Development 8 - Exercises
Unit 4

Exercise 1:

Implement a function

```
let filter = <a>(predicate:  (x:  a) => boolean) => (l:  List<a>):  List<a>
```

that inserts in the output list only the elements for which `predicate` returns true

Exercise 2:

Implement a function

```
let map = <a, b>(f:  (x:  a) => b) => (l:  List<a>):  List<b>
```

that applies the function `f` to all the elements of `l` and returns a list containing the results.

Exercise 3:

Implement a function

```
let fold = <s, a>(f:  (state:  s) => (x:  a) => s) => (init:  s) => (l:  List<a>):
s
```

that applies a function `f` to elements in the same position from `l`, threading an accumulator argument of type `s` through the computation.

Exercise 4:

Implement a function

```
let apply = <a, b>(f:  (x:  a) => b) => (x:  a):  b
```

that applies function ‘f‘ to element ‘x‘.

Exercise 5:

Implement a function

```
let curry = <a, b, c>(f: Tuple<a, b> => c) => (x: a) => (y: b): c
```

that applies function ‘f‘ using as input elements ‘x‘ and ‘y‘ stored as a tuple.

Exercise 6:

Implement a function

```
let mapFold = <a, b>(f: (x: a) => b) => (l: List<a>): List<b>
```

that implements `map` only using `fold`

Exercise 7:

Implement a function

```
let filterFold = <a>(predicate: (x: a) => boolean) => (l: List<a>): List<a>
```

that implements `filter` only using `fold`

Exercise 8:

Implement a function

```
let flatten = <a>(l: List<List<a>>): List<a>
```

that takes a list of lists and places all their elements it into a single one. Use `fold` to implement this function.

Exercise 9:

Implement a function

```
let map2 = <a, b, c>(f: (x: a) => (y: b) => c) => (l1: List<a>) => (l2: List<b>):
List<c>
```

that applies the function `f` to the elements in the same position of two lists of equal length `l1` and `l2`.

Exercise 10:

Implement a function

```
let fold2 = <s, a, b>(f:  (state:  s) => (x:  a) => (y:  b) => s) => (init:  s) =>
(l1:  List<a>) => (l2:  List<b>):  s
```

that applies a function `f` to elements in the same position from `l1` and `l2`, threading an accumulator argument of type `s` through the computation.

Exercise 11:

Implement a function

```
let zip = <a, b>(l1:  List<a>) => (l2:  List<b>):  List<Tuple<a, b>>
```

that take two lists with the same length and creates a list of pairs containing the elements that are in the same position from both lists. Implement this function by using normal recursion and then by using `fold2`

Exercise 12:

Implement a function

```
let map2Safe = <a, b, c>(f:  (x:  a) => (y:  b) => c) =>(l1:  List<a>) =>
(l2:  List<b>):  List<Option<c>>
```

that applies the function `f` to the elements in the same position of two lists`l1` and `l2`, possibly with different length. If an element of one list does not have a correspondent element in the second list, then the function returns `None`.

Example: Summing the elements of `[1, 2, 3, 4]` and `[4, 5]` with `map2Safe` returns `[Some(5), Some(7), None, None]`