Homework 2

Functional Programming

due: 2024-02-21

Place your solutions in a module named `Homework2` in a file with path `homework/Homework2.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.

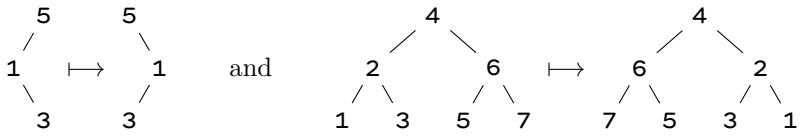
Problem 1
Write a(ny possible terminating) function with each of the following types:

```
fun1  :  Either a a -> a
fun2  :  Pair (Pair a b) c -> Pair a (Pair b c)
fun3  :  Pair a (Either b c) -> Either (Pair a b) (Pair a c)
```

Problem 2
Recall the type constructor for (node-labeled binary) trees. Write a generic function that reflects the structure of a tree.

```
reflect  :  Tree a -> Tree a
```

For example:



Problem 3
Write a recursive function that returns the greatest number in a list, if there is one:

```
greatest  :  List Integer -> Maybe Integer
```

For example:

```
Homework2> greatest []
Nothing
Homework2> greatest [1, 2, 3, 3, 2, 1]
Just 3
```

Hint: You can use the function `max : Integer -> Integer -> Integer` to get the larger of two `Integer`s.

Problem 4
Define a *type isomorphism* between the types `Maybe a` and `Either Unit a`, generic in the parameter type `a`. Recall that this means defining back-and-forth functions,

```
forward  :  Maybe a -> Either Unit a
backward :  Either Unit a -> Maybe a
```

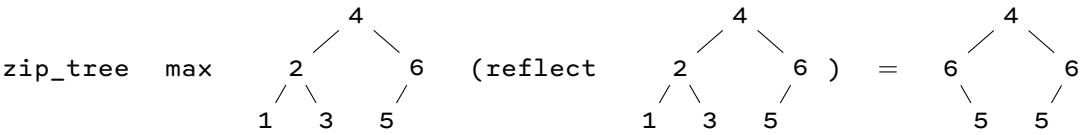
such that:

- `backward (forward x)` evaluates to `x` for any `x : Maybe a`, and
- `forward (backward y)` evaluates to `y` for any `y : Either Unit a`.

Problem 5
Write the *zip* function for trees, which has the following type:

```
zip_tree  :  (a -> b -> c) -> Tree a -> Tree b -> Tree c
```

which should behave as follows:



Problem 6
Use recursion to write the *flatten* function for lists:

```
flatten_list  :  List (List a) -> List a
```

which should behave as follows:

```
Homework2> -- flatten an empty list of lists:
Homework2> flatten_list []
[]
Homework2> -- flatten a non-empty list of empty lists:
Homework2> flatten_list [[] , [] , []]
[]
Homework2> -- flatten a non-empty list of non-empty lists:
Homework2> flatten_list  [[1,2,3] , [4,5,6] , [7,8,9]]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Problem 7
Now rewrite the flatten function for lists using the *fold* for lists, which has the following type:

```
fold_list  :  (c : a -> t -> t) -> (n : t) -> List a -> t
```

Note: Your solution should *only* call this fold function and *not* use any pattern matching nor recursion. In other words, you should write your function by completing the goals `?c` and `?n` below.

```
flatten_list'  :  List (List a) -> List a
flatten_list'  =  fold_list ?c ?n
```

Problem 8
Write the *fold* function for natural numbers, `fold_nat`. Then describe in words (as a comment) what this function does.