Homework 1

Functional Programming (ITI0212)

due: 2022-03-02

Place your solutions in a module named `Homework1` in a file with path `homework/Homework1.idr` within a repository called `iti0212-2022` on the TalTech GitLab server (`https://gitlab.cs.ttu.ee/`). Your solutions will be pulled automatically for marking shortly after the due date.

At the start of the file include a comment containing your name as it appears in your university records. Precede each problem's solution with a comment specifying the problem number.

The solution file that you submit should load without errors. If you encounter a syntax or type error that you are unable to resolve, please use comments or holes to isolate them from the part of the file interpreted by Idris.

Problem 1
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
```

Problem 2
The *Ackermann function* is a famously fast-growing total computable function with the following type:

```
ack  :  Nat -> Nat -> Nat
```

and recursively defined by:

$$\text{ack}\ m\ n = \begin{cases} n+1 & \text{if } m = 0 \\ \text{ack}\ (m-1)\ 1 & \text{if } m \neq 0 \text{ and } n = 0 \\ \text{ack}\ (m-1)\ (\text{ack}\ m\ (n-1)) & \text{otherwise} \end{cases}$$

Write the Ackermann function in Idris using pattern matching. Make sure Idris agrees that your function is total and confirm that it returns correct results for some low argument values according to `https://en.wikipedia.org/wiki/Ackermann_function#Table_of_values`.

Problem 3
Write (any possible) total functions with each of the following types:

```
fun1  :  (c -> a) -> (c -> b) -> c -> Pair a b
fun2  :  Pair (Pair a b) c -> Pair a (Pair b c)
fun3  :  Pair a (Either b c) -> Either (Pair a b) (Pair a c)
fun4  :  Pair (a -> b) (c -> d) -> Either a c -> Either b d
```

Problem 4
Write a higher-order function that uses a given function to transform the element at the specified index of a list:

```
transform  :  (f : a -> a) -> (index : Nat) -> List a -> List a
```

If the index is out-of-bounds for the list then your function should behave like the identity function. For example:

```
> transform S 0 [1 , 2 , 3]
[2, 2, 3]
> transform S 1 [1 , 2 , 3]
[1, 3, 3]
> transform S 2 [1 , 2 , 3]
[1, 2, 4]
> transform S 3 [1 , 2 , 3]
[1, 2, 3]
```

Problem 5
Write a function that capitalizes the first character of each word of a string. For example:

```
> titlecase "it was the best of times it was the worst of times"
"It Was The Best Of Times It Was The Worst Of Times"
```

You may assume that the words are composed of letters and are separated by whitespace. The following standard library functions will be helpful, you should `:doc` them:

- `words : String -> List String`,
- `unwords : List String -> String`,
- `unpack : String -> List Char`,
- `pack : List Char -> String`,
- `toUpper : Char -> Char`.

The functions `toUpper`, `pack` and `unpack` are in the module `Prelude`, which is imported automatically by default. The functions `words` and `unwords` are in the module `Data.String`, which you will need to `import` in order to use.

tip: you can write this as a one-liner using your function from problem 4, function composition, and the prelude function `map : (a -> b)-> List a -> List b`, which applies the given function to each element of the given list.

Problem 6
Write the `zip` function for the type of node-labeled binary trees:

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

Note: recall that we met `Tree` types in lab 3.

Problem 7
Write the `fold` function for `Tree` types, call it `fold_tree`. You will need to work out its type as well as its definition.

Problem 8
Use the `fold` function for trees that you wrote in problem 7 to rewrite the `size` function from lab 3 as a fold.

```
size  :  Tree a -> Nat
size  =  fold_tree ?g1 ?g2
```

Note: your solution should be one line with expressions substituted for the two goals above. It should contain no case analysis nor recursion, the `fold_tree` function should already take care of those things.