Chapitre 1 : oCaml

Dispose d'un interpréteur (ocaml), d'un compilateur bytecode (ocamlc), et d'un compilateur natif (ocamlopt).

- **opam** is a source-based package manager for OCaml.
- utiliser opam pour installer ocaml et utop (Universal Toplevel)

démos

• valeurs et types de bases ;

```
2 + 2;;
2,3 + 3,5;; // 2.3 +. 3.5;;
true && true ;; // pas de majuscule
'a' // "a" = character // string
"bonjour" ^ " tout le monde" ;;
"bonjour" ^ " tout le monde" ^ '!';;
```

• expressions, variables, et inférence de type ;

```
let pi = 3.1415
```

• fonctions et curryfication ;

```
(^);;
let f x y = x + y ;;
let f (x : int) (y : int) : int = x + y ;;
```

en python :

```
def f(x, y): // en ocaml, pas de parenthèses, pas de virgules
    return x + y // suite d'instructions remplacée par une expression unique
```

curryfication
"Généralement en programmation fonctionnelle, opération qui transforme une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction prenant le reste des arguments."

```
let addition x y = x + y ;;
```

Fonction qui prend deux arguments `int`

```
addition 2 ;;
```

Fonction qui prend **un** argument `int`

```
let ajoute_deux  = addition 2 ;;
ajoute_deux 3 ;;
```

Autre exemple (fonction qui renvoie une fonction) :

```
let bonjour (nom : string) : string = "Bonjour " ^ nom ;;
let hello (nom : string) : string = "Hello " ^ nom ;;
(* bonjour et hello sont de types string -> string *)

(* Saluer renvoie une fonction de type string -> string *)
let saluer (langue : string) : (string -> string) =
  match langue with
  | "francais" -> bonjour (* On renvoie juste la fonction, on ne l'appelle pa
  | _ -> hello ;;

let saluer_en_anglais = saluer "anglais" ;; (* La fonction hello sera renvoyé
saluer_en_anglais "Jeanne" ;;(* donne "Hello Jeanne" *)
```

fonction locale (modifier la portée de la variable):

```
let cube x = x * x * x in (cube 2) + (cube 3);;
cube 3;;
```

• listes et fonctions récursives ;

```
'a list. :
```

Fonctions récursives : pas de boucle for ou while en ocaml, c'est la récursivité qui est utilisée pour répéter des opérations mot clé : rec.

- un cas de base
- un cas récursif

```
let rec somme (n : int) : int =
  if n = 0 then
    0
  else
    n + (somme (n - 1))
```

Listes : ne contiennent des éléments que d'un seul type

```
let liste_vide : float list = []
let petite_liste : float list = 12.5 :: []
let moyenne_liste : float list = 1.7 :: 8.6 :: 88.56 :: []
let moyenne_liste : float list = [ 1.7 ; 8.6 ; 88.56 ]
let grosse_liste = moyenne_liste @ petite_liste
```

• filtrage (pattern matching) ;

pattern matching + récursivité

```
let rec fois_deux (l : int list) : int list =
  match l with
  | [] -> []
  | element :: suite -> (element * 2) :: (fois_deux suite) ,,
```

déconstruction :

```
match ma_liste with
| avant @ [ 1 ; 2 ] @ apres -> "La liste contient 1 et 2"
| _ -> "La liste ne contient pas 1 et 2"
```

Module List (plein de fonctions utiles genre head, sort, etc.)

```
List.map : ('a -> 'b) -> 'a list -> 'b list
associe à tout élément d'une liste de type a un autre élément de type b
List.map int_of_char [ 'a' ; 'b' ; 'Z' ];; (*code ascii*)
```

• types algébriques (ADT) et filtrage ;
on décrit les différentes formes que peuvent prendre notre type

```
type taille = Petit | Moyen | Grand
type garniture = Vegetarien | Poulet | Boeuf
type sauce = Fromage | Algerienne | Curry | Harissa | Samurai | Blanche

type tacos = taille * garniture * sauce
```

pattern matching :

```
let prix_viande (v : viande) : float =
  match v with
  | Vegetarien -> 4.5
  | Poulet -> 5.0
  | Boeuf -> 6.0
```

• types récursifs ;

```
type liste_entier =
  | Vide
  | Element of int * liste_entier
```

• filtrage avec garde ;

```
let egal n1 n2 =
  match n1,n2 with
   | n,p when n=p  -> "vrai"
   | _ -> "faux" ;;
```

• inférence de type et filtrages non-exhaustifs ;
Dans ocaml, on n'est pas obligé de préciser le type de tous les objets qu'on manipule : celui-ci est inféré automatiquement au moment de la compilation :

```
let add x y = x + y;; (* pas besoin de préciser : x et y sont des int et le r
```

autre exemple : construire un arbre avec un type polymorphe

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree

let rec sum_tree tree =
  match tree with
  | Leaf -> 0
  | Node (value, left, right) -> value + sum_tree left + sum_tree right

let example_tree =
  Node (1, Node (2, Leaf, Leaf), Node (3, Leaf, Node (4, Leaf, Leaf)))
```

- enregistrements et déclarations directs dans les ADT ;

```
type t_etudiant = {nom : string; prenom : string; age : int; annee : int};;
```