Homework 3

Functional Programming

due: 2024-03-06

Place your solutions in a module named `Homework3` in a file with path `homework/Homework3.idr` within your course git repository. Please submit *only* your Idris source file. At the beginning of the file include a comment containing your name. Precede each problem's solution with a comment specifying the problem number and `public export` each definition that you are asked to write so that it can be `import`ed for testing.

Whether or not it is complete, the solution file that you submit should load without errors. If you encounter a syntax or type error that you are unable to resolve, use comments or goals to isolate it from the part of the file that is interpreted by Idris.

Your solutions will be pulled automatically for marking shortly after the due date.

Problem 1
Recall the type constructor for (node-labeled binary) trees. Write an implementation of the `Eq` interface for `Tree`s whose element types themselves implement this interface:

```
implementation  Eq a => Eq (Tree a)  where
```

Your `(==)` method should return `True` just in case the two trees have the same shape and also have equal elements at each position.

Problem 2
Write a function that returns the list of indices at which an item occurs in a list:

```
indices  :  ?constraint_on_a => a -> List a -> List Nat
```

As part of the problem you should determine what constraint is needed on the parameter type `a`.

Your function should behave as follows:

```
Homework3> indices 2 [1,2,3,2,1]
[1, 3]
Homework3> indices True $ map ( `mod` 2 == 0) [1,2,3,4,5,6]
[1, 3, 5]
```

Problem 3
As we have seen, the default implementation of the `Eq` interface for `List` types compares lists for equality *element-wise*. Write a named implementation of the `Eq` interface for `List` types that compares them *multiset-wise*:

```
implementation  [multisetwise] Eq a => Eq (List a)  where
```

that is, two lists should be considered equal just in case each element that occurs in one list occurs with the same multiplicity in the other list:

```
Homework3> (==) @{multisetwise} [1,2,3] [3,2,1]
True
Homework3> (==) @{multisetwise} [1,2,2,3] [3,2,2,1]
True
Homework3> (==) @{multisetwise} [1,2,3] [1,2,3,3]
False
```

Hint: The following standard library functions may be helpful:

```
elem             :  Eq a => a -> List a -> Bool
Data.List.delete :  Eq a => a -> List a -> List a
```

Alternatively, you can use the function we defined in lecture:

```
multiplicity  :  Eq a => a -> List a -> Nat
```

Problem 4
In lecture we saw how to sort a list whose element type is ordered using the *quicksort* algorithm. For this problem you will write a list sorting function using the *mergesort* algorithm:

```
mergesort  :  Ord a => List a -> List a
```

This sorting algorithm consists of the following steps:

1. *split* the argument list into two strictly shorter lists (if possible, otherwise it is already sorted),
2. *sort* each of these two lists recursively,
3. *merge* the two sorted lists into one sorted list.

For step 3 you can use the function,

```
merge_list  :  Ord a => List a -> List a -> List a
```

that you wrote in lab 5. For step 1 you can define a helper function,

```
split_list  :  List a -> Pair (List a) (List a)
```

that divides a list into two strictly shorter lists (unless the argument list has fewer than two elements, in which case both of the result lists cannot be shorter than the argument list).

Problem 5
Write functions with each of the following types:

```
joinIO :  IO (IO a) -> IO a

mapIO  :  (a -> b) -> IO a -> IO b

appIO  :  IO (a -> b) -> IO a -> IO b
```

Do this without using standard library functions that we haven't discussed yet in this course. Specifically, your definitions should be written in terms of the monadic combinators that we learned about,

- `pure  :  a -> IO a` and
- `(>>=) :  IO a -> (a -> IO b) -> IO b`

or the `do`-notation equivalent, if you prefer.

Problem 6
Write a function,

```
insist  :  IO (Maybe a) -> IO a
```

which runs its argument computation repeatedly until it returns `Just` a result, at which point it returns that result.

For example:

```
Homework3> :exec insist try_number >>= printLn
Please enter a number: no
Please enter a number: forty two
Please enter a number: 42
42
```

Problem 7
Suppose that we have a list of computations, each of type `IO (Either error Unit)`, which when run may yield either the result `Right ()` if it completes normally or else `Left e`, where `e` is an element of some error type, if something goes wrong. Write a function that takes a list of such computations and returns a computation that tries to run them in order, but stops if it encounters an error, returning the error and discarding any pending computations from the list:

```
tryIOs  :  List (IO (Either error Unit)) -> IO (Maybe error)
```

For example, if

```
success , failure  :  IO (Either String Unit)
success  =  putStrLn "okay" >>= const (pure (Right ()))
failure  =  putStrLn "oops" >>= const (pure (Left "error"))
```

then:

```
Homework3> :exec tryIOs [success] >>= printLn
okay
Nothing
Homework3> :exec tryIOs [success , success] >>= printLn
okay
okay
Nothing
Homework3> :exec tryIOs [success , failure , success] >>= printLn
okay
oops
Just "error"
```