Secure Programming via Libraries (T3)
ECI 2011 - Day 1

Alejandro Russo

Chalmers University of Technology

Exercise 1 Type classes

In the lectures, we define the following data type

```
data Nationality = Argentinian | Swedish
```

We would like to introduce equality among values of such data type. More specifically, we would like to provide == in such a way that

```
*Overview> Argentinian == Argentinian
True
*Overview> Argentinian == Swedish
False
*Overview> Swedish == Swedish
True
*Overview> Swedish == Argentinian
False
```

The data type `Nationality` is defined in the file `Overview.hs` available in the web page (check the material for the first day of the course).

1. Load the file `Overview.hs` in the Haskell interpreter GHCI, i.e. type in the interpreter `:l Overview.hs`. Try the following,

   ```
   *Overview> Argentinian == Argentinian
   ```

   What is the error that you get and what does it mean?

2. To define the equality between values of type `Nationality`, you should provide an instance of the class `Eq`. More precisely, we have that

   ```
   instance Eq Nationality where
     Argentinian == Argentinian
     ....
   ```

   In the definition above, we only define == for the cases when both arguments are `Argentinian`. Complete the definition of == for the other cases.

Exercise 2 The monad `IO`

Write a function called `quiz2` that asks for a non-prime number and returns the next two prime numbers. More precisely,

```
*Overview> quiz2
Give me a number (no prime):
4
The following two primes for that number are:
5
7

*Overview> quiz2
Give me a number (no prime):
10
The following two primes for that number are:
11
13
```

1. Implement the function `quiz2` using the monadic operations `return`, bind, `getLine`, and `putStrLn`. You might want to use the function `nextPrime` defined in `Overview.hs`, `toInt` (to convert an string into a number), and `show` (to convert a number into an string).

Exercise 3 Flow sensitivity

In this exercise, we are going to exercise the concepts of flow-sensitivity in monitors for information-flow security. We assume a purely dynamic monitor, i.e., a monitor that **only** inspects the current instruction being executed in order to determine if it is safe to execute it or not. This monitor is flow-sensitive, i.e., it might changes the security level associated with variables depending on the content that they store at certain points of the execution of programs. More specifically, when the monitor finds an explicit flow (an assignment) of the form

```
v = e
```

then, the monitor determines the security level of the expression `e` and associates that security level to `v`. We assume that constants are public values. On the other hand, when the monitor detects assignments inside a branching instruction, it assigns the security level of the assigned variables to the join of the branch's conditional and the security level of the expression being assigned. For example, assuming that `h` is a secret variable, then if the monitor executes the following then-branch

```
v = 1 ;
if h then v = 0 ;
```

then, variable `v` is considered as a secret variable. In contrast, if `h` is false, the security level of variable `v` remains public.

Assuming the monitor described above, consider the following code where functions `secret_input` and `public_output` introduce secret values and produce public outputs, respectively.

```
l = 1 ;
t = 0 ;
h = secret_input () ;
if h then t = 1 ;
if !t then l = 0 ;
public_output(l) ;
```

1. What are the security levels associated to variables `l`, `t`, and `h` when executing the program considering `h` being true and false?
2. Does the program above leak information when being monitored?
3. If you answer affirmatively the previous question, do you have some ideas how to fix the problem?