universität innsbruck

Functional Programming | WS 2021 | LVA 703025

Exercise Sheet 4, 10 points | Deadline: Wednesday, November 3, 2021, 6am

- 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 .hs-file(s) of Exercises 1 and 2 in OLAT.
- Your .hs-file should be compilable with ghci.

Exercise 1 *Nested Lists and Either* **5 p.**

1. Study the slides of week 4, pages 19 & 20 to understand the consequences of the definition of the predefined string type.

```haskell
type String = [Char]
(++) :: [a] -> [a] -> [a]    -- and not: String -> String -> String
head :: [a] -> a             -- and not: String -> Char
```

   Given a function `concat :: [[a]] -> [a]`, briefly explain the type of the following six Haskell expressions or give a reason why these expressions result in a type error.

```haskell
e1 = concat [1 :: Int, 2, 3]
e2 = concat ["one", "two", "three"]
e3 = concat [[1 :: Int, 2], [], [3]]
e4 = concat [["one", "two"], [], ["three"]]
e5 = concat e3
e6 = concat e4
```

(1 point)

2. Define a function `suffixes` that computes the list of all suffixes of a list. Particularly, the following identities should hold:

```haskell
suffixes [1, 2, 3] = [[1,2,3], [2,3], [3], []]
suffixes "hello"   = ["hello", "ello", "llo", "lo", "o", ""]
```

   Hint: structural recursion suffices.

(1 point)

3. Define a function `prefixes` that computes the list of all prefixes of a list. Particularly, the following identities should hold:

```haskell
prefixes [1, 2, 3] = [[1,2,3], [1,2], [1], []]
prefixes "hello"   = ["hello", "hell", "hel", "he", "h", ""]
```

   Hint: you might need an auxiliary function; structural recursion is not recommended for `prefixes`.

(2 points)

4. Utilize the `Either` type to create a menu that generates the list of prefixes, suffixes or a meaningful error message depending on its input. Particularly, the following identities should hold:

```haskell
menu 'p' [1,2,3]  = Right [[1,2,3],[1,2],[1],[]]
menu 'p' "hello"  = Right ["hello","hell","hel","he","h",""]
menu 's' [1,2,3]  = Right [[1,2,3],[2,3],[3],[]]
menu 's' "hello"  = Right ["hello","ello","llo","lo","o",""]
menu 'c' "hello"  = Left  "(c) is not supported, use (p)refix or (s)uffix"
```

(1 point)

Solution 1

```
{-
e1 is not type correct, since [[a]] is not more general than [Int]
e2 :: String   -- substitute a/Char (recall String = [Char], so [String] = [[Char]])
e3 :: [Int]    -- substitute a/Int
e4 :: [String] -- substitute a/String
e5 is not type correct, since [[a]] is not more general than [Int], see e3 and e1
e6 :: String   -- substitute a/Char, see e2 and e4
-}

suffixes :: [a] -> [[a]]
suffixes [] = [[]]
suffixes xs@(_ : ys) = xs : suffixes ys

{-
init is a predefined function that drops the last element of a non-empty list,
i.e., it computes the initial part of a list;
it could also be user-defined or named differently, e.g., dropLast

init :: [a] -> [a]
init [x] = []
init (x : xs) = x : init xs
-}

prefixes :: [a] -> [[a]]
prefixes [] = [[]]
prefixes xs = xs : prefixes (init xs)

menu :: Char -> [a] -> Either String [[a]]
menu 'p' xs = Right (prefixes xs)
menu 's' xs = Right (suffixes xs)
menu a _ = Left ("(" ++ [a] ++ ") is not supported, use (p)refix or (s)uffix")
```

Exercise 2 *Polymorphic Expressions* **5 p.**

1. Define a polymorphic datatype to represent expressions involving addition, multiplication and numbers. In particular `expr1` and `expr2` should be accepted.

   ```
   expr1 = Times (Plus (Number (5.2 :: Double)) (Number 4)) (Number 2)
   expr2 = Plus (Number (2 :: Int)) (Times (Number 3) (Number 4))
   expr3 = Times (Number "hello") (Number "world")
   ```

   Is `expr3` type correct as well? Provide a brief explanation. (1 point)

2. Write a polymorphic function `numbers` that given an expression constructs a list of numbers that occur in the expression. For example `numbers expr1 = [5.2,4,2]` and `numbers expr2 = [2,3,4]`.

   Also provide a type for your function that is as general as possible. (1 point)

3. Write a polymorphic function `eval` to evaluate an expression. For example `eval expr1 = 18.4` and `eval expr2 = 14`.

   Also provide a type for your function that is as general as possible. (1 point)

4. Write a polymorphic function `exprToString` that converts an expression into a string that represents the expression. The string should insert parentheses only if they are required. For example:

   - `exprToString expr1 = "(5.2 + 4.0) * 2.0"`
   - `exprToString expr2 = "2 + 3 * 4"`

   Also provide a type for your function that is as general as possible. (2 points)

Solution 2

```haskell
data Expr a =
    Number a
  | Times (Expr a) (Expr a)
  | Plus (Expr a) (Expr a)
  deriving Show

numbers :: Expr a -> [a]
numbers (Number x) = [x]
numbers (Times e1 e2) = numbers e1 ++ numbers e2
numbers (Plus e1 e2)  = numbers (Times e1 e2)

eval :: Num a => Expr a -> a
eval (Number x) = x
eval (Times e1 e2) = eval e1 * eval e2
eval (Plus e1 e2) = eval e1 + eval e2

paren :: Expr a -> String -> String
paren (Plus _ _) s = "(" ++ s ++ ")"
paren _ s = s

exprToString :: Show a => Expr a -> String
exprToString (Number x) = show x
exprToString (Plus e1 e2) =
  exprToString e1 ++ " + " ++ exprToString e2
exprToString (Times e1 e2) =
  paren e1 (exprToString e1) ++ " * " ++ paren e2 (exprToString e2)
```