Lab Sheet 5

Recursion and Modules

STS
Software
Technology
Systems

Functional Programming - Winter 2022/2023 - November 23, 2022 - Schupp/Lübke

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 tasks on lab sheet. You are encouraged to talk to your neighbors and find solutions together, as well to ask the tutor for help. **Towards the end of the session, the tutor will briefly discuss your solutions with you. To pass the lab, you should complete two thirds of the tasks (rounding half up)**.

How to complete an exercise successfully?

In order to complete an exercise sheet successfully, you must upload your answers using INGInious **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 INGInious must have the format as specified in the task sheets (usually ".hs", ".lhs", or ".txt"). Furthermore, INGInious will only consider your last submission. 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 successful!

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.

For this lab, you need to have `QuickCheck` installed. You can check whether it is already installed by entering `ghci` and then executing `import Test.QuickCheck`. If no error messages appear, it is installed. Otherwise, leave GHCi (`:q`), and then execute `cabal install --lib QuickCheck`.

If you are in Linux Pool 3b and get an error saying that `cabal` is not found/installed/..., you need the (enclosed) *install-cabal.sh* file. Open a shell in the directory where the file is placed, and then execute `source install-cabal.sh`. Afterwards, you can proceed to install `QuickCheck` following the instructions above.

Task 1 Follow DR-2 from the lecture (Lecture 5) and use recursion to implement the function `myOr`, which should behave as the Prelude function or. The function or takes a list of Booleans and returns True exactly when True is included in the list. Otherwise it returns False.

For this exercise, please provide at least one generalized (using generated inputs) test.

Task 2 Consider the following faulty implementation of `myFindIndex`, which gets a value `x` and a list of values `ys`, and somehow returns a position of `x` in `ys`.

```
> myFindIndex x ys = helper (zip [0..] ys)
>   where
>     helper [] = -1
>     helper zs | x == snd z = fst z
>               | otherwise = helper zs
>       where z = head zs
```

Provide a complete design recipe for `myFindIndex`, including contract, purpose, examples, and tests. Run your tests to find all errors in the implementation. Explain all errors you found, how you found them, and how you fixed them. **Note:** If the type signature is very general, then quickCheck is free to use a variety of types, including (). The easiest way to get a reasonable type is to hardcode the element type in the contract of `myFindIndex` to something like Int or (Bool, Bool, Bool). Alternatively, you can look at the tests for `mySplitAt` below.

Task 3 Assume you are writing your own module `My` and would like to use all of the module Numeric except for floatToDigits, since this would lead to a name clash. Provide two different ways to import the module without causing a name clash.

Task 4 * The below implementation of the Prelude function splitAt (named `mySplitAt`) is recursively defined.

```haskell
-- Contract
mySplitAt :: Int -> [a] -> ([a], [a])

-- Purpose
--  mySplitAt n xs returns a tuple where first element is the length n prefix of
--  the list xs, and the second element is the remainder of the list xs.

-- Examples
example_mySplitAt_1 = mySplitAt 1 [1,2,3] == ([1],[2,3])
example_mySplitAt_2 = mySplitAt 4 [1,2,3] == ([1,2,3],[])
example_mySplitAt_3 = mySplitAt (-1) [1,2,3] == ([],[1,2,3])

-- Definition
mySplitAt n xs | n <= 0 = ([], xs)
mySplitAt _ [] = ([], [])
mySplitAt n (x : xs) = let (ys,zs) = mySplitAt (pred n) xs in (x:ys, zs)

-- Tests
prop_mySplitAtKeep n xs = let (ps,qs) = mySplitAt n xs in ps ++ qs == xs
    where types = xs :: [Int]
prop_mySplitAtLength :: Int -> [Int] -> Bool
prop_mySplitAtLength n xs = length (fst (mySplitAt n xs)) == expected
    where expected = min (max 0 n) (length xs)
prop_mySplitAtDropTail n xs = as == take n' xs && bs == drop n' xs
    where n' = n `mod` succ (length xs)
          (as, bs) = mySplitAt n' xs
          _ = xs :: [Int]
```

a) Perform a manual step-by-step evaluation for `mySplitAt` to complete the following table:

	Input		Output
recursion depth	`n`	`(x:xs)`	`(x:ys, zs)`
0	2	[1, 2, 3]	([1,2] , [3])
1	1	[2,3]	([2] , [3])
2	0	[3]	([ ] , [3])

Hint: Fill out the input columns first (top-down) and the output columns afterwards (bottom-up).

b) Argue why the function `mySplitAt` terminates (1-3 sentences)

The function mySplitAt terminates because it has two clauses that handle the base cases where the input list is empty or the input integer is less than or equal to 0.
When either of these conditions is met, the function returns a result without making any further recursive calls.
This ensures that the function will eventually stop executing and return a result, regardless of the input.

* This is an advanced, optional task, but we strongly advise you to give it a try!