Introduction au langage OCaml

I. Deux paradigmes de programmation

On distingue en général de façons principales d'envisager la programmation, deux **paradigmes** principaux :

La programmation impérative se base sur l'idée qu'un ordinateur est une machine à mémoire vive. Ainsi, un programme va être une succession d'instruction qui vont modifier l'état de l'ordinateur, c'est-à-dire les valeurs dans différentes cases mémoire.

La programmation fonctionnelle est basée sur la notion de fonctions, i.e. de transformations de données que l'on va composer. Avec cette approche, on ne se préoccupe pas de la gestion de la mémoire. De plus, dans un langage fonctionnel, il n'y a pas de distinction entre instruction et expression, tout est expression.

On peut aussi mentionner la **programmation objet** qui concerne plus la manière dont sont organisées les données que l'on manipule. Mais c'est un aspect complémentaire et relativement indépendant de la distinction impératif/fonctionnel.

Le langage Python est plutôt un langage impératif, alors que le langage OCaml est plutôt fonctionnel. Cependant, comme beaucoup de langages modernes, chacun des deux langages possède des capacités relevant à la fois de l'impératif et du fonctionnel.

Exemple Prenons l'exemple de la factorielle. On peut, *grosso modo*, la définir de deux manières :

$$n! = \prod_{k=1}^{n} k \qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Approche impérative Pour calculer $n!$, on procède ainsi :

1. On réserve une case mémoire dans laquel on va stocker un entier r que l'on initialise à 1.
2. Pour toutes les valeurs de k allant de 1 à n, à l'aide d'une boucle, on multiplie r par k.
3. À la fin, r contient $n!$.

Approche fonctionnelle Pour calculer $n!$, deux cas sont possibles :

- ⋆ Si $n = 0$, le résultat vaut 1,
- ⋆ Sinon, on calcule $(n-1)!$ (à l'aide d'un appel de fonction) et on renvoie le résultat obtenu multiplié par n.

II. Présentation du langage

Voici de premiers éléments du langage, que l'on continuera de développer dans les chapitres suivants.

1. Types de données

La langage OCaml est un langage où la notion de **type** est primordiale, via un principe de *sureté de typage* : le compilateur vérifie pour tous les appels de fonctions que les arguments ont le bon type. Ainsi, on est sûr que lors de l'exécution du programmen, il n'y a aura pas de problème de données n'ayant pas le bon type.[1]

1.1. Types simples

Entiers et flottants

```
# 1 + 1 ;;
- : int = 2
# 2.7 *. 3.4 ;;
- : float = 9.18
# 2 ** 4 ;;
Line 1, characters 0-1:
Error: This expression has type int but an expression was
       expected of type float
  Hint: Did you mean `2.'?
# 2. ** 4. ;;
- : float = 16.
```

Remarque Les entiers sont signés, et compris entre $[\![-2^{62}, 2^{62}-1]\!]$ (dans une architecture 64 bits). Dans un autre langage, l'intervalle des valeurs possibles est en général $[\![-2^{63}, 2^{63}-1]\!]$ mais OCaml utilise un chiffre binaire pour la gestion de la mémoire.

Booléens

[1] Python ne fait pas ce genre de vérification à la compilation, mais à l'exécution.

```
# true ;;
- : bool = true
# not true ;;       (* négation *)
- : bool = false
# false || true ;; (* disjonction *)
- : bool = true
# true && false ;; (* conjonction *)
- : bool = false
# 1 = 2 ;;          (* test d'égalité *)
- : bool = false
# 3 <> 4 ;;         (* test de non-égalité *)
- : bool = true
```

Remarque La conjonction et la disjonction sont paresseuses : si leur valeur peut être déterminée à partir du premier argument, le second n'est pas évalué.

```
# true || (1 / 0) = 2 ;;
- : bool = true
```

Caractères et chaînes de caractères

Contrairement à Python, et comme l'immense majorité des langages de programmation, OCaml distingue les caractères (que l'on entoure de `'`) des chaînes de caractères (entourées de `"`).

```
# 'a' ;;
- : char = 'a'
# "Bonjour" ;;
- : string = "Bonjour"
# String.length "Bonjour" ;;
- : int = 7
```

1.2. Définition de variables

Faisons une courte pause pour présenter la définition de variables.

Commençons par la définition de variables globales dont la syntaxe est « `let` identificateur `=` expression » que l'on peut compléter de définitions supplémentaires à l'aide de `and`.

Un nom de variable (on parlera d'*identifiant*) commence obligatoirement par une lettre **minuscule**, et peut ensuite contenir des lettres (minuscules ou majuscules), des chiffres et les caractères « _ » et « ' ».[2]

[2] Pour être rigoureux, un identifiant peut aussi commencer par un *underscore*.

```
# let x = 3 ;;
val x : int = 3
# x + 1 ;;
- : int = 4
# let y = 'a' and z = false || true ;;
val y : char = 'a'
val z : bool = true
```

On peut aussi faire des définitions locales en ajoutant `in`.

```
# let y = 2 + 2 in y * y ;;
- : int = 16
# y ;;
- : char = 'a'
```

1.3. Types composés

Paires et tuples

```
# let a = (1, "poulet") ;;
val a : int * string = (1, "poulet")
# fst a ;;
- : int = 1
# snd a ;;
- : string = "poulet"
```

On peut aussi définir des n-uplets, pour $n \geq 3$, mais on ne peut plus utiliser `fst` et `snd`.

```
# (2.0, true, 1) ;;
- : float * bool * int = (2., true, 1)
# fst (2.0, true, 1) ;;
Line 1, characters 4-18:
Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type 'd * 'e
```

Par contre, on peut décomposer un tuple à l'aide d'un `let` (et voyez comment le tiret bas, l'*underscore*, peut servir de joker) :

```
# let a = (2.0, true, 1) ;;
val a : float * bool * int = (2., true, 1)
# let (x, _, y) = a ;;
val x : float = 2.
val y : int = 1
```

C'est un cas particulier d'une construction plus générale sur laquelle nous revien-

drons.

Listes et tableaux

En OCaml, nous avons de types de données très utiles, les listes et les tableaux.

★ **Listes**

- ★ Structure homogène (un seul type de données) ;
- ★ Taille variable ;
- ★ Accés à la tête et au reste en temps constant.

```
# [1; 2; 3] ;;
- : int list = [1; 2; 3]
# 1 :: [2; 3] ;;
- : int list = [1; 2; 3]
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

★ **Tableaux**

- ★ Structure homogène ;
- ★ Taille fixe ;
- ★ Accès à n'importe quel élément en fonction de sa position en temps constant.

```
# let a = [| 2; 3; 0; 4 |] ;;
val a : int array = [|2; 3; 0; 4|]
# a.(2) ;;
- : int = 0
# a.(1) <- 5 ;;
- : unit = ()
# a ;;
- : int array = [|2; 5; 0; 4|]
```

Notons que contrairement à la majorité des langages de programmation, en Python, les deux structures sont fusionnée en une seule (qui sont en fait des *tableaux dynamiques*).

2. Fonctions

On peut définir une fonction à l'aide du mot `function`.

```
# function x -> x + 1 ;;
- : int -> int = <fun>
# let f = function x -> [x] ;;
val f : 'a -> 'a list = <fun>
# f(1) ;;
- : int list = [1]
# f 3 ;;
- : int list = [3]
```

On peut écrire `fun` comme raccourci.[3] De plus, si l'on définit une fonction avec un nom, on peut utiliser une écriture plus simple.

```
# let g x = (x, x + 1) ;;
val g : int -> int * int = <fun>
# g 4 ;;
- : int * int = (4, 5)
```

Attention, il n'y a pas de `return` en OCaml. Une fonction évalue une unique quantité, et c'est cette quantité qui est le résultat.

2.1. Fonctions à plusieurs arguments

On peut être tenté d'écrire, par exemple :

```
# let diff (x, y) = x - y ;;
val diff : int * int -> int = <fun>
```

Il s'agit en fait d'une fonction à un unique argument, qui est une paire d'entiers. Une « vraie » fonction à deux arguments s'écrira :

```
# let prod x y = x * y ;;
val prod : int -> int -> int = <fun>
```

On dit que `prod` est *currifiée*[4], alors que l'on pourrait non-currifiée en écrivant

```
function (x, y) -> x * y.
```

On privilégiera les versions currifiées. Un avantage est que l'on peut considérer l'application partielle d'une fonction.

```
let fois2 = prod 2 ;;
val fois2 : int -> int = <fun>
# fois2 3 ;;
- : int = 6
```

2.2. Expressions conditionnelles

Il est possible de faire des tests et donc d'avoir des expressions conditionnelles en

[3] Il y a de petites différences avec `function`, nous y reviendront.
[4] Du nom du mathématicien Haskell Curry

OCaml, à l'aide de constructions du type « `if ... then ... else ...` ».

```
# let plus_grand a b = if a < b then b else a ;;
val plus_grand : 'a -> 'a -> 'a = <fun>
# let delta p = if p then 1 else 0 ;;
val delta : bool -> int = <fun>
```

Par contre, contrairement à d'autres langages, la partie `else` est obligatoire, car dans l'approche fonctionnelle, quelle que soit la branche suivie, on doit renvoyer une valeur, et les deux branches doivent renvoyer des valeurs de même type.

```
# if true then 1 ;;
Line 1, characters 13-14:
Error: This expression has type int but an expression was
       expected of type unit because it is in the result of
       a conditional with no else branch
```

2.3. Récursivité

On peut définir des fonctions qui s'appellent elles-mêmes, on parle alors de fonctions récursives, en ajoutant le mot-clé `rec` après le `let`.

Voici, par exemple, une implémentation de la factorielle :

```
let rec factorielle n =
  if n <= 1 then
    1
  else
    n * factorielle (n - 1)
;;
```

C'est une notion très importante, nous y reviendrons longuement.

2.4. Polymorphisme

Dans les exemples précédents, on a vu des cas où le type d'une fonction n'était pas entièrement déterminé.

```
# fun (x, y) -> y ;;
- : 'a * 'b -> 'b = <fun>
# snd ;;
- : 'a * 'b -> 'b = <fun>
# let f x = [x] ;;
val f : 'a -> 'a list = <fun>
# f 1 ;;
- : int list = [1]
# f "Bonjour";;
- : string list = ["Bonjour"]
```

On parle alors de fonctions **polymorphes**, pour lesquelles le type du résultat dépend explicitement du type des arguments.

Concrètement, les variables de type `'a`, `'b`, ... (que l'on nomme en général *alpha*, *béta*, ...) sont déterminées en fonction des arguments.

Un exemple important de fonctions polymorphes est constitué par les opérateurs de comparaison (les opérateurs infixes s'étudient en les entourant de parenthèses).

```
# let egal a b = a = b ;;
val egal : 'a -> 'a -> bool = <fun>
# (=) ;;
- : 'a -> 'a -> bool = <fun>
# (<>) ;;
- : 'a -> 'a -> bool = <fun>
# (<) ;;
- : 'a -> 'a -> bool = <fun>
# (>=) ;;
- : 'a -> 'a -> bool = <fun>
```

3. Exercices

Exercice 1 Déterminer les types des fonctions suivantes :

1. `fun (x, y) -> (x, x) ;;`
2. `fun f g x ->  f (g x) ;;`
3. `fun f g x -> (f x) + (g x) ;;`

Exercice 2 Que renvoie l'interpréteur ?

```
# let h (f, g) = fun x -> f (g x) ;;
```

Exercice 3 Déterminer les types des fonctions suivantes, mais cette fois, les noms de variables ne donnent aucune information :

1. `fun x y z -> x (y z) ;;`
2. `fun x y z -> (x y) z ;;`
3. `fun x y z -> x y z ;;`
4. `fun x y z -> x (y z x);;`

5. `fun x y z -> (x y) (z x);;`

Exercice 4 – Suite récurrence

Écrire une fonction `suite_recurrente` telle que `suite_recurrente f n a` renvoie le terme u_n où $(u_k)_{k\in\mathbf{N}}$ est la suite définie par

$$u_0 = a \qquad \forall\, k \in \mathbf{N},\, u_{k+1} = f(u_k).$$

Quel est son type ?

Exercice 5 – Entiers de Hamming

Un entier est dit *de Hamming* s'il ne comporte que des facteur 2, 3 et 5 dans sa décomposition en facteurs premiers.

Écrire une fonction `est_hamming : int -> bool` qui indique si un entier est de Hamming ou non.

On pourra utiliser les opérations de quotient et de reste d'une division entière.

```
# 2048 / 12 ;;
- : int = 170
# 2048 mod 12 ;;
- : int = 8
```