Linguagens de Programação: Lab Session 2

Important: Before you tackle the exercises listed below, please make sure that you have covered the material discussed in Lecture 1 and Lecture 2.

Exercise 1

Consider the following two inductively defined types.

```
Inductive mumble : Type :=
  | a : mumble
  | b : mumble → nat → mumble
  | c : mumble.

Inductive grumble (X:Type) : Type :=
  | d : mumble → grumble X
  | e : X → grumble X.
```

Which of the following are well-typed elements of `grumble X` for some type `X`?

- `d (b a 5)`
- `d mumble (b a 5)`

- `d bool (b a 5)`
- `e bool 5`
- `e bool true`
- `e mumble (b c 0)`
- `e bool (b c 0)`
- `c`

If you define these types in Coq, place their definitions inside a new Module, so the type constructors `a`, `b`, `c`, etc. do not interfere with subsequent exercises.

Exercise 2

In the lectures, we have seen how we can define polymorphic lists and functions that operate on polymorphic lists. Here, we will use the datatype defined in Coq and import the following modules:

```
Require Import Coq.Lists.List.
Import ListNotations.
```

2.1. Define the function `tl` that returns the *tail* of a list. For example:

`tl [1;2;3] = [2;3]`.

2.2. Define the function `removelast` that removes the last element of a list.

2.3. Define the function `firstn` that returns the first `n` elements of a list. For example:

`firstn 3 [1;2;3;4;5] = [1;2;3]`

2.4. Define the function `skipn` that skips the first `n` elements of a list. For example:

`skipn 3 [1;2;3;4;5] = [4;5]`

2.5. Define the function `last` that returns the last element of a list. Use the option type to deal with the case where the list is empty. For example:

`last [] = None`

`last [1;2;3] = Some 3`

2.6. Define the function `seq` that takes two natural numbers, `start` and `len`, and creates a list of size `len` with consecutive numbers starting from `start`. For example:

```
seq 3 4 = [3;4;5;6]
```

2.7. Define the function `split` that takes a list of pairs and returns a pair of lists (sometimes, also called *unzip*). For example:

```
split [(1,true);(2,false);(3,true)] =
([1;2;3],[true;false;true]).
```

2.8. Define the function `append` that takes two lists of the same type and appends the second to the first. For example:

```
append [1;2;3] [4;5;6] = [1;2;3;4;5;6]
```

2.9. Define the function `rev` that reverses a list. For example:

```
rev [1;2;3] = [3;2;1]
```

2.10. Define the function `existsb` that, given a boolean test and a list, checks whether there is any element in the list that satisfies the test. For example:

`existsb (fun e ⇒ e <=? 3) [2;4;5] = true`

To test the function with the notation `<=?`, import the module `PeanoNat`:

```
Require Import Coq.Arith.PeanoNat.
```

2.11. Define the function `forallb` that, given a boolean test and a list, checks whether *all* the elements in the list satisfy the test. For example:

`forallb (fun e ⇒ e <=? 3) [2;4;5] = false`

2.12. Define the function `find` that, given a boolean test and a list, returns the first element that satisfies the test. If there are no elements that satisfy the test, the function should return `None`. For example:

`find (fun e ⇒ e <=? 3) [6;4;1;3;7] = Some 1` and `find (fun e ⇒ e <=? 3) [6;4;4;5;7] = None`

2.13. Define the function `partition` specified as:

partition : forall X : Type, (X -> bool) -> list X -> list X * list X

Given a set `X`, a test function of type `X → bool` and a `list X`, `partition` should return a pair of lists. The first member of the pair is the sublist of the original list containing the elements that satisfy the test, and the second is the sublist containing those that fail the test. The order of elements in the two sublists should be the same as their order in the original list.

2.14. Define the function `list_prod` that takes two lists, `l1` and `l2`, and returns all the possible pairs of elements `(x,y)`, where `x` are elements of `l1` and `y` elements of `l2`. For example:

```
list_prod [1; 2] [true; false] =
[(1,true); (1,false); (2,true);
(2,false)]
```

2.15. So far, when defining polymorphic functions, we have always listed the types involved between curly brackets as in:

Fixpoint repeat {X:Type} (x:X) (count:nat) : list X

If we tell Coq to set implicit arguments, then we can use the commands `Section` and `Variable` to make the types of our functions simpler. For example, we could write:

```
Set Implicit Arguments.
Section ListFunctions.
  Variable X : Type.

  Fixpoint repeat (x:X) (count:nat) :
list X :=
    match count with
    | 0 ⇒ []
    | S n ⇒ x :: repeat x n
    end.

End ListFunctions.
```

In a new file, follow this approach and rewrite all your solutions to the exercises above to make use of this mechanism.