# Put CAML in high order

***"Functionals" or "higher-order functions"***

## 1 Higher-order?

**Exercise 1.1 (Try those examples)**

```
# let succ = function x -> x + 1 ;;
# succ 5 ;;
# (function x -> x + 1) 5 ;;

# let times = function x -> function y -> x * y ;;
# let double = times 2 ;;
# double 15 ;;

# let double_fun f x = double (f x) ;;
# double_fun succ 2 ;;
# double_fun (function x -> x + 1) ;;
# let double_succ = double_fun succ ;;
# double_succ 2 ;;
```


**Exercise 1.2 (Sigma)**

1. Write a function that computes the following sum (without multiplication...):
$$\sum_{i=0}^{n} i$$
2. Write a function that computes the following sum of the squares:
$$\sum_{i=0}^{n} i^2$$
3. Higher order functions allow us to generalize the sum to any function. Write a function that, given any function $f$, computes:
$$\sum_{i=0}^{n} f(i)$$

**Exercise 1.3 (Loop - Bonus)**

1. Write the function `loop` that, when applied to a predicate $p$, a function $f$ and an integer $x$, returns the value $f^n(x)$ with $n$ the lowest integer that satisfies $p(f^n(x)) = true$.
2. Use `loop` to define the function `find_power` that, given two integers $x$ and $n$, returns the first power of $x$ greater than $n$.

# 2 The one where functions apply to lists' elements

Give, for each function, an example of application.

## Iterators

**Exercise 2.1** (`map`)

Write the function `map` that applies a given function $f$ to all elements of a list and returns the list of results.
`map f [`$a_1$`; ...; `$a_n$`]` returns `[f `$a_1$`; f `$a_2$`;...; f `$a_n$`]`

**Exercise 2.2** (`iter`)

Write the function `iter` that applies a given function $f$ in turn to all elements of a list.
`iter f [`$a_1$`; ⋯ ; `$a_n$`]` ≡ `f `$a_1$`; f `$a_2$`;...; f `$a_n$` ; ()`

## List scanning

*predicate* = boolean function that represents a property

**Exercise 2.3** (`for_all`)

Write the function `for_all` that checks if all elements of a list satisfy the given predicate `p`.

**Exercise 2.4** (`exists`)

Write the function `exists` that checks if at least one element of a list $l$ satisfies the given predicate $p$.

## List searching

**Exercise 2.5** (`find`)

Modify the function `exists` in order to return the first element of the list that satisfies the predicate.

**Exercise 2.6** (`filter`)

Write a function that returns the list of all the elements of a list that satisfy the given predicate `p`.

**Exercise 2.7** (`partition`)

Write a function that takes a predicate $p$ and a list $l$ as parameters. It returns a pair of lists $(l_1, l_2)$, where $l_1$ is the list of all the elements of $l$ that satisfy the predicate `p`, and $l_2$ is the list of those that do not satisfy `p`.

## On two lists

### Exercise 2.8 (map2)

1. Write the CAML function `map2` with the following specifications:
   - It takes a a two-argument function, $f$, and two lists, $[a_1; a_2; \cdots; a_n]$ and $[b_1; b_2; \cdots; b_n]$, as parameters.
   - It returns 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. Use the function `map2` to define the function `add_list` that "adds" two integer lists.

   *Example of result:*

```
# add_list [1;2;3;4] [2;3;4;5] ;;
- : int list = [3; 5; 7; 9]

# add_list [1;2;3;4] [1;2;3] ;;
Exception: Invalid_argument "Different lengths".
```

3. Use the function `map2` to define the function `combine` that transforms two lists into a list of pairs: `combine` $[a_1; \cdots; a_n]$ $[b_1; b_2; \cdots; b_n]$ is $[(a_1, b_1); ...; (a_n, b_n)]$.

   *Example of result:*

```
#  combine [1;2;3;4] ['a';'b';'c';'d'];;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c'); (4, 'd')]
```

### Exercise 2.9 (find2 – *C1♯ 05/2021*)

1. Write the CAML function `find2 p l1 l2` with the following specifications:
   - It takes a two-argument boolean function, `p`, and two lists, $[a_1; a_2; \cdots; a_n]$ and $[b_1; b_2; \cdots; b_n]$, as parameters.
   - It returs the first pair of elements that verifies the predicate $p$: the first pair of elements $(a_i, b_i)$ such that $p\ a_i\ b_i$ is true.
   - It raises an exception if no pair $(a_i, b_i)$ such that $p\ a_i\ b_i$ is true has been found or if the two lists have different lengths.
2. Use the function `find2` to write a function `first_shared l1 l2` that returns the first element present in both lists at the same position.

# 3 Bonus

**Exercise 3.1 (Recursive process)**

Many recursive patterns (not tail-recursive) can be generalized by the following function:

```
# let rec process_list f init =
    function
        [] -> init
      | e :: l -> f e (process_list f init l);;
```

- In the case of an empty list, the function returns a base value: `init`
- In the case of a non empty list `e::l`, the result depends on `e` and on the process of the list `l`.

The function `process_list` can be used to calculate new functions, without recursivity. For instance, the function that returns the length of a list is defined by:

```
# let length l = process_list (function e -> function call -> 1 + call) 0 l ;;

val length : 'a list -> int = <fun>
```

1. (a) What is the type of `process_list`?

   (b) What does `process_list` $f$ *init* `[`$e_1;...;e_n$`]` calculate?

2. Use the function `process_list` to define, without recursivity, the following functions:

   (a) `sum` calculates the sum of all elements of an integer list.

   (b) `concatenate` takes a string list as parameter and returns a string, result of the concatenation of the list strings. Give some application examples of `concatenate`.

   (c) `append` concatenates two lists.

   (d) `map` calculates $[f(e_1);...;f(e_n)]$ when applied to $f$ and $[e_1;...;e_n]$.

   (e) `for_all` takes a boolean function $p$ and a list as parameters. It checks whether all elements of the list satisfy $p$.

   (f) **Bonus:** `sum_list` takes a list containing real functions $[f_1;...;f_n]$ and returns the function $f_1 + \cdots + f_n$. Give examples of `sum_list` applications. Give examples of the application of those results.

**Exercise 3.2 (`fold_left`)**

Write the function `fold_left` defined by:

```
# val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

fold_left f a [b1;...; bn] (* is *) (f a b1) b2)...) bn
```