Homework 1

Functional Programming (ITI0212)

Deadline: Thursday 19th March 2026

Task 1

The *Ackermann function* is a famously fast-growing (total, computable) function that takes two natural numbers and returns a natural number, and is 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 Lean using pattern matching. Your function should be **total** (i.e. don't use the `partial` keyword!). Confirm that it returns correct results for some low argument values according to `https://en.wikipedia.org/wiki/Ackermann_function#Table_of_values`.

Task 2

Write generic functions with each of the following types:

`diag` : $\alpha \to \alpha \times \alpha$

`anyway` : $\alpha \oplus \alpha \to \alpha$

`assocr` : $(\alpha \times \beta) \times \gamma \to \alpha \times (\beta \times \gamma)$

`distrib` : $\alpha \times (\beta \oplus \gamma) \to (\alpha \times \beta) \oplus (\alpha \times \gamma)$

Task 3

Write the generic function that consolidates a `List` of `(Option` α`)`s into an element of `Option (List` α`)`. It should return `some` (list) just in case the argument is a list of `some`s. For example:

```
#eval consolidate []
=> some []
#eval consolidate [some 1 , some 2 , some 3]
=> some [1, 2, 3]
#eval consolidate [some "A", none, some "C"]
=> none
```

Task 4

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

`transform : (f :` $\alpha \to \alpha$`) (i : Nat) (xs : List` α`) : List` α

For example:

```
#eval transform S 0 [1, 2, 3]
=> [2, 2, 3]
#eval transform S 1 [1, 2, 3]
```

```
=> [1, 3, 3]
#eval transform S 2 [1, 2, 3]
=> [1, 2, 4]
#eval transform S 3 [1, 2, 3]
=> [1, 2, 3]
```

Task 5
Write a function that capitalizes the first character of each word of a string.

```
#eval 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 separated by whitespace. The following standard library functions may be helpful:

- `String.splitOn : String → List String` (with an optional third argument giving the string to split on, which is a space by default)
- `String.intercalate " " : List String → String`, which joins a list of strings into a string by intercalating a space (note the partial application of `String.intercalate` to `" "`, which is the string to intercalate).
- `String.toList : String → List Char`
- `List.asString : List Char → String`,
- `Char.toUpper : Char → Char`.

Tip: Your function from task 4 may be helpful, as well as standard higher-order functions like function composition ($\circ$) and `List.map`.

Task 6
Recall the inductive type `Tree` from Lab 3. Write the `fold` function for `Tree`s.

You will need to first work out what type this function should have. Recall that a fold can be conceived of as replacing the element constructors of an inductive type with functions and values, yielding a value of another type β.

To write the type of the fold function for an inductive type:

1. Examine the types of its element constructors, e.g.

   `[] : List` α
   `(::) :` $\alpha \to$ `List` $\alpha \to$ `List` α

2. In each constructor replace the type itself with a new type parameter, which will be the return type, e.g.

   `n :` β
   `c :` $\alpha \to \beta \to \beta$

3. The fold function has one argument for each constructor-replacing term and return a function from the type being folded to the parameter type β:

   `fold_list : (c :` $\alpha \to \beta \to \beta$`) → (n :` β`) → List` $\alpha \to \beta$

Task 7
Reimplement the `size : Tree` $\alpha \to$ `Nat` function from Lab 3 as a one-line function using your fold function for `Tree`s from Task 6.

Hint: you will need to use a function literal.