Lecture 12 – Code Generation

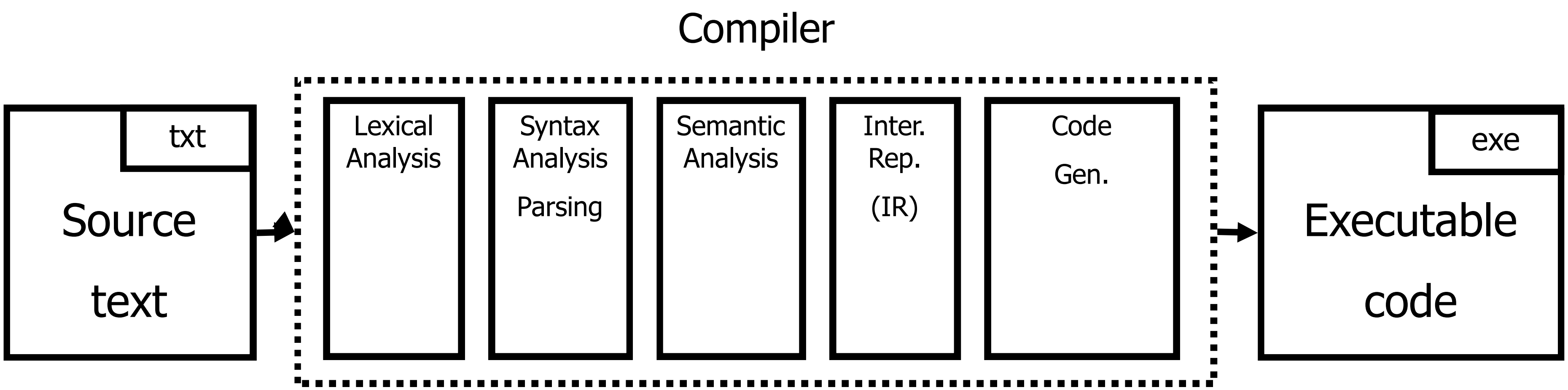
THEORY OF COMPILATION

Eran Yahav

www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec12.pptx

Reference: Dragon 8. MCD 4.2.4

You are here


Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
Inter.
Rep.
(IR)
Code
Gen.
exe
Executable
code

simple code generation

- registers
  - used as operands of instructions
  - can be used to store temporary results
  - can (should) be used as loop indexes due to frequent arithmetic operation
  - used to manage administrative info (e.g., runtime stack)

- number of registers is limited
- need to allocate them in a clever way

simple code generation

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

- further assume that we have all registers available for our use
  - ignore registers allocated for stack management

simple code generation

- translate each 3AC instruction separately
- A register descriptor keeps track of the variable names whose current value is in that register.
  - we use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty.
  - As code generation progresses, each register will hold the value of zero or more names.
- For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found.
  - The location may be a register, a memory address, a stack location, or some set of more than one of these
  - Information can be stored in the symbol-table entry for that variable

simple code generation

For each three-address statement x := y op z,

1. Invoke *getreg* (x := y op z) to select registers R_x, R_y, and R_z.
2. If Ry does not contain y, issue: "LD R_y, y' ", for a location y' of y.
3. If Rz does not contain z, issue: "LD R_z, z' ", for a location z' of z.
4. Issue the instruction "OP R_x,R_y,R_z"
5. Update the address descriptors of x, y, z, if necessary.
   - R_x is the only location of x now, and
     R_x contains only x (remove R_x from other address descriptors).

updating descriptors

- 1. For the instruction LD R, x
  a) Change the register descriptor for register R so it holds only x.
  b) Change the address descriptor for x by adding register R as an additional location.
- 2. For the instruction ST x, R
  - change the address descriptor for x to include its own memory location.
- 3. For an operation such as ADD Rx, Ry, Rz, implementing a 3AC instruction x = y + *z*
  a) Change the register descriptor for Rx so that it holds only *x.*
  b) Change the address descriptor for x so that its only location is Rx. Note that the memory location for x is *not* now in the address descriptor for x.
  c) Remove Rx from the address descriptor of any variable other than x.
- 4. When we process a copy statement x = y, after generating the load for y into register Ry, if needed, and after managing descriptors as for all load statements (rule 1):
  a) Add x to the register descriptor for Ry.
  b) Change the address descriptor for x so that its only location is Ry .

example

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

R1	R2	R3	A	B	C	D	t	u	v
			A	B	C	D			

t = A – B
LD R1,A
LD R2,B
SUB R2,R1,R2

R1	R2	R3	A	B	C	D	t	u	v
A	t		A,R1	B	C	D	R2		

u = A – C
LD R3,C
SUB R1,R1,R3

R1	R2	R3	A	B	C	D	t	u	v
u	t	C	A	B	C,R3	D	R2	R1	

v = t + u
ADD R3,R2,R1

R1	R2	R3	A	B	C	D	t	u	v
u	t	v	A	B	C	D	R2	R1	R3

A B C D = live outside the block
t,u,v = temporaries in local storate

example

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

R1	R2	R3	A	B	C	D	t	u	v
u	t	v	A	B	C	D	R2	R1	R3

A = D
    LD R2, D

R1	R2	R3	A	B	C	D	t	u	v
u	A,D	v	R2	B	C	D,R2		R1	R3

D = v + u
    ADD R1,R3,R1

R1	R2	R3	A	B	C	D	t	u	v
D	A	v	R2	B	C	R1			R3

exit
    ST A, R2
    ST D, R1

R1	R2	R3	A	B	C	D	t	u	v
D	A	v	A,R2	B	C	D,R1			R3

A B C D = live outside the block
t,u,v = temporaries in local storate

design of getReg

- many design choices

- simple rules:
  - If y is currently in a register, pick a register already containing y as Ry. No need to load this register.
  - If y is not in a register, but there is a register that is currently empty, pick one such register as Ry.
- complicated case:
  - y is not in a register, but there is no free register.

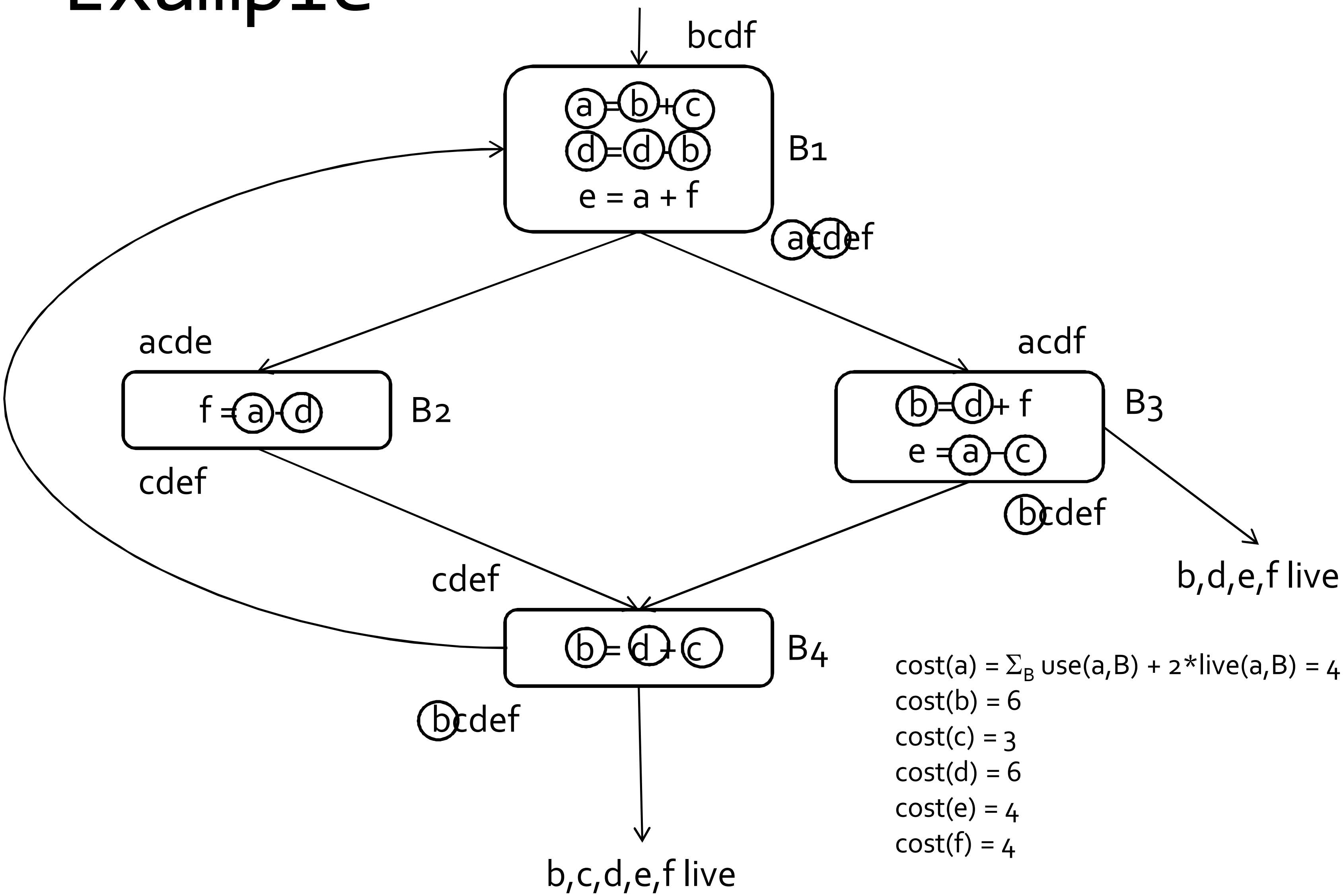
design of getReg

- instruction: x = y + z
- y is not in a register, no free register
- let R be a taken register holding value of a variable v
- possibilities:
  - if the value v is available somewhere other than R, we can allocate R to be Ry
  - if v is x, the value computed by the instruction, we can use it as Ry (it is going to be overwritten anyway)
  - if v is not used later, we can use R as Ry
  - otherwise: spill the value to memory by ST v,R

global register allocation

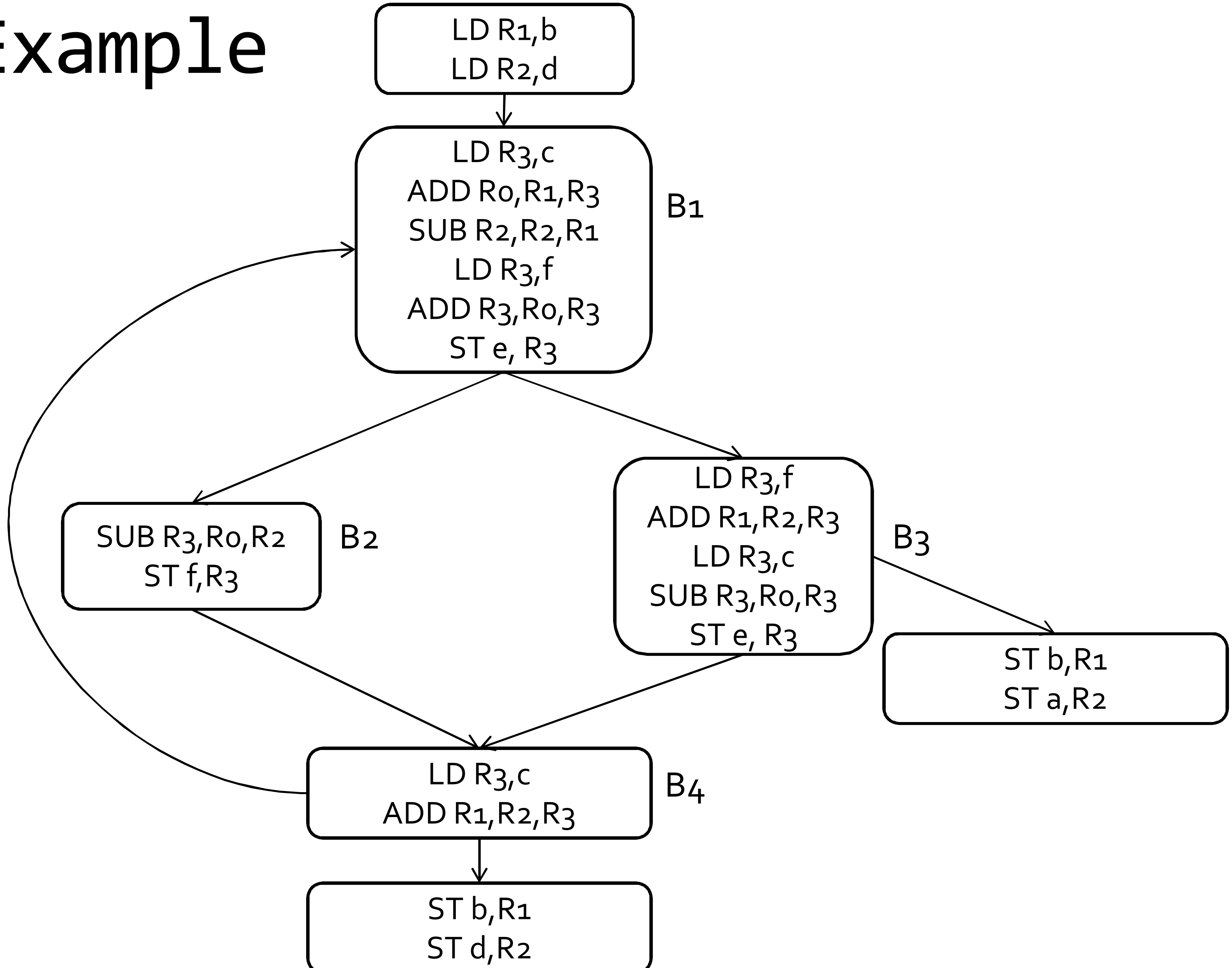
- so far we assumed that register values are written back to memory at the end of every basic block
- want to save load/stores by keeping frequently accessed values in registers
  - e.g., loop counters
- idea: compute "weight" for each variable
  - for each use of v in B prior to any definition of v add 1 point
  - for each occurrence of v in a following block using v add 2 points, as we save the store/load between blocks
  - cost(v) = $\Sigma_B$use(v,B) + 2*live(v,B)
    - use(v,B) is is the number of times v is used in B prior to any definition of v
    - live(v, B) is 1 if v is live on exit from B and is assigned a value in B
  - after computing weights, allocate registers to the "heaviest" values

Example



cost(a) = Σ_B use(a,B) + 2*live(a,B) = 4
cost(b) = 6
cost(c) = 3
cost(d) = 6
cost(e) = 4
cost(f) = 4

Example



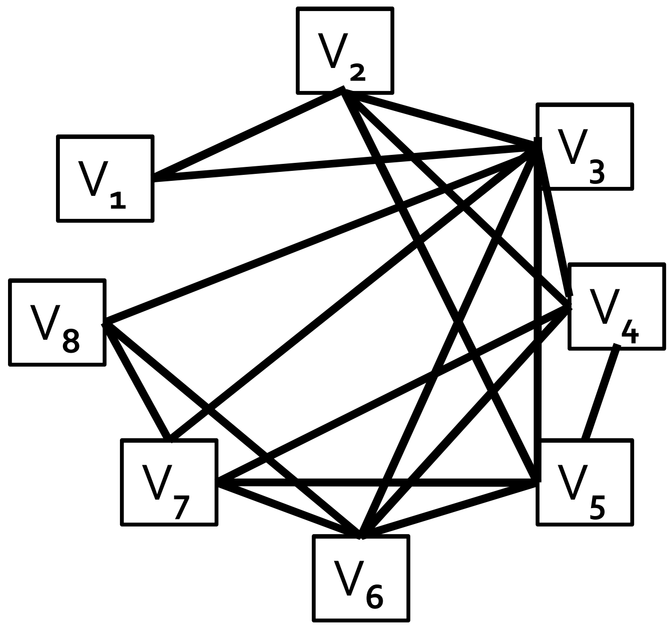
LD R1,b
LD R2,d
LD R3,c
ADD R0,R1,R3
SUB R2,R2,R1
LD R3,f
ADD R3,R0,R3
ST e, R3
B1
SUB R3,R0,R2
ST f,R3
B2
LD R3,f
ADD R1,R2,R3
LD R3,c
SUB R3,R0,R3
ST e, R3
B3
ST b,R1
ST a,R2
LD R3,c
ADD R1,R2,R3
B4
ST b,R1
ST d,R2

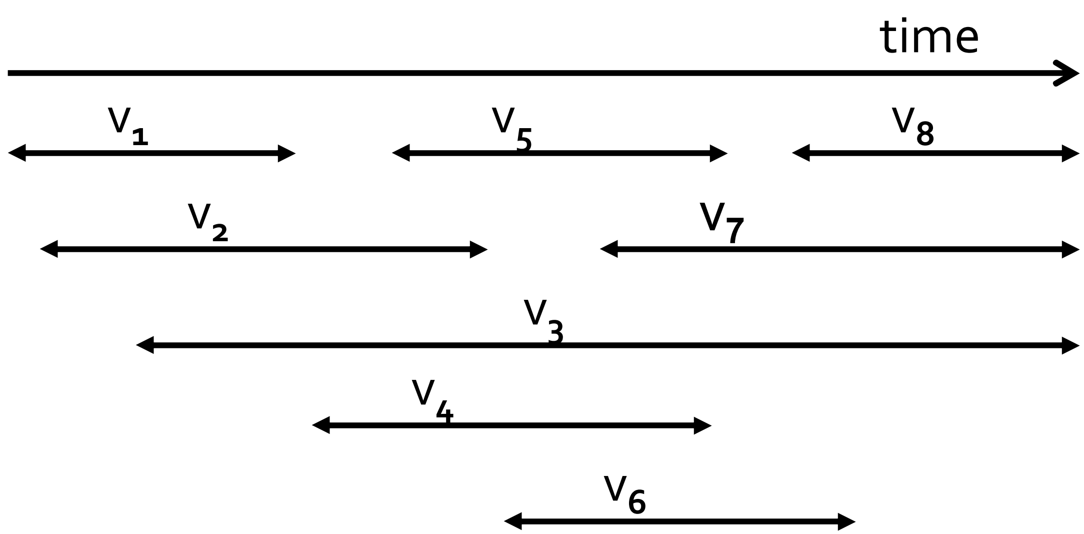
Register Allocation by Graph Coloring

- Address register allocation by
  - liveness analysis
  - reduction to graph coloring
  - optimizations by program transformation

- Main idea
  - register allocation = coloring of an interference graph
  - every node is a variable
  - edge between variables that "interfere" = are both live at the same time
  - number of colors = number of registers

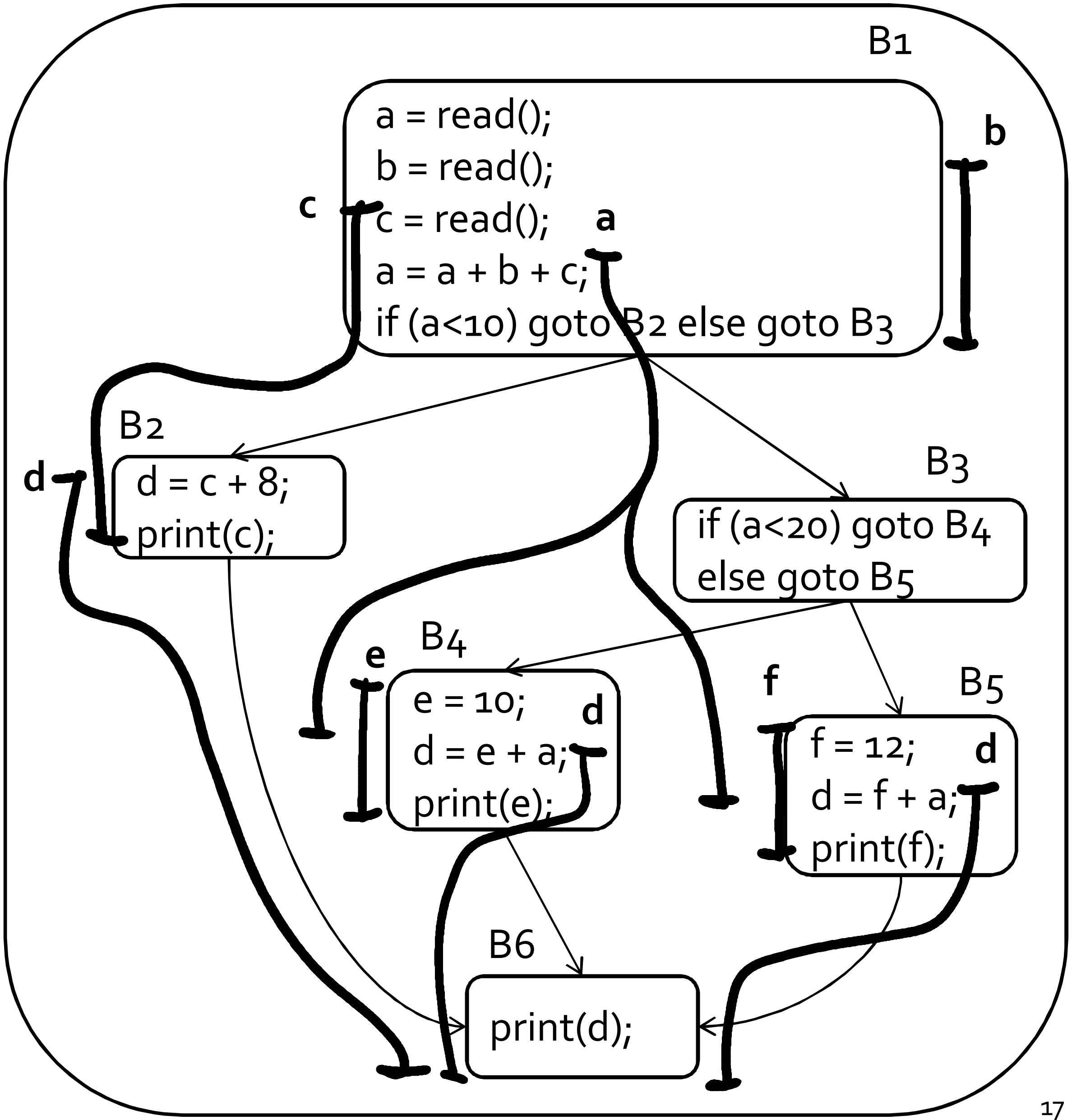
Example



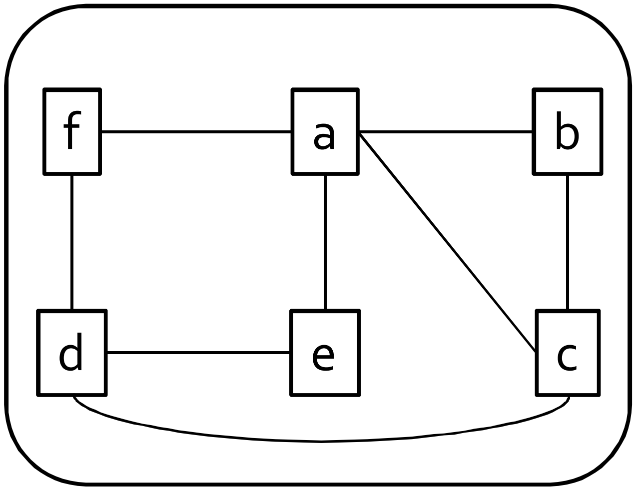


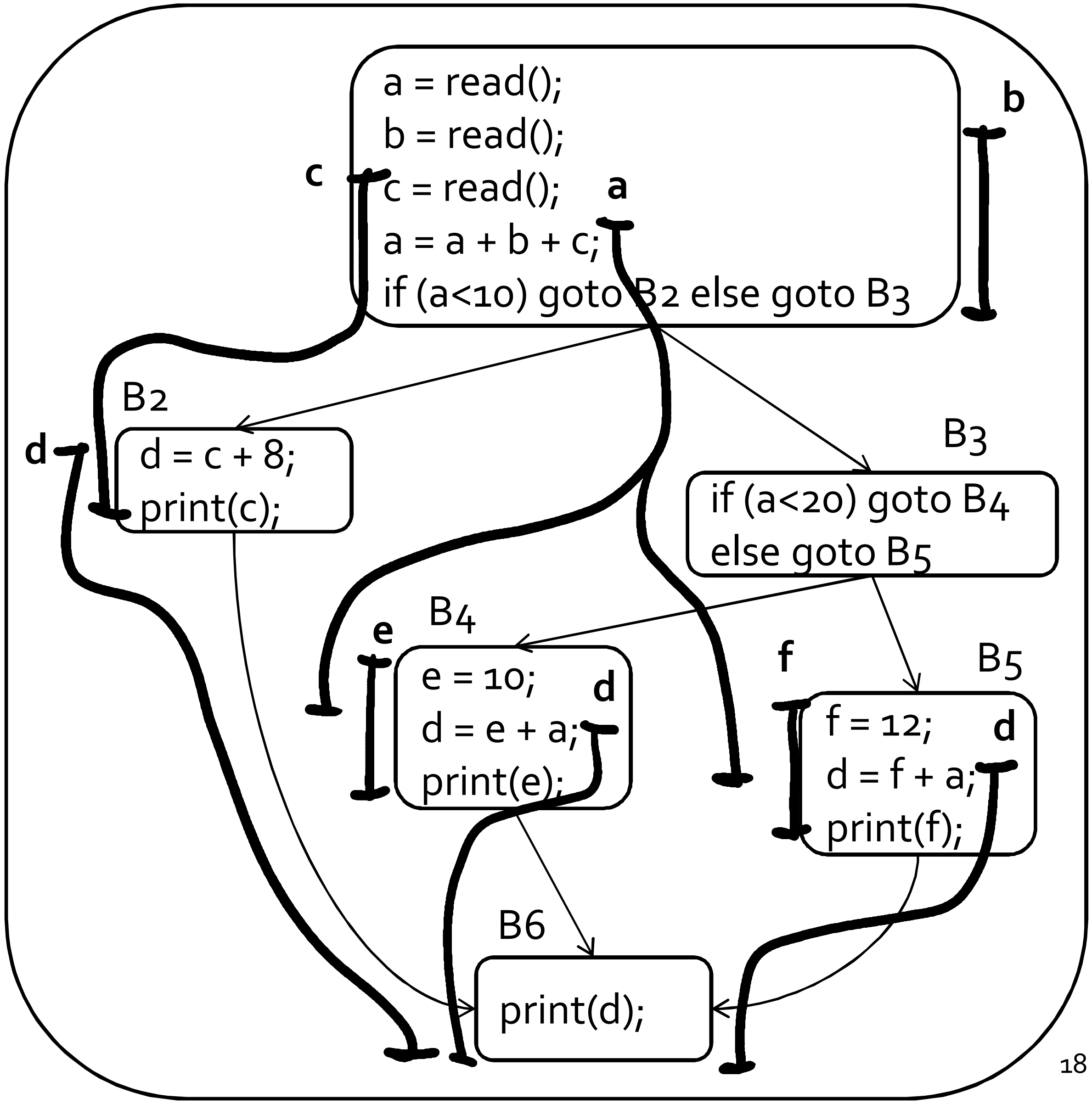
Example

```
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: Interference Graph





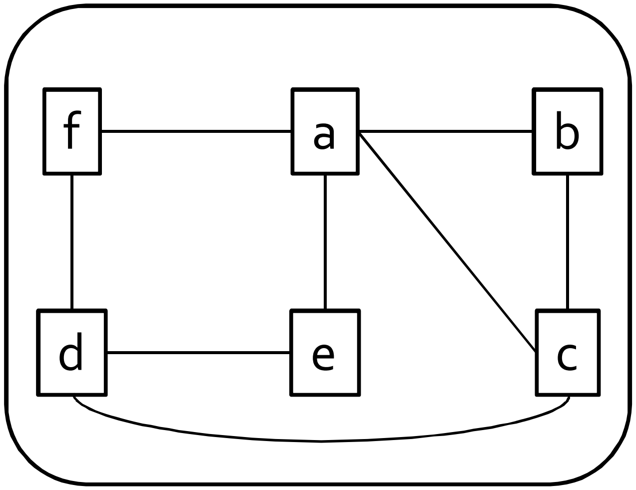
Register Allocation by Graph Coloring

- 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
  - bad news: problem is NP-complete
  - good news: there are pretty good heuristic approaches

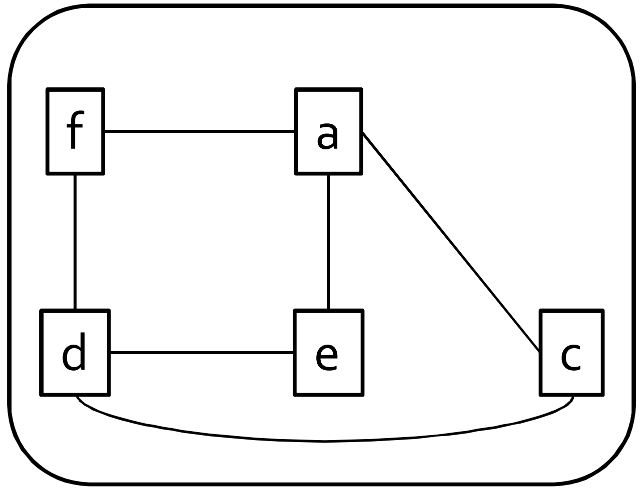
Heuristic Graph Coloring

- 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 the least connected node
  - remove least connected node from the graph
  - color the reduced graph recursively
  - re-attach the least connected node

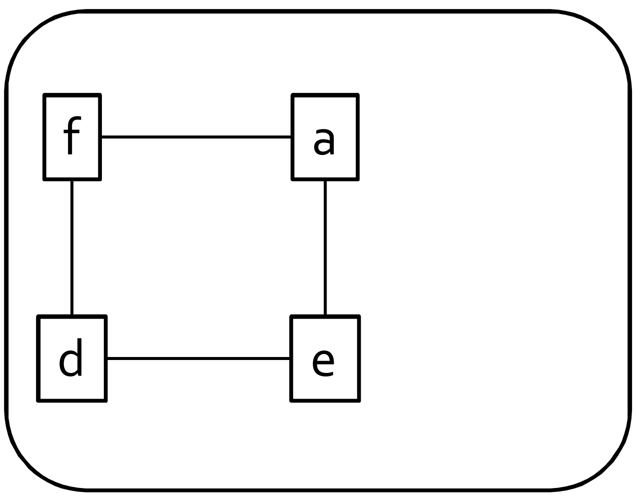
Heuristic Graph Coloring



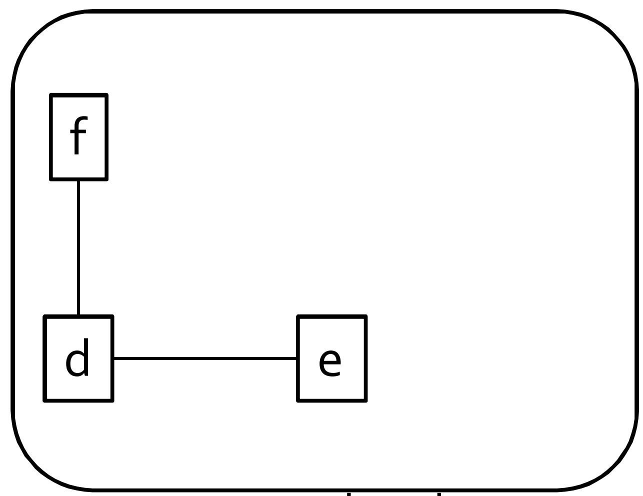
stack: ε



stack: b

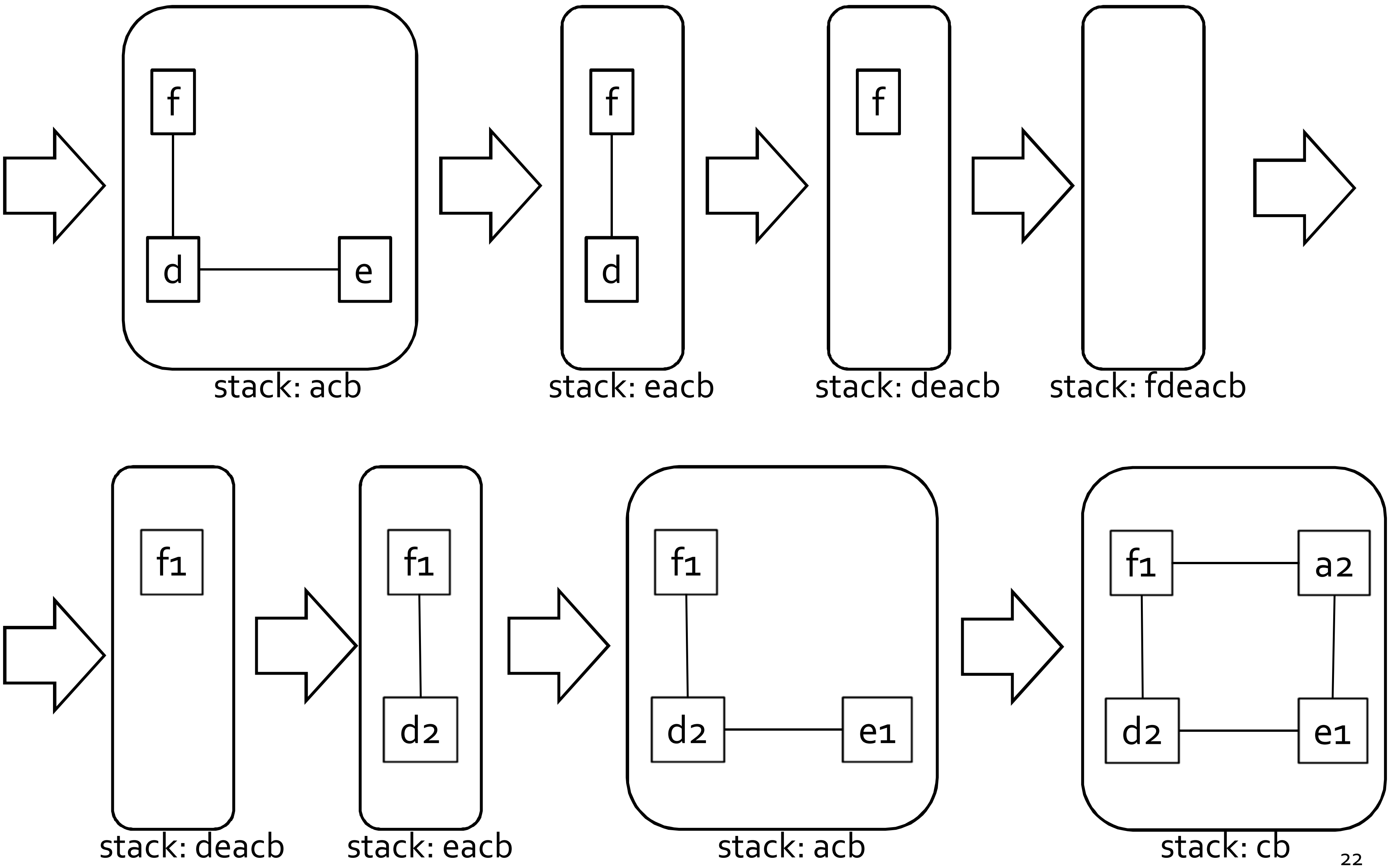


stack: cb

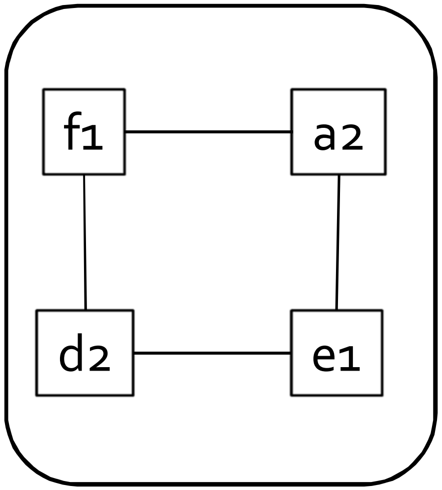


stack: acb

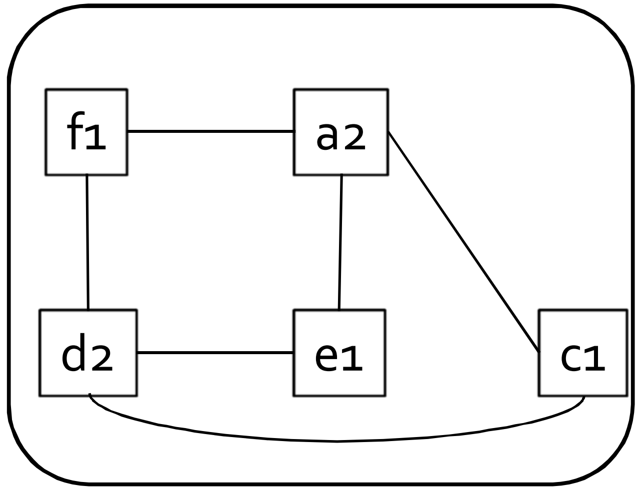
Heuristic Graph Coloring



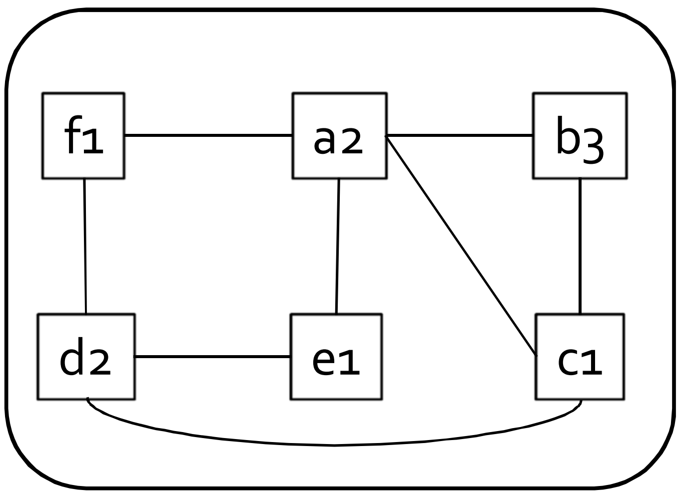
Heuristic Graph Coloring



stack: cb



stack: b



stack: ε

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

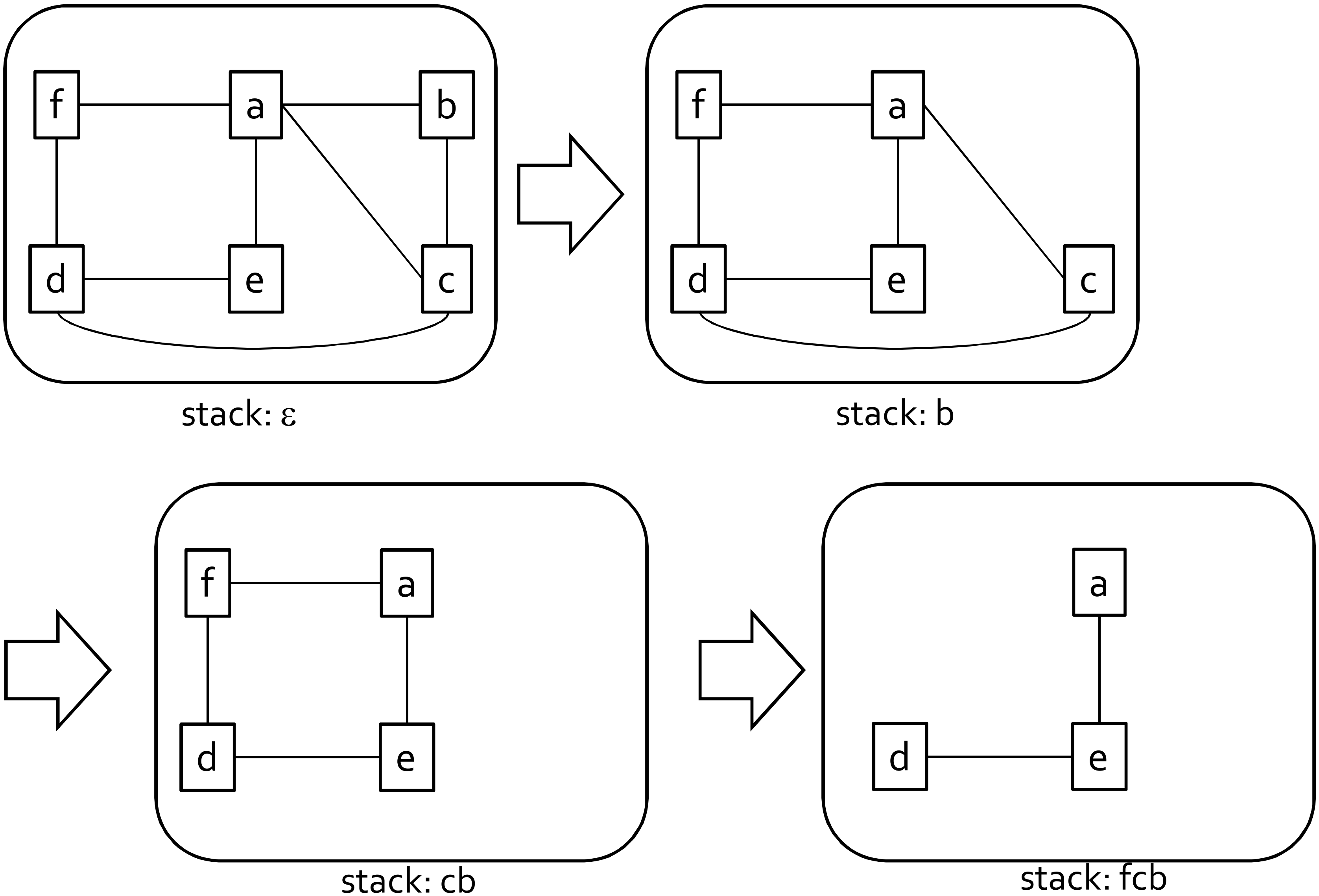
Supercompilation

- exhaustive search in the space of (small) programs for finding optimal code sequences
  - often counter intuitive results, not what a human would write
  - can be very efficient
- generate/test paradigm

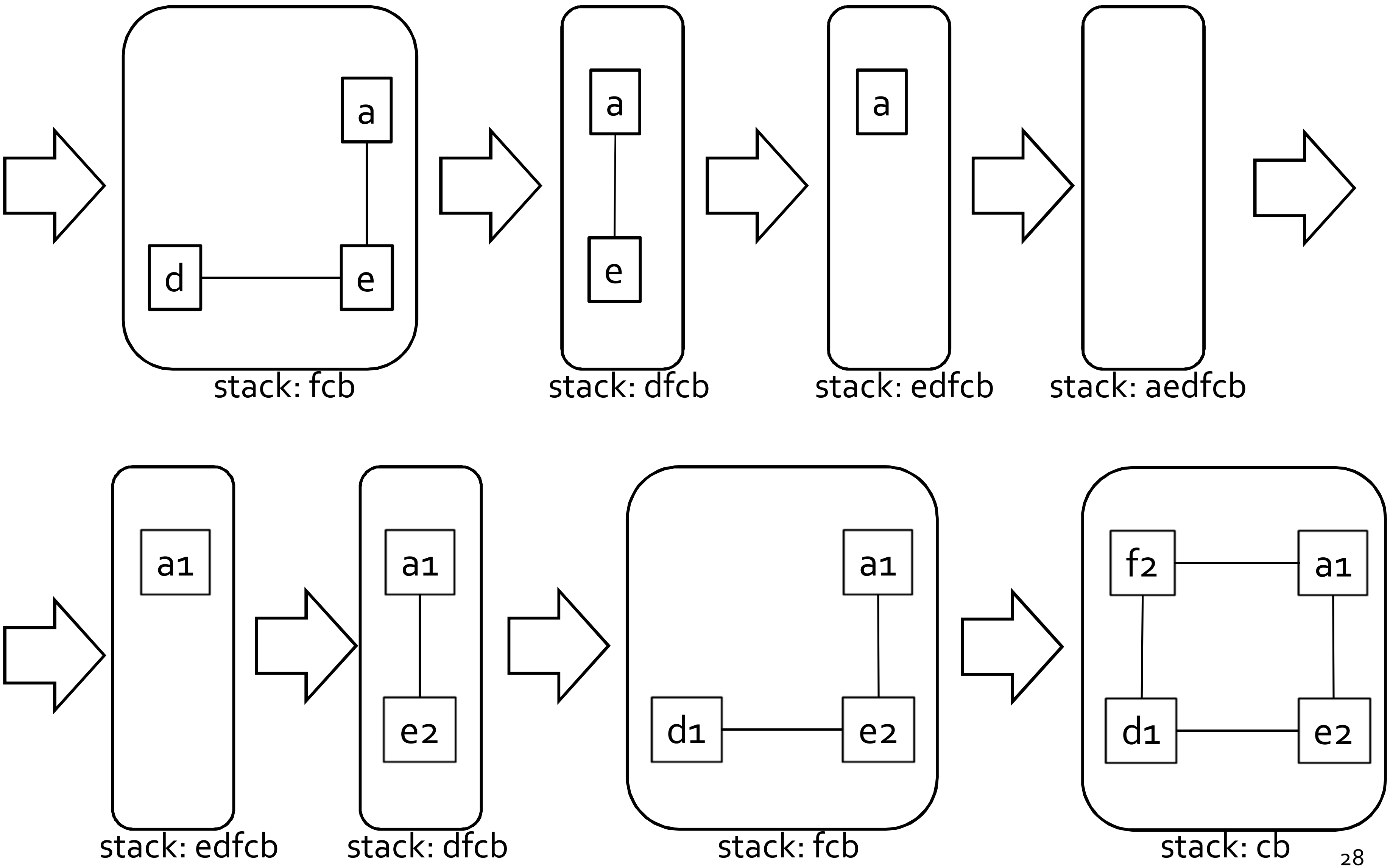
```
; n in register %ax
  cwd              ; convert to double word:
                   ;   (%dx,%ax) = (extend_sign(%ax), %ax)
  negw %ax         ; negate: (%ax,cf) = (-%ax,%ax != 0)
  adcw %dx,%dx     ; add with carry: %dx = %dx + %dx + cf
; sign(n) in %dx
```

The End

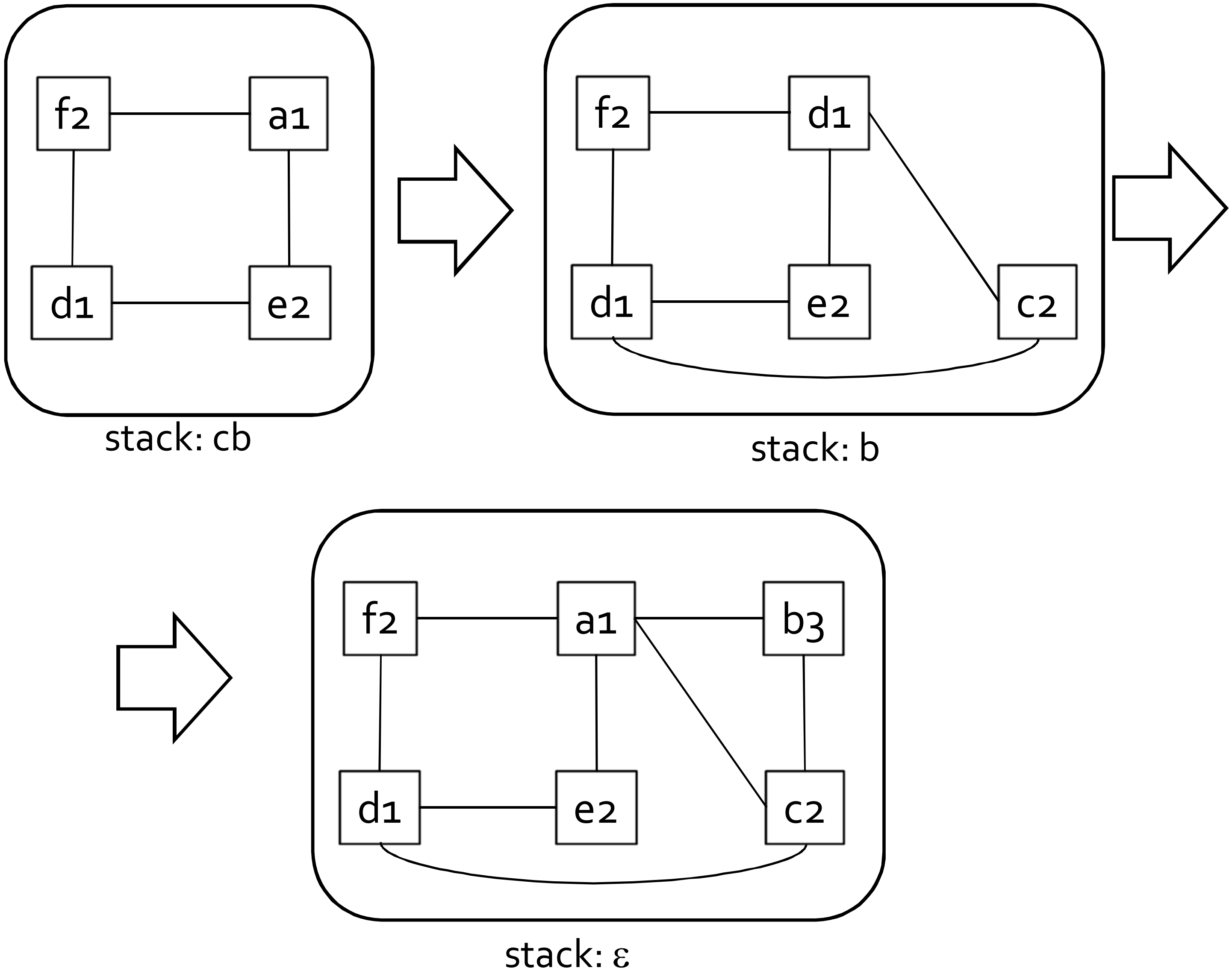
Heuristic Graph Coloring



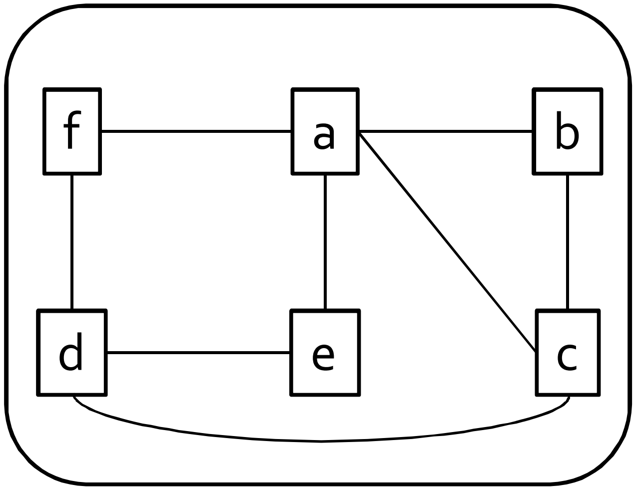
Heuristic Graph Coloring



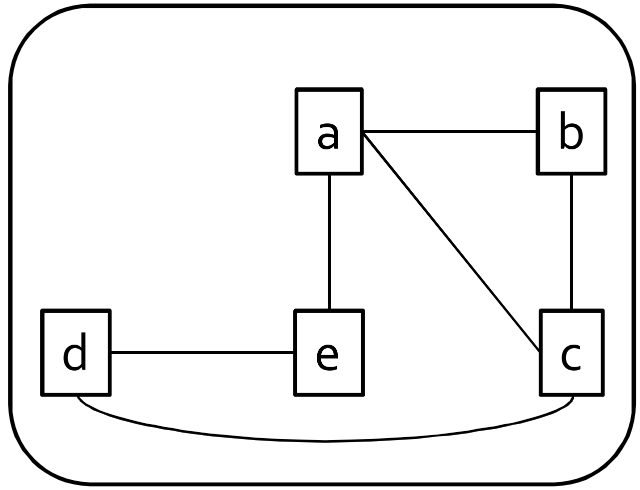
Heuristic Graph Coloring



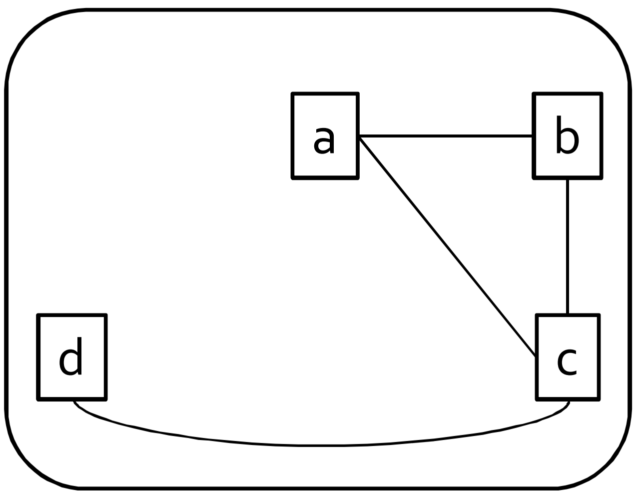
Heuristic Graph Coloring



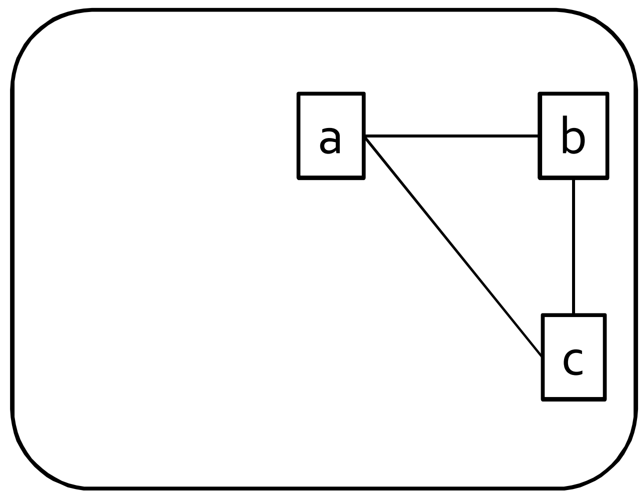
stack: ε



stack: f

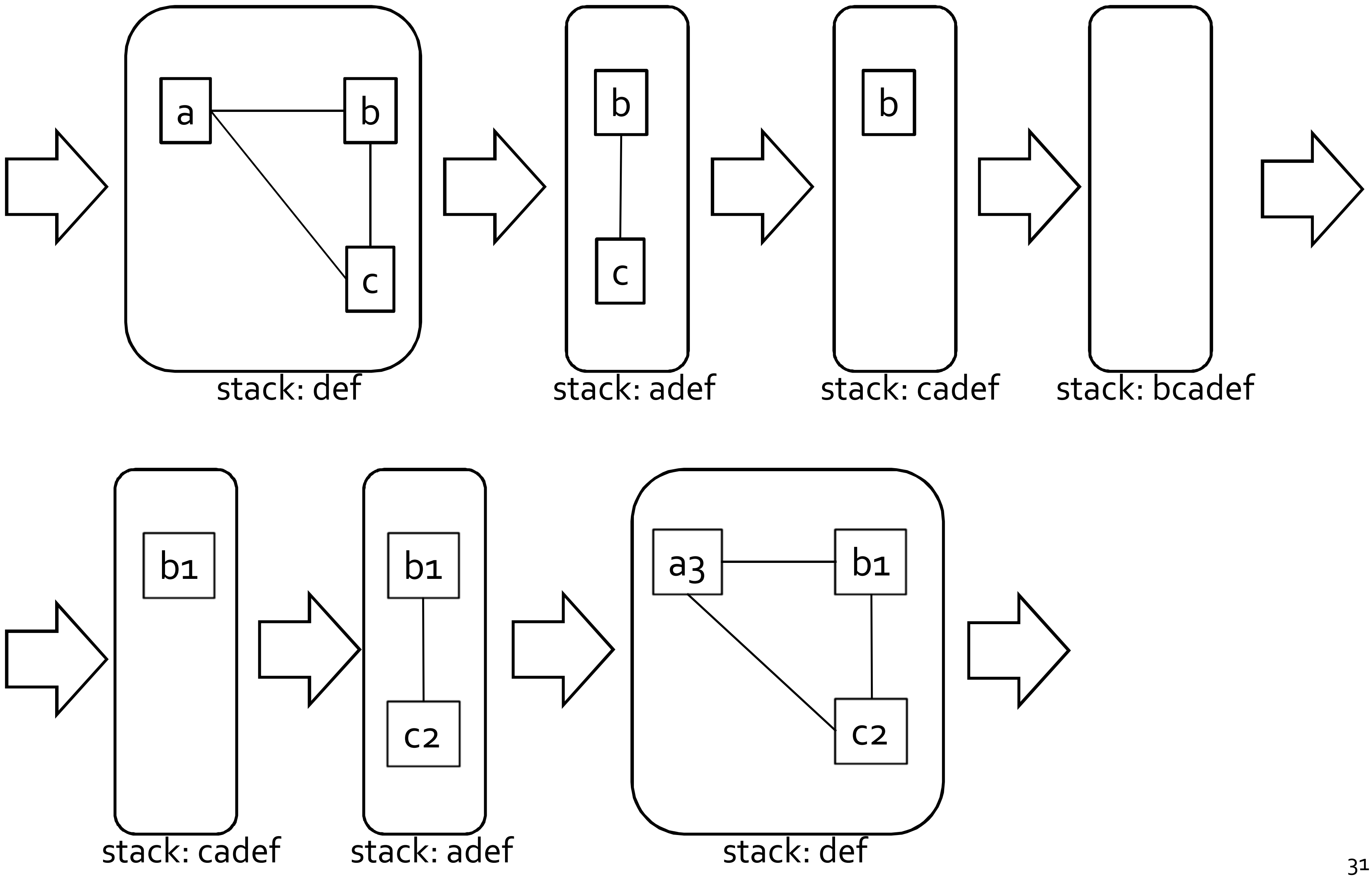


stack: ef

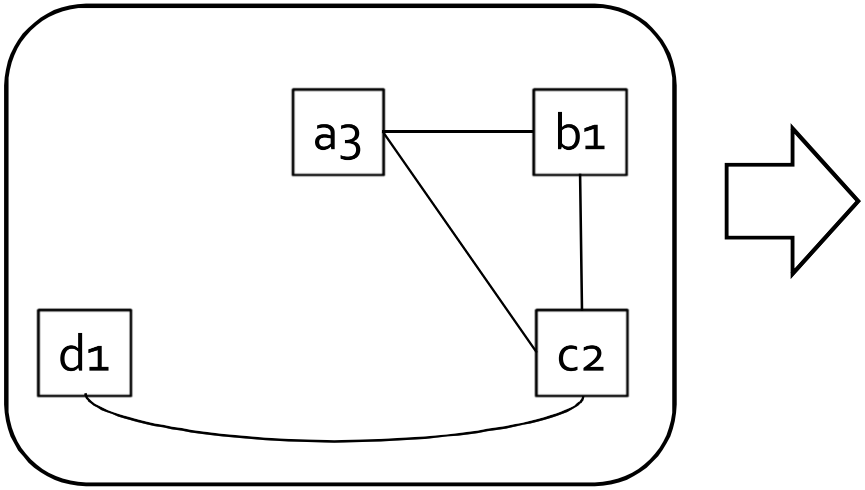


stack: def

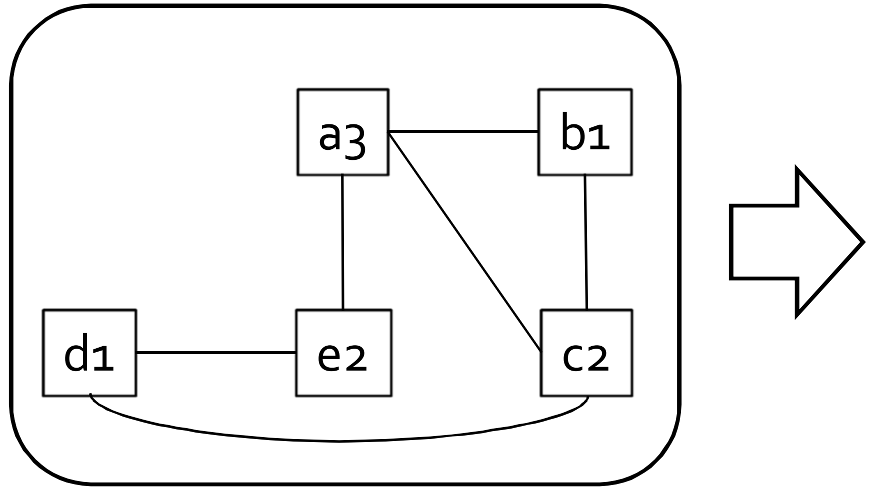
Heuristic Graph Coloring



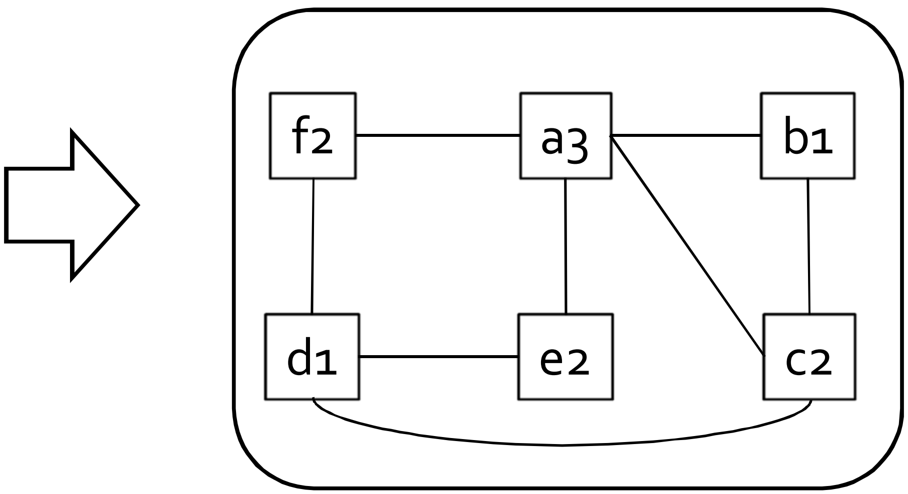
Heuristic Graph Coloring



stack: ef



stack: f



stack: ε