LINFO1131 Practical Exercises
Lab 1: Back to the basics

Observations

In this lab session we assume that you understand and can make good use of the following concepts:

- Using Oz, its browser and understanding the error console;
- Lists, records and nested structures composed of both;
- Functors, and compiling oz programs with `ozc`;
- Procedures, functions, their difference, and tail-recursion.

Now we will refresh your Oz skills around:

- Higher order programming;
- Threads and declarative concurrency;
- Message passing.

Higher order programming

1. Write a function `{Filter L F}` that returns a list with all the elements from `L` that pass the filtering function `F`. Where `F` is a function that receives one argument and returns a boolean.
2. Using function `Filter`, write a function that takes a list of integers and returns only the even numbers.

Threads and declarative concurrency

1. Analyse the following code and, before running it on the OPI, give the values that are bound to variables `X`, `Y` and `Z`, at the end of the execution:

   ```
   local X Y Z in
      thread if X==1 then Y=2 else Z=2 end end
      thread if Y==1 then X=1 else Z=2 end end
      X=1
   end
   ```

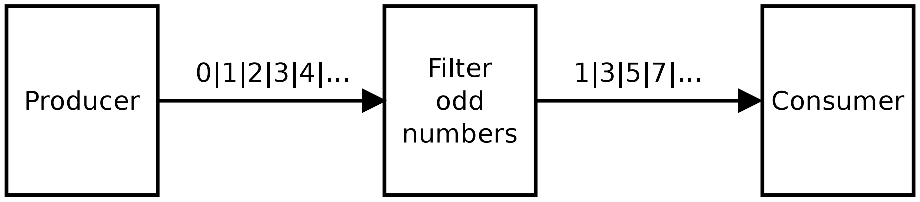
   Do the same for the following code:

```
local X Y Z in
   thread if X==1 then Y=2 else Z=2 end end
   thread if Y==1 then X=1 else Z=2 end end
   X=2
end
```

Verify your answers in the OPI. Of course, you will have to introduce a call to the browser. In which part of the code can you do this?

2. Implement a program that asynchronously generates a stream of 10,000 integers, filters out the odd number of the stream and sums up the elements of the resulting stream. The producer, filter, and consumer should all run in separate threads.

   The following figure describes the algorithm, where `Producer` is the process generating the stream of integers and `Consumer` the one summing up the filtered elements.



3. The following code synchronizes two procedures to play ping-pong. The synchronization is done using a list. Each procedure blocks on an unbound list and waits that the other procedures binds it to a head and a tail. Once this happens, the procedures browses either `ping` or `pong`, and then binds the rest of the list so that the other process is unblocked.

```
proc {Ping L}
   case L of H|T then T2 in
    {Delay 500} {Browse ping}
    T=_|T2
    {Ping T2}
   end
end

proc {Pong L}
   case L of H|T then T2 in
    {Browse pong}
    T=_|T2
    {Pong T2}
   end
end
```

   Consider now the following code call these two procedures:

```
declare L in
thread {Ping L} end
thread {Pong L} end
L=_|_
```

- What happens with the execution?
- Why does the game blocks?
- How do you fix the program?

More message passing

1. Ports are a kind of communication channel. You can send messages to a port, and read the messages received by the port. Every port has a stream associated to it. Sending a message to a port results in adding the message to port's stream. To create a port we use the procedure `{NewPort S P}`, which creates port `P`, and it associates stream `S` to `P`. Remember that a stream is just an infinite list. To send messages to a port, we use the procedure `{Send P Msg}`, which sends `Msg` to `P`. The following example send messages `foo` and `bar` to port `P`.

   ```
   declare
   P S
   {NewPort S P}

   {Send P foo}
   {Send P bar}
   ```

   But we haven't done anything with the stream associated to the port yet. Try browsing it with `{Browse S}`. What do you observe?

2. Now you have to implement `WaitTwo`. The idea is that a call to `{WaitTwo X Y}` will return 1 if `X` is bound before `Y`, and 2 otherwise. Implement it using ports. Remember that `WaitTwo` is not deterministic. Do you know why? In other words, is there any observable non-determinism?

3. Implement a server that receives messages in the form `Msg♯Ack`, where `Ack` is an unbound variable. The messages arrive via a `Port`. To acknowledge the reception of the message, the server binds the variable `Ack` to `unit` after waiting for a random time between 500 and 1500 milliseconds. The delay is only a way to simulate latency in the communication between two machines.

4. Implement a function `{SafeSend P M T}` that takes a port `P`, a message `M`, and a time value `T` (in milliseconds). The function sends a pair `M♯Ack` to the port `P`. If the receiver acknowledges receipt by binding `Ack` within `T` milliseconds, the function returns `true`. Otherwise, the function returns `false`.