Bonus Exercises Week 1

Jan van Eijck
CWI & ILLC, Amsterdam

September 3, 2014

```
module Lab1Bonus

where
```

The `foldr` function:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

Here is what happens if you call `foldr` with a function f, and identity element z, and a list $[x_1, x_2, x_3, \ldots, x_n]$:

$$\text{foldr } f \ z \ [x_1, x_2, ..., x_n] = (f \ x_1 \ (f \ x_2 \ (f \ x_3 \ \ldots \ (f \ x_n \ z) \ldots).$$

And the same thing using infix notation:

$$\text{foldr } f \ z \ [x_1, x_2, ..., x_n] = (x_1 \ `f` \ (x_2 \ `f` \ (x_3 \ `f` \ (\ldots (x_n \ `f` \ z) \ldots).$$

The `and` function can be defined using `foldr` as follows:

```
and = foldr (&&) True
```

Exercise 1

1. Define `length` in terms of `foldr`.
2. Define `elem x` in terms of `foldr`.
3. Find out what `or` does, and next define your own version of `or` in terms of `foldr`.
4. Define `map f` in terms of `foldr`.
5. Define `filter p` in terms of `foldr`.
6. Define `(++)` in terms of `foldr`.
7. Define `reversal` in terms of `foldr`.

While `foldr` folds to the right, the following built-in function folds to the left:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

If you apply `foldl` to a function $f :: \alpha \to \beta \to \alpha$, a left identity element $z :: \alpha$ for the function, and a list of arguments of type β, then we get:

$$\text{foldl } f \; z \; [x_1, x_2, ..., x_n] = (f \ldots (f(f(f \; z \; x_1) \; x_2) \; x_3) \ldots x_n)$$

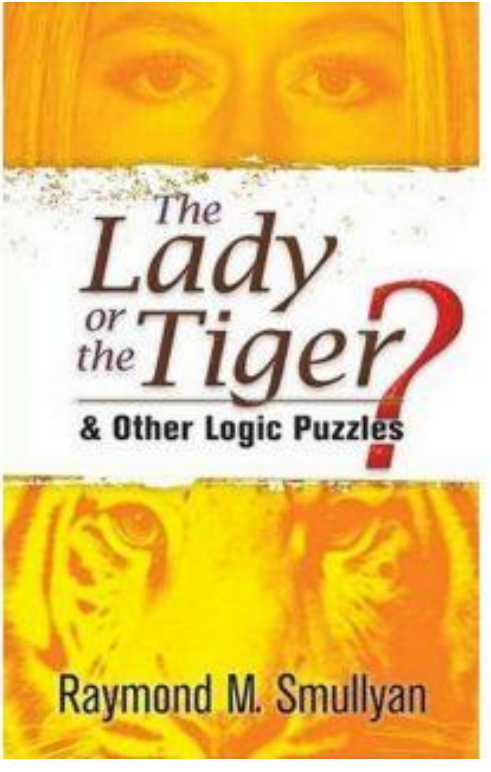
Or, if you write f as an infix operator:

$$\text{foldl } f \; z \; [x_1, x_2, ..., x_n] = (\ldots (((z \; `f` \; x_1) \; `f` \; x_2) \; `f` \; x_3) \; \ldots \; `f` \; x_n)$$

Exercise 2 Give an alternative definition of reversal (the function for reversing a list), in terms of `foldl`.

Exercise 3 One of `foldr`, `foldl` may work on infinite lists. Which one? Why? (Look up some explanations in http://www.haskell.org/haskellwiki/.)

Puzzles from Smullyan ?



The First Puzzle

There are two rooms, and a prisoner has to choose between them. Each room contains either a lady or a tiger. In the first test the prisoner has to choose between a door with the sign “In this room there is a lady, and in the other room there is a tiger”, and a second door with the sign “In one of these rooms there is a lady and in the other room there is a tiger.” A final given is that one of the two signs tells the truth and the other does not.

Haskell Statement of the Puzzle

```
data Creature = Lady | Tiger
     deriving (Eq,Show)

sign1, sign2 :: (Creature,Creature) -> Bool
sign1 (x,y) = x == Lady && y == Tiger
sign2 (x,y) = x /= y
```

Haskell solution

```
solution1 :: [(Creature,Creature)]
solution1 =
 [ (x,y) | x <- [Lady,Tiger],
           y <- [Lady,Tiger],
           sign1 (x,y) /= sign2 (x,y) ]
```

Running this reveals that the first room has a tiger in it, and the second room a lady:

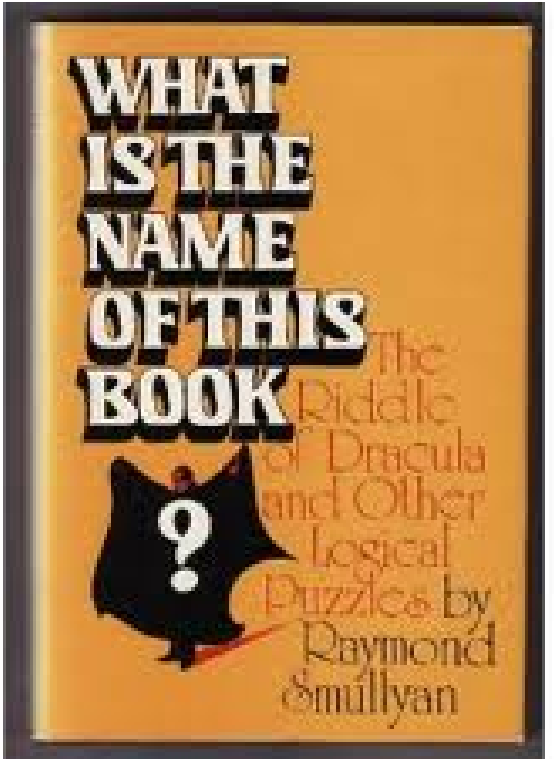
```
*Lab1Bonus> solution1
[(Tiger,Lady)]
```

The Second Puzzle

The second puzzle of the book runs as follows. Again there are two signs. The sign on the first door says: "At least one of these rooms contains a lady." The sign on the second door says: "A tiger is in the other room." This time either the statements are both true or both false.

Exercise 4 Give a Haskell implementation of `solution2` that solves the puzzle. You will also have to write functions for the new signs, of course.

Knights and Knaves



The First Puzzle

On the island of knights and knaves made famous in another logic puzzle book by Raymond Smullyan **?**, there are two kinds of people. Knights always tell the truth, and knaves always lie. Of course, if you ask inhabitants of the island whether they are knights, they will always say "yes."

Suppose John and Bill are residents of the island. They are standing next to each other, with John left and Bill right. John says: "We are both knaves." Who is what?

Haskell Solution

```
data Islander = Knight | Knave deriving (Eq,Show)

john :: (Islander,Islander) -> Bool
john (x,y) = (x,y) == (Knave,Knave)

solution3 :: [(Islander,Islander)]
solution3 = [(x,y) | x <- [Knight,Knave],
                     y <- [Knight,Knave],
                     john (x,y) == (x == Knight) ]
```

This reveals that John is a knave and Bill a knight:

```
Lab1Bonus> solution3
[(Knave,Knight)]
```

Another Knights and Knaves Puzzle

In this puzzle, again John is on the left, Bill on the right. John says: "We are both of the same kind." Bill says: "We are both of different kinds." Who is what?

Exercise 5 Implement a Haskell solution.

Becoming a Puzzle Designer

Exercise 6 Use the Haskell puzzle solving tools to design a few puzzles of your own.

Method: work backward from puzzle answers specifications that have a unique solution to suitable puzzle forms.

You can start to build some routine by designing variations on the ladies and tigers theme.

Crime Scene Investigation

A group of five school children is caught in a crime. One of them has stolen something from some kid they all dislike. The headmistress has to find out who did it. She questions the children, and this is what they say:

Matthew Carl didn't do it, and neither did I.

Peter It was Matthew or it was Jack.

Jack Matthew and Peter are both lying.

Arnold Matthew or Peter is speaking the truth, but not both.

Carl What Arnold says is not true.

Their class teacher now comes in. She says: three of these boys always tell the truth, and two always lie. You can assume that what the class teacher says is true. Use Haskell to write a function that computes who was the thief, and a function that computes which boys made honest declarations.

Hint: represent the declarations of the boys as properties for specifying the guilty boy, as follows:

```
data Boy = Matthew | Peter | Jack | Arnold | Carl
           deriving (Eq,Show)

boys = [Matthew, Peter, Jack, Arnold, Carl]

matthew, peter, jack, arnold, carl :: Boy -> Bool
matthew = \ x -> not (x==Matthew) && not (x==Carl)
peter   = \ x -> x==Matthew || x==Jack
jack    = \ x -> not (matthew x) && not (peter x)
arnold  = \ x -> matthew x /= peter x
carl    = \ x -> not (arnold x)

declarations = [matthew,peter,jack,arnold,carl]
table = zip declarations boys
```

Exercise 7 Now write a function `solution` that lists the boys that could have done it, and a function `honest` that lists the boys that have made honest declarations, for each member of the solution list.