INGI1131 Practical Exercises
Lab 6: Laziness

In this lab session we will study the advantages of lazy execution. In Deep Purple's song called "Lazy" (yes, a very old song from last century), the lazy girl is some one who just stay in bed doing nothing, no work at all. Note that this kind of laziness is only about *when to do the work*, and not about *not doing the work*. So, don't take it as a promotion for being a lazy student. We don't care about when you do your work, but it is important that the work gets done when it is requested.

1. Warming up exercise:

   - Write a function `{Gen I}` that lazily generates a list of all integers starting from `I`.
   - Browse the first 3 natural numbers using the function `Gen`.
   - Write the function `{GiveMeNth N L}` that returns the Nth element of a list `L` generated with `Gen`.

2. Implement a function `{Primes}` that lazily returns the infinite list of prime numbers in increasing order. In order to implement `Primes`: Write a lazy version of `Filter`:

   ```
   fun {Filter Xs P}
      case Xs of
         nil then nil
      [] X|Xr then
         if {P X} then
            X|{Filter Xr P}
         else
            {Filter Xr P}
         end
      end
   end
   ```

   Write a lazy version of the function Sieve shown in Lecture 4:

   ```
   fun {Sieve Xs}
      case Xs of nil then nil
      [] X|Xr then
         X|{Sieve {Filter Xr fun {$ Y} Y mod X \= 0 end}}
      end
   end
   ```

   Then, generate an infinite list of increasing natural numbers using function `Gen` from the first exercise to feed the `Sieve`.

3. Implement a function `{ShowPrimes N}` that uses `{Primes}` to show the first `N` prime numbers.

4. Consider the following definition of Gen, Filter and Map:

```
fun {Gen I N}
   {Delay 500}
   if I==N then [I] else I|{Gen I+1 N} end
end
fun {Filter L F}
   case L of nil then nil
   [] H|T then
      if {F H} then H|{Filter T F} else {Filter T F} end
   end
end
fun {Map L F}
   case L of nil then nil
   [] H|T then {F H}|{Map T F}
   end
end
```

And now feed the following program:

```
declare Xs Ys Zs
{Browse Zs}
{Gen 1 100 Xs}
{Filter Xs fun {$ X} (X mod 2)==0 end Ys}
{Map Ys fun {$ X} X*X end Zs}
```

Zs is only displayed when Xs and Ys are completely determined. What we want, instead, is to display Zs incrementally.

a Do it by adding **thread** ... **end**

b Do it without explicitly creating threads, but by using lazy functions

5. {Minimum L} is a function that returns the smallest element in list L. Consider the following implementation of Minimum:

```
fun {Insert X Ys}
   case Ys of
      nil then [X]
   [] Y|Yr then
      if X < Y then
     X|Ys
      else
     Y|{Insert X Yr}
      end
   end
end
```

```
fun {InSort Xs} %% Sorts list Xs
   case Xs of
      nil then nil
   [] X|Xr then
      {Insert X {InSort Xr}}
   end
end
fun {Minimum Xs}
   {InSort Xs}.1
end
```

a What is the complexity of `Minimum`? (You can confirm by adding `Show` statements to `Insert` and `InSort` to see how many times they are called).

b What is the complexity if we make `Insert` and `InSort` lazy?.

6. {`Maximum L`} is a function that returns the greatest element in `L`. Consider the following implementation of `Maximum`:

```
fun {Last Xs}
   case Xs of
      [X] then X
   [] X|Xr then {Last Xr}
   end
end
fun {Maximum Xs}
   {Last {InSort Xs}}
end
```

For this implementation of `Maximum`, does it make any difference to implement `Insert` and `InSort` lazily? Why?

7. Make sure you understood the exercise on thread termination from last week (specially the concurrent MapRecord). We are sorry to be so insistent, but we are the assistants, and in "The Secret Book of the Teaching Assistant" there is a rule that says we have to be a bit annoying. So, not our fault! Blame Canada!

8. Let us study now a lazy version of the bounded buffer. This exercise is taken from the session of last week, but adding laziness to it. Solve the exercise of one producer `DGenerate` with two consumers `DSum01` and `DSum02` connected with bounded buffers. This time consider the lazy implementation of the bounded buffer which goes as follows:

```
fun {Buffer In N}
   End=thread {List.drop In N} end
   fun lazy {Loop In End}
      case In of I|In2 then
     I|{Loop In2 thread End.2 end}
      end
   end
in
   {Loop In End}
end
```

- Do you have to change the implementation of `DGenerate`, `DSum01` and-or `DSum02`?
- Why? Call Gustavo and try to explain him your conclusions. If and only if Gustavo is way too busy, then call Boriss.

Here is the code from last week (without the bounded buffer).

```
declare
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
```