Structure de code L'implémentation est écrite dans un fichier `.ml`, la signature dans un fichier `.mli`. Les commentaires sont écrits dans des blocs `(* ... *)`, les commentaires de documentation dans des blocs `(** ... *)`. Les fichiers `.ml` sont des suites de définitions de la forme `let` ou `type` ou `exception`.

Types de données de base

Type	Valeurs possible	Commentaire
`unit`	`()`	vide
`int`	`2 0x452 -12`	entiers relatifs
`float`	`0.2 -1.2e25`	flottants
`bool`	`true false`	booléens
`char`	`'x' '2' 'à'`	caractères
`string`	`"AZaz09="`	chaînes de caractères
`'a list`	`[1;2;3] 2 :: []`	listes
`'a array`	`[|-0.2;3.14|]`	tableau
`t1*...*tn`	`(25.1, 0, [])`	n-uplets
`'a option`	`None (Some 2)`	option

Les opérations arithmétiques sur les entiers sont `+ - * / mod abs`, sur les flottants on a `+. -. *. /. ** abs_float`.

Manipulation de chaînes de caractères On accède à un élément d'une chaîne de caractère par `s.[n]`. Les chaînes de caractères sont immuables. On concatène deux chaînes de caractères par l'opérateur `^`. On dispose de la fonction `String.length` de complexité constante.

Types enregistrements

```
type t1 = { champ : int; mutable champ2 : float }
let (x : t1) = {champ = 2; champ2 = 2.5}
let _ = x.champ2 <- 3.14
```

Types somme :

```
type t2 = Cons1 | Cons2 of int * t2
let (x : t2) = Cons2(2, Cons1)
```

Définition de fonctions

```
let f arg1 arg2 arg3 = <expn>
let g = fun x y -> x + 2 * y
let h = function
  | Cons1       -> -1
  | Cons2 (n,x) -> n + h x
  | _           -> failwith "Message d'erreur"
```

Manipulation de listes La syntaxe des listes est : `[]` la liste vide, `tete :: queue` ou `[x0;x1;...]` sinon. On évalue les fonctions suivantes avec la syntaxe `List.nom_fonction` :

Fonction	Spécification
`length : 'a list -> int`	Renvoie la longueur en $\mathcal{O}(\|l\|)$
`rev : 'a list -> 'a list`	Renvoie la liste retournée

On ajoute les fonctions `mem, exists, for_all, filter, map, iter` du module `List`, et l'opérateur de concaténation `@`. Les complexités des fonctions sont à connaître.

Manipulation de tableaux La syntaxe des tableaux est `[|x0;x1;...|]`. On évalue les fonctions suivantes avec la syntaxe `Array.nom_fonction` :

Fonction	Spécification
`length : 'a array -> int`	Longueur (en $\mathcal{O}(1)$)
`make : int -> 'a -> 'a array`	Crée un tableau de taille n initialisé à x
`copy : 'a array -> 'a array`	Renvoie une copie superficielle

On ajoute les fonctions `make_matrix, init, mem, exists, for_all, map, iter` du module `Array`. On accède à une case de tableau par `t.(n)`, on la modifie par `t.(n) <- 25`.

Structures de contrôle Les `if` peuvent se succéder :

```
if n mod 2 = 0 then n / 2 else 3 * n + 1;;
if cond1 then quelque_chose_1
else if cond2 then quelque_chose_2
else if cond3 then quelque_chose_3;;
(* sans else, on renvoie () *)

try code_avec_exceptions
with Exception1 -> ...
   | Exception2 -> ...
```

Les motifs sont testés dans l'ordre et peuvent être de la forme :

```
match exp with
| constante                -> ...
| Constructeur (arg1, ...) -> ...
| [1;x] -> ... (* liste de taille 2 de tête 1 *)
| { champ = 1; ... }       -> ...
  (* pas obligé de préciser tous les champs *)
| motif when condition     -> ...
  (* quand le motif et la condition sont vérifiés *)
| _ -> ... (* motif universel *)
```

OCaml impératif On représente les instructions par des expressions de type `unit`. Par exemple, on dispose des fonctions `print_<type>` pour les types `int, float, string`. Quand on évalue l'expression `e1 ; e2`, on évalue d'abord `e1` de type `unit`, puis on évalue et on renvoie `e2`.

Références On définit une référence par la fonction `ref : 'a -> 'a ref`. On accède à la valeur de `x` par `!x`. On modifie une référence `x : 'a ref` par `x := 1`.

Ensembles et dictionnaires On peut utiliser les ensembles fonctionnels par le module `Set` avec le foncteur `Set.Make(...)`, les dictionnaires fonctionnels par le module `Map` avec le foncteur `Map.Make(...)`, les tables de hachage par le module `Hashtbl` avec le foncteur `Hashtbl.Make(...)`.

Boucles

```
for k = 0 to 25 do (* 25 compris *)
  faire_quelque_chose k (* corps de type [unit] *)
done (* l'expression entière est de type [unit] *)
(* [for k = 25 downto 0 do] est aussi possible *)
while condition do
  ... (* corps de type [unit] *)
done (* l'expression entière est de type [unit] *)
```

Divers `begin ... end` est sémantiquement identique à `( ... )`, utilisé surtout dans les `if then else`. `Sys.argv` est un `string array` des arguments de l'exécutable compilé. On lit un fichier avec `open_in : string -> in_channel`, on le ferme avec `close_in : in_channel -> unit`. On écrit un fichier avec `open_out` et on le ferme avec `close_out`.

Les sources à connaître pour aller plus loin :

- le programme de l'option MPSI / MP : il comprend une annexe détaillant les connaissances exigibles avant et après rappel
- `cs3110.github.io/textbook/`, un cours d'introduction à la plupart des concepts importants en OCaml
- `v2.ocaml.org/releases/4.14/htmlman/index.html`, la documentation officielle du langage