Compilation and Program Analysis(#7):

Register Allocation + Data Flow Analyses

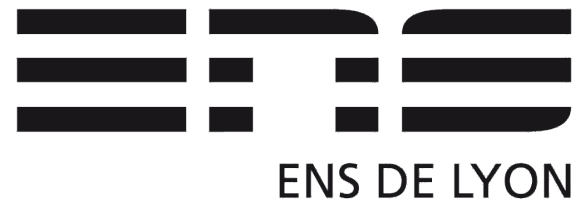
Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

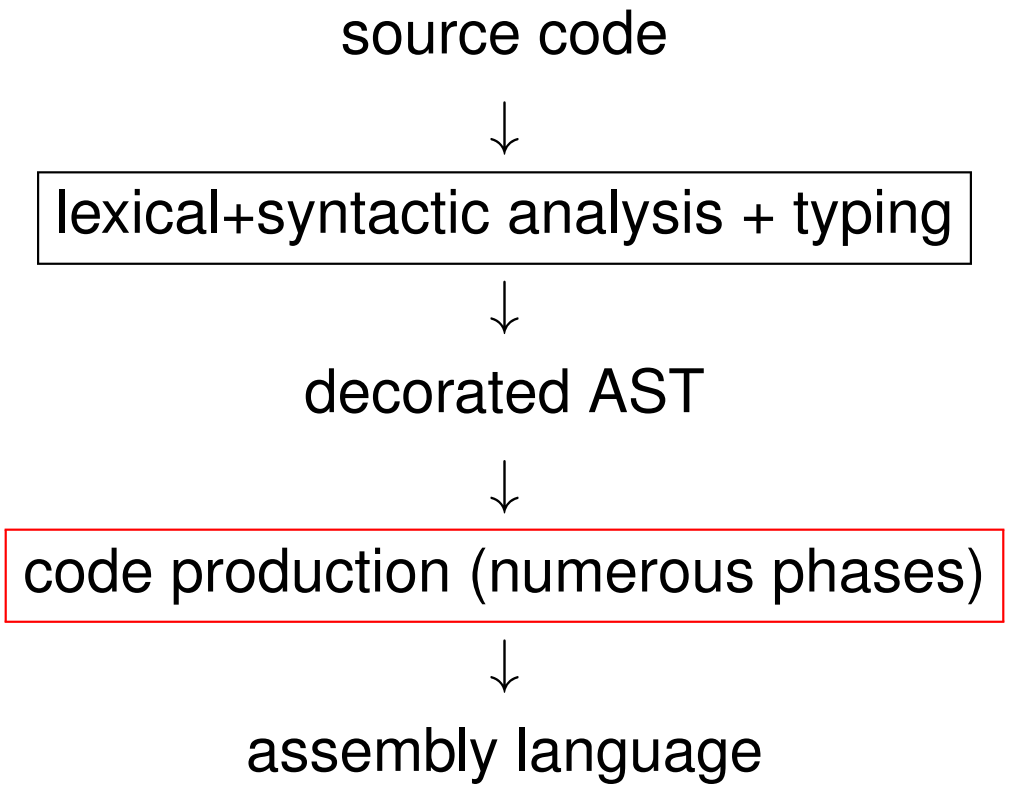
Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

2018-2019



Where are we ?



- We work on IRs (Middle-end).

Credits

Fernando Pereira's course on register allocation :

`http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/RegisterAllocation.pdf`

What for ?

- Finding storage locations to the values manipulated by the program ▶ registers or memory.
- registers are fast but in small quantity.
- memory is plenty, but slower access time.

▶ A good register allocator should strive to keep in registers the variables used more often.

"Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important - if not the most important - of the optimizations."

Hennessy and Patterson (2006) - [Appendix B; p. 26]

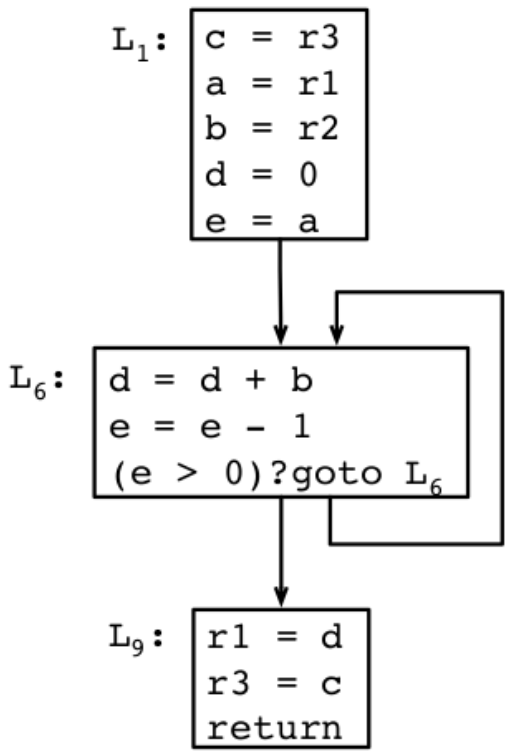
What for ?

Expected behavior of **register allocation** :

- Input : a CFG with basic blocks with 3-address code (and pseudo-registers, aka temporaries)
- Output : same CFG but without pseudo-registers :
  - replace with physical registers as much as possible.
  - if not **splill**, ie allocate a place in memory.
  - all copies assigned to the same physical registers (“moves”) can be removed : **coalescing**

Register constraints

Some variable are assigned to some specific registers (compiler, architecture constraints)



- r1,r2,r3 are used to pass function arguments here.

The key notion : liveness

Observation

Two variables that are simultaneously **alive** must be assigned different registers.

(formal definition of alive follows)

Register assignment is NP-complete

Theorem

Given P and K general purpose registers, is there an assignment of the variables P in registers, such that (i) every variable gets at least one register along its entire live range, and (ii) simultaneously live variables are given different registers ?

Gregory Chaitin has shown, in the early 80's, that the register assignment problem is NP-Complete (register allocation via coloring, 1981)

3-phase algorithm

- **Liveness analysis**
  - When is a given value necessary for the rest of the computation ?
- **Interference graph**
  - A graph that encodes which pseudo-registers cannot be mapped to the same location.
- **Graph coloring** then register allocation.
  - The effective allocation : physical registers and stack allocation for pseudo-registers.

Liveness analysis

In the sequel we call **variable** a pseudo-register or a physical register.

Definition (Alive Variable)

In a given program point, a variable is said to be alive if the value she contains may be used in the rest of the execution.

May : non decidable property ▶ overapproximation.

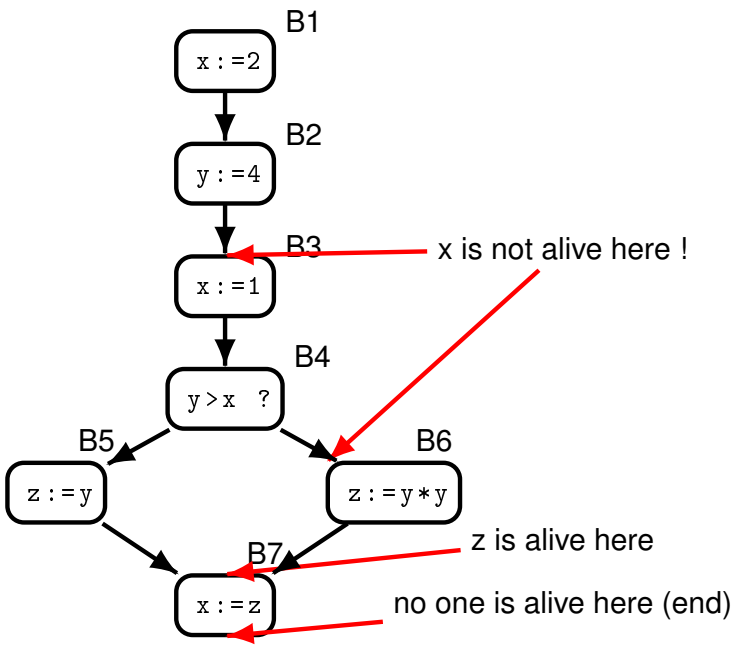
Important remark : here a block = a statement/program point.
We have the same kind of analyses with block=basic block.

An example for live ranges

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

```
x:=2;
y:=4;
x:=1;
if (y>x)
   then z:=y
   else z=y*y ;
x:=z;
```



- The information flow is **backward** : from uses to definitions.

Data flow expressions

Definition

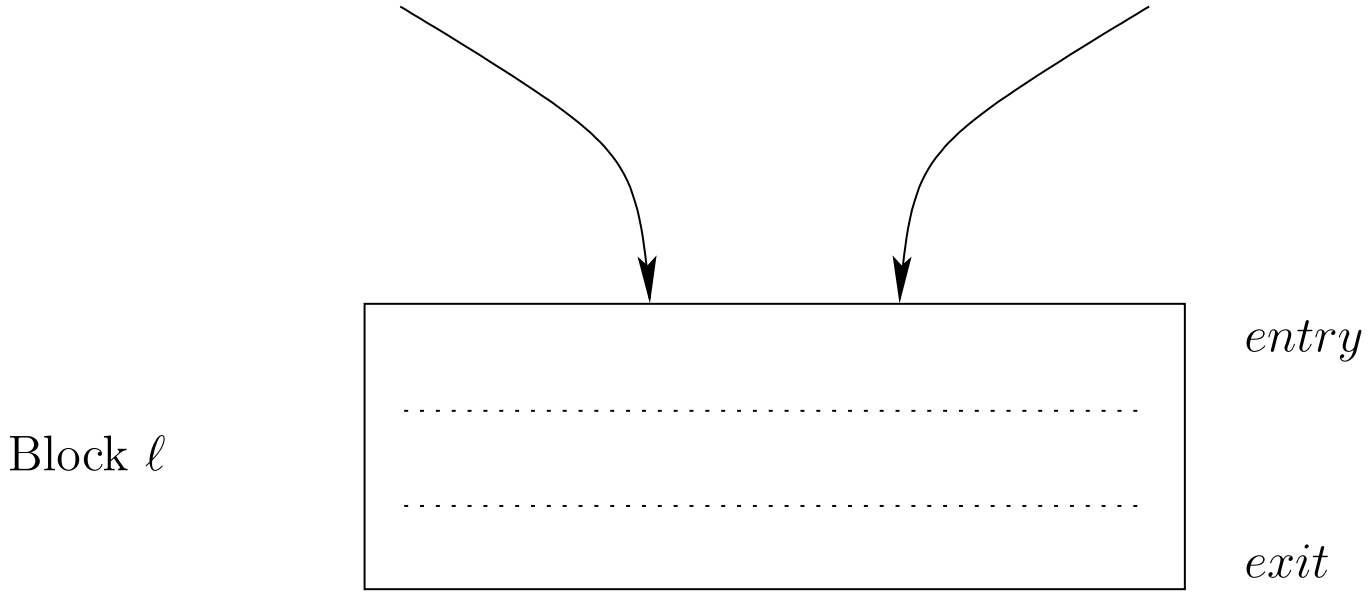
A variable that appears on the left hand side of an assignment is **killed** by the block. Tests do no kill variables.

Definition

A **generated** variable is a variable that appears in the block.

- Sets : $kill_{LV}(block)$ and $gen_{LV}(block)$

Data flow expressions



$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = final \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

Data flow equation : solving

Here :

- Initialise LV sets to $\emptyset$.
- Compute LV_{entry} sets, then LV_{exit}, and continue.
- Stop when a fix point is reached.

▶ (vector of) Sets are strictly growing, and the live range set is at most the set of all variables, thus **this algorithm terminates**.

Steps

$LV_{entry}(\ell)$ denoted by $In(\ell)$, $LV_{entry}(\ell)$ by $Out(\ell)$ initilisation to emptysets is not depicted.

			Step 1		Step 2		Step 3 (stable)
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$
1	$\{x\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{y\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{y\}$	$\emptyset$
3	$\{x\}$	$\emptyset$	$\emptyset$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$
4	$\emptyset$	$\{x,y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$
5	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$	$\{z\}$	$\emptyset$	$\{z\}$	$\emptyset$	$\{z\}$

Final result and use

Backward analysis and we want the smallest sets, here is the final result : (we assume all vars are dead at the end).

ℓ	$LV_{entry}(\ell)$	$LV_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\emptyset$

- Use : Dead code elimination ! Note : can be improved by computing the use-defs paths. (see Nielson/Nielson/Hankin)

Common subexpressions / Available expressions

Avoiding the computation of an (arithmetic) expression :

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```

AE : genesis of the analysis

An expression is available at a control point if its current value has already been computed earlier in the execution :

- Sets of expressions.
- How does information originate ? from a use that do not redefine any operand !
- How does information propagate ? from top to bottom.
- How do we join info after tests ?

Some defs

Definition

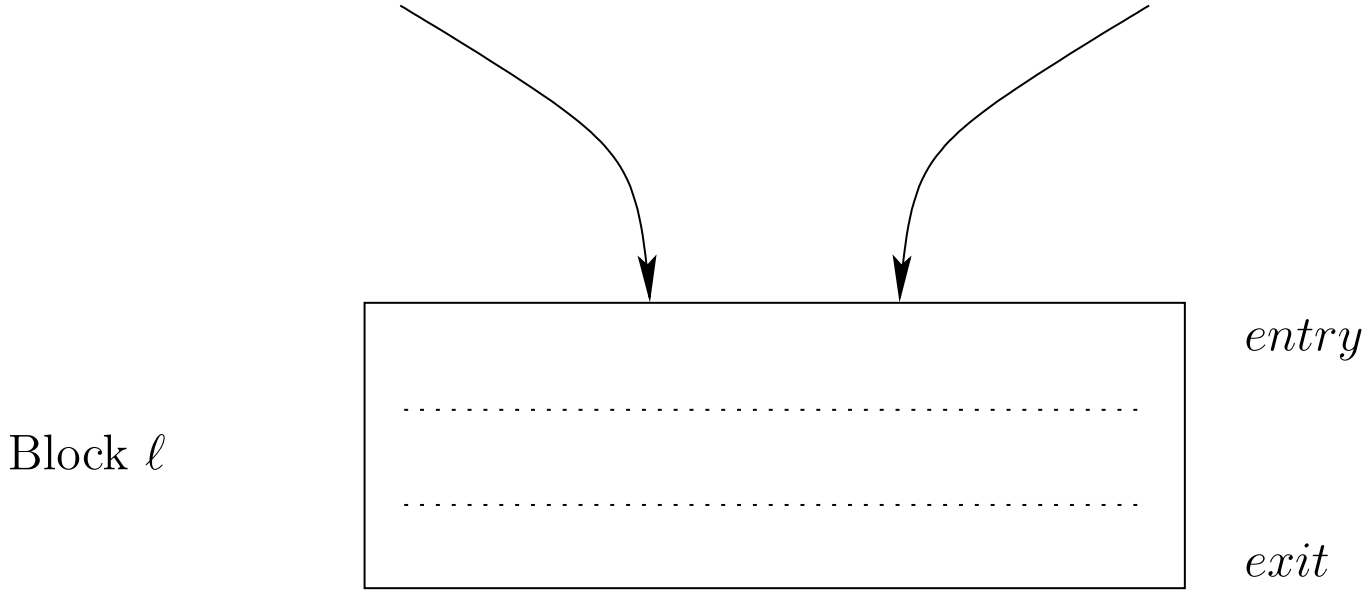
An expression is **killed** in a block if any of its variables is defined in the block.

Definition

A **generated** expression is an expression evaluated in the block and none of its variables is killed in the block.

- Sets : $kill_{AE}(block)$ and $gen_{AE}(block)$

Data flow expressions

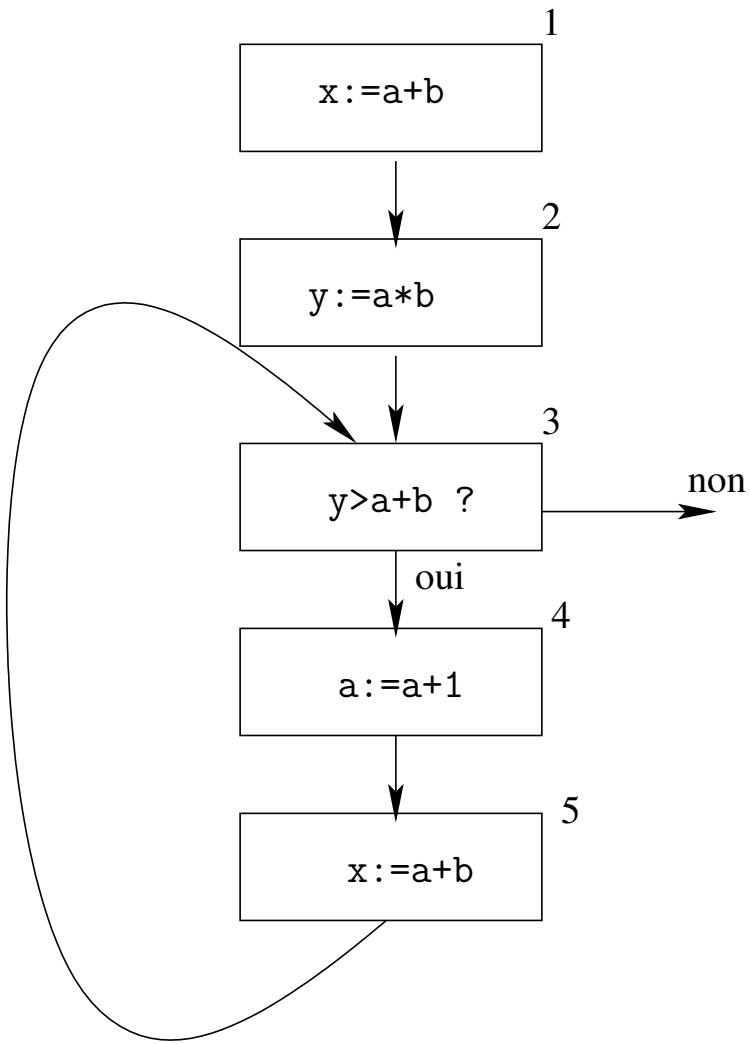


$$AE_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = init \\ \bigcap\{AE_{exit}(\ell') | (\ell', \ell) \in flow(G)\} & \end{cases}$$

$$AE_{exit}(\ell) = \left(AE_{entry}(\ell) \backslash kill_{AE}(\ell)\right) \cup gen_{AE}(\ell)$$

On the example - equations

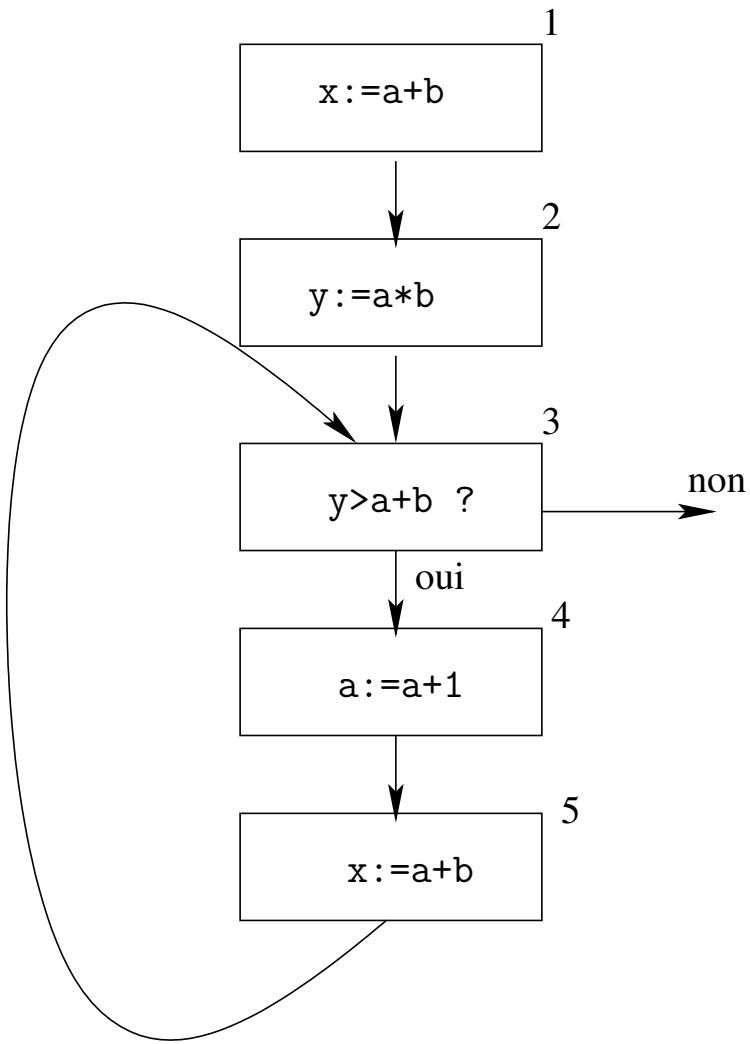
ℓ	$kill_{AE}(\ell)$	$gen_{AE}(\ell)$
1	$\emptyset$	{a+b}
2	$\emptyset$	{a*b}
3	$\emptyset$	{a+b}
4	{a+b, a*b, a+1}	$\emptyset$
5	$\emptyset$	{a+b}



On the example - final solution

ℓ	$AE_{entry}(\ell)$	$AE_{exit}(\ell)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

- a+b is available on entry to the loop, not a*b
- Improvement of code generation



Digression : common points

- Computing growing sets from $\emptyset$ via *fixpoint iterations*. (or the dual)
- Sets of equations of the form (collecting semantics) :

$$(\ell) = \bigcup_{(\ell',\ell)\in E} f((\ell'))$$

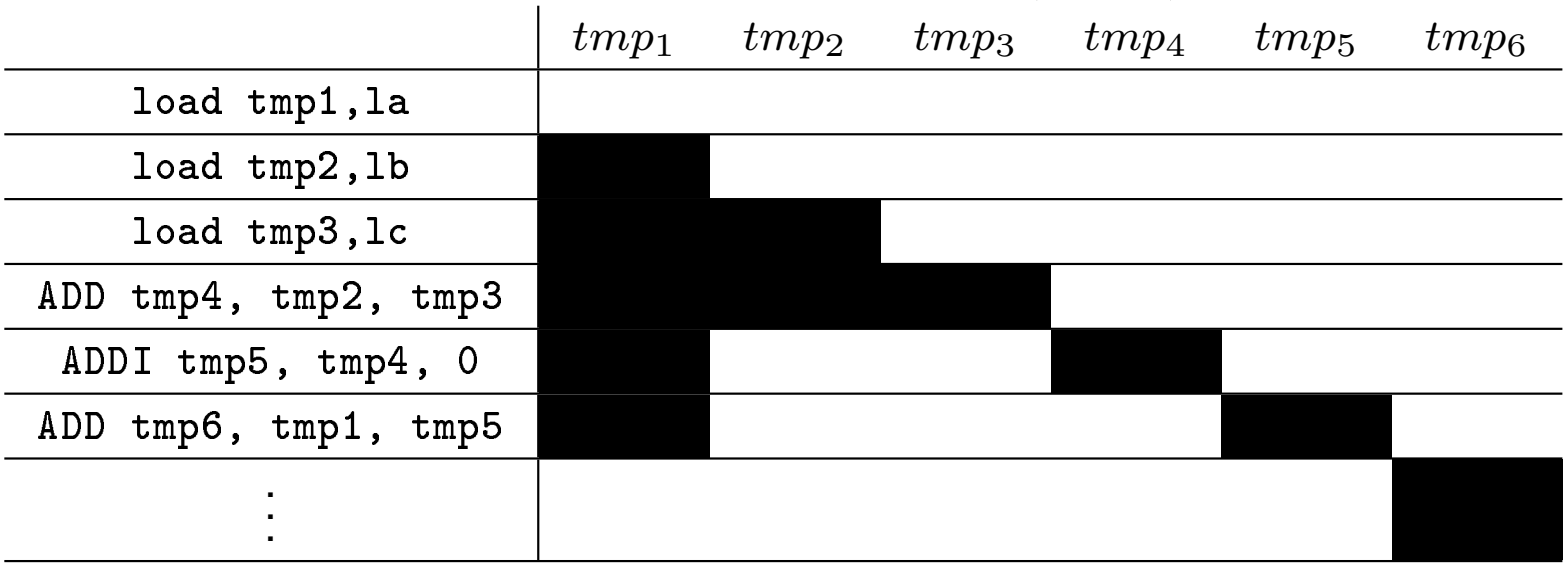
where f is computed w.r.t. the *program statements*.

- is an **abstract interpretation** of the program (see the course on Abstract Interpretation, later)

See also : Kam, J. B. and J. D. Ullman, "Monotone Data Flow Analysis Frameworks", Actal Informatica 7 :3 (1977), pp. 305-318.

Interference

The liveness analysis gives us for $a + (b + c)$:

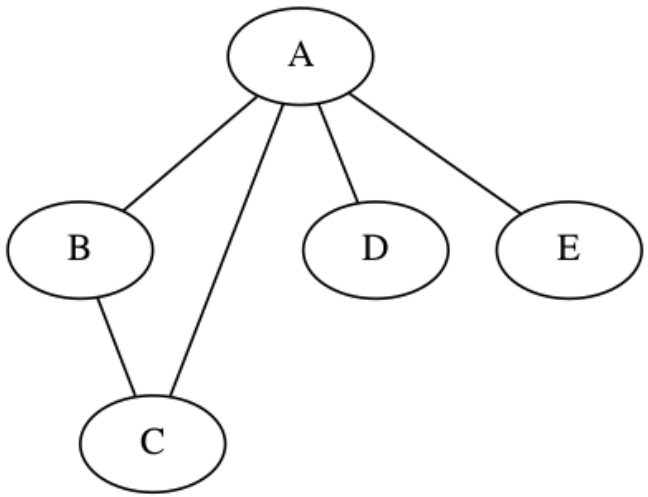


- `tmp1` is in conflit with `tmp2` (because of instruction 3) denoted by $tmp_1 \bowtie tmp_2$.

Important remark : technically, `ADD tmp5, tmp4, 0` is a **move instruction**

Interference graph

A denotes `tmp1`, ... $\bowtie$ defines a graph :

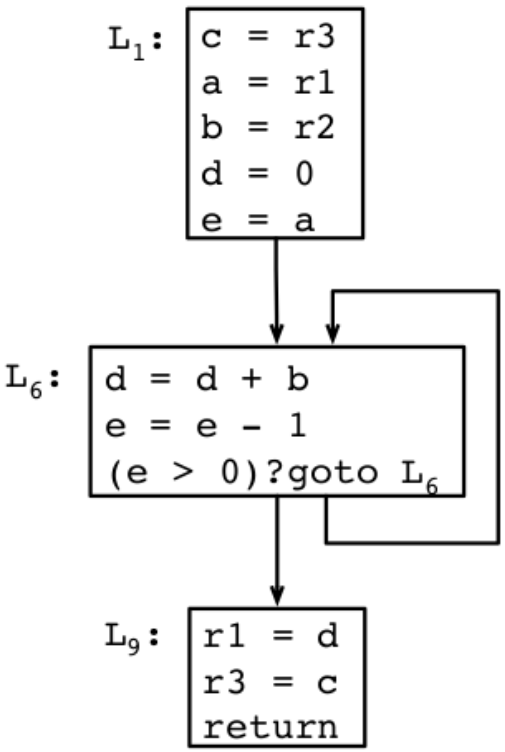


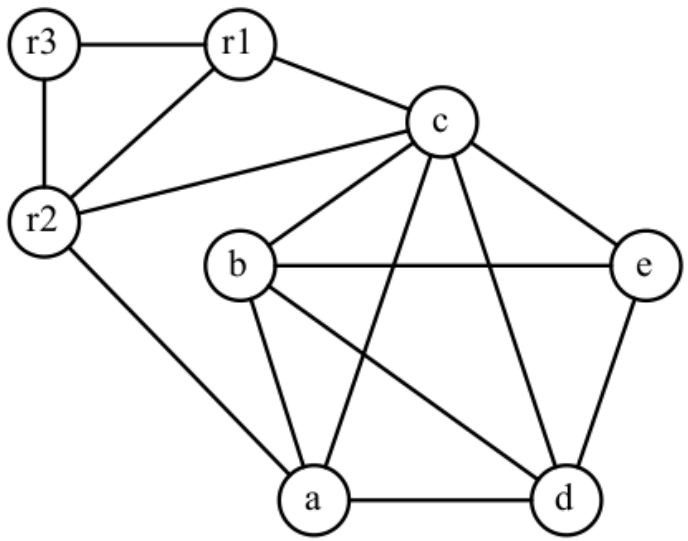
We want a **correct allocation** with respect to $\bowtie$:

$tmp_1 \bowtie tmp_2 \implies Alloc(tmp_1) \neq Alloc(tmp_2)$.

- Graph coloring.

Running example





Kempe's simplification algorithm 1/2

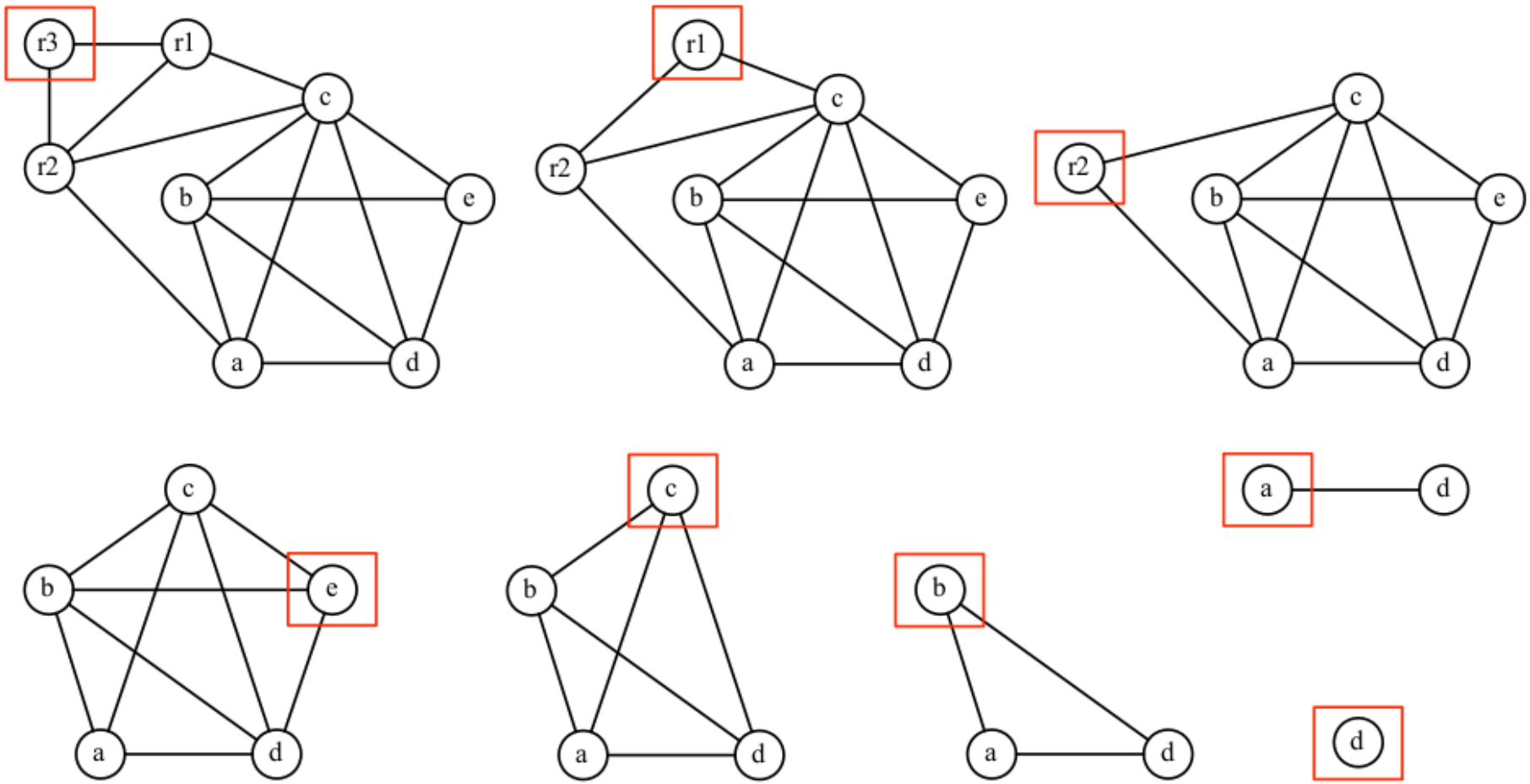
On the interference graph (without coalesce edges) :

Proposition (Kempe 1879)

Suppose the graph contains a node m with fewer than K neighbours. Then if $G' = G \setminus \{m\}$ can be colored, then G can be colored as well.

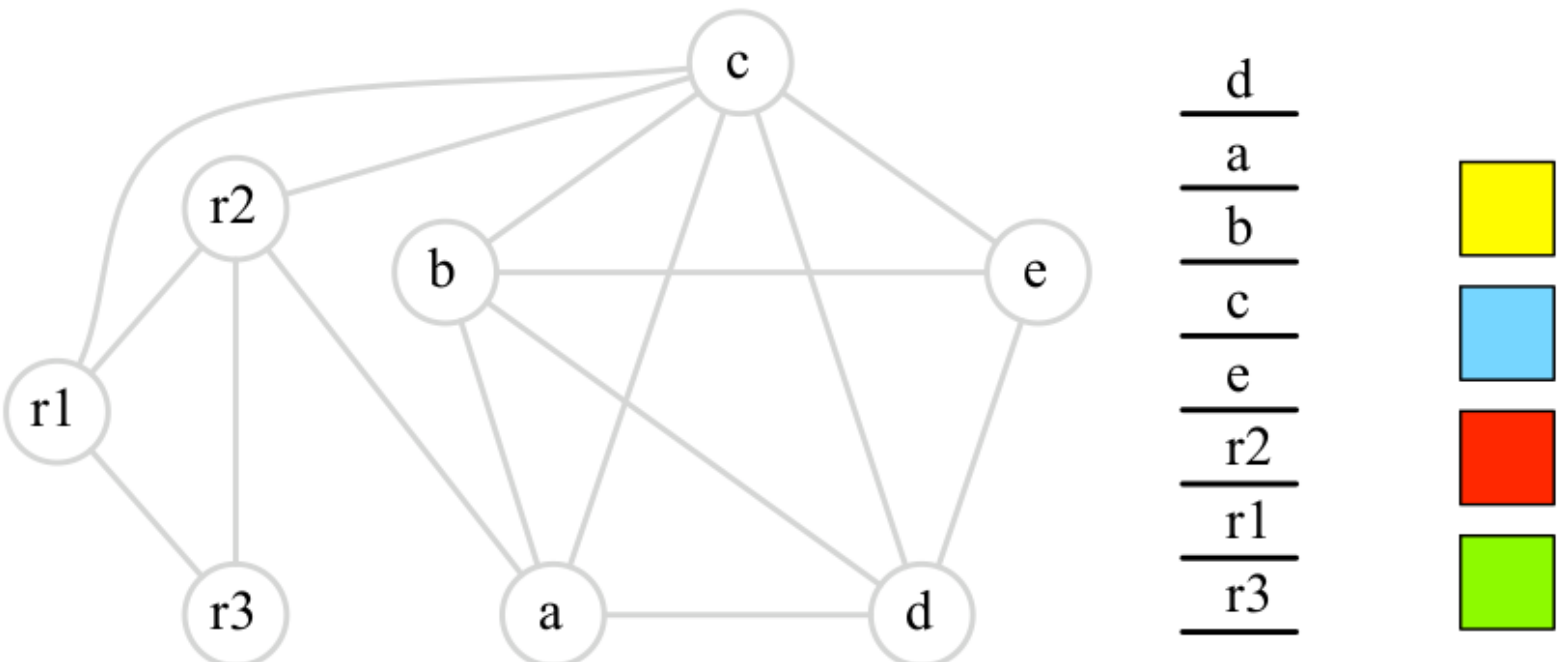
- Pick a low degree node, and remove it, and continue until remove all (the graph is K-colorable) or . . .

Kempe's simplification algorithm 2/2

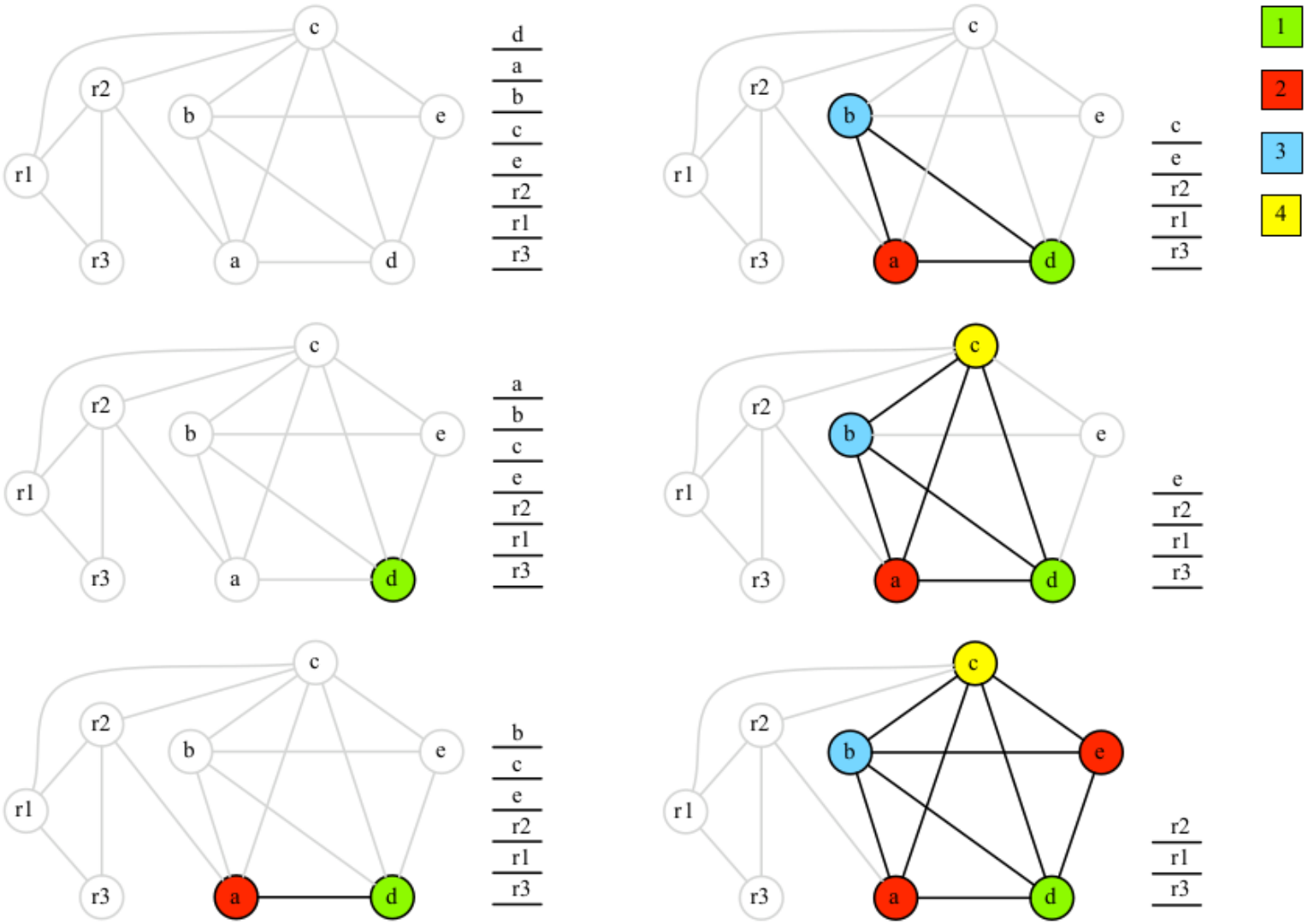


Let's color !

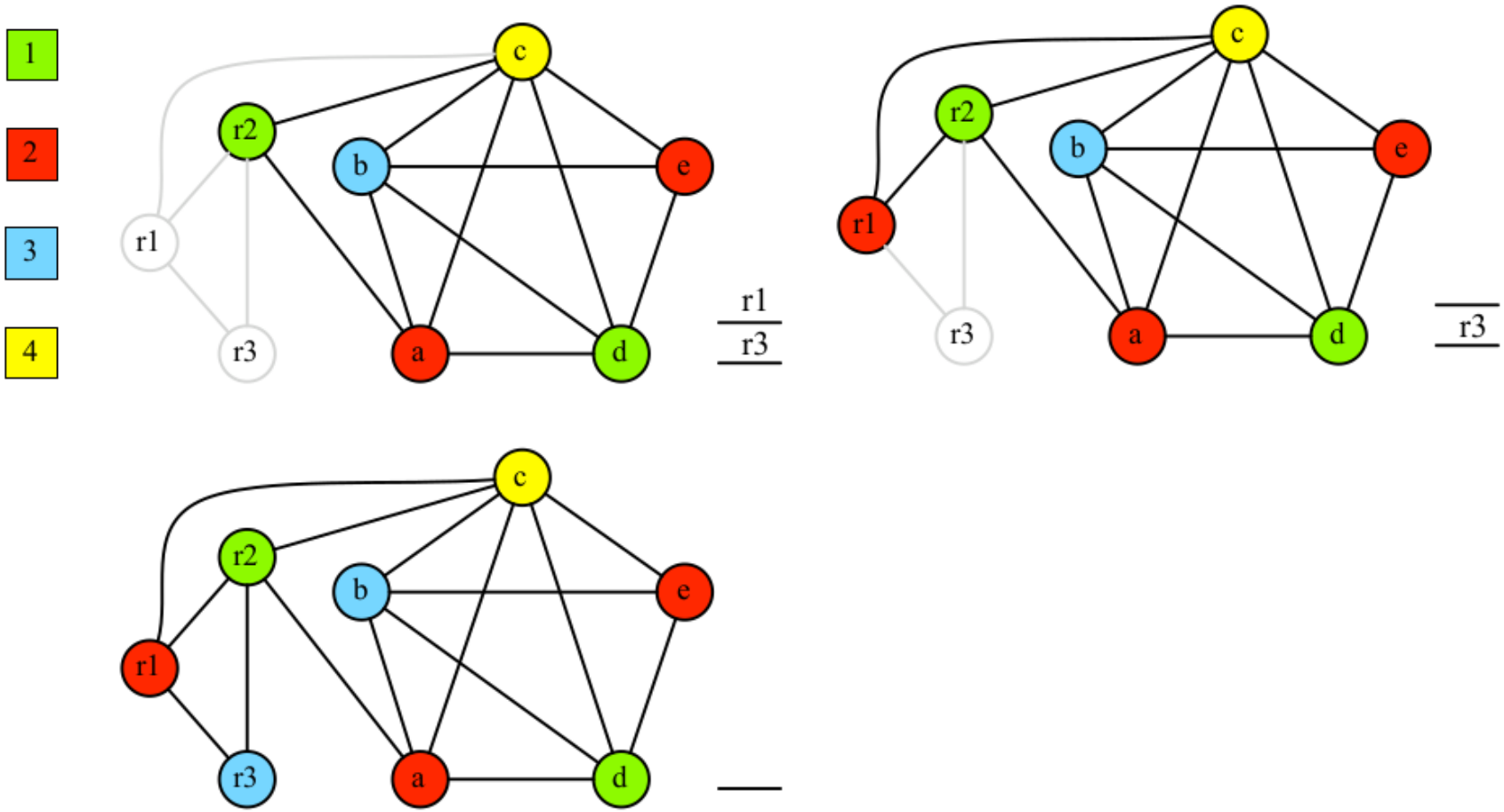
- We assign colors to the nodes greedily, in the reverse order in which nodes are removed from the graph.
- The color of the next node is the first color that is available, *i.e.* not used by any neighbour.



Greedy coloring example 1/2



Greedy coloring example 2/2



If the graph is not colorable

Non-colored variables are named **spilled pseudo-registers**.

Idea : Modify the code to lower the number of simultaneously alive registers. Plenty of solutions, the simplier is to reserve a *dedicated place for a given spilled variable*, and store and load from memory :

```
ADD temp5, temp4, temp3
...
ADD temp6, temp5, #5
```

becomes :

```
ADDINMEMORY [placefortemp5], temp4, temp3
...
ADDxx         temp6, [placefortemp5], #5
```

But we do not have this kind of instruction in our machine !

One solution for non-colored variables

We invent 2 versions of the same variable (**live-range splitting**), and modify the code into :

```
ADD temp51, temp4, temp3
ST temp51 [placeinmemory]
..
LD temp52 [placeinmemory]
ADD temp6, temp52, #5
```

- But now we have to allocate these two new variables !

We relaunch the coloring algorithm. This is called iterative register coloring. (see Exercise Sheet 6b).

Second solution

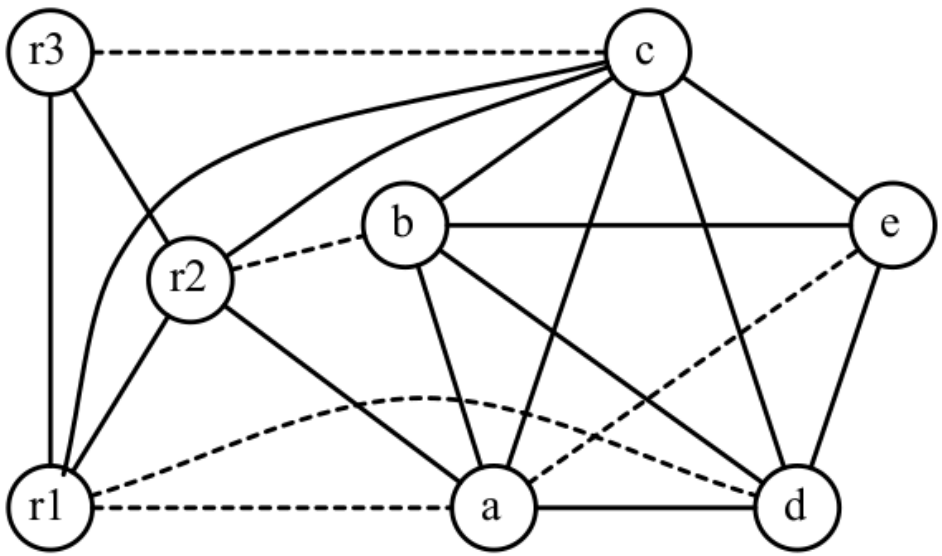
- Remove all colored variables from the graph.
- Relaunch the coloration algorithm with an infinite number of colors.
- Use these colors to compute offsets in the stack.

Physical Memory Allocation

We will invent physical memory places from the stack pointer (like in Lab 4).

Other Algorithms

- **Linear scan** : greedy coloring of interval graphs. (see Fernando Pereira's slides on register allocation : 18 to 35)
- **Iterative Register Coalescing** (George/Appel, TOPLAS, 1996) (same, from slides 44), which uses "coalesce edges" (variables are related by move instructions).
- Plenty of other heuristics for splilling.



A nice result

Chordal graphs are P-colorable

For certain classes of graphs, graph coloring is P. This is the case for **cordal graphs** where every cycle with 4 or more edges has a chord (connects 2 vertices in the cycle but not part of the cycle).

Important result (Sebastian Hack) : Programs in strict SSA form have this property.

- Pereira Palsberg Register allocation (APLAS 2005).