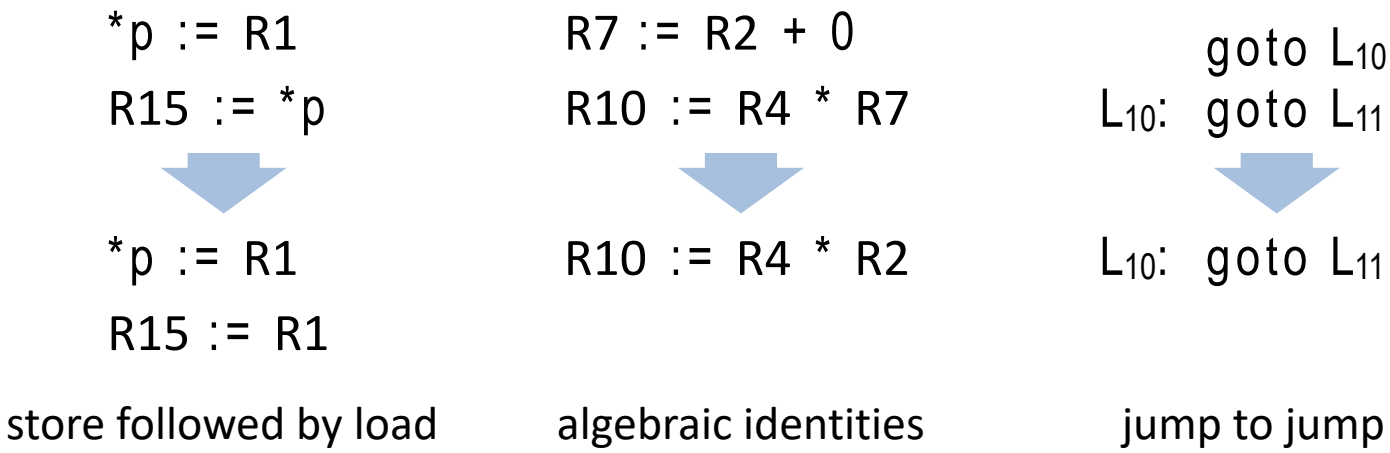
Loads *address* of operand into target register.

Instruction Selection

- How do we choose which instructions to emit from our IR?
- Greatly depends on how we design our IR. These are two of the accepted methods:
  - Peephole Matching (suitable for 3AC)
  - Tiling (suitable for tree-structured IR)

Peephole Matching

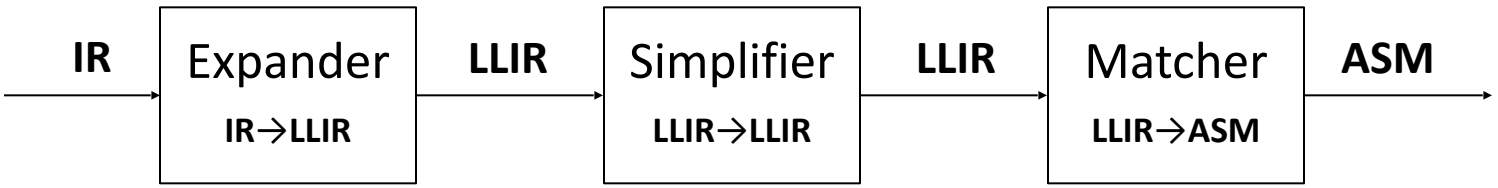
- Basic idea: discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a small sliding window ("peephole") over code and search for improvement
- Classic examples:



Peephole Matching

- How to implement it?

- Modern instruction selectors break problem into three tasks: (*Davidson*, 1989)



Peephole Matching — Example

Original IR Code

$t_1 := 2 * y$

$w := x - t_1$

Register allocation

$x \mapsto r_{17}$ $t_1 \mapsto r_{14}$

$y \mapsto r_{13}$ $w \mapsto r_{18}$

Expand

LLIR Code

$r_{10} := 2$
$r_{11} := @y$
$r_{12} := bp + r_{11}$
$r_{13} := *(r_{12})$
$r_{14} := r_{10} * r_{13}$
$r_{15} := @x$
$r_{16} := bp + r_{15}$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

Peephole Matching — Example

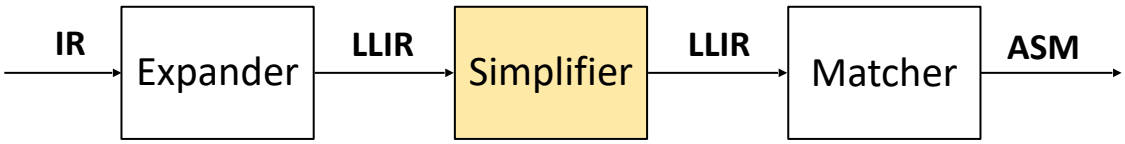
LLIR Code

$r_{10} := 2$
$r_{11} := @y$
$r_{12} := bp + r_{11}$
$r_{13} := *(r_{12})$
$r_{14} := r_{10} * r_{13}$
$r_{15} := @x$
$r_{16} := bp + r_{15}$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

Simplify

LLIR Code

$r_{13} := *(bp + @y)$
$r_{14} := 2 * r_{13}$
$r_{17} := *(bp + @x)$
$r_{18} := r_{17} - r_{14}$
$*(bp + @w) := r_{18}$



Peephole Matching — Example

LLIR Code

$r_{13} := *(bp + @y)$

$r_{14} := 2 * r_{13}$

$r_{17} := *(bp + @x)$

$r_{18} := r_{17} - r_{14}$

$*(bp + @w) := r_{18}$

Match

Assembly (x86) Code

```
movl @y(%ebp), %esi
movl %esi, %edi
shll $1, %edi
movl @x(%ebp), %ecx
movl %ecx, %edx
subl %edi, %edx
movl %edx, @w(%ebp)
```

Peephole Matching — Example

- Simplifier rules

$r_1 := \mathbf{c}$
$r_2 := r_3 + r_1$
⇒ $r_2 := r_3 + \mathbf{c}$

c – constant

$r_1 := r_2 + e$
$r_3 := *(r_1)$
⇒ $r_3 := *(r_2 + e)$

e – constant or register

$r_1 := 2 * r_2$
$r_3 := r_4 - r_1$
⇒ $r_3 := r_4 - r_2 - r_2$

Normally, there would be many such rules; this is only a example.

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{10} := 2$
$r_{11} := @y$
$r_{12} := bp + r_{11}$
$r_{13} := *(r_{12})$
$r_{14} := r_{10} * r_{13}$
$r_{15} := @x$
$r_{16} := bp + r_{15}$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

$r_{10} := 2$
$r_{12} := bp + @y$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{10} := 2$
$r_{12} := bp + @y$
$r_{13} := *(r_{12})$
$r_{14} := r_{10} * r_{13}$
$r_{15} := @x$
$r_{16} := bp + r_{15}$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

$r_{10} := 2$
$r_{13} := *(bp + @y)$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{10} := 2$
$r_{13} := *(bp + @y)$
$r_{14} := r_{10} * r_{13}$
$r_{15} := @x$
$r_{16} := bp + r_{15}$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

$r_{12} := *(bp + @y)$
$r_{14} := 2 * r_{13}$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{13} := *(bp + @y)$
$r_{14} := 2 * r_{13}$
$r_{15} := @x$
$r_{16} := bp + r_{15}$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{13} := *(bp + @y)$
$r_{14} := 2 * r_{13}$
$r_{15} := @x$
$r_{16} := bp + r_{15}$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

$r_{14} := 2 * r_{13}$

$r_{16} := bp + @x$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{13} := *(bp + @y)$
$r_{14} := 2 * r_{13}$
$r_{16} := bp + @x$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

$r_{14} := 2 * r_{13}$

$r_{17} := *(bp + @x)$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{13} := *(bp + @y)$
$r_{14} := 2 * r_{13}$
$r_{17} := *(bp + @x)$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

$r_{17} := *(bp + @x)$
$r_{18} := r_{17} - r_{13} - r_{13}$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

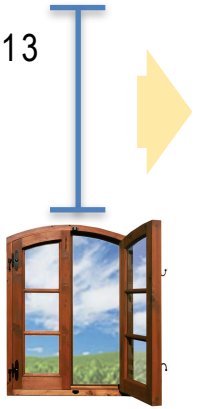
$r_{13} := *(bp + @y)$
$r_{17} := *(bp + @x)$
$r_{18} := r_{17} - r_{13} - r_{13}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{13} := *(bp + @y)$
$r_{17} := *(bp + @x)$
$r_{18} := r_{17} - r_{13} - r_{13}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$



$r_{18} := r_{17} - r_{13} - r_{13}$

$r_{20} := bp + @w$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

$r_{13} := *(bp + @y)$
$r_{17} := *(bp + @x)$
$r_{18} := r_{17} - r_{13} - r_{13}$
$r_{20} := bp + @w$
$*(r_{20}) := r_{18}$

$r_{18} := r_{17} - r_{13} - r_{13}$
$*(bp + @w) := r_{18}$

Peephole Matching — Example

Steps of the Simplifier

LLIR Code

r_{13} := *(bp + @y)
r_{17} := *(bp + @x)
r_{18} := r_{17} – r_{13} – r_{13}
*(bp + @w) := r_{18}

Peephole Matching — Example

LLIR Code

$r_{10} := 2$
$r_{11} := @y$
$r_{12} := bp + r_{11}$
$r_{13} := *(r_{12})$
$r_{14} := r_{10} * r_{13}$
$r_{15} := @x$
$r_{16} := bp + r_{15}$
$r_{17} := *(r_{16})$
$r_{18} := r_{17} - r_{14}$
$r_{19} := @w$
$r_{20} := bp + r_{19}$
$*(r_{20}) := r_{18}$

Simplify

LLIR Code

$r_{13} := *(bp + @y)$
$r_{17} := *(bp + @x)$
$r_{18} := r_{17} - r_{13} - r_{13}$
$*(bp + @w) := r_{18}$

Peephole Matching

Expander

register-transfer language

- Turns IR code into a low-level IR (LLIR) such as RTL
- Operation-by-operation, template-driven rewriting
- LLIR form includes all direct effects (*e.g.*, setting flags)
- Significant, albeit constant, expansion of size

Peephole Matching

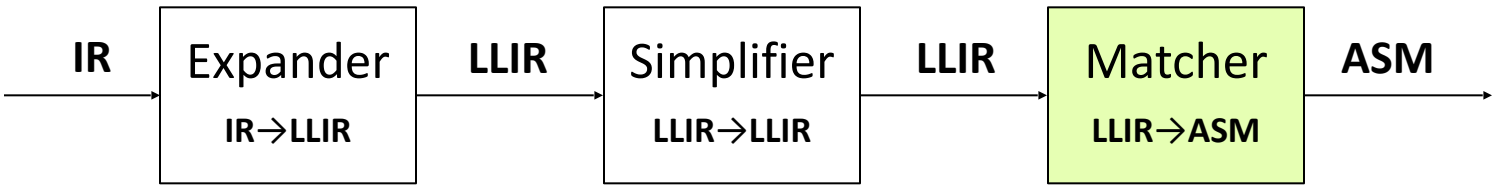
Simplifier

- Looks at LLIR through window and rewrites it
- Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
- Performs local optimization within window

Peephole Matching

Matcher

- Compares simplified LLIR against a library of patterns
- Picks low-cost pattern that captures effects
- Must preserve LLIR effects, may add new ones
- Generates the assembly code output



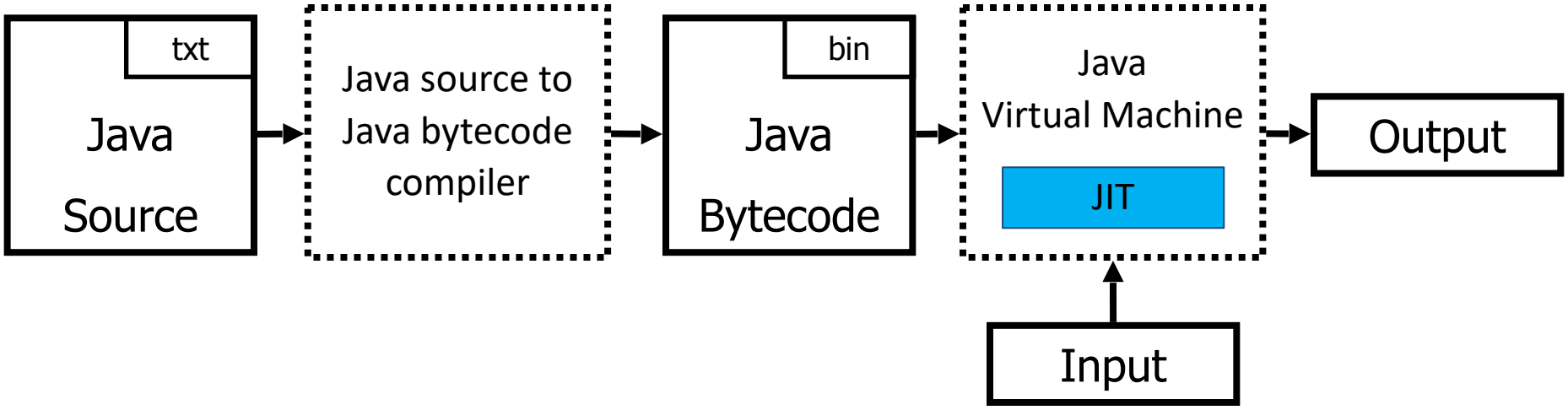
Summary

- Register allocation
  - Use live variable analysis to determine which variables are needed at each program point
  - Compute interference between variables and reduce the problem to graph coloring
    - A heuristic solution for graph coloring is used
- Instruction selection
  - Use Peephole matching directly on 3AC
    - Rewrite instructions in fixed sliding window

Aside

Just-in-time Compiler

(Java example)

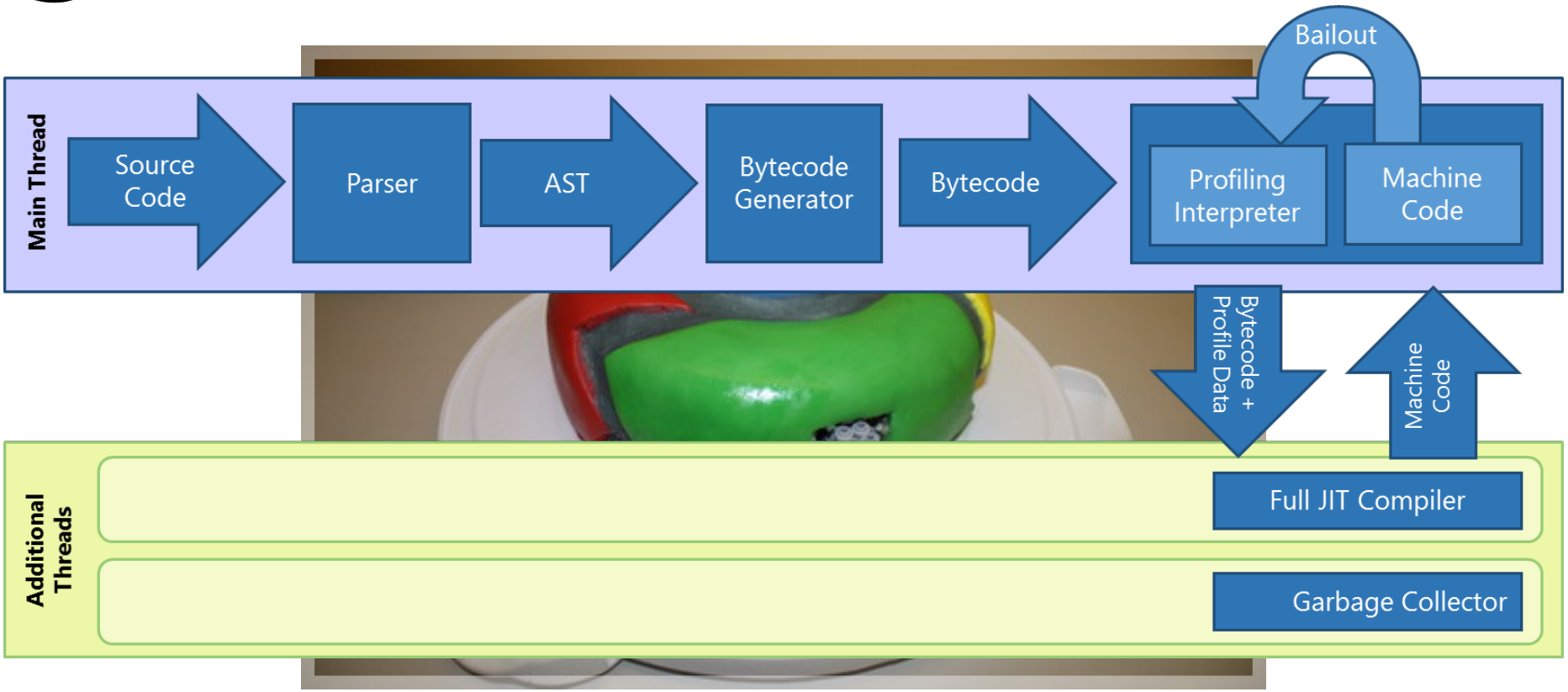


Just-in-time (JIT) compilation: bytecode interpreter (in the JVM) compiles program fragments during interpretation to avoid expensive re-interpretation.

Aside

Just-in-time Compiler

(Javascript example)



- The compiled code is optimized dynamically at runtime, based on ***runtime behavior***

Coming Up