INGI1131 Practical Exercises
Lab 13: Shared State Concurrency and Locks

This lab session is dedicated to work with concurrency and explicit state, so now things get *sophisticated*. If you still don't feel familiar with explicit state, you can have a look at some exercises from lab session 11.

1. Consider the following implementation of `Port` and `Send` using Oz cells.

   ```
   proc {NewPort S P}
       P = {NewCell S}
   end

   proc {Send P Msg}
       NewTail
   in
       @P = Msg|NewTail
       P := NewTail
   end
   ```

   Is it correct? Is it free of race conditions? If not, how can you fix it. Do you need a lock?

2. Let us implement a concurrent counter. The counter has an increment operation. We would like this operation to be safely callable from any number of concurrent threads. Consider the following simple implementation that uses a shared cell `C` and `Exchange`:

   ```
   local X in X = C := X+1 end
   ```

   This attempted solution does not work (try it!). Explain why the above program does not work and propose a fix.

3. Implement a class `BankAccount` with four methods: `init`, `deposit(Amount)`, `withdraw(Amount)` and `getBalance($)`. Do you need to use locks for any of these methods?

4. Implement a procedure to transfer money from one account to another one. Feel free to implement it as an external procedure, as class method, or whatever you think is more convenient. Make sure you can launch concurrent money transfers without losing correctness.

5. Consider the two following implementations of `NewQueue`. The first one uses only one lock. Therefore, performing enqueue and dequeue are exclusive operations.

```
fun {NewQueue1 ?Insert ?Delete}
   X C={NewCell q(0 X X)}
   L={NewLock}
in
   proc {Enqueue X}
      N S E1 in
      lock L then
         q(N S X|E1)=@C
         C:=q(N+1 S E1)
      end
   end
   proc {Dequeue ?X}
      N S1 E in
      lock L then
         q(N X|S1 E)=@C
         C:=q(N-1 S1 E)
      end
   end
end
```

The second implementation of `NewQueue` uses two locks: one for the head of the queue, and one for the last tail. One lock is in charge of synchronizing all the enqueue operations, and the other one the dequeue operations. With this implementation you can perform enqueue and dequeue concurrently.

```
fun {NewQueue2 ?Insert ?Delete}
   X C={NewCell q(0 X X)}
   L1={NewLock}
   L2={NewLock}
in
   proc {Insert X}
      N S E1 in
      lock L1 then
         q(N S X|E1)=@C
         C:=q(N+1 S E1)
      end
   end
   proc {Delete ?X}
      N S1 E in
      lock L2 then
         q(N X|S1 E)=@C
         C:=q(N-1 S1 E)
      end
   end
end
```

Are both implementations correct? Explain!