Functional Programming in Lean

Prof. Dr. Peter Thiemann
Marius Weidner

University of Freiburg
Winter 2026

Sheet 2

Due: Monday, 2026-11-09

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 `Template02.lean`.

Exercise 2.1 (Map, filter, and fold)

In the lecture you saw three famous higher-order functions, i.e. functions which take functions as arguments or return functions as results.

1. `List.map` applies a function to each element of a list and returns the list of the results.

   ```
   >>> [1, 2, 3, 4].map (fun x => x * 2)
   [2, 4, 6, 8]
   ```

2. `List.filter` applies a predicate to each element of a list and returns the elements for which the predicate returns `true`.

   ```
   >>> [1, 2, 3, 4].filter (fun x => x > 2)
   [3, 4]
   ```

3. `List.foldr` folds a list with a binary function. For example, it sums up a list as follows.

   ```
   >>> [1, 2, 3, 4].foldr (fun x acc => x + acc) 0
   10
   ```

   Intuitively, the fold replaces each `::` by the function and the empty list `[]` by the initial value `0`. The fold of `1 :: (2 :: (3 :: (4 :: [])))` computes `1 + (2 + (3 + (4 + 0)))`.

These functions express many recursive functions on lists in a more concise way. To use them, you have to spot when a recursive definition fits the pattern of one of them. For example, the following recursive function sums up a list of integers.

```
def sumRec : List Int → Int
  | [] => 0
  | x :: xs => x + sumRec xs
```

With a fold, the same function is a one-liner.

```
def sum' (xs : List Int) : Int :=
  foldr' (fun x acc => x + acc) 0 xs
```

First reimplement map, filter, and fold with the following type signatures.

```
def map' (f : α → β) : List α → List β
def filter' (p : α → Bool) : List α → List α
def foldr' (f : α → β → β) (init : β) : List α → β
```

Then implement each of the following functions twice. The version with the suffix `Rec` uses recursion and pattern matching, e.g. `incRec`. The version without the suffix uses `map'`, `filter'`, and `foldr'`, e.g. `inc`.

We append a single quote only to functions that also exist in the standard library, e.g. `and'` for `List.and`. This avoids name clashes and tells you which functions you don't have to write yourself in later exercises.

1. Write a function `inc`, which increments each element in a list of integers.

   Examples:

   ```
   >>> inc [1, 2, 3]
   [2, 3, 4]
   ```

2. Write a function `evenList`, which returns all even integers in a list.

   Examples:

   ```
   >>> evenList [1, 2, 3, 4]
   [2, 4]
   ```

3. Write a function `shortStrs`, which takes a list of integers and returns the string representations of those integers whose string representation has at most two characters. The function `toString` converts an integer into a string.

   Examples:

   ```
   >>> shortStrs [1, 10, 100, 1000, -1, -10]
   ["1", "10", "-1"]
   ```

4. Write two functions `and'` and `or'`, which take a list of booleans. `and'` returns whether all booleans are `true`, whereas `or'` returns whether at least one boolean is `true`.

   Examples:

   ```
   >>> and' [true, true, true]
   true
   >>> and' [true, false, true]
   false
   ```

```
>>> or' [false, false, false]
false
>>> or' [false, true, false]
true
```

5. Write two functions `all'` and `any'`, which take a list and a predicate on its elements. `all'` returns whether all elements satisfy the predicate, whereas `any'` returns whether at least one element does.

   Examples:

   ```
   >>> all' [1, 20, 3] (fun x => x < 10)
   false
   >>> any' [1, 20, 3] (fun x => x < 10)
   true
   ```

6. Write a function `length'`, which takes a list and returns its length.

   Examples:

   ```
   >>> length' [10, 20, 30]
   3
   ```

7. Write three functions `idMap`, `idFilter`, and `idFold`, which take a list and return it unchanged. Each function applies `map'`, `filter'`, or `foldr'` to the list, with arguments chosen such that the result is the input list. Write only this one version of each function.

   Examples:

   ```
   >>> idMap [1, 2, 3]
   [1, 2, 3]
   ```

8. Write two functions `mapFold` and `filterFold`, which behave like `map'` and `filter'` but use `foldr'` instead of recursion.

Exercise 2.2 (Three-dimensional vectors)

Higher-order functions are useful for all kinds of data, not only for lists. The following structure represents vectors in three-dimensional space.

```
structure V3 (α : Type) where
  x : α
  y : α
  z : α
  deriving Repr, BEq

def exampleV3 : V3 Int := ⟨5, 3, 8⟩
```

`deriving Repr, BEq` lets Lean print vectors with `#eval` and compare them with `==`. `Repr` and `BEq` are type classes, which we have not covered yet.

Vectors support the usual arithmetic operations, which apply the operation on numbers to each coordinate. For example, negation, addition, and multiplication can be defined as follows.

```
def negV3 (v : V3 Int) : V3 Int :=
  ⟨-v.x, -v.y, -v.z⟩

def addV3 (v w : V3 Int) : V3 Int :=
  ⟨v.x + w.x, v.y + w.y, v.z + w.z⟩

def mulV3 (v w : V3 Int) : V3 Int :=
  ⟨v.x * w.x, v.y * w.y, v.z * w.z⟩
```

```
>>> negV3 ⟨2, -4, 8⟩
{ x := -2, y := 4, z := -8 }
>>> addV3 ⟨1, 2, 3⟩ ⟨10, 20, 30⟩
{ x := 11, y := 22, z := 33 }
```

This code repeats itself. `addV3` and `mulV3` both lift a binary operator on numbers pointwise to vectors, and `negV3` does the same for a unary operator.

Write two functions `mapV3` and `liftV3`, which lift a unary and a binary operator to vectors. Use them to implement negation, addition, subtraction, multiplication, and division for `V3 Int`, named `negV3`, `addV3`, `subV3`, `mulV3`, and `divV3`.

```
def mapV3 (f : α → β) (v : V3 α) : V3 β
def liftV3 (f : α → β → γ) (v : V3 α) (w : V3 β) : V3 γ
```

Exercise 2.3 (Tic-tac-toe)

In this exercise we implement the classic game tic-tac-toe. We represent the game state as follows.

```
inductive Token where
  | X | O | E
  deriving Repr, BEq

open Token

abbrev Board := List (List Token)
```

A `Board` is a matrix of tokens. A token is either `X` or `O` for the two players, or `E` for an empty cell. `open Token` lets us write `X` instead of `Token.X`. The following board shows a game that player `X` has won.

```
def exampleBoard : Board :=
  [ [X, O, E],
    [X, O, E],
    [X, E, E] ]
```

Lean prints tokens with their full names, such as `Sheet02.Token.X`. For this reason the examples below compare results with `==`.

The end of the template contains a graphical interface, which runs in the infoview. Once your functions work, put the cursor on the `#widget` line at the end of the template. The infoview then shows the board, and a click on a cell places the token of the current player. You do not need to understand the code of the interface.

Complete the game by implementing the following functions.

1. Write a function `emptyBoard : Nat → Board`, which takes a number `n` and returns an empty $n \times n$ board.

   Examples:

   ```
   >>> emptyBoard 2 == [[E, E], [E, E]]
   true
   ```

   Hint: `List.replicate` creates a list by repeating an element.

   ```
   >>> List.replicate 3 'a'
   ['a', 'a', 'a']
   ```

2. Write a function `isFull : Board → Bool`, which returns whether the board contains no empty token `E`. The interface uses it to detect a draw.

3. Write a function `showBoard : Board → String`, which converts a board into a string for display. It shows an empty token as an underscore.

   Examples:

   ```
   >>> showBoard exampleBoard
   "X O _\nX O _\nX _ _"
   >>> IO.println (showBoard exampleBoard)
   X O _
   X O _
   X _ _
   ```

   Hint: `String.intercalate` joins a list of strings with a separator.

   ```
   >>> ", ".intercalate ["a", "b", "c"]
   "a, b, c"
   ```

4. Write a function `setToken : Nat × Nat → Token → Board → Option Board`, which places a token at a position `(x, y)`, i.e. in column `x` of row `y`. In a pure language, updating the board means returning a new board, which is similar to the old one.

   A player may only place a token on an empty cell. For this reason `setToken` returns an `Option Board`. It returns `some` updated board, or `none` if the cell is not empty or the position lies outside the board.

   Examples:

```
>>> setToken (1, 0) X (emptyBoard 3)
    == some [[E, X, E], [E, E, E], [E, E, E]]
true
>>> setToken (1, 0) O [[E, X, E], [E, E, E], [E, E, E]]
    == none
true
>>> setToken (55, 32) X (emptyBoard 3) == none
true
```

Hint: `setToken` is much easier to write once you can update a single element of a list. Consider the following function `mapAt`, which applies a function only to the element at the given index.

```
def mapAt : Nat → (α → α) → List α → List α
```

The standard library calls this function `List.modify`.

```
>>> [1, 2, 3, 4].map (fun x => x * 100)
[100, 200, 300, 400]
>>> [1, 2, 3, 4].modify 1 (fun x => x * 100)
[1, 200, 3, 4]
```

`setToken` needs a slightly more general function, because the update may fail. Write a function `mapAtM : Nat → (α → Option α) → List α → Option (List α)`. In `mapAtM i f xs`, the function `f` returns either `some` new element or `none`. If `f` returns `none`, or if the index `i` lies outside the list, then `mapAtM i f xs` returns `none` as well.

Examples:

```
def f (x : Nat) : Option Nat :=
  if x > 1 then some (x * 100) else none
>>> mapAtM 1 f [1, 2, 3, 4]
some [1, 200, 3, 4]
>>> mapAtM 0 f [1, 2, 3, 4]
none
```

With two calls of `mapAtM`, `setToken` becomes a one-liner.

Hint: `Option.map` applies a function to the value inside an `Option`. Think of an `Option` as a list with at most one element.

```
>>> Option.map (fun x => x + 1) (some 2)
some 3
```

5. Write a function `winner : Board → Token`, which returns `X` or `O` if that player has a full row, column, or diagonal, and `E` otherwise. You don't need to detect a draw, because `isFull` covers that case.

   Examples:

```
>>> winner (emptyBoard 3) == E
true
>>> winner [[X, X, X], [E, E, E], [O, O, E]] == X
true
>>> winner [[O, E, E], [E, O, E], [X, X, O]] == O
true
```

Hint: First write a function that returns the winner of a single row. Lean's core library has no function to transpose a matrix, so build the columns and diagonals yourself. `List.range n` returns `[0, 1, ..., n - 1]`, and `xs.getD i d` returns the element at index `i`, or `d` if the index lies outside the list.

```
>>> List.range 3
[0, 1, 2]
>>> [X, O].getD 5 E == E
true
```

The function `all'` from Exercise 2.1 helps as well.