universität innsbruck

Functional Programming	WS 2022	LVA 703025

Exercise Sheet 3, 10 points — Deadline: Wednesday, October 26, 2022, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_03.hs` provided on the proseminar page.
- Upload your modified .hs file for Exercises 1.1, 1.2 and 2 in OLAT.
- Your .hs file should be compilable with `ghci`.

Exercise 1 *Pattern Matching and Function Definitions* — 4 p.

Consider the following definitions, describing boolean expressions and the conjunction function as shown on slide 17 of lecture 3:

```
data BoolExpr = And BoolExpr BoolExpr
    | Or BoolExpr BoolExpr
    | Impl BoolExpr BoolExpr
    | Not BoolExpr
    | Atom Bool

conjLecture :: Bool -> Bool -> Bool
conjLecture True b = b
conjLecture False _ = False
```

1. Give an alternative implementation `conj :: Bool -> Bool -> Bool` of `conjLecture`. (0.75 points)
2. Implement logical disjunction, implication, and negation as functions
   - `disj :: Bool -> Bool -> Bool`
   - `impl :: Bool -> Bool -> Bool`
   - `not2 :: Bool -> Bool`

   following the example of the conjunction function from above, that is, not using the built-in logical operators. Try to minimize the number of defining equations using pattern matching.

   Hint: The truth tables of those functions are:

   Disjunction ($\vee$):

a	b	$a \vee b$
F	F	F
T	F	T
F	T	T
T	T	T

   Implication ($\rightarrow$):

a	b	$a \rightarrow b$
F	F	T
T	F	F
F	T	T
T	T	T

   Negation ($\neg$):

a	$\neg a$
F	T
T	F

   (2.25 points)
3. Find a pattern such that the expression

   `And (Atom True) (Impl (Not (Atom True)) (Atom False))`

   yields the substitutions `a/True`, `b/(Not (Atom True))` when matched against it. (1 point)

Exercise 2 *Recursive Functions* **6 p.**

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

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

as well as the function `eval` that evaluates an `Expr` into an `Integer` (slide 21 of lecture 3):

```
eval (Number x)    = x
eval (Plus e1 e2) = eval e1 + eval e2
eval (Negate e)    = - eval e
```

Moreover, consider the datatype `data Nat = Zero | Plus1 Nat` representing natural numbers (that is, non-negative integers) as consecutive applications of "+1." For example, "2" is represented as `Plus1 (Plus1 Zero)`.

1. Implement a recursive function `normalize :: Expr -> Expr` that eliminates all occurrences of `Negate` from an `Expr` such that `eval` results in the same `Integer` for `e` and `normalize e`.

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

   Hint: It might be useful to first implement a *smart constructor* for `Negate`, that is, a function say `neg` that when applied to an `Expr` behaves like `Negate` with respect to `eval` but never actually adds the constructor `Negate`. For example, `neg (Number 1) = Number (-1)`. (3 points)

2. Implement a function `showNat :: Nat -> String` that computes a readable `String` representation of `Nat`s. Take care that "0" is only used in the result when it is necessary. (Remember that `String`s in Haskell are enclosed in double quotes "`"`" and are concatenated using the "`++`" operator.)

   Examples: `showNat Zero = "0"`, `showNat (Plus1 (Plus1 (Plus1 Zero))) = "1+1+1"` (1 point)

3. Implement a function `nat :: Integer -> Nat` that takes an `Integer` and turns it into the corresponding `Nat` (use `Zero` as result for negative integers).

   Example: `nat 2 = Plus1 (Plus1 Zero)` (1 point)

   Hint: Note that in Haskell you can use `>`, `>=`, `<`, `<=`, and `==` to compare two `Integer`s. 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
   ```

4. Implement a function `exprToNat :: Expr -> Nat` that turns a given `Expr` into the corresponding `Nat` (use `Zero` for expressions that do not correspond to a `Nat`).

   Examples:
   `exprToNat (Plus (Number 1) (Number 1)) = Plus1 (Plus1 Zero)`
   `exprToNat (Plus (Number 2) (Negate (Number 1))) = Plus1 Zero` (1 point)