The goal of the exercise is to let you implement two different versions of semaphores in Scala (using Pilib) and convince yourselves that they are equivalent, by translating them in $\pi$-calculus and using the ABC tool.

# 1 Implementations

In both implementations we represent a semaphore by an object with two methods: $get$ and $release$. The idea is that a client process will call the method $get$ before entering its critical section and call the method $release$ when exiting its critical section, in order to assure mutual exclusion. Furthermore we want a *counting* semaphore (an earlier $release$ can not unlock a later $get$) with non-blocking $release$.

For convenience we use the class $Signal$ as defined in the course 8:

```
class Signal extends Chan[Unit] with {
    def send = write(());
    def receive = read;
}
```

**First implementation**

The first implementation uses two signals $g$ (get) and $r$ (release) which are used to synchronize calls to methods $get$ and $release$.

- When invoked, methods $get$ and $release$ are waiting for the right to return by sending their corresponding signal.
- There is an external process $Sched$ with two states which will schedule the methods of the semaphore. It can loop in its initial state by repeatedly receiving $r$ or go, by receiving $g$, to a state where it can only accept $r$ to return in its initial state.

**Second implementation**

The second implementation uses two signals $a$ (available) and $na$ (non available) and regards a state of the semaphore as a "soup" of signals.

- A call to $get$ in presence of the signal $a$ in the soup will consume it, add the signal $na$ in the soup and return.
- A call to $release$ in presence of the signal $a$ will consume it, add the signal $a$ and return.
- A call to $release$ in presence of the signal $na$ will consume it, add the signal $a$ and return.

Remark: we add a signal $s$ to the the soup by performing

```
spawn<s.send>
```

$Q_1$: *Write both implementations in Scala using Pilib and test them quickly*

## 2 Testing equivalence using the bisimulation checker

Now we would like to prove that the two implementations are equivalent by showing that their translations to the $\pi$-calculus are bisimilar.

The standard way of translating an object with methods $m_i$ to the $\pi$-calculus is to define one recursive agent per method and put them in parallel as in the following schema:

```
...
M_i = m_i(done).(M_i | Body_i)
...
O = (M_1 | ... | M_n)
```

When a process wants to call method $m_i$ it sends on channel $m_1$ a channel $done$ and waits on this channel. The body of the method returns to the caller by sending the result on this channel. Such a channel is called a *continuation*.

$Q_2$: *Translate both implementations in $\pi$-calculus in the format of the ABC tool.*

$Q_3$: *Try to check that the two resulting $\pi$-processes are bisimilar using ABC. What happens ?*

The problem is that both implementations are infinite state.

$Q_4$: *Could you explain why ?*

In such a case the tool can loop forever. The fact that checking does not terminate is a good sign but not really convincing...

## 3 Getting finite processes

So we will try to work with finite processes. There are essentially two solutions:

- Either keeping the implementations as they are and testing explicit use of the semaphores by a same process which will call only a finite number of times the methods $get$ and $release$.
- Or modifying the implementations in order to limit the number of times we can call methods $get$ and $release$, and testing directly the new implementations (hint: instead of using only one agent for each method use a finite family of agent indexed by a integer).

$Q_5$: *Implement and test both of these techniques.*