I Introduction et paradigmes

Le langage OCaml (pour Objective Categorical Abstract Machine Language) est un langage de programmation développé par l'INRIA depuis 1984.

Un paradigme est une manière de penser et de représenter le monde. En informatique, on distingue plusieurs paradigmes, pas forcément incompatibles, sur la façon de programmer. Un langage de programmation est généralement conçu pour utiliser plus spécifiquement certains paradigmes. Nous en citons ici trois importants :

- La **programmation impérative** consiste à effectuer des opérations successives modifiant l'état de la mémoire. Les objets informatiques sont donc modifiables et l'ordre dans lequel les opérations sont effectuées a une importance.
- La **programmation fonctionnelle** considère le calcul informatique comme l'évaluation de fonctions mathématiques, l'exécution d'un programme étant l'évaluation de différentes fonctions définies au fur et à mesure, qui ne peuvent pas modifier l'état de la mémoire. Dans ce paradigme, il n'y a pas de différence de nature entre un objet simple (entier, chaîne de caractère, tableau) et une fonction : tous sont vus comme des éléments d'ensembles mathématiques différents.
- La **programmation orientée objet** voit les données informatiques comme des objets comportant des attributs qui les définissent, et des méthodes propres à ces objets qui peuvent modifier leurs attributs.

En plus de ces grands paradigmes, on peut trouver des « sous-paradigmes » plus ou moins adaptés à ces premiers. Citons notamment :

- La **programmation itérative** consiste à la répétition d'un bloc d'instruction, de manière conditionnelle (boucle `while`) ou inconditionnelle (boucle `for`).
- La **programmation récursive** est une méthode qui résout des problèmes par la résolution de sous-problèmes similaires mais de taille plus petite. En pratique, cela se manifeste par une fonction qui s'appelle elle-même.

Ainsi, le langage Python est un langage principalement impératif qui utilise essentiellement de la programmation itérative. Il y est cependant possible de faire de la programmation récursive. Le langage OCaml est un langage principalement fonctionnel et orienté objet (d'où le « Objective ») qui est plus adapté pour de la programmation récursive, mais qui permet l'utilisation de boucles.

II Un langage fortement typé

Contrairement à Python qui effectue souvent une conversion automatique des types manipulés (on parle de langage faiblement typé), ou qui permet à une fonction de renvoyer des valeurs de types différents (parfois une liste, parfois un entier), OCaml est beaucoup moins permissif et effectue un contrôle strict des types. Ainsi, une fonction ne peut être appliquée qu'à des arguments du même type, et ne renvoie toujours qu'un seul type d'élément. Il existe cependant des fonctions permettant d'effectuer cette conversion.

Nous présentons ici les types élémentaires de OCaml.

II.1 Type `unit`

Ce type correspond au `None` de Python, c'est à dire le « rien ». Il n'existe qu'une seule valeur de ce type, et elle est notée avec des parenthèses :

```
# ();;
- : unit = ()
```

Lorsqu'une fonction ne renvoie aucune valeur, mais se contente de modifier l'environnement (afficher une image, modifier une valeur dans un tableau, imprimer du texte, ...), le type renvoyé par cette fonction est le type `unit`.

II.2 Entiers

Sur un ordinateur 64 bits, le type `int` sont les entiers de l'intervalle $[\![-2^{62}; 2^{62}-1]\!]$. Les opérations usuelles sont désignées par `+`, `-`, `*`, `/` (division entière, c'est-à-dire quotient de la division euclidienne) et `mod` (modulo, c'est-à-dire reste de la division euclidienne). Pour faire l'analogie, ces deux dernières opérations correspondent aux `//` et `%` de Python (on prendra garde à ne pas intervertir les notations des deux langages).

```
# 5 + 2 * 7;;
- : int = 19
# 19 mod 4;;
- : int = 3
# 19 / 4;;
- : int = 4
# max_int;;
- : int = 4611686018427387903
# min_int;;
- : int = -4611686018427387904
# 2305843009213693952 * 2;;
- : int = -4611686018427387904
```

II.3 Flottants

Comme en Python, le type `float` permet de représenter les nombres réels (ou plutôt leur approximation décimale). Les opérateurs sont notés pour ces derniers : `+.`, `-.`, `*.`, `/.` et `**` (puissance). On retrouve en outre des fonctions mathématiques usuelles : `sqrt` (racine carrée), `sin`, `cos`, `tan`, `exp`, `log` (logarithme népérien), ...

```
# 3.54 *. 2.;;
- : float = 7.08
# sqrt 4.;;
- : float = 2.
# sin 3.14;;
- : float = 0.0015926529164868282
```

Notons que les opérateurs des entiers ne fonctionnent pas avec des flottants et réciproquement :

```
# 3.54 *. 2;;
Characters 9-10:
  3.54 *. 2;;
          ^
Error: This expression has type int but an expression was expected of
type float
# 3.54 * 2.;;
Characters 1-5:
  3.54 * 2.;;
  ^^^^
Error: This expression has type float but an expression was expected of
type int
```

Des conversions sont cependant possibles entre les types :

```
# int_of_float 5.13;;
- : int = 5
# float_of_int 3;;
- : float = 3.
```

II.4 Booléens

Les deux valeurs possibles du type `bool` sont `true` et `false` (sans majuscule, contrairement à Python). Les opérateurs booléens sont `&&` (ET logique), `||` (OU logique) et `not` (négation). Comme en Python, l'évaluation des booléens est paresseuse. Notons que le test d'égalité se fait par un simple `=` au lieu de `==` en Python et que le test de différence se fait par `<>`.
⚠ Si le `or` est synonyme du `||`, on prendra garde à ne pas utiliser le `and` qui a une autre signification en OCaml.

```
# 5 > 3;;
- : bool = true
# false || 3. = 1.5 *. 2.;;
- : bool = true
# (2 < 1) && not (6. > 4.);;
- : bool = false
```

II.5 Tuples

Les tuples (correspondant aux *uplets* en français) sont similaires à ceux de Python : on construit un élément appartenant à un produit cartésien de plusieurs ensembles, pouvant ici être de types différents. Comme en Python, on les note :

```
# (1, true, 3.);;
- : int * bool * float = (1, true, 3.)
```

Notons que les parenthèses sont facultatives :

```
# let x = 1, 2, 3.14;;
val x : int * int * float = (1, 2, 3.14)
```

Si on veut accéder à l'une des composantes d'un tuple, il n'y a pas d'autre choix que de leur donner un nom à chacune, de la façon suivante :

```
# let a, b, c = x;;
val a : int = 1
val b : int = 2
val c : float = 3.14
```

L'exception est le cas des couples (et uniquement les couples) où l'on dispose des fonctions `fst` et `snd`.

```
# let x = 1, true;;
val x : int * bool = (1, true)
# fst x;;
- : int = 1
# snd x;;
- : bool = true
```

II.6 Tableaux

Les tableaux en OCaml sont de type `array`. Leur utilisation ressemble à celle de la classe `list` en Python, à quelques restrictions près :

- Ils sont de taille fixe, choisie à la création du tableau. On ne trouve donc pas d'équivalents aux fonctions `append`, `pop`, `remove`, ...
- Tous les éléments d'un tableau doivent être du même type. Il n'y a par contre pas de restriction sur le type d'un tableau.

```
# [|true; false; false|];;
- : bool array = [|true; false; false|]
# [|1; 2; 3|];;
- : int array = [|1; 2; 3|]
```

L'encadrement se fait par `[| |]` et la séparation par `;`. On prendra donc garde à éviter les erreurs du type :

```
# [|1,2,3|];;
- : (int * int * int) array = [|(1, 2, 3)|]
```

Ce dernier tableau étant un tableau à un seul élément, qui est un triplet d'entiers.

L'accès à un élément du tableau ou la modification d'un élément du tableau se font en temps constant selon la syntaxe suivante :

```
# let t = [|1; 2; 3|];;
val t : int array = [|1; 2; 3|]
# t.(2);;
- : int = 3
# t.(1) <- 4;;
- : unit = ()
```

Cette dernière opération renvoie un type `unit` : elle ne renvoie pas de valeur, mais se contente de modifier l'état de la mémoire.

On utilise `Array.length` pour connaître la taille d'un tableau, et `Array.make` pour créer un tableau ne contenant qu'une seule valeur répétée plusieurs fois :

```
# Array.length t;;
- : int = 3
# Array.make 5 true;;
- : bool array = [|true; true; true; true; true|]
```

II.7 Caractères, chaînes de caractères

Comme dans d'autres langages de programmation, les textes en OCaml sont encodés dans des chaînes de caractères. Une des particularités est qu'une **chaîne de caractères** (type `string`) est composée de plusieurs **caractères** (type `char`) et qu'il y a une différence entre les deux. On notera avec des guillemets doubles les premières et des guillemets simples les seconds :

```
# let s = "Louis-le-Grand";;
val s : string = "Louis-le-Grand"
# let c = 'l' and d = "l";;
val c : char = 'l'
val d : string = "l"
```

Les chaînes de caractères se comportent globalement comme les tableaux (taille fixe, mais contenu modifiable). L'accès à un caractère donné se faisant par `s.[i]` :

```
# String.length s;;
- : int = 14
# s.[3] <- 'r';;
- : unit = ()
# String.make 6 'b';;
- : string = "bbbbbb"
```

III Constante globale, constante locale, références

De par la programmation fonctionnelle, la notion de **variable** est traitée de manière particulière en OCaml. Nous présenterons donc ici la manipulation de **constantes**, c'est-à-dire de valeurs non modifiables.

III.1 Constantes globales

La syntaxe utilisée pour définir une constante est : « `let` *nom* `=` *expression* » :

```
# let a = 4;;
val a : int = 4
# a * 3;;
- : int = 12
```

On peut toutefois redéfinir une constante en utilisant un nom déjà utilisé. Notons que la définition est statique et non rétroactive :

```
# let b = a + 2;;
val b : int = 6
# let a = 2;;
val a : int = 2
# b;;
- : int = 6
```

III.2 Constantes locales

Dans certains cas, on n'a besoin d'une valeur que dans l'expression qui suit sa définition, par exemple dans une fonction, pour un calcul intermédiaire, etc. La syntaxe pour la déclaration de constantes locales est « `let` *nom* `=` *expression* `in` ». Le nom utilisé peut-être le même que celui d'une constante globale sans que cela modifie le contenu de cette dernière.

```
# let x = let a = 3.14 in a *. a ** a;;
val x : float = 114.10081408374859
# a;;
- : int = 2
```

III.3 Déclaration simultanée

Il est possible de déclarer plusieurs constantes simultanément. Cela se fait avec le mot clé `and` (qui n'est donc pas un opérateur booléen) :

```
# let a = 5 and b = 6;;
val a : int = 5
val b : int = 6
```

Attention, comme son nom l'indique, cette déclaration est simultanée, donc l'une des constantes ne peut pas dépendre de l'autre :

```
# let c = 5 and d = 2 * c;;
Characters 22-23:
  let c = 5 and d = 2 * c;;
                        ^

Error: Unbound value c
```

III.4 Références

Les références sont des cases mémoires d'un type défini à l'avance dont le contenu peut être modifié. Elles permettent l'utilisation de variables et donc d'effectuer de la programmation impérative. En OCaml, on distinguera le nom de la case mémoire et son contenu.

```
# let x = ref 1;;
val x : int ref = {contents = 1}
# x;; (* La référence *)
- : int ref = {contents = 1}
# !x;; (* et son contenu *)
- : int = 1
```

La modification d'une référence peut dépendre de son contenu. On utilise la syntaxe suivante :

```
# x := !x + 2;;
- : unit = ()
# !x;;
- : int = 3
# incr x; !x;;
- : int = 4
# decr x; !x;;
- : int = 3
```

Exercice 1

Déterminer le résultat de l'évaluation des instructions suivantes :

```
let x = 1 and y = 3 in let x = 2 in x + y;;

let x, y, z = 1, 2, 3 in x + y + z;;

let x, y, z = 1, (2, 3) in x + y + z;;

let x, y, z = true, 2, 3 in x or y = z;;

let x, y, z = true, 2, 3 in x and y < z;;

let t = [|1, 2, 3.|] in t.(0);;
```

IV Tests conditionnels

La syntaxe des tests est classique, comme le montre l'exemple suivant :

```
# let x = if 2 > 3 then 5 else 3;;
val x : int = 3
```

Comme dans beaucoup de langages, le `else` et le deuxième bloc d'instructions sont facultatif, mais OCaml étant fortement typé, il est alors nécessaire que le premier bloc d'instruction soit de type `unit` :

```
# let x = if 2 > 3 then 5;;
Characters 23-24:
  let x = if 2 > 3 then 5;;
                        ^

Error: This expression has type int but an expression was expected of
type unit
```

Si un bloc d'instruction contient plus d'une instruction, il est nécessaire d'utiliser un parenthésage. Ce dernier peut se faire avec des parenthèses, ou avec les mots clés `begin...end` :

```
# let x = if 2 > 3 then (incr a; 5) else begin print_int 3; 3 end;;
3val x : int = 3
```

Notons toutefois que les instructions supplémentaires (à part la dernière) sont toujours de type `unit` et sont séparées par des points-virgules.

V Programmation itérative

Les boucles `for` et `while` fonctionnent comme en Python, mais la syntaxe est un peu différente. Le bloc d'instruction qui est répété est toujours encadré par les mots clés `do...done`. Le bloc d'instruction doit nécessairement être de type `unit`, c'est-à-dire se contenter de modifier l'environnement, sans renvoyer de valeur.

V.1 Boucles inconditionnelles

La syntaxe est « `for` *variable* `=` *valeur initiale* `to` *valeur finale* `do` *bloc d'instruction* `done` » :

```
# for i = 1 to 5 do
    print_int i
  done;;
12345- : unit = ()
```

Dans le cas d'une boucle à valeurs décroissantes, le `to` est remplacé par `downto`.

V.2 Boucles conditionnelles

La syntaxe est « `while` *condition* `do` *bloc d'instruction* `done` » :

```
# let i = ref 5;;
val i : int ref = {contents = 5}
# while !i > 0 do
    print_int !i;
    decr i
  done;;
54321- : unit = ()
```

VI Fonctions

VI.1 Introduction

Les fonctions sont définies à l'aide d'une syntaxe proche de la notation mathématique correspondant à $f : x \longmapsto x + 2$:

```
# let f x = x + 2;;
val f : int -> int = <fun>
# f 4;;
- : int = 6
```

Les parenthèses autour des variables sont facultatives, mais peuvent être nécessaires selon le contexte :

```
# f 4 * 2;;
- : int = 12
# (f 4) * 2;;
- : int = 12
# f (4 * 2);;
- : int = 10
```

VI.2 Fonctions à plusieurs arguments

De manière simple, il suffit d'écrire les noms des arguments avant le signe = dans la définition de la fonction :

```
# let g x y = x + y;;
val g : int -> int -> int = <fun>
# g 3 4;;
- : int = 7
```

Notons que dans la deuxième ligne, les types indiqués sont ceux des arguments dans l'ordre où ils sont écrits, sauf le dernier qui est le type du résultat de la fonction. On appelle cette ligne la **signature** de la fonction.
On peut, de manière similaire, définir la fonction à l'aide d'un tuple :

```
# let h (x, y) = x + y;;
val h : int * int -> int = <fun>
# h (3, 4);;
- : int = 7
```

Ici, il s'agit en fait d'une fonction à un seul argument qui est de type `int * int`. Informatiquement, il y a donc une différence entre les deux fonctions.

La fonction `g` est dite *curryfiée* (en référence à Haskell Curry), et la fonction `h` est dite *non curryfiée*. La première version sera privilégiée car elle présente les avantages de faciliter l'utilisation de fonctions partielles. Par exemple, pour définir `f` avec `g` ou `h`, la syntaxe aurait été :

```
# let f = g 2;;
val f : int -> int = <fun>
# let f x = h (2, x);;
val f : int -> int = <fun>
```

VI.3 Fonctions anonymes

Comme en Python, il est possible de définir des fonctions anonymes suivant la syntaxe :

```
# function x -> x + 2;;
- : int -> int = <fun>
```

En Python, cela correspond à `lambda x: x + 2`.

Ainsi, les fonctions `f`, `g` et `h` auraient eu les définitions suivantes avec le mot clé `function` :

```
# let f = function x -> x + 2;;
val f : int -> int = <fun>
# let g = function x -> function y -> x + y;;
val g : int -> int -> int = <fun>
# let h = function (x, y) -> x + y;;
val h : int * int -> int = <fun>
```

Dans cet exemple, on comprend qu'écrire une version curryfiée d'une fonction à beaucoup d'arguments peut vite devenir pénible. Pour éviter cela, on peut utiliser le mot clé `fun` :

```
# let g = fun x y -> x + y;;
val g : int -> int -> int = <fun>
```

Ainsi, `fun x y z ...` est équivalent à `function x -> function y -> function z ...`

VI.4 Polymorphisme

Certaines fonctions n'imposent aucune contrainte sur le type des arguments. Elles sont dites **polymorphes**.

```
# max;;
- : 'a -> 'a -> 'a = <fun>
# fst;;
- : 'a * 'b -> 'a = <fun>
# let f = function x -> (x, x);;
val f : 'a -> 'a * 'a = <fun>
# let g = fun x y -> (x, y);;
val g : 'a -> 'b -> 'a * 'b = <fun>
```

Dans les exemples ci-dessus, on constate que les lettres du début de l'alphabet sont utilisées pour désigner un type qui n'est pas encore déterminé.

VI.5 Filtrage

Dans certains cas, les tests conditionnels ont plus que deux possibilités à envisager. Dans ce cas, il peut être intéressant de lister toutes les possibilités concernant certaines valeurs. C'est ce qu'on appelle le **filtrage** ou *matching* en anglais. Par exemple, pour définir la fonction $f : \mathbb{N} \longrightarrow \mathbb{N}$ telle que $\begin{cases} f(0) = 0 \\ f(1) = 2 \\ f(n) = 2n+1 \text{ si } n \geq 2 \end{cases}$, la syntaxe est la suivante :

```
# let f n = match n with
    | 0 -> 0
    | 1 -> 2
    | n -> 2 * n + 1;;
val f : int -> int = <fun>
```

Il faut bien prendre garde d'aller du plus spécifique au plus général. En effet, si l'un des cas traite déjà d'un cas qui est traité après, ce dernier sera inutile :

```
# let f n = match n with
    | 0 -> 0
    | n -> 2 * n + 1
    | 1 -> 2;;
Characters 62-63:
    | 1 -> 2;;
      ^

Warning 11: this match case is unused.
val f : int -> int = <fun>
```

Il s'agit ici d'un avertissement et non pas d'une erreur : la fonction peut être utilisée, mais ne renverra peut-être pas ce qu'on avait prévu.

Lors de l'écriture d'une fonction anonyme, le mot clé `match` n'est pas écrit :

```
# let f = function
    | 0 -> 0
    | 1 -> 2
    | n -> 2 * n + 1;;
val f : int -> int = <fun>
```

Lorsque l'on souhaite regrouper tous les cas non encore traités, on peut utiliser le symbole `_`. Par exemple, considérons la fonction définie par :

$$\begin{array}{rcl} f : \mathbb{N} \times \mathbb{N} & \longrightarrow & \mathbb{N} \\ m, n & \longmapsto & \begin{cases} 0 & \text{si } m = 0 \\ 1 & \text{si } m \neq 0 \text{ et } n = 0 \\ 2m+n & \text{si } m < n \\ 3 & \text{sinon} \end{cases} \end{array}$$

Alors elle s'écrira en OCaml :

```
# let f m n = match m, n with
      | 0, _ -> 0
      | _, 0 -> 1
      | m, n when m < n -> 2 * m + n
      | _ -> 3;;
val f : int -> int -> int = <fun>
```

Notons la présence du mot-clé `when` pour rajouter un test conditionnel sur les deux variables. Les deux dernières lignes de la fonction auraient été équivalentes à :

```
      | m, n -> if m < n then 2 * m + n else 3;;
```

⚠ Pour vérifier que les deux valeurs filtrées sont égales, il est nécessaire d'utiliser le mot-clé `when` :

```
# let f m n = match m, n with
      | k, k -> 0
      | _ -> 1;;
Characters 38-39:
      | k, k -> 0
             ^
Error: Variable k is bound several times in this matching
# let f m n = match m, n with
      | k, l when k = l -> 0
      | _ -> 1;;
val f : 'a -> 'a -> int = <fun>
```

VI.6 Récursivité

Même si elle sera étudiée plus en détail par la suite, nous donnons ici la syntaxe d'une fonction récursive, c'est-à-dire d'une fonction qui s'appelle elle-même. Il faut retenir que c'est le mot-clé `rec` qui permet de préciser qu'une fonction est récursive. Par exemple, la fonction $f : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $\begin{cases} f(0) = 1 \\ f(n) = 2f(n-1) + 3 \text{ si } n > 0 \end{cases}$ sera écrite :

```
# let rec f = function
      | 0 -> 1
      | n -> 2 * f (n - 1) + 3;;
val f : int -> int = <fun>
```

On utilise le mot-clé `and` pour définir des fonctions inter-dépendantes :

```
# let rec pair = function
      | 0 -> true
      | n -> impair (n - 1)
  and impair = function
      | 0 -> false
      | n -> pair (n - 1);;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>
```

Exercice 2

Déterminer les signatures (les types retournés) des fonctions suivantes :

```
let f g h x = (g x) + (h x);;

let f g h x = g (h x);;

let f x y = function
   | 0, 0 -> 1
   | x, y -> x + y;;

let rec f g n = match n with
   | 0 -> (function x -> x)
   | n -> (function x -> (g ((f g (n - 1)) x)));;
```

Exercice 3

Écrire en OCaml :

- La fonction `f1` qui à $f : \mathbb{R} \longrightarrow \mathbb{R}$ associe $g = \dfrac{f}{2}$.
- La fonction `f2` qui à $f : \mathbb{N} \longrightarrow \mathbb{N}$ associe $x \longmapsto f(x+1)$.
- La fonction `f3` qui à $f : \mathbb{R} \longrightarrow \mathbb{N}$ et $x \in \mathbb{R}$ associe $f(x^2) + f(x)^2$.

Exercice 4

Écrire une fonction `maximum` qui recherche la valeur maximale dans un tableau donné en argument. La fonction doit être de signature `'a array -> 'a = <fun>`.
Écrire une fonction `indice_max` qui renvoie l'indice du maximum dans un tableau donné en argument. La fonction doit être de signature `'a array -> int = <fun>`.

Remarque A

> Les opérateurs de comparaison : `<`, `>`, `<=`, `>=` et `=` sont polymorphes.

Exercice 5

Écrire une fonction `miroir` qui prend en argument une chaîne de caractères et renvoie la chaîne miroir. Par exemple, `miroir "Bonjour"` renverra `"ruojnoB"`. En déduire une fonction `palindrome` qui teste si une chaîne de caractère est un palindrome (qui se lit de la même manière dans les deux sens).

Exercice 6

Écrire une fonction **récursive** `puissance` qui prend en arguments deux entiers naturels `a` et `n` et qui calcule a^n. On prendra pour convention que $a^0 = 1$ et $0^n = 0$ si $n \neq 0$.