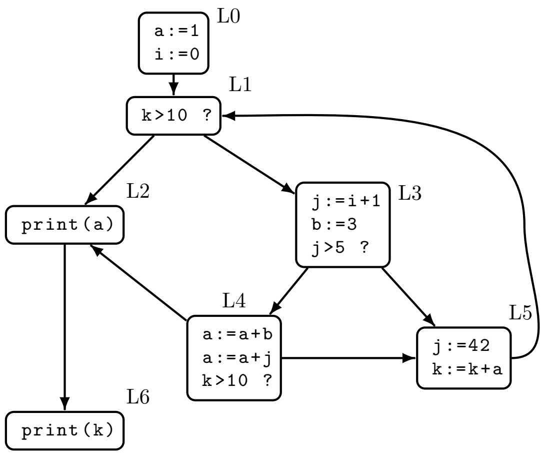
Exercise session 5

CFG, register allocation, code generation

5.1 Liveness

EXERCISE #1 ► **Liveness by hand - CC 2016**

On donne un graphe de flot, et on rappelle qu'*une variable est vivante en sortie d'un bloc si il existe un chemin partant de ce bloc qui mène à une utilisation de cette variable sans redefinition de cette variable.*



bloc	variables vivantes en sortie du bloc
L0	
L1	
L2	
L3	
L4	
L5	
L6	$\emptyset$

1. Sans calcul, remplir le tableau avec les variables vivantes en sortie de chaque bloc.
2. En déduire le code mort.

EXERCISE #2 ► **Constant propagation**

In this exercice we will study the following program:

```
z:=3
x:=1
while (x>0) {
  if (x=1) then
     y:=7;
  else
     y:=z+4;
  x:=3
  print y
}
```

1. Draw the CFG where blocks are statements.
2. Which of variables x,y,z are constants ? What modification can be done on the code if we know this information?
3. How is a "i'm a constant" information generated ? killed ? Is it a forward or backward dataflow propagation?
4. How to merge two incoming informations $x \mapsto 1$ and $x \mapsto 2$ in a test ?
5. Give an algorithm to compute all the constants available at the begining of each block.

5.2 Register Allocation

EXERCISE #3 ► **Code production**

(Skip if you already made this exercise in the previous exercise session).

code	t1	t2	t3	t4	t5	t6	t7	t8	t9
`ldr t1,[n]`									
`ldi t2,1`									
`add t3,t1,t2`									
`ldr t4,[n]`									
`mul t5,t4,t3`									
`ldi t6 2`									
`ldr t7,[n]`									
`mul t8,t6,t7`									
`add t9,t5,t8`									
`...`									

1. Draw the interference graph (nodes are variables, edges are liveness conflicts).
2. Color this graph with three colors using the algorithm of the course (`http://laure.gonnord.org/pro/teaching/MIF08_Compil1617/07-RegisterAlloc.pdf`, slides 27-30).
3. Generate the (final) code.

EXERCISE #4 ▶ **Code production and register allocation**

The three address code generation produced the following code for a given function (the three arguments of the function are given through the help of registers R1,R2,R3, and the result-a pair- is stored in R2,R3):

```
a := r2
b := r3
c := r1
r1 := 5
d := r1
L1: r1 := d
d := d + r1
r1 := d
if d<10 goto L1
r3 := b
r2 := a
return [LIVE-OUT: r2,r3]
```

Figure 5.1: Program to analyse

- For non-spilled variable: replace the temporary with its associated color/register.
- For a spilled variable (say, $temp5$ here, assigned to color 2):
  `ADD temp6 temp1 temp5`
  becomes (we use $R5$ and $R7$ to make load and stores for spilled variables):
  `LDR R5 R6 #-2`
  `ADD alloc(temp6) alloc(temp1) R5`
  where $alloc(temp1)$ denotes the allocation of $temp1$ (if it is a spilled variable, we have to first load its value in $R7$).

Figure 5.2: Final Code generation

1. Give “Live-in” sets of each instruction (without fixpoint computation).
2. Draw the interference graph (nodes are variables, edges are liveness conflicts).
3. Color this graph with $K = 3$ colors using the algorithm seen in the course. Preferably color the register R1 with color 1, R2 with color 2. . . .
4. For spilled variables, recolor the graph with an infinite number of registers.
5. Generate the (final) code with the same algorithm as in Lab#5 (see Figure 5.2)

5.3 Function code generation

EXERCISE #5 ▶ **Function code generation**

Generate the code for the program (use R0 for parameters, R6 for the stack, R7 is reserved for storing the PC):

```
function (x:int) {
    var y:int
    y:=5
    returns (x+y) }
```

```
main(){
    z:=f(12)+1
}
```