universität innsbruck

Functional Programming	WS 2023/2024	LVA 703025
Exercise Sheet 9, 10 points		Deadline: Tuesday, December 12, 2023, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_09.hs` provided on the proseminar page.
- Upload your `*.hs` files in OLAT. (Upload each file separately, and do not use zip, etc.)
- Your `*.hs` files must be compilable with `ghci`.

Exercise 1 *Scope of Variable/Function Names* **4 p.**

The following exercises are about the scope of variables and functions.

1. In the Haskell program below, analyze the scope of `radius` in the three functions `operationA`, `operationB`, and `operationC`. Moreover, state which `radius` (global or local) each function refers to and justify your answers. (1 point)

```haskell
radius :: Double
radius = 10  -- global radius

computeVolume :: Double -> Double
computeVolume rad = (4/3)*pi*rad^3

operationA :: Double -> Double
operationA radius = computeVolume radius

operationB :: Double
operationB  = computeVolume radius

operationC :: Double -> Double
operationC = computeVolume
```

2. Analyze the implementation of `reverseList` in the program below. Does it work as expected? Perform the same variable renaming as in the slides from week 9. (1 point)

```haskell
reverseList :: [a] -> [a]
reverseList xs =
    let reverseListAux xs ys = case xs of
            (x:xs) -> reverseListAux xs (x:ys)
            _ -> ys
    in reverseListAux xs []
```

3. Given the following program:

```haskell
squareRootTwo :: Double -> Integer -> Double
squareRootTwo guess n
    | n == 0 = guess
    | otherwise = squareRootTwo ((guess + 2/guess) / 2) (n-1)

squareRootTwoA :: Double -> Integer -> Double
squareRootTwoA guess n
    | n == 0 = guess
    | otherwise = squareRootTwoA ((guess + 2/guess) / 2) (n-1) where n=n

squareRootTwoB :: Double -> Integer -> Double
squareRootTwoB guess n
    | n == 0 = guess
    | otherwise = let n = n-1 in squareRootTwoB ((guess + 2/guess) / 2) n
```

(a) Consider the funtion `squareRootTwo` above which approximates $\sqrt{2}$ based on an initial guess for n iterations. Do `squareRootTwoA` and `squareRootTwoB` work as expected? Justify your answers. (1 point)

(b) Is it considered good practice to have global and local variables/functions of the same name? (1 point)

Solution 1

1. `operationA` has a local variable 'radius', which overwrites the global one. `operationB` does not have a local variable 'radius', hence it uses the global 'radius'. `operationC` is a partial function application and will use whatever argument is passed when the function is invoked.

2. `reverseList` works as expected, i.e., it computes the reverse of some input list.

   ```haskell
   reverseList :: [a] -> [a]
   reverseList xs_1 =
       let reverseListAux xs_2 ys_1 = case xs_3 of
               (x_1:xs_4) -> reverseListAux xs_5 (x_2:ys_2)
               _ -> ys_3
       in reverseListAux xs_6 []
   ```

   - `xs_6` refers to `xs_1`
   - `xs_3` refers to `xs_2`
   - `xs_5` refers to `xs_4`
   - `x_2` refers to `x_1`
   - `ys_2` and `ys_3` refer to `ys_1`

3.
   - `squareRootTwoA` does not terminate. Although the body of the functions looks similar, the where clause does not use the function variable `n`. Rather it uses a local recursive definition which will never terminate, similar to `fun1 = fun1`.
   - `squareRootTwoB` will also never terminate for the same reason. The `n` in the **`let`** expression is defined recursively and not by the `n` passed as a function variable.
   - It is not considered good practice to have global and local function variables with the same name. This is because it can lead to so-called "shadowing" where a global variable is shadowed by the local variable. This can cause confusion and potential bugs.

Exercise 2 *Modules and Property-Based Testing with LeanCheck* **6 p.**

The easiest way to install additional packages for Haskell is the Haskell Tool Stack, called `stack` on the command line. If `stack` is not installed on your system, then please do so.[1] If you installed GHC via `ghcup`, then this is possible by invoking `ghcup install stack`.

[1] https://docs.haskellstack.org/en/stable/install_and_upgrade/

1. First work through the LeanCheck README[2] and then its tutorial[3] so that you are able to use the package and answer basic questions about it. (2 points)

2. Install the LeanCheck package for *property-based testing* via (0.5 points)

   ```
   $ stack install leancheck
   ```

   Make sure that the package is actually available by starting GHCi via

   ```
   $ stack ghci
   ```

   and then entering

   ```
   ghci> import Test.LeanCheck
   ghci> :t holds
   holds :: Testable a => Int -> a -> Bool
   ```

3. Define a module `Tree` that exports the type `Tree` (and its constructors) from Sheet 05 and also the functions `fillXs` and `splitAtLevel`. (0.5 points)

4. Write a `Listable` instance for `Tree a`. (1 point)

5. Use LeanCheck's `check` function to test whether the following property holds:

   > For arbitrary integers `i`, trees `t` and `s`, and lists of trees `ss`, we have that whenever `splitAtLevel i t == (s, ss)`, then also `fillXs s ss == (t, ss)`.

   (2 points)

   Hint: You can do this by following these steps:

   (a) Import LeanCheck and your module `Tree`.

   (b) Insert the `Listable` instance for `Tree a` from above.

   (c) Implement a function

   ```
   prop_splitAtLevel_implies_fillXs ::
     Int -> Tree Int -> Tree Int -> [Tree Int] -> Bool
   ```

   that encodes the property from above. Note that LeanCheck provides the notation `==>` for logical implication. That is, `x ==> y` means "*whenever* `x`*, then also* `y`".

   (d) Use LeanCheck's `check` function to test your property.

Solution 2

3.
```
module Tree (Tree(..), splitAtLevel, fillXs) where

data Tree a = Node a (Tree a) (Tree a) | X deriving (Eq, Show)

splitAtLevel :: Int -> Tree a -> (Tree a, [Tree a])
splitAtLevel _ X = (X, [X])
splitAtLevel i t@(Node x l r)
  | i <= 0 = (X, [t])
  | otherwise = (Node x t1 t2, ts1 ++ ts2)
  where
    (t1, ts1) = splitAtLevel (i - 1) l
    (t2, ts2) = splitAtLevel (i - 1) r

fillXs :: Tree a -> [Tree a] -> (Tree a, [Tree a])
fillXs t [] = (t, [])
fillXs X (t : ts) = (t, ts)
fillXs (Node x l r) ts = (Node x t1 t2, ts2)
  where
    (t1, ts1) = fillXs l ts
    (t2, ts2) = fillXs r ts1
```

[2] https://github.com/rudymatela/leancheck/blob/master/README.md
[3] https://github.com/rudymatela/leancheck/blob/master/doc/tutorial.md

4. ```
   instance Listable a => Listable (Tree a) where
     tiers = cons0 X \/ cons3 Node
   ```

5. ```
   import Test.LeanCheck
   import Tree

   instance Listable a => Listable (Tree a) where
     tiers = cons0 X \/ cons3 Node

   prop_splitAtLevel_implies_fillXs ::
     Int -> Tree Int -> Tree Int -> [Tree Int] -> Bool
   prop_splitAtLevel_implies_fillXs i t s ss =
     splitAtLevel i t == (s, ss) ==> fillXs s ss == (t, ss)
   ```

   ```
   ghci> check prop_splitAtLevel_implies_fillXs
   ```