# INGI1131 Practical Exercises
# Lab 5: Threads and Declarative Concurrency

During this lab session we will continue synchronizing concurrent threads. We will see producer and consumer, logic gates, bounded buffer, and detection of thread termination.

1. **Producer-Consumer** We start with simple exercises for warming up on stream communication.

   a Create a function {`Numbers N I J`} that returns a list of `N` random integers from `I` to `J`. Use the function {`OS.rand`} that returns a random integer. As an extra constraint, force function `Numbers` to wait half a second before generating every number.

   b Create a function called `SumAndCount` that takes a list of integers as input, and produces two results: the sum of all the elements of the list, and the amount of elements that were in the list. As an extra constraint, `SumAndCount` should take 250 milliseconds to process every element.

   c Make the output of `Numbers` be the input of `SumAndCount`, where each function run on its own thread. How much time does the whole program takes in order to execute everything? (answer in terms of the N).

   d Write a function {`FilterList Xs Ys`} that takes `Xs` as input, and return another list where none of the elements belongs to `Ys`. These kind of functions can be used to filter emails using a black list of known spammers.

   e Concatenate `Numbers`, `FilterList` and `SumAndCount` to filter some numbers from the random generation, and show intermediate outputs on the Browser to see the evolution of the lists.

2. **Digital Logic Simulation**

   a A logic gate is a concurrent function that takes one or more streams as input with binary values, apply a logic function to the input, and returns the resulting stream as output. The simplest logic gate is the `NotGate`, which takes one stream as input, and negates every element of it. Implement `NotGate` starting by defining the function `Not` that receives a binary integer and returns its negation.

   b Implement logic gates `AndGate` and `OrGate`. Both of them take two streams as input, and return the result of applying logic operations *and* and *or* to each pair of inputs. And example of calling the `AndGate` can be seen in Figure 1.

   c We can represent one-output logic gates with binary trees of the form
   `gate(value:<logic operator> <input 1> ... <input n>)`
   Note that the amount of inputs vary according to the logic operator. Inputs can

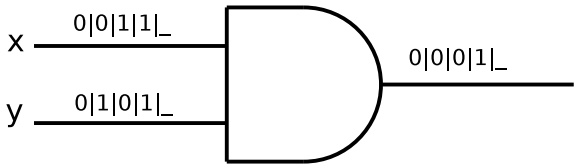


Figure 1: The 'and' logic gate.

be represented as other logic gates, or as `input(v)` where `v` is to be found on the stream inputs. A way of writing the `NotGate` using these trees is simply

```
gate(value:'not' input(x))
```

The gate that represents Figure 1 is the following:

```
gate(value:'and' input(x) input(y))
```

The following binary tree represents the logic gate of Figure 2:

```
gate(value:'or'
   gate(value:'and'
      input(x)
      input(y))
   gate(value:'not'
      input(z)))
```

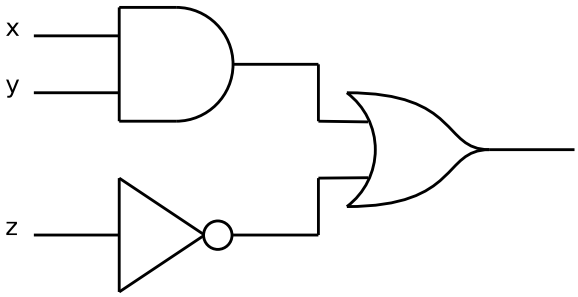


Figure 2: Composition of logic gates.

Implement a function {`Simulate G Ss`} that returns the output of the circuit corresponding to `G` under the inputs `Ss`.

Example of execution: Assume that `G` is bound to the tree above. The call
{`Simulate G input(x:1|0|1|0|_ y:0|1|0|1|_ z:1|1|0|0|_)`}
should return `0|0|1|1|_`. Notice that Simulate must be a concurrent process. Further determination of the input should imply further determination of the output. When feeding the following piece of code (again, assume that `G` is bound to the tree above):

```
declare Ss
{Browse {Simulate G Ss}}
Ss=input(x:1|0|1|0|_ y:0|1|0|1|_ z:1|1|0|0|_)
```

`0|0|1|1|_` must appear in the Browser.

d General question. What is the relationship between logic gates and producer and consumers?

3. **Thread Termination**

a As you may already known, the function {`Map L F`} takes a list `L` and a function `F` as arguments. It applies `F` to every element of `L`, and it returns a list with those results. Now, let us suppose that you call `Map` inside a thread, and you want to use `Show` to display the result on the emulator. A first attempt to write the code would be as follows:

```
declare
L1 L2 F

L1 = [1 2 3]
F = fun {$ X} {Delay 200} X*X end
thread L2 = {Map L1 F} end
{Show L2}
```

The problem is that if `Show` is called before the thread has finished, the result will not appear on the emulator. This is because `L2` will not be bound to the resulting list. Use the function {`Wait X`}, which waits until variable `X` is bound to a value, to call {`Show L2`} only when the thread has finished its execution.

b Exactly the same problem, but now using three threads. You want to call `Show` only when all threads are finished. Note that we do not need an intermediate variable `F` to crate a function and pass it as argument. You can create it just when the function Map is called.

```
declare
L1 L2 L3 L4
L1 = [1 2 3]
thread L2 = {Map L1 fun {$ X} {Delay 200} X*X end} end
thread L3 = {Map L1 fun {$ X} {Delay 200} 2*X end} end
thread L4 = {Map L1 fun {$ X} {Delay 200} 3*X end} end
{Show L2#L3#L4}
```

c `MapRecord` is a function that, given a record `R1` and function `F`, returns the records that result of applying `F` to each field of `R1`. Consider the following definition of Map on records:

```
proc {MapRecord R1 F R2}
   A={Record.arity R1}
   proc {Loop L}
      case L of nil then skip
      [] H|T then
         thread R2.H={F R1.H} end
         {Loop T}
      end
   end
in
   R2={Record.make {Record.label R1} A}
   {Loop A}
end
```

- `Record.arity` returns a list where every element is a field name of the record. For instance: {`Record.arity rec(foo1:1 foo2:2)`} returns `[foo1 foo2]`
- `Record.make` creates a record based on two arguments: a label, and a list with the arity of the record. The values of the fields are unbound.

Under this definition, If you feed the following code:

```
{Show {MapRecord
       '#'(a:1 b:2 c:3 d:4 e:5 f:6 g:7)
       fun {$ X} {Delay 1000} 2*X end}}
```

You will get a tuple of undetermined values on the emulator window.

i. Redefine `MapRecord` by adding an extra parameter that is bound to **unit** once all the threads has finished their execution.

ii. Use this last version of `MapRecord` to wait for the execution of all the threads before showing the result.

d **Bounded Buffer**

i. Do the exercise about Foo and Bar in lab session 4 if you have not done it yet. In that exercise you have to write your own bounded buffer, and that will help you to understand better the next exercise.

ii. Consider now that you have two consumers and one producer. How can you use bounded buffers to avoid having a memory overflow in the case where one consumer go faster than the other one? And by the way, why do we have an overflow situation if one consumer is faster?
In the example shown below, we cannot know a priori which consumer will go faster since the delay at each iteration is random. Let $n_1$ and $n_2$ be the number of elements consumed by `DSum01` and `DSum02` at some point in time. How can we ensure that $|n_1 - n_2|$ is always smaller than or equal to the size of the bounded buffer?

*Hint: You are not limited to use only one bounded buffer.*

```
declare
proc {Buffer N ?Xs Ys}
   fun {Startup N ?Xs}
      if N==0 then Xs
      else Xr in Xs=_|Xr {Startup N-1 Xr} end
   end
   proc {AskLoop Ys ?Xs ?End}
      case Ys of Y|Yr then
         Xr End2
      in
         Xs=Y|Xr % Get element from buffer
         End=_|End2 % Replenish the buffer
         {AskLoop Yr Xr End2}
      end
   end
   End={Startup N Xs}
in
   {AskLoop Ys Xs End}
end
proc {DGenerate N Xs}
   case Xs of X|Xr then X=N {DGenerate N+1 Xr} end
end
fun {DSum01 ?Xs A Limit}
   {Delay {OS.rand} mod 10}
   if Limit>0 then
      X|Xr=Xs
   in
      {DSum01 Xr A+X Limit-1}
   else A end
end
fun {DSum02 ?Xs A Limit}
   {Delay {OS.rand} mod 10}
   if Limit>0 then
      X|Xr=Xs
   in
      {DSum02 Xr A+X Limit-1}
   else A end
end
local Xs Ys V1 V2 in
   thread {DGenerate 1 Xs} end % Producer thread
   thread {Buffer 4 Xs Ys} end % Buffer thread
   thread V1={DSum01 Ys 0 1500} end % Consumer thread
   thread V2={DSum02 Ys 2 1500} end % Consumer thread
   {Browse [Xs Ys V1 V2]}
end
```