High-order

1 Toolbox

Reminders and usefull functions

Fonctions you can use are indicated in the examples below:

```
♯ (int_of_char 'A', int_of_char 'Z') ;;
 - : int * int = (65, 90)

♯ (char_of_int 97, char_of_int 122);;
 - : char * char = ('a', 'z')

♯ Char.escaped 'A' ;;
 string = "A"

♯  'F' > 'A' && 'F' < 'Z';;
 - : bool = true
```

```
♯ let s = "a string" ;;
 val s : string = "a string"

♯ String.length s ;;
 - : int = 8

♯ (s.[0], s.[7]) ;;
 - : char * char = ('a', 'g')
```

1.1 char and string

char

1. Write the function `char_type c` that returns a string according to the type of character `c`: `"lower"` or `"upper"` if `c` is a lowercase or uppercase letter, `"other"` otherwise.

```
 val char_type : char -> string = <fun>
♯ (char_type 'a', char_type 'G', char_type ' ') ;;
 - : string * string * string = ("lower", "upper", "other")
```

2. Write the function `uppercase` that transforms a lowercase letter in uppercase and returns other characters unchanged.

```
 val uppercase : char -> char = <fun>
♯ (uppercase 'a', uppercase 'G', uppercase ' ') ;;
 - : char * char * char = ('A', 'G', ' ')
```

3. Write the function `lower` that transforms an uppercase letter in lowercase and returns other characters unchanged.

```
 val lowercase : char -> char = <fun>
♯ (lowercase 'A', lowercase 'h', lowercase ',') ;;
 - : char * char * char = ('a', 'h', ',')
```

4. Write the function `swap_alpha c` that returns the symmetrical letter of `c` in the alphabet if `c` is a letter.

```
 val swap_alpha : char -> char = <fun>
♯ (swap_alpha 'F', swap_alpha 'c', swap_alpha ' ') ;;
 - : char * char * char = ('U', 'x', ' ')
```

5. Write the function `char_rotn n c` that applies a right shift of value `n` (to the right if `n` positive) to the character `c` in case `c` is a letter.

```
 val rotn: int -> char -> char = <fun>
♯ (rotn 3 's', rotn 13 'A', rotn (-5) 'z', rotn 5 'u', rotn 5 ' ');;
 - : char * char * char * char * char = ('v', 'N', 'u', 'z', ' ')
```

Reminder: partial application

```
♯ let rot13 = rotn 13 ;;
 val rot13 : char -> char = <fun>

♯ rot13 'A' ;;
 - : char = 'N'
```

`string` ↔ `list`

1. Write the function `string_of_list` that transforms a character list into a string.

```
val string_of_list : char list -> string = <fun>

♯ string_of_list ['I'; 'm'; 'p'; 'l'; 'o'; 'd'; 'e'] ;;
- : string = "Implode"
```

2. Write the function `list_of_string` that transforms a string into a character list.

```
val list_of_string : string -> char list = <fun>

♯ list_of_string "Explode" ;;
- : char list = ['E'; 'x'; 'p'; 'l'; 'o'; 'd'; 'e']
```

1.2 High-order

1. Write the function `map f [`a_1`; ...; `a_n`]` that applies function `f` to a_1`; ...; `a_n and builds the list `[f `a_1`; f `a_2`;...; f `a_n`]` with the results returned by `f`.

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

Use `map` to write the function `uppercase_list` that converts each lowercase letter in a list in uppercase.
Write as well the function `lowercase_list`.

```
val uppercase_list : char list -> char list = <fun>
val lowercase_list : char list -> char list = <fun>

♯ uppercase_list ['a'; 'B'; 'c'; 'd'; 'e'; ' '; '0'] ;;
- : char list = ['A'; 'B'; 'C'; 'D'; 'E'; ' '; '0']
♯ lowercase_list ['A'; 'b'; 'C'; 'D'; 'E'; ' '; ','] ;;
- : char list = ['a'; 'b'; 'c'; 'd'; 'e'; ' '; ',']
```

2. Write the function `iter f [`a_1`; ...; `a_n`]` that applies function `f` in turn to a_1`;...; `a_n.
It is equivalent to `begin f `a_1`; f `a_2`;...; f `a_n`; () end`.

```
val iter : ('a -> unit) -> 'a list -> unit
```

3. Write the function `map2 f` $[a_1; a_2; \cdots ; a_n]$ $[b_1; b_2; \cdots ; b_n]$ that builds the list: $[f\ a_1\ b_1\ ; f\ a_2\ b_2\ ;\ \cdots\ ;\ f\ a_n\ b_n]$.
It raises an exception if the two lists have different lengths.

2 A little High-order: Ciphers

2.1 Caesar cipher

In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code or Caesar shift, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a right shift of 3, `A` would be replaced by `D`, `B` would become `E`, `X` would become `A` and so on. The method is named after Julius Caesar, who used it in his private correspondence.
wikipedia

1. Using the previous functions, write the function `caesar_encode n s` that applies the Caesar cipher with a shift value of `n` to the string `s`.

```
val caesar_encode : int -> string -> string = <fun>

♯ caesar_encode 3 "Krisboul" ;;
- : string = "Nulverxo"

♯ caesar_encode 13 "Chiffrement" ;;
- : string = "Puvsserzrag"
```

2. Using the previous functions, write the function `caesar_decode n s` that unode the string `s` encoded with a shift value of `n` to the string `s`.

```
val caesar_decode : int -> string -> string = <fun>

♯ caesar_decode 3 "Nulverxo";;
 - : string = "Krisboul"
♯ caesar_decode 5 "Ymjxj antqjsy ijfiqnsjx mfaj antqjsy jsix." ;;
 - : string = ...
```

2.2 Vigenère cipher

The Vigenère cipher is a method of encrypting alphabetic text by using a series of different Caesar ciphers based on the letters of a keyword. It is a simple form of polyalphabetic substitution. This encryption introduced the notion of a key. A key is usually in the form of a word or phrase. To cipher our text, for each character we will use a letter from the key to do the substitution. Obviously, the longer and varied the key is the better the text will be encrypted.

wikipedia[1]

An example with "Vigenere cipher" the string to encode and "abc" the key:

```
string to encode :  Vigenere Cipher
key, repeated :     abcabcab cabcab
                    ---------------
result :            Vjieogrf Eiqjes
```

1. Write the functions `char_encode_vigen c ckey` and `char_decode_vigen c ckey` that encodes (respectively decodes) a character from the message with a character `c` from the key `ckey`.

```
 val char_encode_vigen : char -> char -> char = <fun>
 val char_decode_vigene : char -> char -> char = <fun>
♯ char_encode_vigen 'T' 'E' ;;
 - : char = 'X'
♯ char_encode_vigen 'T' 'M' ;;
 - : char = 'F'
# char_decode_vigen 'V' 'L' ;;
 - : char = 'K'
```

2. Write the function `gen_key_list list key` that builds the list with characters of `key` repeated and aligned to the letters in `list`.

```
 val gen_key_list : char list -> char list -> char list = <fun>

♯ gen_key_list
['V'; 'i'; 'g'; 'e'; 'n'; 'e'; 'r'; 'e'; ' '; 'C'; 'i'; 'p'; 'h'; 'e'; 'r'] ['K'; 'e'; 'y'];;
- : char list =
['K'; 'e'; 'y'; 'K'; 'e'; 'y'; 'K'; 'e'; ' '; 'y'; 'K'; 'e'; 'y'; 'K'; 'e']
```

3. Write the `vigenere` function that takes a function (encoding or decoding), the key and the message we need to cipher or deciphers.

```
 val vigenere : (char -> char -> char) -> string -> string -> string = <fun>

♯ vigenere char_encode_vigen "abc" "Vigenere Cipher" ;;
 - : string = "Vjieogrf Eiqjes"

♯ vigenere char_decode_vigen "Caml" "Vo Dpeuddg ie okvuyg" ;;
 - : string = "..."
♯ vigenere char_decode_vigen "ACDC" "Wjb wgnb fecokngv?" ;;
 - : string = "..."
```

[1] https://en.wikipedia.org/wiki/Tabula_recta

3 Build House: High Order Work

3.1 Assembly line work

The following lists represent the raw materials needed to build our house.

```
let sand = ['B' ; 'b' ; 'L' ; 'r' ; 'B'; 'B' ; '\n'] ;;
let water = ['y' ; 'o' ; 'y' ; 'y' ; 'i'; 'y'; '\n'] ;;
let brick = ['y' ; 's' ; 's' ; 's' ; 's'; 'e'; '\n'] ;;
let wood = ['h' ; 'r' ; 'w' ; 'w' ; 'r'; 'h'; '\n'] ;;
let coca = ['s' ; 'l' ; 'r' ; 'x' ; 'l'; 's'; '\n'] ;;
```

Our house is built from materials tacked together, here is a list:

```
♯ let house = [sand ; water ; brick ; wood ; coca];;
 val house : char list list = ...
```

Now, we have to indicate to the workers in each order they have to perform tasks to build a house.

```
♯ let workers = [lowercase ; swap_alpha ; rot13 ; rotn 5 ; rotn 20];;
 val workers : (char -> char) list = [<fun>; <fun>; <fun>; <fun>; <fun>]
```

3.1.1 Let's get to work!

Now that our workers know in what order they have to work, all you have to do is tell them how to work. You have to write a function `chain` that take a function list (like `workers`) and a list of character lists (like `house`) as parameters. It applies each function to each character in the corresponding list and returns the character list that contains the result as in the following example:

```
chain [f1, f2] [['a' ; 'c'; 'd' ; 'c'] ; ['s'; 'u'; 'p']]
-> [[f1 'a'; f1 'c'; f1 'd'; f1 'c'] ; [f2 's'; f2 'u'; f2 'p']]
```

```
val chain : ('a -> 'b) list -> 'a list list -> 'b list list = <fun>
```

As you can well imagine, this function must be applied to `workers` and `house`, in order to retrieve the new list of lists that will represent the foundations of our house.

```
♯ let fundations = chain workers house;;
 val fundations : char list list = ...
```

3.1.2 Finalisation

As foundations are completed, now you just have to build (display) your house. For instance, here are the characters to display according to the materials:

```
'm' (wall)       ->  '|'
'f' (floor)  ->  '_'
'l' (left roof) ->  '/'
'r' (right roof) ->  '\'
'w' (window) ->  '+'
'b' (blanck) ->  ' '
'\n' (end of ground) ->  '\n'
```

Write the function `print_house f house` with `f` the function that displays materials.

```
val val print_house : ('a -> 'b) -> 'a list list -> unit = <fun>
```

Keep going with high-order: iter is your friend!