Homework 1

Functional Programming (ITI0212)

due: 2023-03-06

Place your solutions in a module named `Homework1` in a file with path `homework/Homework1.idr` within a repository called `iti0212-2023` on the TalTech GitLab server (`https://gitlab.cs.ttu.ee/`). Submit only your Idris source file. Do not include any build artifacts, such as `.tmp`, `.ttc`, or `.ttm` files.

At the beginning of the file include a comment containing your name. Precede each problem's solution with a comment specifying the problem number.

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 holes 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
The *Fibonacci function* was discovered over 2200 years ago and describes a surprising array of phenomena. It has the following type:

```
fib  :  Nat -> Nat
```

and is recursively defined as:

$$\text{fib } n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \text{fib } (n-1) + \text{fib } (n-2) & \text{otherwise} \end{cases}$$

Write the Fibonacci function in Idris using pattern matching on the argument `Nat`. Confirm that it returns correct results for some low argument values according to `https://oeis.org/A000045`.

Problem 2
Write a recursive definition for the exponentiation function on the natural numbers, m^n:

```
exp  :  Nat -> Nat -> Nat
```

For example:

```
Homework1> exp 2 0
1
Homework1> exp 2 1
2
Homework1> exp 2 2
4
Homework1> exp 2 3
8
```

Hint: Think about in which argument this function is recursive, and use your inductive definition of multiplication from lab 2 as a guide.

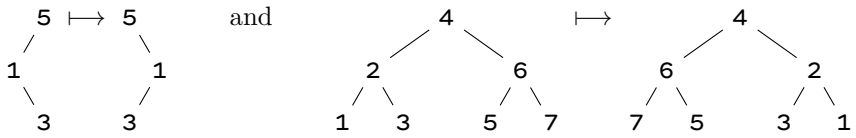
Problem 3
Write a(ny possible terminating) function with each of the following types:

```
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 4
Recall the type constructor for *node-labeled binary trees* from lab 3. Write a function that reflects the structure of a tree.

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

For example:



Problem 5
Write a function that returns the greatest `Integer` in a list, if there is one:

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

For example:

```
Homework1> greatest []
Nothing
Homework1> greatest [1]
Just 1
Homework1> 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 6
Define a *type isomorphism* between the types `Maybe a` and `Either Unit a`, generically in the parameter type `a`. Recall from lab 3 that this means defining back-and-forth functions,

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

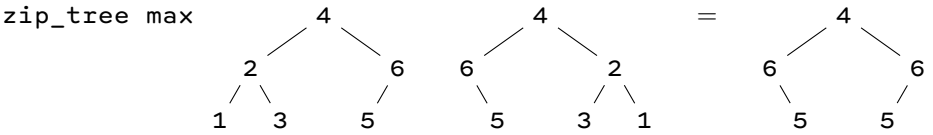
such that:

- `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 7
Write the `zip` function for `Tree`s, which has the following type:

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

which should behave as follows:



Problem 8

- Write the flatten function for lists:

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

  which should behave as follows:

  ```
  Homework1> -- flatten an empty list of lists:
  Homework1> flatten_list []
  []
  Homework1> -- flatten a non-empty list of empty lists:
  Homework1> flatten_list [[] , [] , []]
  []
  Homework1> -- flatten a non-empty list of non-empty lists:
  Homework1> flatten_list  [[1,2,3] , [4,5,6] , [7,8,9]]
  [1, 2, 3, 4, 5, 6, 7, 8, 9]
  ```

- Now rewrite the flatten function for lists using the fold for lists, which was presented in lecture 4 and has the following type:

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

  Note: Your solution should only call this function and not use any pattern-matching or recursion. In other words, you should write this function by completing the goals `?n` and `?c` below.

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

Hint: `:doc Prelude.List.(++)`.