Lab Sheet 2

Types and Functions

Functional Programming - Winter 2018/2019 - November 1, 2018 - Schupp/Lehmann

How to succeed with the labs and exercises?

Labs and exercise sheets are published every week on the course homepage at StudIP. As described in the first lecture, each successfully completed lab and exercise earns you bonus points towards your final score in this semester's exam. Keep in mind that you only get bonus points if you would pass the exam without them. **Cheating does not help you - but we will!**

How to complete a lab successfully?

In the lab, you will solve the lab sheet with fellow student(s). To find out who your teammates are, please look at the group pdf in StudIP's download area. **During the session, one participant of every group will be selected, who must then explain one of their task solutions to a lab assistant**.

How to complete an exercise successfully?

In order to complete an exercise sheet successfully, you must upload your answers using CAT on StudIP **before the deadline** printed on the exercise sheet. We will not consider any solutions handed in after the deadline! Furthermore, you must solve and hand in the exercises **individually** and your Haskell code **must compile** and **pass certain amounts of tests** as specified. During the exercise session, we develop possible solutions together. Please participate! We encourage you to ask and answer questions from fellow students.

Technically, Haskell files you submit using CAT on StudIP must have the format as specified by CAT (usually ".hs", ".lhs", or ".txt"). Furthermore, CAT will store your last submission only. Therefore, if you first submit successfully (your code compiles and tests are passed) and afterwards unsuccessfully (your code does not compile or certain tests fail again), your last submission counts, and - if it does not compile - will therefore be ignored. Make sure your last submission was successfull!

How to get additional information?

We encourage you to discuss past and present exercise sheets with us. Either approach us during the exercise session, or visit us during the weekly office hours. We are also available via e-mail or on the StudIP forum. We try to reply as quick as possible and in general, you should get a reply the next weekday, but we cannot guarantee this.

Exercise 1 Write the following function definitions to the file `Lab2Err.hs` using, e.g., "SciTE for Haskell" or "Atom." Please enable visible whitespace and line numbers under "View."

```
succ :: int -> int
succ x = x + 1

Pred :: Int -> Int
Pred = - 1 + x

tupleup :: Int -> (Int, (Int, Int)
tupleup i = i, (i, i)

myNameIs :: [Char] -> Bool
myNameIs str = str = 'Nobody'

newLine :: String -> String
newLine str = str ++ '\n'
```

Load the file using (Win)GHCi, look at the error messages, and fix all errors.

Hint: If you cannot localize the error, work at each function definition separately by commenting out the remaining definitions.

Exercise 2 Which of the following Haskell expressions are well typed? For each expression that you think is well typed, write down which type you expect it to have. Use GHCi's `:type` command to check your results **afterwards**.

a) `['1','2','3']` List of chars

b) `(["False", "True"], [False, True], ['0','1'])`

c) `[1] ++ ['a']`

d) `[(False, '0'), (True, '1')]`

e) `[("False", '0'), ("True", '1')]`

f) `("1, 2",("3"))`

g) `[['1'],("True")]`

h) `[tail, init, reverse]`

Exercise 3

a) Explain the difference between `a`, `'a'`, `['a']`, and `"a"` in terms of types.

b) Explain why `(not 'a')` is a type error and informally argue how Haskell arrives at the error message.

c) Given a function `foo` of type `Char -> String` and a value `bar` of type `Char`, what is the type of `(foo bar)` and how do you determine this?

Exercise 4 Consider the following function definition:

```
square n =  n * n
```

Provide *contract*, *purpose*, *examples* and *tests* as described in the lecture (design by recipe).

Exercise 5 * What would be the effect of replacing `<=` by `<` in the following definition of `qsort`?

```
qsort [] =  []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                  smaller = [a | a <- xs, a <= x]
                  larger = [a | a <- xs, a > x]
```

Hint: Consider the example `qsort [2,3,2,1,2,2,1]`.