# INGI1131 Practical Exercises
# Lab 7: More laziness and the limitations of declarative concurrency

This is the last lab session purely dedicated to declarative concurrency, and we will try to explore the limitations of it. We start by reviewing the questions of the mid-term exam, and then we will continue with more exercises on laziness. Finally, we will explore the limitations of declarative concurrency. For this lab we have an extra time limitations: you can only work on the questions of the exam during the first hour. After that, you have to carry on with the other exercises of the lab.

1. Consider the following program:

```
fun lazy {Ints N} N|{Ints N+1} end

fun lazy {Sum2 Xs Ys}
   case Xs#Ys of (X|Xr)#(Y|Yr) then (X+Y)|{Sum2 Xr Yr} end
end

S=0|{Sum2 S {Ints 1}}
```

   Answer the following questions about this program:

   - What is displayed when the statement {`Browse S.2.2.1`} is executed?
   - Give a function of $i$ that returns the value of element $s_i$ of `S` $= [s_0\ s_1\ \cdots]$. Explain why this function is correct by reasoning on the relationships between the elements of the infinite streams in the program.
   - Translate the program into kernel language. That is, all functions should be written as procedures, all nested expressions should be written as sequences of statements, and all the `lazy` annotations should be translated with threads and `WaitNeeded` instructions.
   - What lazy suspensions are created when the program is executed (without browse) and what variables are they attached to? What is the execution when the expression `S.2.1` is executed? Explain what suspensions are activated and what new suspensions are created.

2. For this question, answer the following questions:

   - Explain how a time-varying digital signal can be modeled by a stream.
   - Define the term *combinational logic* and give an example program that does a combinational logic operation with at least two gates and at least one input and one output. Draw the circuit that is equivalent to the program.

- Define the term *sequential logic* and give an example program that does a sequential logic operation with at least two gates and at least one input and one output. Draw the circuit that is equivalent to the program. What is the difference between sequential logic and combinational logic?
- Write a program that models a circuit that oscillates, i.e., it returns an infinite stream of alternating 0's and 1's.
- **Bonus** What happens if the program of the previous point is written using lazy functions?

3. In the following example we are launching 100 processes. Each process has a random type associated to it, a job to do, and a flag to indicate when the process has finished. The job to be done is simply a procedure that waits a random amount of time, and then binds the flag to `unit` indicating that the process has finished.

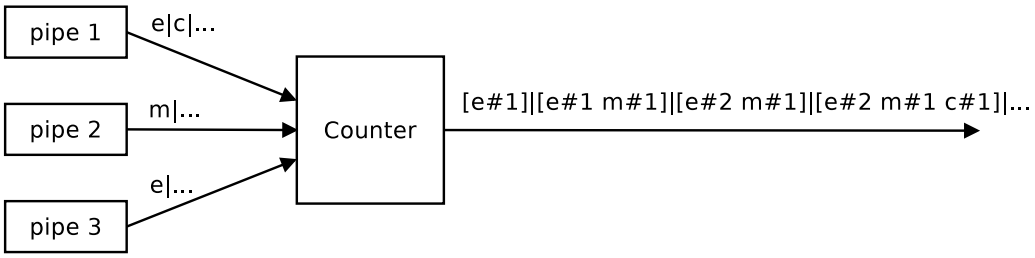
```
declare
%% Delay random time. Print job's type. Bind the flag.
proc {Job Type Flag}
   {Delay {OS.rand} mod 1000}
   {Browse Type}
   Flag=unit
end

%% BuildPs binds Ps to a tuple of process descriptions.
%% Each process is assigned a random type
proc {BuildPs N Ps}
   Ps={Tuple.make '#' N}
   for I in 1..N do
      Type={OS.rand} mod 10
      Flag
   in
      Ps.I=ps(type:Type job:proc {$} {Job Type Flag} end flag:Flag)
   end
end

%% Launching 100 processes
N=100
Ps={BuildPs N}
for I in 1..N do
   thread {Ps.I.job} end
end
```

Suppose that we want to be notified when all the threads of a given type are done. Write procedure {`WatchPs I Ps`} that prints the message ``‘‘all the threads of type I are finished’’`` when all the threads of type `I` have finished.

4. Implements the procedure {`WaitOr X Y`} that waits until `X` or `Y` is bound to a value.

5. A little bit more difficult. Implements the *"function"* {`WaitOrValue X Y`} that waits until `X` or `Y` is bound, and returns the value of the first bounded variable. Is it possible to implement such function and remain in the declarative concurrency paradigm?

6. In lab session 4, we implemented a counter of rescued miners. Its input was a single stream of characters. Each character represented the activity of each miner. The counter could tell how many miners were rescued classifying them by their activity. Luckily for our declarative concurrent program, they rescued the miners using a single pipe for taking them out. Now, let us imagine that the three machines that were working on the rescue operation succeeded on making the pipe, so they could rescue the miners using three pipes. We still want to use a single counter as it is depicted on the following figure.




   Implement the counter and the pipes in a declarative concurrent way. You can start by implementing only 2 pipes.

   - What happens if one of the pipes stop rescuing miners?
   - Can you extends the solution to $N$ pipes?