Exercise 2

NTNU

TDT4165 fall 2018

1 Parametric polymorphism

1.1 Theory

Up until now, we have mostly written functions that works on one type. This is very inefficient, since we then have to write a specialized function for every type. Say we have a function that filters a string on a condition. We could write it like this:

```
filterStr :: (Char -> Bool) -> [Char] -> [Char]
filterStr _ []   = []
filterStr f (x:xs)
    | f x       = x : filterStr f xs
    | otherwise = filterStr f xs
```

This function takes a function from Char to Bool and a list of characters. Remember that a list of characters is the same as a string. It then filters out all characters that satisfy our condition **f x**. In other words, it returns a list of characters where the function from Char to Bool returns True.
If we now wanted to filter a list of integers, we would have to write a new filter function to accommodate this. Very tedious, right?
Luckily we have a way to mitigate this. In our function, we only use functions that apply to lists of any type. In fact, we are only using **(:)**, which we know works on any list. This means our function will also work on any list. We replace all occurrences of **Char** with **a**.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []   = []
filter f (x:xs)
    | f x      = x : filter f xs
    | otherwise = filter f xs
```

Here, **a** represents any type. This is syntactic sugar for **forall a. (a -> Bool) -> a -> [a] -> [a]**. There is nothing special about the character **a**, we could have chosen any lower case string we wanted.

1.2 Exercises

Complete the two tasks under **TASK 1** in the code skeleton. The skeleton is located in the file **ex2code/src/Lib.hs**.

2 Higher-order functions

2.1 Theory

In Haskell, functions are first class citizens, they are treated like any other type. They can be passed to functions and returned from them. A function that takes a function as an argument and/or returns a function is called a **higher-order function**. Our functions **filterStr** and **filter**, above, are higher-order functions.

2.2 map

Implement the function **map :: (a -> b) -> [a] -> [b]** that takes a function from any type **a** to any type **b** and a list of type **a**. It should iterate through the list, apply the functions to all of the elements and return a list of type **b**. Do not use the built in function **map**.

```
--example
Prelude> map show [123, 456, 789]
 ["123", "456", "789"]
```

Note that **a** and **b** in the type signature does not mean that a and b have to be of different types. This means that **they can be**, not that it's necessary that they are.

```
--example
Prelude> map (+5) [1..5]
[6,7,8,9,10]
```

(+) takes in two numbers of the same type and adds them together. So here we have an example where a and b in the type signature of map is of the same type.

2.3 iterate

Implement the function **iterate :: (a -> a) -> a -> [a]** which produces an infinite list by repeatedly applying a function **f :: a -> a** to a value. The result we want is this:

$$iterate = [n, fn, f(fn), f(f(fn))..]$$

3 Currying

3.1 Theory

By default all functions in Haskell takes one argument and returns one value. This might sound counter-intuitive, since we've worked with many functions that seemingly takes many arguments. To explain how this works, we need to digress a little bit and look at lambda expressions.

3.2 Lambda expressions

A lambda expression is a function that can be defined without being bound to an identifier. You might know them already, possibly under the name **anonymous functions**. Many languages have the possibility to define and call functions without naming them, e.g. Python, JavaScript and Java. In Haskell, we define an anonymous function like this:

```
Prelude> let lamb = \x y -> x + y
Prelude> lamb 3 4
7
```

The lambda expression got bound to an identifier **lamb** in this example, but this is not needed.

```
--example
Prelude> map (\x -> x ++ "set") ["clo", "as", "re"]
["closet", "asset", "reset"]
```

The functions we write are all desugared into lambdas when our code is parsed.

```
-- when we write
f x y z = x+y+z
-- our code gets desugared into a lambda expression
f = \x y z -> x+y+z
-- which desugars into nested lambda expressions taking one argument
f = \x -> \y -> \z -> x+y+z
```

This is known as **currying**. If we give an argument to our curried function *f*, we get a lambda expression back.

```
f2 = f 3
-- now f2 = \y -> \z -> 3+y+z
```

This is called **partial function application**.

3.3 Partial application

3.4 Filter negative numbers

Complete the two functions **filterPos** and **filterPosMany** in **src/Lib.hs**.

3.5 flip3

In Prelude, we have a function **flip :: (a -> b -> c) -> b -> a -> c** that can be applied to a function with two arguments, reversing the order these arguments are applied. This can be a handy trick if you want your functions to line up nicely, for example if you want to map over a list using a function that takes a list as the first argument.

```
Prelude> import Data.List
Prelude Data.List> :t intersperse
intersperse :: a -> [a] -> [a]
Prelude Data.List> let flipersperse = flip intersperse
Prelude Data.List> :t flipersperse
flipersperse :: [a] -> a -> [a]
Prelude Data.List> Prelude.map (flipersperse "intersperse") ":.!"
["i:n:t:e:r:s:p:e:r:s:e","i.n.t.e.r.s.p.e.r.s.e","i!n!t!e!r!s!p!e!r!s!e"]
```

Implement the function **flip3** that flips the arguments of a function with three arguments.

```
Prelude> flip3 (\x y z -> x : y : [z]) 'c' 'b' 'a'
"abc"
```

4 Infinite lists

4.1 Newton's method for finding square roots

In this task, we will find a way to determine if a number is a perfect square, i.e. the square of an integer. To achieve this, we need to find a way to calculate the square root. Haskell has a built in function for this, but being the awesome and curious programmers that we are, we are going to ignore that fact.
Implement a function that approximates the square root of a number. We suggest using Newton's method for this (defined below). You are free to choose other methods, but your function needs to give better approximations for better guesses. You are also free to choose your own type signatures, although some tasks have suggestions.

$$x_1 = x_0 - \frac{(x_0)^2 - x}{2x_0}$$

where x_1 is an approximation of the square root, given a guess x_0 and a number we approximate the square root of: x.

4.2 Creating the infinite list

It is time to make use of a function we wrote earlier, namely *iterate*. We want to call our Newton's method function (or what you chose to implement in the last task) over and over, so we get better and better approximations of *sqrt x*. A good way to to this, is to make an infinite list using iterate. Implement a function that returns this infinite list of approximations.

4.3 Check if a Double is an integer

Write a function, **Double -> Bool**, that checks if a double is an integer. Give it an appropriate name. f 1.0 should return True, while f 1.5 should return False.

4.4 Approximating the square root

Write a function, **approx :: Double -> [Double] -> Double**, that iterates through the list. Every iteration, it checks if the difference between the first and second element in the list is less than a given threshold, given as an argument to the function. If the difference between two successive values in the list is less than the threshold, we return the second of these two values.

```
--example where 16 is our x
Prelude> approx 0.0000001
    [5.0,4.1,4.001219512195122,4.0000001858445895,4.000000000000004,4.0...]
    --infinite list
4.0
```

4.5 Check for perfect squares

Write a function **isPerfSq :: Double -> Bool** which ties all the functions together to check if a number is a perfect square. Experiment with the threshold to find an accurate solution and choose a guess, either by defining it in terms of x or by hardcoding a value.

```
--example
Prelude> isPerfSq 16
True
Prelude> isPerfSq 5
False
```

4.6 Tweaking your solution

You are provided with a function **accuracy :: Int -> Bool** which tests **isPerfSqr** against an infinite list of perfect squares. It generates an infinite list of all numbers **isPerfSq** claims are perfect squares, and filters out numbers where **isPerfSqr** returns False. If **accuracy 200** returns True, that means

that **isPerfSqr** has correctly identified the first 200 perfect squares (and correctly rejected the numbers in between). This function is meant to aid you in choosing a satisfactory threshold and guess.

5 Theory questions

- What is the difference between a null pointer reference (e.g. null in java) and **None :: Maybe a**?
- Is $ a keyword or a function? What does it do?
  (hint: try typing **:t ($)** in GHCi)
- Explore (.) the same way. What is the difference between (.) and ($)?
- Try running **take 50 $ map fib [1..]** where fib is the unoptimized fibonacci function from exercise 1. If you did not complete this exercise, you will have to write a function **fib :: Int -> Int** first.

  - What does this code snippet do?
  - Do you get the whole result at once? Why or why not?
  - What happens if you run the Python code below? What changed and why?

```
def fib(a):
  if a == 0:
    return 0
  elif a == 1:
    return 1
  else:
    return (fib(a-1) + fib(a-2))

res = []
for i in map(fib,
     [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19, 34]):
  res.append(i)

print(res)
```

- In what situations can infinite lists be useful?
- Try typing **:t (,)** into GHCi. What is the type of (,) and what does this mean?
- How would you go about creating a curried function in Python? Is it possible?

- Point-free style is a way of programming without mention of the arguments we are working with. An example is in order:

```
print :: Show a => a -> IO ()
print item = putStrLn . show item
```

In our print function, we are composing the functions **show :: Show a => a -> String** and **putStrLn :: String -> IO ()**, giving us a function that takes an argument of a showable type and prints it. Using point-free style, our function would look like this:

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

Why is this possible in Haskell?