LINFO1131 Practical Exercises
Lab 1, extra exercises

Bounded buffers

1. Foo was a legendary student famous for his ability of drinking an incredible amount of beer. In fact, nobody knew his real limit. In addition, Foo was able to drink a pint in 12 seconds in a regular basis. Actually, some students could drink a single pint faster than that, but they could not hold the pace for too long. Something like sprinters versus long distance runners.

   Bar was a renowned barman able to take a glass, serve the beer, and put it on the table in only 5 seconds, without dropping anything, and with two fingers of foam. The drinking sessions of Foo were organized as follows. Bar constantly served beers to Foo during one hour, and Foo drank them one after the other without having a single pause. To prevent beer from getting warm, Bar never put more that 4 beers on the table at the same time.

   Model this scenario as a producer/consumer algorithm, where Bar is the producer and Foo is the consumer. What happens at second 36? Do you need to model the table as well? How many beers did Foo drank in the hour. Search on the web and try to learn something about *bounded buffers*.

2. 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
```

Thread Termination

1. 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.

2. 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}
```

3. `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.

- Redefine `MapRecord` by adding an extra parameter that is bound to **unit** once all the threads has finished their execution.
- Use this last version of `MapRecord` to wait for the execution of all the threads before showing the result.