Homework 3

Functional Programming (ITI0212)

due: 2021.04.16

Place your solutions in a module named `Homework3` in a file with path `homework/Homework3.idr` within your `iti0212-2021` repository on the TalTech GitLab server (`https://gitlab.cs.ttu.ee/`). Your solutions will be pulled automatically for marking. At the start of the file include a comment containing your name and the Idris version you are using. Precede each problem's solution with a comment specifying the problem number.

Problem 1
An *endomorphism* is a function whose argument and result types are the same. Write `Semigroup` and `Monoid` instances for endomorphisms:

```
implementation  Semigroup (a -> a)  where

implementation  Monoid (a -> a)  where
```

such that:

```
> (( + 1) <+> ( * 2) <+> neutral) 3
8
> (not <+> neutral <+> not) False
False
> (neutral <+> Maybe <+> List) Nat
List (Maybe Nat)
```

Problem 2
Complete the definition of the function `applicify`, which takes any binary operation and extends it to any applicative type constructor:

```
applicify  :  {t : Type -> Type} -> Applicative t =>
  (op : a -> a -> a) -> t a -> t a -> t a
```

Using this function you can easily define operators such as:

```
infixl 7 +?
(+?)  :  Num a => Maybe a -> Maybe a -> Maybe a
(+?)  =  applicify (+)

infixl 7 +*
(+*)  :  Num a => {n : Nat} -> Vect n a -> Vect n a -> Vect n a
(+*)  =  applicify (+)
```

which behave as follows:

```
> Just 3 +? Just 4 +? Just 5
Just 12
> Just 3 +? Nothing +? Just 5
Nothing
```

```
> [1,2,3] +* [4,5,6] +* [7,8,9]
[12, 15, 18]
```

Problem 3
Use interactive editing in order to write functions with each of the following types:

```
mapPair  :  (f : a -> a') -> (g : b -> b') ->
  Pair a b -> Pair a' b'

mapDPair  :  (f : a -> a') -> (g : {x : a} -> b x -> b' (f x)) ->
  DPair a b -> DPair a' b'
```

Problem 4
Write a function that when given a `Nat` arity `n` and a type, computes the type of `n`-ary operations on the given type:

```
ary_op  :  (arity : Nat) -> Type -> Type
```

For example:

```
> 0 `ary_op` Nat
Nat
> 1 `ary_op` Nat
Nat -> Nat
> 2 `ary_op` Nat
Nat -> Nat -> Nat
> the (3 `ary_op` Nat) (\ x, y, z => (x + y) * S z) 3 4 5
42
```

Using the following type constructor found in the standard library (in `Data.List` for Idris 1 and in `Data.List.Elem` for Idris 2),

```
data  Elem : a -> List a -> Type  where
  Here  :  Elem z (z :: xs)
  There :  Elem z xs -> Elem z (x :: xs)
```

we can interpret the type `Elem z xs` as the proposition that the element `z` occurs somewhere within the list `xs`. Import the library containing this definition in order to solve the next two problems about list concatenation.

Hint: consider the recursive structure of the list concatenation function and think about which argument to induct on in order to best follow it.

Problem 5
Prove that if an element occurs within a given list then it also occurs within the concatenation of that list with any other list:

```
in_left  :  Elem z xs -> (ys : List a) -> Elem z (xs ++ ys)
```

Problem 6
Prove that if an element occurs within a given list then it also occurs within the concatenation of any other list with that list:

```
in_right :  Elem z ys -> (xs : List a) -> Elem z (xs ++ ys)
```