EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Summerterm 2023
Prof. Torsten Grust, Denis Hirn
WSI — Database Systems Research Group

Assignment 1

Hand in this assignment until **Friday, 05. May 2023, 10:00** at the latest.

Ⓔxam-style Exercises

Exercises marked with Ⓔ are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Running out of ideas?

Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the `#forum` channel on our Discord server and check the tag for this assignment—maybe you'll find just the help you need.

Rules for this and all future assignments

- In general, the only acceptable file format is plain text (`*.txt`, `*.hs` for Haskell code). Files in other formats are not graded, unless explicitly stated differently.
- All code you submit must compile. Code that does not compile (in particular: does not typecheck) might not be graded.
- You should aspire to code that does not produce any warnings when compiled with the GHC option `-Wall`.
- Please submit code that is nicely and consistently formatted and well-documented[a]. Every top-level function definition has to include a type signature and a comment.

[a] To have an idea of "nicely formatted code", you can find a short style guide here: https://github.com/tibbe/haskell-style-guide/blob/master/haskell-style.md.

Task 1: Types (1 credit)

Please answer the following questions about Haskell's type system.

(a) Consider the following types:

i. `a -> b -> c -> d`
ii. `a -> (b -> c) -> d`
iii. `a -> b -> (c -> d)`

Which pairs of types are equivalent and which are not? Explain.

(b) Can you give multiple definitions of a function of type `(a, b) -> a` that **behave differently**, that is, return different values for the same argument? Explain briefly. Assume that your function actually has to return a value (i.e. no crashes, no infinite loops and recursions).

(c) Consider a function of type `[a] -> a`. Recall that `a` represents an *arbitrary* type—the function must thus work correctly for lists over *any* element type. Could it be a function which

i. … returns the largest element of the list?
ii. … computes the sum of all list elements?
iii. … returns a *constant* value?
iv. … performs I/O operations (e.g. prints a value to the terminal)?

Explain your answers.

(d) The following function is supposed to extract the first character from a given string.

```
getFirstLetter :: [Char] -> [Char]
getFirstLetter s = head s
```

Fix any type errors.

(e) Given the the functions `fst :: (a, b) -> a` and `snd :: (a, b) -> b`, derive the type of the following expression:

```
snd . snd . fst
```

Task 2: Finger Exercises (1 credit)

(a) Define an infix operator that implements logical implication. You can use the Boolean operators `(&&)`, `(||)`, `not`, or a conditional expression (`if` e_1 `then` e_2 `else` e_3). The new operator's precedence should be less than the three Boolean operators' above. Please give some example expressions to show this behavior.

```
(==>) :: Bool -> Bool -> Bool
```

(b) Define a function `distance` to calculate the Euclidean distance between to Points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$.

```
distance :: (Double, Double) -> (Double, Double) -> Double
```

(c) Write a function `gcdEuclid` i j, such that computes the greatest common divisor of two integers $i, j > 0$ using Euclid's algorithm[1].

```
gcdEuclid :: Int -> Int -> Int
```

Include a brief comment on how your implementation would behave if parameters i or j are ≤ 0.

Task 3: Safe Head (1 credit)

Consider the Haskell function `head :: [a] -> a`, which returns the first element of a list. `head` is not able to return a value if the list is empty. `Haskell` would report an error at runtime.
Now imagine the function `headMaybe` with the following type:

```
headMaybe :: [a] -> Maybe a

> headMaybe [1, 2, 3]
Just 1
> headMaybe "abc"
Just 'a'
> headMaybe []
Nothing
```

`headMaybe` returns `Nothing` on failure and `Just` x on success, where x is the head of the argument list.
Define the function `headMaybe` so that it behaves as in the description above.

> **Hint**
> You can use the built-in predicate `null :: [a] -> Bool` to test whether a given list is empty.

[1] https://en.wikipedia.org/wiki/Euclidean_algorithm#Implementations