Lab 9: Dependent types in action

Functional Programming (ITI0212)

This week we are learning about programming with dependent types. We saw how to write expressions that compute the types of other expressions, including the type of the `filter` function for `Vect` types and that of the `printf` function.

A useful technique for writing recursive functions is to use an *accumulator*, which is an additional argument that keeps track of how the value computed by a function changes with each recursive call. The value of the accumulator is then used to compute the result when a base case is reached.

Task 1
Write the ternary boolean *majority* function, which returns the `Bool` that occurs most often among its arguments, and whose type can be written using the `ary_op` type constructor from lecture:

Task 2
There is a similar *majority* function that takes a list of booleans as an argument (let's stipulate that ties go to `true`): Write this function in such a way that it makes *exactly one* pass over its argument list, which is optimal. Note that functions like `length`, `filter`, `count` (or `accepts`) *each* make one pass over a list, as you can confirm by printing them.

For example:

```
#eval list_majority [true,false]
true
#eval list_majority [true,false,false]
false
```

Hint: you can use a helper function that takes an additional *accumulator* argument that keeps track of what you know about the majority so far. When you reach the base case of an empty list you can use this accumulator to decide which boolean wins. As a bonus, your function will most likely be *tail recursive*, which means that it can run in constant space on a stack-based interpreter.

Task 3
Write a function `as_fin` that takes two `Nat` arguments and tries to interpret the first as an element of the `Fin` type determined by the second.

For example:

```
#check as_fin 2 4
as_fin 2 4 : Option (Fin 4)
#check as_fin 2 3
as_fin 2 3 : Option (Fin 3)
#check as_fin 2 2
as_fin 2 2 : Option (Fin 2)
```

Task 4
Let's practise defining our own inductive types. As a warm-up, define the recursive function `is_even` which takes a natural number and returns a boolean if the given number is even. Of course, you could also define it a non-recursive version.

Then, define an inductive type `Even` that represents the set of even natural numbers. Try to look at the definition of `Nat` and refine the type for this purpose.

Compare your two definitions and try to understand the differences. Do you think one is better than the other? Why?

Task 5
Define an inductive type `Sorted` that represents the set of sorted lists of natural numbers:

```
inductive Sorted : List Nat → Prop
```

Try to think about the structure of the type `List` i.e. what does it mean for an empty list to be sorted? And what about a list with one element?