INGI1131 Practical Exercises
Lab 9: Message Passing

In this lab session we carry on with the practical use of ports, how to use functional building blocks as concurrency pattern.

1. Implement a server receiving the following kind of messages:

   - `add(X Y R)`: Makes the addition of `X` and `Y` binding the result to `R`.
   - `pow(X Y R)`: Computes `X` to the power of `Y` and binds the result to `R`.
   - `'div'(X Y R)`: Divides `X` by `Y` and binds the result to `R`.

   You can consider all the input values as integers. If the message received by the server does not match any of the 3 messages described above, it should print `'message not understood'` in the emulator output.

   The following code can be used to test the server. Note that the function that you have to implement is `LaunchServer`. You may need to add some instructions to ensure that calls to `Show` are always displaying valid information. Which kind of protocol are we using?

   ```
   declare
   A B N S
   Res1 Res2 Res3 Res4 Res5 Res6

   S = {LaunchServer}
   {Send S add(321 345 Res1)}
   {Show Res1}

   {Send S pow(2 N Res2)}
   N = 8
   {Show Res2}

   {Send S add(A B Res3)}
   {Send S add(10 20 Res4)}
   {Send S foo}
   {Show Res4}
   A = 3
   B = 0-A
   {Send S 'div'(90 Res3 Res5)}
   {Send S 'div'(90 Res4 Res6)}
   {Show Res3}
   {Show Res5}
   {Show Res6}
   ```

2. A marketing student, called Charlotte, is making a study about the beer consumption during the 24 heures vélo. She has a list of *ports* to contact every student at the UCL. She got it from a secret API that can be obtained from the student's repertoire at *uclouvain.be*. For her study, she sends a message to every student to know how many beers each of them consumed. Once she gathers all the information, she will know: a) how many beers where consumed in total, b) what was the average consumption, c) what was the minimum and d) the maximum of consumption.

   What follows are two possible implementations for the student *agent*. Pick one of them and write a small program to help Charlotte do her task. The first student works like as RMI. The second one replies sending a message to Charlotte's port.

```
fun {StudentRMI}
   S
in
   thread
      for ask(howmany:Beers) in S do
         Beers = {OS.rand} mod 24
      end
   end
   {NewPort S}
eng}

fun {StudentCallBack}
   S
in
   thread
      for ask(howmany:P) in S do
         {Send P {OS.rand} mod 24}
      end
   end
   {NewPort S}
end
```

   The following code can help you to create a list of students ready to answer Charlotte's questions. The code has a bug, so you have to fix it.

```
fun {CreateUniversity Size}
   fun {CreateLoop I}
      if I =< Size then
         {Student}|{CreateLoop I+1}
      end
   end
in
   %% Kraft dinner is full of love and butter
   {CreateLoop 1}
end
```

3. **Port objects** This is a very important concept that we will continue using during the next lab sessions, and it will be also useful for the project. As you may know, some of the important properties of objects are *state* and *encapsulation*. These two properties (among other we will study later on the course) are also present in port objects. A port object start as any object with an initial state. Every time it receives a message, it uses the message and its state to call a function that implements the encapsulated behaviour of the object. The function returns the new state of the object, and it is ready to receive the next message. This is exactly as a method call in other object-oriented languages. The following code is an implementation of a port object:

```
fun {NewPortObject Behaviour Init}
   proc {MsgLoop S1 State}
      case S1 of Msg|S2 then
         {MsgLoop S2 {Behaviour Msg State}}
      [] nil then skip
      end
   end
   Sin
in
   thread {MsgLoop Sin Init} end
   {NewPort Sin}
end
```

   The function returns a port. It receives as arguments a function that we call `Behaviour`, and the initial state `Init`.

   Create a function `Porter` that counts how many people get into a concert. `Porter` uses a port object that receives three kind of messages: `getIn(N)`, `getOut(N)` and `getCount(N)`. Message `getIn(N)` increments the internal counter with `N` persons. Message `getOut(N)` decrements the counter as expected. Now, the porter can be asked at any time how many people is inside with the message `getCount(N)`, which binds variable `N` to the current count of people.

4. Implement the function `NewStack` with no arguments, that returns a port object representing a stack. Use the function `NewPortObject` given bellow (or use your own if you feel like). Then, implement the three main operations that can be performed over a stack. {`Push S X`} that push the element `X` into the stack `S`. {`Pop S`} that returns the element in the top of the stack `S`, and {`IsEmpty S`}, that returns a boolean value telling if the stack `S` is empty or not. Each of these procedures should send the corresponding message to the port object created by `NewStack`.

5. Now that you know how to implement a stack, implementing a queue will not be difficult at all. Implement the function `NewQueue` with no arguments, that returns a port object representing a FIFO queue. The following operations must be implemented:
   {`Enqueue Q X`} that enqueue the element `X` into the queue `Q`. {`Dequeue Q`} that returns the first element in the queue. {`IsEmpty Q`} returning if `Q` is empty or not, and {`GetElements Q`} that returns all the elements of the queue.

6. Write another version of the program to count the rescued miners (see previous labs). In that program, implement the Counter Server using `NewPortObject`. Note that apart from the internal state, the count server also needs to output a stream of lists, with the amount of characters counted so far. Write a function {`Counter Output`} that returns a port object. The variable `Output` corresponds to the stream with the history of counted miners.

   Hint: Think about the output as the stream of another port. Then, the function `Behaviour`, that changes the state of the server, will first send the state to the output port and then return the new state.