Programming languages – Haskell

Laboratory instruction A (2021/22)

T. Goluch

Exercises should be carried out in the given order. After completing each exercise, please present lecturer result and save the function/program code in the solution file. If any of exercises seems too difficult, you can skip them and move on to the next one. Unless stated otherwise as a solution, include the implemented code to the final solution file.

1) Exercise

a) (0.5 pts.)

Run REPL loop: `ghci -flocal-ghci-history`[1] (Haskell Platform) or `stack ghci`/`cabal repl` (The Haskell Tool Stack) alternatively use the site: https://repl.it/repls/ExtraneousAdorableLink.

Create constants `down` = 3 i `up` = 14 and list `l` consisting of increasing integers from `down` to `up`.
Calculate sum `s` of elements of list `l` using the appropriate function working on the list.
Calculate square `s2` of sum `s` (elements of list `l`).
Using the appropriate function, create a list `l2` of squares of elements of list `l`.
Calculate the sum `sl2` of elements of list `l2`.
Calculate the difference between `s2` (square of sum of elements of `l`) and `sl2` (sum of the squares of elements `l`), is it `9394`?

As a solution, either do printScreen or include the contents of the *ghci_history* (applies to Haskell Platform).

b) Exercise (0.5 pts.)

Write the same as the function `sqrSum_sumSqr` taking two numbers defining range of increasing integers and returning difference between square of sum and the sum of squares. For a call `sqrSum_sumSqr 3 14` should return result `9394`. As a solution include the code to the final solution file.

c) Exercise (0.5 pts.)

Write a program (zad3.hs) containing function `sqrSum_sumSqr`. Load file as a module and call function: `sqrSum_sumSqr 3 14` should return result `9394`. Add `main` function printing result of call of function `sqrSum_sumSqr 3 14` and then compile and run executable file. As a solution, do printScreen of successfully compiled and executed the binary.

---

[1] It's temporarily replace the command history file location on the current folder. The file ".ghci_history" can be uploaded as a solution to the exercises implemented in GHCI.

2) Exercise (0.5 pts.)

Implement your own function `sumInt` doing the same as sum function working on lists (adding list items). Narrow function to `Int` types by adding the appropriate header. Load file as module and run function: `sumInt [2,4..12]`. Check if result is `42` and if function type `> :t` is: `sumInt :: [Int] -> Int`. As a solution, do printScreen of successfully loaded and executed module.

3) Exercise (0.5 pts.)

Implement function `exist` checking that element `e` passed as first parameter is on list `l` passed as the second parameter. If element `e` exists, function should return `"yes"` otherwise it should return `"no"`. For instance
`exist 8 [4, 6 ,8, 8]` should return `"yes"`, and
`exist 5 [4, 6 ,8, 8]` should return `"no"`.
Please do not use `elem` function working on lists.

4) Exercise

a) (0.5 pts.)

Implement `setDiff` function that returns difference of sets a\b[2] given as lists `a` and `b`.
For instance `setDiff [3,5,6] [1,2,6]` should return `[3,5]`.
Please do not use `\\` operator and `union` or `intersect` functions from the Data.List module.

b) Exercise (0.5 pts.)

Implement `setSymDiff` function that returns symmetrical difference of sets[3], you can use `setDiff` function from previous exercise. For instance `setDiff [3,5,6] [1,2,6]` should return `[3,5,1,2]`.
Please do not use `\\` operator and `union` or `intersect` functions from the Data.List module.

5) Exercise

a) (0.5 pts.)

For a given list `l` consisting of elements e_i use the appropriate language construction to obtain `l_ind` list of the same length as `l` consisting of two-element tuples (pairs). Each pair consists of the element e_i of list `l` at first position and its index `i` at second position. The index `i` is a number from 1 that indicates the position of element on the list. For example: `l = [1,3,-4,3]` should return: `[(1,1),(3,2),(-4,3),(3,4)]`

b) Exercise (0.5 pts.)

Implement `ind` function which prints indexes of all occurrences of the element given as the first parameter in the list given as the second parameter. For example: `ind 3 [1,3,-4,3]` should return: `[2,4]`

6) Exercise (0.5 pts.)

Complete: `foldl … … [(1,3),(3,-6),(3,9),(3,8)]` function by providing the appropriate first two parameters to obtain the sum `(10,14)` of the 2-dimensional space vectors given as the third parameter.

7) Exercise (0.5 pts.)

---

[2] https://en.wikipedia.org/wiki/Difference_(set_theory)
[3] https://en.wikipedia.org/wiki/Symmetric_difference

Implement `getVarL` function printing 2^{length} two-element lists (variations with repetitions), where lenght = the number of list elements passed as a parameter. Each such 2-element list includes all possible 2-element variations with repetitions of input list elements. The program retrieves elements from the list given as a parameter and adds to each of them one element from the same list, along with the repetition. For example: `getVarL [1,2]` should return: `[[1,1],[1,2],[2,1],[2,2]]`

8) Exercise (0.5 pts.)

Implement `isSorted` that checking if the integer list given as a parameter is sorted. For example: `isSorted [3,1,2]` should return: `False`, and `isSorted [1,2,3]` should return: `True`

9) Exercise

a) (0.5 pts.)

Implement `remFromL` function removing the element given as the first parameter from the list given as the second parameter. For example: `remFromL 3 [1,3,-4,3]` should return: `[1,-4]`

b) Exercise (0.5 pts.)

Implement `getVarNoRepL` function printing length * length – 1 two-element lists (variations without repetitions), where lenght = number of elements of the list passed as a parameter. Each such 2-element list includes all possible 2 element variants without repetitions of input list elements. The program takes in turn the elements from the list given as a parameter and adds to each of them one other element from the list, without repeating. For example: `getVarNoRepL [1,2,3]` should return: `[[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]`

c) Exercise (1 pts.)

Implement `permuteL` function printing all (length !) permutations of the list given as a parameter. You can reimplement the previous task and instead of adding the second element to the pair, add all (length - 1) ! permutations of the list with removed element. This will require a recursive call of `permuteL` and handle the situation ending recursion (passing an empty list as a parameter). For example: `permuteL [1,2,3]` should return: `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`