INGI1131 Practical Exercises
Lab 4: Threads and Declarative Concurrency

Once again we sit down in front of the computer to solve the exercises of the lab session, and we wonder why. Well, the objective of this lab session is to learn how to synchronize concurrent parts of a program, and to understand why the result is declarative.

1. Consider

program A

```
local X Y Z in
   X = Y+Z
   Y = 1
   Z = 2
   {Browse X}
end
```

program B

```
local X Y Z in
   X = plus(Y Z)
   Y = 1
   Z = 2
   {Browse X}
end
```

   a Even though in both cases we are defining `X` in terms of `Y` and `Z`, the first program blocks, while the second doesn't. Why?

   b On the exercise 3.2, of lab session 3, we needed to change the order of the binding instructions in order to fix the problem. Now, without changing the order of instructions, use a thread in program A to get `X` bound.

2. On August 5, this year, 33 miners were trapped more than 600 meters under the ground after an accident in a mine in Atacama, Chile. This week, more than two months after the accident, they have been rescued, one by one, all 33 alive and in good condition. To celebrate such event, we will build a small program to keep track of the rescue operation classifying the miners by the role they played while they were trapped in the mine. There were mechanics, electricians, cooks, nurses, etc. You have to implement a program that keeps track of the occurrences of those activities. To simplify it, we will use one character to represent each role, as it is depicted in Figure 1. Therefore, the mine will *produce* a stream of characters. The function `Counter` receives such stream as input, and it generates an output stream where each position of the stream corresponds to the total count so far. Each position of the output stream is a list of tuples `C#N`, where `N` is the number of time that `C` appears in the input stream so far. You have to implement `Counter`.

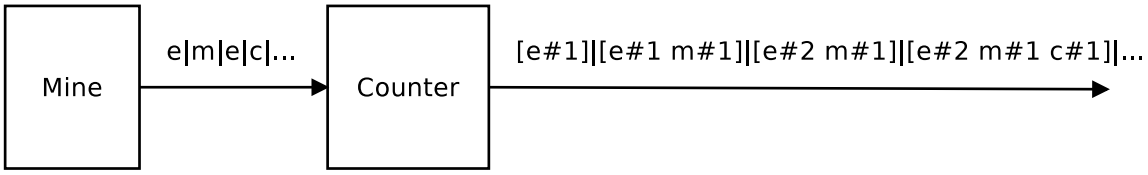


Figure 1: Example of counting characters.

Notice that Counter must be a concurrent process. If the following lines are fed:

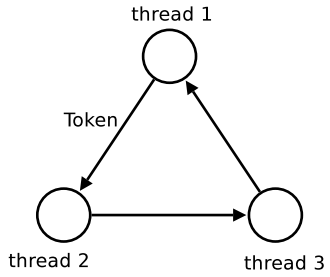
```
local
   InS
in
   {Browse {Counter InS}}
   InS=e|m|e|c|_
end
```

One should see in the browser `[e#1]|[e#1 m#1]|[e#2 m#1]|[e#2 m#1 c#1]|_`

3. The procedure assingTheToken Id Tin ToutPassingTheToken Id Tin Tout, given below, takes three arguments: `Id` is an integer (identification number), `Tin` and `Tout` are dataflow variables. The procedure `PassingTheToken` (see the code below) waits for the variable `Tin` to be bound to a pair `H|T`. Then it prints `Id#H`, suspends for a second, binds the variable `Tout` to the pair `H|X` (where `X` is the newly created local variable), and recursively call itself (i.e. `PassingTheToken Id T X`).

```
proc {PassingTheToken Id Tin Tout}
   case Tin of H|T then X in
      {Show Id#H}
      {Delay 1000}
      Tout = H|X
      {PassingTheToken Id T X}
   [] nil then
      skip
   end
end
```

Your assignment is to create three threads (see figure below), where each of the threads call the `PassingTheToken` procedure. The first thread will pass the token to the second thread, the second thread will pass the token to the third, and finally the third thread will pass the token back to the first thread, and so on...



4. 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*.

5. 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.

a Why don't you observe the same when you use `Browse` instead of `Show`?

b Redefine `MapRecord` by adding an extra parameter that is bound to `unit` once all the threads has been executed.

c Use this last version of `MapRecord` to wait for the execution of all the threads before showing the result.