

Introduction à OCaml

September 26, 2021

1 Principe généraux du langage

OCaml est un langage fonctionnel. Les fonctions jouent un rôle primordial par opposition aux objets dans un langage objets ou aux instructions dans un langage impératif (*c'est un peu plus compliqué que cela on y reviendra dans le chapitre sur les paradigmes*).

OCaml est un langage fortement typé dans lequel les types, même s'il sont omis dans le programme car inféré par le compilateur, jouent un rôle très important.

Quand on évalue dans l'interprète n'importe quelle valeur OCaml, elle a un type et OCaml répond comme ici `type = valeur`.

```
[1]: 3
```

```
[1]: - : int = 3
```

2 Les types de base

2.1 Entier

Comme on vient de le voir, le type des entiers relatifs est `int`. Contrairement à des langages comme Python, les entiers en OCaml ont une longueur fixe en mémoire. Sur ce serveur, ils sont compris entre -2^{62} et $2^{62} - 1$. Pour les manipuler, on utilise les opérations élémentaires `+`, `-`, `*` et `/`. Cette dernière division correspond au quotient dans la division euclidienne, on accède au reste avec `mod` : `12 mod 5` s'évalue ainsi sur 2.

2.2 Les flottants

Le type `float` correspond aux nombres à virgule flottante. Il s'agit d'une représentation des nombres réelles qui, bien que ne permettant d'en manipuler qu'un nombre fini, permet tout de même de faire des calculs numériques. Les opérations qu'on vient de voir sur les entiers ne *s'appliquent pas sur des float*. Avec OCaml les opérations ont un unique type pour les opérandes et la valeur calculée. Il y a donc d'autres opérateurs pour les flottants, ce sont les mêmes que précédemment mais avec un `.` à la fin qui rappelle la notation de la virgule : `+. , -. , *. et /..`. De plus, on a l'exponentiation `a ** b` pour calculer a^b , des fonctions mathématiques usuelles comme `sqrt`, `cos`, `sin`, ...

On peut passer d'un entier à un flottant à l'aide de la fonction `float_of_int` ainsi `float_of_int 3` correspond au flottant 3.0. On convertit dans l'autre sens avec `int_of_float` mais cette opération

réalise une troncature.

2.3 Les chaînes de caractères et les caractères

Les caractères en OCaml sont représentés avec des accolades 'a' et les chaînes avec des guillemets "bonjour". Le type des caractères est `char` et celui des chaînes `string`. On verra quand ce sera utile la manipulation des chaînes.

2.4 Unit

Si tout à un type en OCaml, comment faire avec des expressions qui n'ont aucune valeur ? Par exemple une instruction d'affichage, comme ici pour faire un saut de ligne. On voit que la valeur `()` est celle renvoyée et également celle passée en paramètre, c'est l'unique valeur du type `unit` correspondant.

```
[30]: print_newline ();
```

```
[30]: - : unit = ()
```

Une expression qui n'a aucune valeur est appelée communément une instruction en référence aux langages de programmations impératifs. On peut utiliser l'opérateur `;` pour enchaîner des instructions : `instr1 ; ... ; instrn ; expr` la valeur sera alors la valeur de `expr`. Si jamais une des instructions n'en est pas une et a une valeur différente de `()`, OCaml produira un avertissement.

2.5 Les booléens et les expressions logiques

Comme dans les autres langages, les expressions logiques sont très importantes car elles permettent d'exprimer des conditions pour les instructions conditionnelles ou les boucles. En OCaml, il s'agit du type `bool` qui contient deux éléments `true` et `false`. On utilise très rarement ces valeurs directement dans un programme. En effet, elles sont produites par les relations de comparaison usuelles comme `=`, `<`, `<=`, `>` et `>=`. Contrairement aux opérations arithmétiques vues précédemment, ces comparaisons fonctionnent pour tous les types. On peut ainsi écrire `3 = 2` (qui s'évalue à `false`) ou `5.1 = 5.1` (qui s'évalue à `true`).

Les opérations logiques usuelles sont

- `not` pour la *négation* : `not b` s'évalue à `true` si et seulement si `b` vaut `false`
- `&&` pour la *conjonction* : `b1 && b2` s'évalue à `true` si et seulement si `b1` et `b2` valent `true`
- `||` pour la *disjonction* : `b1 || b2` s'évalue à `true` si et seulement si `b1` ou `b2` valent `true`

Une des manières les plus fréquentes d'utiliser les expressions booléennes est l'expression conditionnelle :

```
if condition
then expr1
else expr2
```

qui s'évalue en `expr1` si la condition vaut `true` et `expr2` sinon. **ATTENTION** le type de `expr1` doit être celui de `expr2` et c'est alors le type de l'expression conditionnelle. Ainsi on ne peut pas écrire

```
if b then 2 else 3.2
```

Quand la branche `else` ne comporte que la valeur `()` du type `unit`, ce qui correspond au cas où on ne veut rien faire dans le cas où la condition n'est pas vérifiée, on peut l'omettre. Ainsi,

```
if condition then expr else ()
```

peut s'écrire plus simplement

```
if condition then expr
```

ATTENTION le type de `expr` doit impérativement être `unit`.

Quand on enchaîne plusieurs instructions conditionnelles, ce n'est pas le plus clair :

```
if b1 then if b2 then e1 else if b3 then e2 else e3 else e4
```

On pourra alors rajouter des parenthèses pour clarifier :

```
if b1 then (if b2 then e1 else (if b3 then e2 else e3)) else e4
```

Il est aussi possible d'utiliser `begin/end` à la place de `()` :

```
if b1
then begin
  if b2
  then e1
  else begin if b2 then e2 else e3 end
end else e4
```

3 Les variables

Les variables ont un statut très différent en `OCaml` par rapport aux autres langages, ce sont de simples alias pour des valeurs définies par une expression. On fait la distinction entre les variables définies globalement et celles définies localement dans l'évaluation d'une expression.

3.1 Variable globale

On écrit, par exemple,

```
let x = 3
```

pour définir une variable `x` représentant la valeur 3 dans la suite du programme. Ce nom `x` peut alors être utilisé partout où on aurait pu écrire 3. La variable a ainsi une valeur et un type unique. Ils ne pourront jamais changer. Tout ce qu'on peut faire, c'est définir une nouvelle variable appelée `x`, mais alors ça ne changera aucune des valeurs précédemment calculés avec l'ancien `x` (ce qui est assez rassurant en fait).

On remarque que lorsqu'on définit une variable, `OCaml` répond

```
val variable : type = valeur
```

Ici on va avoir une autre variable de type `int` contenant la valeur 6

```
let y = x + 3
```

si on écrit ensuite

```
let x = 5
```

cela ne changera pas la valeur de `y` comme on peut le constater ici :

```
[23]: let x = 3
      let y = x + 3
      let x = 5
```

```
[23]: val x : int = 3
```

```
[23]: val y : int = 6
```

```
[23]: val x : int = 5
```

```
[3]: y
```

```
[3]: - : int = 6
```

3.2 Variable locale

On peut définir une variable utilisée uniquement dans l'évaluation d'une expression.

```
let var = valeur in expr
```

La variable n'est alors accessible que pour cette expression.

4 Les fonctions

OCaml est un langage fonctionnel, ainsi les fonctions sont essentielles et sont des valeurs comme les autres. Pour définir une fonction on utilise une syntaxe proche de la syntaxe de définition des variables sauf qu'on rajoute un certain nombre de paramètres. Par exemple, la fonction suivante calcule le produit de ses trois arguments :

```
[4]: let produit3 x y z = x * y * z
```

```
[4]: val produit3 : int -> int -> int -> int = <fun>
```

On voit que OCaml devine tout seul le type des arguments et le type de la valeur de retour. Le type de la fonction lui-même est un peu étrange avec plusieurs arguments. On y reviendra dans la suite mais pour le moment, on se contentera de dire qu'il est de la forme `type1 -> type2 -> ... ->`

`typen -> typer` où `type1` est le type du premier argument, `type2` celui du deuxième, et ainsi de suite jusqu'à `typer` le type des valeurs de retour de la fonction.

Pour appliquer une fonction, il suffit d'écrire `fonction arg1 ... argn` pas de parenthèses, pas de virgules, juste des espaces.

```
[8]: produit3 1 2 3
```

```
[8]: - : int = 6
```

```
[25]: (* Écrire une fonction carre qui prend un entier x en paramètre et renvoie son
      ↪ carré *)
      (* BEGIN SOLUTION *)
      let carre x = x * x
      (* END SOLUTION *)
```

```
[25]: val carre : int -> int = <fun>
```

```
[26]: assert(carre 2 = 4)
```

```
[26]: - : unit = ()
```

```
[27]: (* Écrire une fonction multadd qui prend trois entiers x y et z en paramètres
      ↪ et renvoie *)
      (* x * (y + z) *)
      (* BEGIN SOLUTION *)
      let multadd x y z = x * (y + z)
      (* END SOLUTION *)
```

```
[27]: val multadd : int -> int -> int -> int = <fun>
```

```
[28]: assert(multadd 2 3 4 = 14)
```

```
[28]: - : unit = ()
```

4.1 Types génériques

Si on définit une fonction comme `let identite x = x` on voit bien qu'aucune contrainte ne peut être imposée arbitrairement sur le type de `x`. En fait, cette fonction pourra s'appliquer sur une valeur de n'importe quelle type, la seule certitude est que la valeur de retour sera du même type que celui de l'entrée. Si on évalue cette fonction, on voit que OCaml répond

```
val identite : 'a -> 'a = <fun>
```

Ce 'a signifie n'importe quel type. Comme prévu, on voit que c'est le même 'a qu'on retrouve en entrée et en sortie.

Essayez de deviner les types des fonctions suivantes avant d'évaluer la cellule :

```
[1]: let f x y = y
      let g x y = 2+x
      let h x y z = x *. z
```

```
[1]: val f : 'a -> 'b -> 'b = <fun>
```

```
[1]: val g : int -> 'a -> int = <fun>
```

```
[1]: val h : float -> 'a -> float -> float = <fun>
```

```
[1]: (* Écrire une fonction max qui prend deux éléments d'un même type et renvoie
      le plus grand.
      On rappelle que les comparaisons ne sont pas typées *)
      (* BEGIN SOLUTION *)
      let max a b = if a > b then a else b
      (* END SOLUTION *)
```

```
[1]: val max : 'a -> 'a -> 'a = <fun>
```

```
[1]: assert (max 0 1 = 1);
      assert (max "a" "b" = "b")
```

```
[1]: - : unit = ()
```

```
[10]: (* Écrire une fonction mediane3 qui prend trois éléments d'un même type, tous
      ↪ distincts
      et renvoie l'élément au milieu dans l'ordre.
      Si on lui passe x < y < z elle renvoie y. *)
      (* BEGIN SOLUTION *)
      let mediane3 x y z =
        if x > y then begin
          if y > z then y
            else begin if x > z then z else x end
          end
        else begin if x > z then x
          else begin if y > z then z else y end
        end
      end
      (* END SOLUTION *)
```

```
[10]: val mediane3 : 'a -> 'a -> 'a -> 'a = <fun>
```

```
[4]: assert(mediane3 1 2 3 = 2);  
assert(mediane3 2 1 3 = 2);  
assert(mediane3 1 3 2 = 2);  
assert(mediane3 3 2 1 = 2);  
assert(mediane3 2 3 1 = 2);  
assert(mediane3 3 1 2 = 2);  
assert(mediane3 'a' 'b' 'c' = 'b')
```

```
[4]: - : unit = ()
```

4.2 Les fonctions comme des valeurs de première classe

Les fonctions en OCaml sont manipulables directement comme n'importe quelle valeur. La fonction suivante prend une fonction `f` et une valeur `x` en paramètre et applique `f` sur `x`. Son type est un peu compliqué mais si on décompose on comprend que `x` a un type quelconque `'a` et comme on applique `f` dessus, il faut que `f` prenne des éléments de type `'a` en argument. Aucune contrainte sur son type de retour, ainsi on indique que c'est un nouveau type générique `'b`. Donc le type de `f` est `'a -> 'b` et le type global de `applique` est `('a -> 'b) -> 'a -> 'b`.

Attention Les parenthèses sont nécessaires. Le type `'a -> 'b -> 'a -> 'b` serait celui d'une fonction à trois arguments dont le premier et le troisième sont de type `'a` et le second de type `'b`.

```
[4]: let applique f x = f x
```

```
[4]: val applique : ('a -> 'b) -> 'a -> 'b = <fun>
```

```
[18]: let compose f g x =  
      (* Compose prend deux fonctions f et g et un élément x et calcule l'image_  
      ↪ par  
      x de la composée de f et de g, donc f(g(x)) en notation mathématique_  
      ↪ usuelle *)  
      (* BEGIN SOLUTION *)  
      f (g x)  
      (* END SOLUTION *)
```

```
[18]: val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
[19]: (* Ce test vérifie que compose est du bon type *)  
let _ = ( compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b ) in  
(* Test sur une composition, on verra la syntaxe utilisée dans la suite *)  
assert (compose ((+) 2) (( * ) 3) 4 = 14)
```

```
[19]: - : unit = ()
```

4.3 Fonctions anonymes

Les fonctions sont tellement importantes en OCaml qu'il est parfois pratique d'en écrire sans avoir à les définir en leur donnant un nom. On parle alors de fonctions anonymes, et cela correspond à la notation mathématique usuelle $x \mapsto x^2$.

Pour définir une fonction en tant que valeur ainsi, il suffit d'écrire

```
fun x -> x*x
```

Une telle valeur peut alors être stockée dans une variable comme n'importe quelle autre valeur :

```
let carre = fun x -> x*x
```

va alors être une définition complètement équivalente à la précédente :

```
let carre x = x*x
```

En fait, cette dernière syntaxe est automatiquement transformée en la syntaxe utilisant une fonction anonyme.

Ainsi, en utilisant la fonction `compose` précédente, on peut écrire

```
compose (fun x -> x+3) (fun x -> x*4) 5
```

qui s'évaluera en $4 \times 5 + 3 = 23$.

Pour des fonctions prenant plusieurs arguments, on utilise la syntaxe

```
fun arg1 ... argn -> valeur
```

Par exemple, la fonction suivante `fun x y -> x+y` réalise l'addition.

4.4 Curryfication

Nous allons revenir sur la notation des types de fonctions à plusieurs arguments ici. En fait, les fonctions n'ont qu'un seul argument en OCaml et sont nécessairement d'un type `a -> b`. Afin de définir des fonctions de plusieurs arguments, on définit des fonctions qui renvoient des fonctions. Ainsi lorsqu'on écrit `fun x y -> x+y` en fait, c'est retranscrit par OCaml en `fun x -> ( fun y -> x+y )`. Si on sait que `x` est de type `int`, alors `fun y -> x+y` a le type `int -> int`. Donc `fun x -> (fun y -> x+y)` a le type `int -> (int -> int)`.

Par défaut, on considère que les parenthèses autour des `->` sont toujours à droite, et on se permet ainsi d'écrire `int -> int -> int`. On retrouve bien le type de la fonction `fun x y -> x+y` vu précédemment.

Attention le type `(int -> int) -> int` est lui bien différent car il correspond au type d'une fonction ayant un argument du type fonctionnel `int -> int` et qui renvoie un entier. C'est le cas de la fonction suivante d'évaluation en 1 :

```
let eval1 f = f 1
```

Le fait que les fonctions à plusieurs arguments soient des fonctions de fonctions permet de pouvoir les appliquer partiellement. En effet, quand on applique `f x1 ... xn` en fait, on évalue `(... (f x1) x2) ...)` `xn`. Mais rien ne nous oblige à donner tous les arguments. Par exemple si on a définit

```
let ajoute x y = x+y
```

l'évaluation `ajoute 3` est une fonction équivalente à `fun y -> 3+y`. On parle, pour ce mécanisme d'évaluation partielle, de curryfication et on dit que `ajoute 3` est une curryfiée de `ajoute`. Cela fait référence à Haskell Curry, précurseur de la programmation fonctionnelle.

5 La récursivité

Jusqu'ici toutes les fonctions ou expressions définies s'évaluent assez directement. On est loin de la richesse qu'on connaît dans d'autres langages. La manière usuelle de retrouver cette expressivité dans un langage fonctionnel est de passer par la récursivité. On dit qu'une fonction ou une expression est récursive quand elle fait référence à elle même. Le cas des expressions récursives est marginal, on ne considère ici que des fonctions.

Naturellement, comme on y fait référence par son nom, cela ne peut concerner que des valeurs définies par `let`. Pour définir une fonction récursive, on utilise `let rec` plutôt que `let`.

On peut ainsi définir la factorielle mathématique usuelle :

```
[20]: let rec fact n =  
      if n = 0  
      then 1  
      else n * fact (n-1)
```

```
[20]: val fact : int -> int = <fun>
```

```
[17]: fact 5
```

```
[17]: - : int = 120
```

Il est possible de demander à OCaml de retracer l'évaluation de `fact 5` pour faire apparaître les appels. On voit qu'on besoin d'**empiler** les appels sur des arguments de plus en plus petit jusqu'à aboutir au cas de base : `fact 0`.

```
[18]: #trace fact;;  
fact 5
```

```
[18]: - : int = 120
```

```
fact is now traced.  
fact <-- 5  
fact <-- 4
```

```
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
fact --> 120
```

```
[21]: (* Écrire une fonction pow2 telle que pow2 n renvoie 2^n *)
      (* BEGIN SOLUTION *)
      let rec pow2 n = if n = 0 then 1 else 2 * pow2 (n-1)
      (* END SOLUTION *)
```

```
[21]: val pow2 : int -> int = <fun>
```

```
[23]: assert(pow2 0 = 1);
      assert(pow2 1 = 2);
      assert(pow2 4 = 16);
      assert(pow2 8 = 256)
```

```
[23]: - : unit = ()
```

```
[5]: (* Écrire une fonction somme_carres telle que somme_carres n renvoie
      la valeur de la somme 1^2 + 2^2 + ... + n^2 *)
      (* BEGIN SOLUTION *)
      let rec somme_carres n =
        if n = 0
        then 0
        else n*n + somme_carres (n-1)
      (* END SOLUTION *)
```

```
[5]: val somme_carres : int -> int = <fun>
```

```
[7]: assert(somme_carres 10 = 385);
      assert(somme_carres 0 = 0);
      assert(somme_carres 1 = 1)
```

```
[7]: - : unit = ()
```

On appelle suite de Fibonacci, la suite F_n définie par $F_0 = 0, F_1 = 1$ et pour n entier, $F_{n+2} = F_{n+1} + F_n$.

```
[2]: (* Écrire une fonction fibo telle que fibo n renvoie Fn *)
      (* BEGIN SOLUTION *)
      let rec fibo n =
        Printf.printf "%d\n" n;
        if n = 0 then 0
        else if n = 1 then 1
        else fibo (n-1) + fibo (n-2)
      (* END SOLUTION *)
```

```
[2]: val fibo : int -> int = <fun>
```

```
[11]: assert (fibo 8 = 21)
```

```
[11]: - : unit = ()
```

```
[4]: (* Ce calcul prend beaucoup de temps, pourquoi ? *)
      fibo 4; flush stdout
```

File "[4]", line 2, characters 0-6:

```
2 | fibo 4; flush stdout
   ~~~~~
```

Warning 10: this expression should have type unit.

```
4
2
0
1
3
1
2
0
1
```

File "[4]", line 2, characters 0-6:

```
2 | fibo 4; flush stdout
   ~~~~~
```

Warning 10: this expression should have type unit.

```
[4]: - : unit = ()
```

Expliquez pourquoi le calcul précédent prend du temps ici.

6 Tuples

Il est possible de considérer, comme en mathématiques, un produit cartésien de différents types. On parlera plutôt de tuple dans le contexte de l'informatique.

Pour construire un tuple il suffit d'écrire plusieurs valeurs ainsi :

```
[1]: (3, 2.2, "bonjour")
```

```
[1]: - : int * float * string = (3, 2.2, "bonjour")
```

On remarque que le type est alors de la forme `type1 * type2 * ... * typen`.

Si la création d'un tuple est facile, son utilisation nécessite de *casser* le tuple pour en récupérer les différentes composantes. Pour cela, on va utiliser une fonctionnalité du `let` : on peut écrire `let motif = valeur` où `motif` est lié à la *structure* de valeur.

Commençons par l'exemple le plus simple :

```
let x, y = (1, "a")
```

est équivalent à la définition :

```
let x = 1
let y = "a"
```

mis à part que ces deux définitions sont simultanées.

On peut tout à fait mettre des tuples dans des tuples.

```
[3]: (3, (2.2, "bonjour"))
```

```
[3]: - : int * (float * string) = (3, (2.2, "bonjour"))
```

Les parenthèses dans le type sont alors très importantes. Pour utiliser une telle valeur, on peut employer une syntaxe similaire à la précédente :

```
let a, (b, c) = (3, (2.2, "bonjour"))
```

Ici, il faut remarquer qu'il est possible de *déconstruire* ce tuple de plusieurs manières. On peut écrire comme précédemment pour avoir chaque composante, ou `let a, t = ...` pour que `t` contienne le couple `(2.2, "bonjour")`.

On peut se servir de cela pour réaliser une *sorte* d'échange de variables :

```
let x, y = y, x in ...
```

En fait, on définit de nouvelles variables, mais en pratique, cela correspond à l'échange usuel. On peut alors simplifier le code `mediane3` :

```
[3]: let mediane3 x y z =
      (* range x < y *)
      let x, y = if x > y then y, x else x, y in
```

```

if z < x then x
  else if z > y then y else z

```

[3]: val mediane3 : 'a -> 'a -> 'a -> 'a = <fun>

[8]: *(* Question difficile - revenez dessus plus tard si vous n'y arrivez pas tout
 ↳ de suite *)*
(Écrire une fonction mediane5 qui prend cinq éléments d'un même type, tous
 ↳ distincts*
et renvoie l'élément au milieu dans l'ordre.
Si on lui passe $x < y < z < t < u$ elle renvoie z .
*On pourra utiliser une fonction min sur le modèle de la fonction max *)*
(BEGIN SOLUTION *)*
 let min a b = if a < b then a else b
 let mediane5 a b c d e =
 let a, b = if a < b then a, b else b, a in
 let d, e = if d < e then d, e else e, d in
 let a, b, d, e = if a < d then a, b, d, e else d, e, a, b in
 if c > b
 then begin
 if b < d
 then min c d *(* car $a < b < d < e$ et $b < c$ donc soit $c < d < e$, $d < e <$
 ↳ c ou $d < c < e$ *)*
 else min b e
 end else begin
 if c > d then min c e else min b d
 end
(END SOLUTION *)*

[8]: val min : 'a -> 'a -> 'a = <fun>

[8]: val mediane5 : 'a -> 'a -> 'a -> 'a -> 'a -> 'a = <fun>

[6]: *(* Ici il y a 120 tests si on veut tout vérifier... *)*

```

assert(mediane5 1 2 3 4 5 = 3);assert(mediane5 1 2 3 5 4 = 3);assert(mediane5 1 2 4 3 5 = 3);assert(mediane5 1 2 4 5 3 = 3);assert(mediane5 1 2 5 3 4 = 3);
→assert(mediane5 1 2 5 4 3 = 3);assert(mediane5 1 3 2 4 5 = 3);
→assert(mediane5 1 3 2 5 4 = 3);assert(mediane5 1 3 4 2 5 = 3);
→assert(mediane5 1 3 4 5 2 = 3);assert(mediane5 1 3 5 2 4 = 3);
→assert(mediane5 1 3 5 4 2 = 3);assert(mediane5 1 4 2 3 5 = 3);
→assert(mediane5 1 4 2 5 3 = 3);assert(mediane5 1 4 3 2 5 = 3);
→assert(mediane5 1 4 3 5 2 = 3);assert(mediane5 1 4 5 2 3 = 3);
→assert(mediane5 1 4 5 3 2 = 3);assert(mediane5 1 5 2 3 4 = 3);
→assert(mediane5 1 5 2 4 3 = 3);assert(mediane5 1 5 3 2 4 = 3);
→assert(mediane5 1 5 3 4 2 = 3);assert(mediane5 1 5 4 2 3 = 3);
→assert(mediane5 1 5 4 3 2 = 3);assert(mediane5 2 1 3 4 5 = 3);
→assert(mediane5 2 1 3 5 4 = 3);assert(mediane5 2 1 4 3 5 = 3);
→assert(mediane5 2 1 4 5 3 = 3);assert(mediane5 2 1 5 3 4 = 3);
→assert(mediane5 2 1 5 4 3 = 3);assert(mediane5 2 3 1 4 5 = 3);
→assert(mediane5 2 3 1 5 4 = 3);assert(mediane5 2 3 4 1 5 = 3);
→assert(mediane5 2 3 4 5 1 = 3);assert(mediane5 2 3 5 1 4 = 3);
→assert(mediane5 2 3 5 4 1 = 3);assert(mediane5 2 4 1 3 5 = 3);
→assert(mediane5 2 4 1 5 3 = 3);assert(mediane5 2 4 3 1 5 = 3);
→assert(mediane5 2 4 3 5 1 = 3);assert(mediane5 2 4 5 1 3 = 3);
→assert(mediane5 2 4 5 3 1 = 3);assert(mediane5 2 5 1 3 4 = 3);
→assert(mediane5 2 5 1 4 3 = 3);assert(mediane5 2 5 3 1 4 = 3);
→assert(mediane5 2 5 3 4 1 = 3);assert(mediane5 2 5 4 1 3 = 3);
→assert(mediane5 2 5 4 3 1 = 3);assert(mediane5 3 1 2 4 5 = 3);
→assert(mediane5 3 1 2 5 4 = 3);assert(mediane5 3 1 4 2 5 = 3);
→assert(mediane5 3 1 4 5 2 = 3);assert(mediane5 3 1 5 2 4 = 3);
→assert(mediane5 3 1 5 4 2 = 3);assert(mediane5 3 2 1 4 5 = 3);
→assert(mediane5 3 2 1 5 4 = 3);assert(mediane5 3 2 4 1 5 = 3);
→assert(mediane5 3 2 4 5 1 = 3);assert(mediane5 3 2 5 1 4 = 3);
→assert(mediane5 3 2 5 4 1 = 3);assert(mediane5 3 4 1 2 5 = 3);
→assert(mediane5 3 4 1 5 2 = 3);assert(mediane5 3 4 2 1 5 = 3);
→assert(mediane5 3 4 2 5 1 = 3);assert(mediane5 3 4 5 1 2 = 3);
→assert(mediane5 3 4 5 2 1 = 3);assert(mediane5 3 5 1 2 4 = 3);
→assert(mediane5 3 5 1 4 2 = 3);assert(mediane5 3 5 2 1 4 = 3);
→assert(mediane5 3 5 2 4 1 = 3);assert(mediane5 3 5 4 1 2 = 3);
→assert(mediane5 3 5 4 2 1 = 3);assert(mediane5 4 1 2 3 5 = 3);
→assert(mediane5 4 1 2 5 3 = 3);assert(mediane5 4 1 3 2 5 = 3);
→assert(mediane5 4 1 3 5 2 = 3);assert(mediane5 4 1 5 2 3 = 3);
→assert(mediane5 4 1 5 3 2 = 3);assert(mediane5 4 2 1 3 5 = 3);
→assert(mediane5 4 2 1 5 3 = 3);assert(mediane5 4 2 3 1 5 = 3);
→assert(mediane5 4 2 3 5 1 = 3);assert(mediane5 4 2 5 1 3 = 3);

```

```
assert(mediane5 1 3 5 7 9 = 5); assert(mediane5 10 13 11 12 14 = 12)
```

```
[6]: - : unit = ()
```

A l'aide de couples, on peut calculer plus rapidement F_n . En effet, on pose $C_n = (F_n, F_{n+1})$ avec $C_0 = (0, 1)$.

```
[22]: (* Écrire une fonction fiboC telle que fiboC n renvoie le couple Cn *)
      (* BEGIN SOLUTION *)
      let rec fiboC n =
        if n = 0 then (0, 1)
        else let a, b = fiboC (n-1) in b, a+b
      (* END SOLUTION *)
```

```
[22]: val fiboC : int -> int * int = <fun>
```

```
[20]: assert (fiboC 8 = (21, 34))
```

```
[20]: - : unit = ()
```

```
[23]: (* Cet appel prend-il autant de temps qu'avec fibo ? Expliquez *)
      fiboC 40
```

```
[23]: - : int * int = (102334155, 165580141)
```

Répondez ici à la question ci-dessus.

7 Filtrage

Un des grandes forces du langage OCaml est de permettre de faire du filtrage de valeur. Cela consiste, étant donnée une valeur, à appliquer différentes expressions selon la *forme* de cette valeur.

Pour cette notion de *forme* on introduit la notion de motif qui sont des expressions sous forme évaluées présentant des variables ou des constantes. Par exemple $(0, x)$ mais pas $1+1$ (qui est 2 sous forme évaluée).

La syntaxe est alors

```
match valeur with
| motif1 -> expr1
| ...
| motifn -> exprn
```

Un premier exemple en reprenant fibo :

```

let rec fibo n =
  match n with
  | 0 -> 0
  | 1 -> 1
  | n -> fibo (n-1) + fibo (n-2)

```

Ici le dernier motif va définir une nouvelle variable `n` qui prendra la valeur de la précédente. Comme cela n'apporte rien, on remplace en général le dernier motif par le motif joker : `_ : | _ -> ....`

La force du filtrage c'est qu'on peut *déconstruire* des types plus complexes. Pour le moment, nous n'avons vu que les tuples. On peut ainsi écrire la fonction suivante qui renvoie `true` si un triplet contient la valeur 0 :

```

let f t = match t with
  | (0, _, _) -> true
  | (_, 0, _) -> true
  | (_, _, 0) -> true
  | _ -> false

```

Les motifs sont parcourus du haut vers le bas, ainsi la fonction suivante appliquée au couple `(x,y)` renvoie `y` si `x` est nulle ou `x` sinon.

```

let f c = match c with
  | (0, y) -> y
  | (x, _) -> x

```

Si on inverse, elle renverra toujours `x` car la seconde ligne ne sera jamais atteinte.

```

let f c = match c with
  | (x, _) -> x
  | (0, y) -> y

```

On peut avoir une structure riche dans les motifs, ainsi `| (x, (_, 0)) ->` va filtrer les couples dont la seconde composante est un couple de deuxième composante nulle en liant à la variable `x` la valeur de la première composante. Par exemple, cela filtrera `(1, (2,0))` en liant 1 à la variable `x`.

En fait, on a déjà vu cela en déconstruisant les tuples et on peut se dire que `let motif = expr in suite` est équivalent à `match expr with | motif -> suite`.

8 Suite de Syracuse

Étant donné $a \in \mathbb{N}^*$, on appelle suite de Syracuse de premier terme a , la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = a$ et pour $n \in \mathbb{N}$:

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Rappel : pour déterminer si un nombre a est pair on peut utiliser le modulo et tester `a mod 2 = 0`.

```

[6]: (* Écrire une fonction suivant telle que suivant u(n) renvoie u(n+1) selon la
      ↪ définition précédente *)
      (* BEGIN SOLUTION *)

```

```
let suivant u = if u mod 2 = 0 then u / 2 else 3 * u + 1
(* END SOLUTION *)
```

[6]: val suivant : int -> int = <fun>

```
[8]: assert (suivant 14 = 7);
      assert (suivant 7 = 22)
```

[8]: - : unit = ()

```
[9]: (* Écrire une fonction syracuse telle que syracuse a n renvoie le n-ième terme
      ↪ de la suite de syracuse
      de premier terme a *)
      (* BEGIN SOLUTION *)
      let rec syracuse a n =
        if n = 0
        then a
        else suivant (syracuse a (n-1))
      (* END SOLUTION *)
```

[9]: val syracuse : int -> int -> int = <fun>

```
[10]: assert (syracuse 3 5 = 4);
       assert (syracuse 1 0 = 1);
       assert (syracuse 1 1 = 4);
       assert (syracuse 1 2 = 2)
```

[10]: - : unit = ()

En évaluant les termes de la suite, on constate qu'elle semble devenir périodique à partir d'un certain rang. Par exemple, en partant de $a = 14$:

14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

Quelle que soit la valeur de u_0 , on retombe sur ce cycle. (Demandez une preuve à votre professeur de mathématiques à 12h05)

On appelle temps de vol pour a , le plus petit n tel que $u_n = 1$ et altitude maximale pour a la plus grande valeur prise par la suite. Ici, le temps de vol pour 14 est 17 et son altitude maximale est 52.

```
[12]: (* Écrire une fonction temps_de_vol telle que temps_de_vol a renvoie le temps
      ↪ de vol pour a.
      On pourra faire le lien entre le temps de vol de a et celui de suivant a *)
```

```

(* BEGIN SOLUTION *)
let rec temps_de_vol a =
  if a = 1 then 0 else 1 + temps_de_vol (suivant a)
(* END SOLUTION *)

```

[12]: val temps_de_vol : int -> int = <fun>

```

[13]: assert (temps_de_vol 1 = 0);
      assert (temps_de_vol 14 = 17)

```

[13]: - : unit = ()

```

[17]: (* Écrire une fonction altitude_maximale telle que altitude_maximale a renvoie
      ↪ l'altitude maximale de a *)
      (* On pourra faire le lien entre altitude_maximale (suivant a) et celle de a *)
      (* BEGIN SOLUTION *)
      let rec altitude_maximale a =
        if a = 1 then 4 else max a (altitude_maximale (suivant a))
      (* END SOLUTION *)

```

[17]: val altitude_maximale : int -> int = <fun>

```

[18]: assert (altitude_maximale 1 = 4);
      assert (altitude_maximale 2 = 4);
      assert (altitude_maximale 14 = 52)

```

[18]: - : unit = ()

```

[21]: (* Écrire une fonction plus_long_temps_de_vol tel que plus_long_temps_de_vol n
      ↪ renvoie
      le plus grand temps de vol parmi les entiers compris entre 1 et n (inclus) *)
      (* BEGIN SOLUTION *)
      let rec plus_long_temps_de_vol n =
        if n = 1 then 0 else max (temps_de_vol n) (plus_long_temps_de_vol (n-1))
      (* END SOLUTION *)

```

[21]: val plus_long_temps_de_vol : int -> int = <fun>

```

[29]: assert (plus_long_temps_de_vol 1 = 0);
      assert (plus_long_temps_de_vol 12 = 19);
      assert (plus_long_temps_de_vol 50 = 111)

```

```
[29]: - : unit = ()
```

9 Facultatif - Entiers de Church

Juste avec les fonctions et les applications, on peut définir des expressions qui se comportent comme les entiers naturels. Ce sont les **entiers de Church**.

On pose ainsi :

```
let zero = fun f -> fun x -> x
let un = fun f -> fun x -> f x
let deux = fun f -> fun x -> f (f x)
let trois = fun f -> fun x -> f (f (f x))
```

Le type de ces entiers est donc $('a \rightarrow 'a) \rightarrow 'a \rightarrow 'a$.

```
[7]: let int_of_church c =
      (* renvoie sous forme de int l'entier représenté par c *)
      (* BEGIN SOLUTION *)
      c (fun n -> n+1) 0
      (* END SOLUTION *)
```

```
[7]: val int_of_church : ((int -> int) -> int -> 'a) -> 'a = <fun>
```

```
[8]: let zero = fun f -> fun x -> x in
let un = fun f -> fun x -> f x in
let deux = fun f -> fun x -> f (f x) in
let trois = fun f -> fun x -> f (f (f x)) in

assert (int_of_church zero = 0);
assert (int_of_church un = 1);
assert (int_of_church deux = 2);
assert (int_of_church trois = 3)
```

```
[8]: - : unit = ()
```

```
[9]: let rec church_of_int n =
      (* renvoie l'entier de Church correspondant à l'int n *)
      (* BEGIN SOLUTION *)
      if n = 0 then fun f -> fun x -> x
      else fun f -> fun x -> f ((church_of_int (n-1)) f x)
      (* END SOLUTION *)
```

```
[9]: val church_of_int : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

```
[10]: for i = 0 to 10 do
      assert (int_of_church (church_of_int i) = i)
done
```

```
[10]: - : unit = ()
```

```
[16]: let successeur a =
      (* prend un entier de Church a et renvoie l'entier correspondant à son
      ↪successeur.
      Bien sûr on ne passera par les entiers ! *)
      (* BEGIN SOLUTION *)
      fun f -> fun x -> a f (f x)
      (* END SOLUTION *)
```

```
[16]: val successeur : (('a -> 'b) -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

```
[17]: for i = 0 to 10 do
      assert (int_of_church (successeur (church_of_int i)) = i+1)
done
```

```
[17]: - : unit = ()
```

```
[11]: let rec ajoute a b =
      (* Prend deux entiers de Church a et b et renvoie l'entier de Church
      correspondant à leur somme. Bien sûr on ne passera par les entiers ! *)
      (* BEGIN SOLUTION *)
      fun f -> fun x -> a f (b f x)
      (* END SOLUTION *)
```

```
[11]: val ajoute : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c = <fun>
```

```
[12]: assert (int_of_church (ajoute (church_of_int 4) (church_of_int 5)) = 9)
```

```
[12]: - : unit = ()
```

```
[13]: let rec multiplie a b =
      (* Prend deux entiers de Church a et b et renvoie l'entier de Church
      correspondant à leur produit. Bien sûr on ne passera par les entiers ! *)
      (* BEGIN SOLUTION *)
      fun f -> fun x -> a (fun x -> b f x) x
      (* END SOLUTION *)
```

```
[13]: val multiply :  
      (('a -> 'b) -> 'c -> 'd) -> ('e -> 'a -> 'b) -> 'e -> 'c -> 'd = <fun>
```

```
[15]: assert (int_of_church (multiply (church_of_int 4) (church_of_int 5)) = 20)
```

```
[15]: - : unit = ()
```