Functional Programming in Lean

Prof. Dr. Peter Thiemann
Marius Weidner

University of Freiburg
Winter 2026

Sheet 1

Due: Monday, 2026-11-02

Important Information:

- Exercises are ungraded and do *not* need to be submitted.
- If you have questions, please post a message in the chat.
- The solutions will be discussed in the tutorial sessions.
- Start from the template `Template01.lean`.

Exercise 1.1 (Warming up)

(a) Write two functions `maxi` and `mini`, which take two `Int`s as arguments and return their maximum and minimum, respectively. Provide type signatures for each function. (Don't use the predefined `min` and `max` functions, obviously!)

(b) Write two functions `max3` and `max3Tupled`, which take three `Int`s as arguments and return their maximum. `max3` should take three arguments (curried function), whereas `max3Tupled` should take a single argument of type `Int × Int × Int` (uncurried function).

(c) Define a function `med`, which takes three `Int`s and returns their median, i.e. the second largest argument.

Exercise 1.2 (List functions)

In this exercise we reimplement some of the list functions from Lean's standard library. This way you practice writing recursive functions with pattern matching, get comfortable with Lean's lists, and explore the standard library at the same time.

We append a single quote to each function name to avoid name clashes, e.g. `null'`. The last item lists the names of the corresponding functions in the standard library.

Write type signatures for the functions and make them reasonably polymorphic. For example, a function which reverses a list works for lists with elements of an arbitrary type α, not only for `List Int`.

1. Write a function `null'`, which takes a list as an argument and returns whether the list is empty.

   Examples:

   ```
   #eval null' ([] : List Nat)  -- true
   #eval null' [1, 2]           -- false
   ```

Lean cannot infer the element type of a bare `[]`, so the first example annotates it.

2. Write a function `sum'`, which takes a list of integers and returns their sum.

   Examples:

   ```
   #eval sum' [1, 2, 3]  -- 6
   ```

3. Write a function `flatten'`, which takes a list of lists and joins them together.

   Examples:

   ```
   #eval flatten' [[1, 2, 3], [4], [5, 6]]  -- [1, 2, 3, 4, 5, 6]
   ```

4. Write a function `elem'`, which takes a value and a list and checks whether the value is in the list.

   Because we haven't covered type classes yet, you can assume the value and list elements are integers.

   Examples:

   ```
   #eval elem' 3 [1, 2, 3, 4]  -- true
   #eval elem' 9 [1, 2, 3, 4]  -- false
   ```

5. Write a function `take'`, which takes a natural number `n` and a list `xs` as arguments and returns a list with the first `n` elements of `xs`.

   Examples:

   ```
   #eval take' 3 [1, 2, 3, 4, 5]  -- [1, 2, 3]
   ```

6. Write a function `drop'`, which takes a natural number `n` and a list `xs` as arguments and returns a list with everything but the first `n` elements of `xs`.

   Examples:

   ```
   #eval drop' 3 [1, 2, 3, 4, 5]  -- [4, 5]
   ```

7. Write a function `zip'`, which takes two lists as arguments and returns a list of pairs of the elements of the argument lists.

   If one list is longer than the other, its extra elements are ignored.

   Examples:

   ```
   #eval zip' [1, 2, 3] [true, false, true]  -- [(1, true), (2,
       false), (3, true)]
   #eval zip' [1, 2, 3, 4] [10, 20]          -- [(1, 10), (2, 20)]
   ```

8. Write a function `reverse'`, which takes a list as an argument and returns the reversed list.

   Examples:

```
#eval reverse' [1, 2, 3]  -- [3, 2, 1]
```

9. Write a function `intersperse'`, which takes a value and a list as arguments and places the value between each two elements of the list.

   This function is particularly useful with strings. A Lean `String` is not a list of characters, so convert it with `String.toList` and back with `String.ofList`.

   Examples:

   ```
   #eval intersperse' 666 [1, 2, 3, 4]  -- [1, 666, 2, 666, 3,
       666, 4]
   #eval String.ofList (intersperse' ',' "abcde".toList)  --
       "a,b,c,d,e"
   ```

10. Check out the documentation of the `List` namespace for many more useful functions. Use `#check` or hover over a name in the editor, or browse the reference manual:
    https://lean-lang.org/doc/reference/latest/

    The functions of this exercise correspond to the following functions of the standard library.

This exercise	Standard library
`null'`	`List.isEmpty`
`sum'`	`List.sum`
`flatten'`	`List.flatten`
`elem'`	`List.elem`
`take'`, `drop'`	`List.take`, `List.drop`
`zip'`	`List.zip`
`reverse'`	`List.reverse`
`intersperse'`	`List.intersperse`

    Some of these functions have a more general type in the standard library. For example, `List.sum` works for every type with addition and zero, and `List.elem` for every type with an equality test. The types don't have to make sense to you now, and we will come back to them once we cover type classes in detail.

Exercise 1.3 (Empty lists)

Some list functions have no sensible result on the empty list. In this exercise you handle the empty list in two ways. The function either returns an `Option`, or it panics.

1. Write a function `initSafe`, which takes a list as an argument and returns the list without its last element. If the list is empty, then `initSafe` should return `none`.

   Hint: You might need to use a `match` expression to pattern match on the result of the recursive call.

Examples:

```
#eval initSafe [1, 2, 3]          -- some [1, 2]
#eval initSafe ([] : List Nat)    -- none
```

2. Write a function `init'`, which takes a list as an argument and returns the list without its last element. If the list is empty, then `init'` should panic.

   Lean has no exceptions in pure code. Instead, `panic! "message"` prints the message and returns a default value of the result type. For lists this default value is `[]`. The program keeps running afterwards.

   Examples:

   ```
   #eval init' [1, 2, 3]             -- [1, 2]
   #eval init' ([] : List Nat)
     -- PANIC at init' ...: init' is undefined on the empty list
     -- []
   ```

3. Write a function `lastSafe`, which takes a list as an argument and returns the last element of the list. If the list is empty, then `lastSafe` should return `none`.

   Examples:

   ```
   #eval lastSafe [1, 2, 3]          -- some 3
   #eval lastSafe ([] : List Nat)    -- none
   ```

The standard library provides `List.getLast?`, which corresponds to `lastSafe`, and `List.dropLast`, which corresponds to `init'`. `List.dropLast` returns `[]` on the empty list instead of panicking.

A panicking version `last' : List α → α` of `lastSafe` is harder to write. On the empty list, `panic!` needs a default value of type α, and an arbitrary type α has none. Lean accepts `last'` only with the extra argument `[Inhabited α]`, which says that α has a default value.

```
def last' [Inhabited α] : List α → α
  | [] => panic! "last' is undefined on the empty list"
  | [x] => x
  | _ :: xs => last' xs
```

We explain this notation when we cover type classes.