Tutorial 2 – Pattern matching and recursion

Summer semester 2026 Florent Schaffhauser

Due by May 18th at 4 PM.

Exercise 1. (Type-checking) Are the following expressions well-typed? If so, what is their type? And if not, what is the problem?

a. `[1; 2; 3]`

b. `true :: true :: nil`

c. `1 + 0`

d. `fun (x : nat) (y : nat) => [x; y]`

e. `fun (x : nat) (y : bool) => [x; y]`

f. `(fun (n : nat) => 2 * n ) 5`

Hint: If you are not sure, ask the prover!

Exercise 2. (Lists) Recall the definition of the type of lists with entries in a type `A`.

```
Inductive list (A : Type) : Type :=
| nil  : list A
| cons : A → list A → list A.

Arguments nil {A}.
Arguments cons {A} a l.

Infix "::" := cons (at level 60, right associativity) : list_scope.
```

Recall next that the following commands gives you access to the notation `[a; b; c]` for the list `a :: b :: c :: nil`.

```
From Stdlib/Coq Require Import List.
  (* Depending on whether you work with Rocq 9.x or an earlier version *)
Import ListNotations.
```

a. Write a function `lgth : list A → nat` that returns the length of a list (if you call it `length`, it might conflict with Rocq's length function from the core library).

b. Write a function `concat : list A → list A → list A` that concatenates two lists. Introduce the infix notation `+++` for it (with the same precedence level and associativity properties as `cons`), and compute `[true; false] +++ [false]` (you should get `[true; false; false]`).

c. A *suffix* of a list `l :list A` is obtained by removing a certain number of initial elements of the list. For instance, the suffixes of the list `[1; 2; 3]` are `[1; 2; 3]`, `[2; 3]`, `[3]` and `[]`. Write a function `suffix : list A → ?` which, given a list `l : list A` returns the list of its suffixes, in decreasing order of length, and test it on the list `[1; 2; 3]`.

Exercise 3. (Equality between booleans) Define equality between booleans as follows.

```
Inductive EqBool : bool → bool → Set :=
| tt : EqBool true true
| ff : EqBool false false.

Notation "b ≡ b'" := (EqBool b b') (at level 120, no associativity).
```

This means that we have a proof, called `tt`, that `true ≡ true` and a proof, called `ff`, that `false ≡ false` . Now recall the definition of the function `negb` .

```
Definition negb (b : bool) : bool :=
  match b with
  | true  => false
  | false => true
  end.
```

a. Prove the following result:

```
Theorem negb_inv (b : bool) : negb (negb b) ≡ b.
```

b. Propose a different definition of equality between Booleans, this time as a `bool`-valued function

```
eqb : bool → bool → bool
```

(with notation `==`). In this definition, `true == true` should evaluate to `true` and so on for other possible `b == b'` where `b` and `b'` are Booleans. Then show that `negb (negb b) == b` *evaluates* to `true` for every value of `(b : bool)`.

Hint: Follow the same approach as in Lecture 3.