universität innsbruck

Functional Programming — WS 2024/2025 — LVA 703025

Exercise Sheet 3, 10 points — Deadline: Tuesday, October 29, 2024, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs file that is provided on the proseminar page.
- Upload your modified `Template_03.hs` file in OLAT.
- Your .hs file must be compilable with ghci.
- **Important changes w.r.t. previous sheets**
  - The template has been split into two parts, the actual template, and the tests.
    - Store both Haskell scripts in the same directory.
    - Do not rename the template file, it must be "`Template_03.hs`".
    - Do not change the first line `module Template_03 ...` in the template file.
    - Do not change `Tests_03.hs` at all.
    - Only upload `Template_03.hs` to OLAT, but not `Tests_03.hs`.
  - It is not required to run the tests on your machine, but of course it is useful to test your source before you upload it to OLAT, and perhaps find some programming mistake via these tests.
  - Since we use some testing framework, it requires a bit of setup to run the tests of this exercise sheets and all upcoming sheets locally. The setup is described below and you only have to do it once.
  - In order to execute the tests on your machine, you need to have `LeanCheck` installed, which is an external module that is not part of GHC's base library.
  - In order to install `LeanCheck` we recommend "`cabal`," a tool that – by default – is installed during the installation of GHC via GHCup.
    - Test whether cabal is working by running "`cabal --v`" in a terminal or shell. It should display you some version number.
    - If this is not the case, run "`ghcup tui`" which opens up a user-interface where you can select to install cabal; or if you did not use GHCup, use some package manager, e.g., "`sudo apt install cabal`" in Ubuntu linux.
  - Installing `LeanCheck` via `cabal` works as follows in a terminal.

    ```
    cabal install leancheck --lib
    ```

  - To finally execute the tests, do the following

    ```
    ghci Tests_03.hs
    > runTests
    ```

    or, if this fails to find `LeanCheck`, run the ghci-REPL from cabal as follows:

    ```
    cabal repl
    > :l Tests_03
    > runTests
    ```

Exercise 1 *Pattern Matching* **2 p.**

Consider the following datatype definitions:

```
data Subject = CS | Math | Physics | Biology
data Programme =
    Bachelor Subject
  | Master Subject
  | Teaching Subject Subject -- teachers need two subjects
data Student = Student
  String -- name
  Integer -- matriculation number
  Bool -- active inscription
  Programme
```

Determine which of the expressions 1.-4. match the patterns in (i) and (ii). For each match give the corresponding substitution. (2 points)

1. `Student "Billy Bane" 123456 True (Bachelor Biology)`
2. `Student "Emily Ear" 666666 False (Bachelor Math)`
3. `Student "Dora Dursley" 654321 False (Teaching CS Physics)`
4. `Student "Frank Fauntleroy" 111111 False (Teaching CS Math)`

(i) `Student name _ _ (Teaching _ Math)`

(ii) `Student name n False p@(Bachelor _)`

Exercise 2 *Function Definitions* **3 p.**

1. Define a function `validVehicle :: Vehicle -> Bool` for computing whether a vehicle satisfies all of the following constraints: (1 point)
   - Cars have a positive horse power.
   - The number of wheels of a truck is between 4 and 288.[1]

   Note that several comparison functions of numbers are available in Haskell, e.g. `==`, `<`, `>`, `<=`, `>=`.
2. Define a function `prodList :: List -> Integer` that takes a list of integers (as defined in the lecture) and returns the product of its elements. The product over an empty list should be 1. (1 point)
3. Define a function `squareSecond :: List -> List` that squares every second element in a given list of integers. For instance, if the input represents list $[2, 7, 9, 3, 15, 5, 7]$, then the output should represent $[2, 49, 9, 9, 15, 25, 7]$. (1 point)

Exercise 3 *Recursive Functions* **5 p.**

Recall the datatype `Expr` of simple arithmetic expressions from slide 4 of lecture 3:

```
data Expr = Number Integer | Plus Expr Expr | Negate Expr
```

1. Implement a recursive function `convert :: Expr -> Expr` that eliminates all occurrences of negative Integers in an `Expr` by replacing them with corresponding `Negate` expressions such that `eval` results in the same `Integer` for `e` and `convert e`. (1 point)

   Example: `convert (Plus (Number (-2)) (Number 1)) = Plus (Negate (Number 2)) (Number 1)`

   Hint: Note that in Haskell you can use `>`, `>=`, `<`, `<=`, and `==` to compare two `Integers`. Each of these comparison functions result in a `Bool`.

   Moreover, the following function distinguishing between two possible results depending on a boolean condition might be useful.

   ```
   ite True  x y = x
   ite False x y = y
   ```

[1] 288 is just a number taken from a vehicle that is mentioned at https://de.wikipedia.org/wiki/Self-Propelled_Modular_Transporter. We did not investigate, whether there is any official classification of a "truck".

2. Implement a recursive function `normalize :: Expr -> Expr` that moves all occurrences of `Negate` from an `Expr` ”down the syntax tree” such that `Negate` may only occur directly above `(Number n)` leaf nodes. As in exercise 3.1, we do not want any negative integers in our result. Again, make sure that `eval` results in the same `Integer` for `e` and `normalize e`. (3 points)

   Example:
   ```
   normalize (Negate (Plus (Number (-2)) (Number 1))) = Plus (Number 2) (Negate (Number 1))
   ```
   Hint: It might make sense to implement auxiliary functions.

3. Implement a function `listToExpr :: List -> Expr` that takes a `List` and returns an `Expr` that represents the addition of all the integers in the list. For the addition of numbers in the empty list you may write `Number 0` but for non-empty lists do not add any numbers which are not in the list, i.e. particularly do not add `Number 0` if the list does not contain a `0`. Once again, as in exercise 3.1, we do not want any negative integers in our result. (1 point)

   Examples:
   ```
   listToExpr Empty = Number 0
   listToExpr (Cons 1 (Cons (-2) Empty)) = Plus (Number 1) (Negate (Number 2))
   ```