Lab 3: Parameterized types and generic functions

Functional Programming (ITI0212)

This week, we saw **parameterized families of inductive types**. These have types of the form `Type -> ... -> Type`. Examples include `List` and `Option`, which take one type parameter (`Type -> Type`), and `Prod` and `Sum`, which take two type parameters (`Type -> Type -> Type`).

We also learned how to write generic functions, which act uniformly on the types of such families. We saw how to use implicit arguments to avoid having to pass type arguments explicitly.

Task 1
Define a function that swaps the elements of a pair:

```
swapPair : Prod α β → Prod β α
```

You can insert Greek letters by typing e.g. `\alpha`. Recall that Lean automatically inserts implicit arguments when it encounters unbound Greek or lowercase letters, so the full type of `swapPair` is

```
swapProd : {α β : Type} → Prod α β → Prod β α
```

If you want to be fancy, you can use the infix notation $\alpha \times \beta$ for `Prod α β`, where $\times$ is inserted as `\times`.

Task 2
Define a function that swaps the left and right components of a `Sum` type:

```
swapSum : Sum α β → Sum β α
```

Lean also defines the notation $\alpha \oplus \beta$ (`α \oplus β`) for `Sum α β`.

Question: Did you have any choice in how you defined the functions in Tasks 1 and 2?

Task 3
Define a recursive function that reverses a list:

```
reverseList : List α → List α
```

For example:

- `reverseList []` evalutes to `[]`
- `reverseList [1, 2, 3]` evaluates to `[3, 2, 1]`

Use recursion and the list concatenation function:

```
List.append : List α → List α → List α
```

(which has infix notation `++`).

Task 4
The following (parameterized) inductive type defines *node-labelled binary trees.*

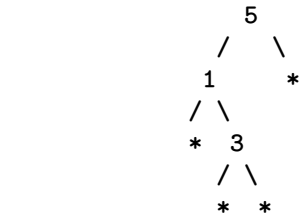
```
inductive Tree (α : Type) where
  | leaf : Tree α
  | node : Tree α -> α -> Tree α -> Tree α
```

That is, a tree with nodes labelled by α is either a leaf, or a node with a (left) tree (with nodes labelled by α), a value of type α and a (right) tree (with nodes labelled by *alpha*).

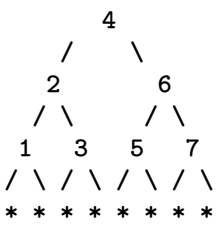
It is customary to draw trees as downward growing diagrams. For example, the term

```
def t1 :  Tree Nat := .node (.node .leaf 1 (.node .leaf 3 .leaf)) 5 .leaf
```

has a node with left branch `(.node .leaf 1 (.node .leaf 3 .leaf))`, label `5` and right branch `leaf`, etc. we depict this as,



where `*` stands for a `leaf`. Write a Lean term for `t2 :  Tree Nat` representing this tree:



Task 5
Define a function that counts number the nodes in a tree:

```
size : Tree α → Nat
```

For example: `size t1` evaluates to 3, and `size t2` evaluates to 7.

Task 6
A *type isomorphism* is a pair of "back-and-forth" functions between two types `f : α → β` and `g : β → α`, such that if we apply either one to the result of applying the other, then we get back the original argument; that is, for any `x : α` and `y : β` we have that `g (f x)` evaluates to `x` and `(f (g y))` evaluates to `y`.

Write a type isomorphism between the types `Nat` and `List Unit`:

```
n_to_lu : Nat → List Unit
lu_to_n : List Unit → Nat
```

They should satisfy, e.g.

- `lu_to_n (n_to_lu 42)` evaluates to `42`
- `n_to_lu (lu_to_n [(), (), ()])` evaluates to `[(), (), ()]`