Prof. Dr. Peter Thiemann
Winter Term 2024/25

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 3

Exercise 1 (Composition and Application Operators)

In the lecture, we have seen operators for function composition `(.)` and function application `($)`.

These functions are defined as follows:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)     -- or alternatively:  f . g = \x -> f (g x)
infixr 9 .

($) :: (a -> b) -> a -> b
f $ x = f x
infixr 0 $
```

The `($)` operator is useful, because it differs from regular function application in the following ways:

- Regular function application is left-associative, whereas `($)` is right-associative (the `r` in `infixr 0 $`), e.g. `f x y` is parsed as `(f x) y` whereas, `f $ x $ y` is parsed as `f (x y)`.
- Regular function application binds stronger than any other infix operator, whereas `($)` binds weaker than any other infix operator (the `0` in `infixr 0 $`), i.e. `f x + y` is parsed as `(f x) + y`, whereas `f $ x + y` is parsed as `f (x + y)`.

Hence, `($)` is often used in Haskell to avoid parentheses, especially in deeper expressions like `f x (g y (h z (j x)))`.

1. Find three different ways to rewrite the following function with `($)` and `(.)` such that no parentheses are necessary (except for writing the operator sections `(>0)` and `(*2)`).

   ```
   f :: [Int] -> Int
   f xs = sum (filter (>0) (map (*2) xs))
   ```

2. What is the type of the following function? What does this function do?

   ```
   (.:) :: ?
   (.:) = (.).(.)
   ```

Exercise 2 (Lazy Evaluation)

1. Write a function `cycle'` that takes a list of elements and returns a list which infinitely repeats those elements.

   Example:

   ```
   >>> take 10 $ cycle' [0,1,2]
   [0,1,2,0,1,2,0,1,2,0]
   ```

2. Write a function `iterate''` that takes a function `f` of type `a -> a` and a value `x` of type `a` and returns an infinite list which at index `i` contains the `i`-fold application of `f` to `x`.

   Example:

   ```
   >>> take 10 $ iterate (+1) 0
   [0,1,2,3,4,5,6,7,8,9]
   ```

3. We already encountered the function `foldr`, which folds a list from the right:

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

   The standard library provides a similar function `foldl`, which folds a list from the left:

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

   Which, if any, of these two functions, when used on infinite lists, (may) terminate?

   If you have identified either of the functions as potentially terminating, then verify your claim by providing a suitable function `f` and starting value `z` such that `foldr f z [0..]` or `foldl f z [0..]` terminates.

4. Determine the worstcase asymptotic runtime for the following functions. Additionally, how often is the `(+1)` function actually evaluated?

   a)
   ```
   f1 :: [Int] -> Int
   f1 = head . map (+1)
   ```

   b)
   ```
   f2 :: [Int] -> Int
   f2 = last . map (+1)
   ```

5. Implement the *Hamming Stream* as an infinite list. The *Hamming Stream* consists of those natural numbers whose prime divisors are all ≤ 5, listed in increasing order without duplicates.

   Example:

   ```
   >>> take 30 hamming
   [1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,40,45,48,50,54,60,64,
    72,75,80]
   ```