Introduction

OCaml est un langage de programmation développé dans les années 1980-1990, en France, par des chercheurs et chercheuses de l'INRIA [1]. Il est utilisé en recherche, dans des domaines comme la théorie de la preuve, ou l'étude des langages de programmation. Quelques applications connues d'OCaml :

— Rocq, un assistant de preuve permettant de formaliser dans un langage particulier des preuves mathématiques
— CompCert, un compilateur C dont la correction a été prouvée (grâce à Rocq)

Langage fonctionnel Python est un langage dit **impératif** car un programme Python est une suite d'instructions **exécutées** les unes à la suite des autres.
OCaml est un langage ***fonctionnel***. La programmation fonctionnelle est un autre paradigme de programmation, dans laquelle un programme est une immense ***expression*** mathématique que l'on **évalue**. Une telle expression peut faire intervenir des constantes (entiers, booléens, chaînes de caractères), des variables, mais aussi des fonctions.
En OCaml, on n'utilisera que très rarement des boucles (on va même se les interdire pendant toute la première partie du cours), on privilégiera l'utilisation de **fonctions récursives**.

1 Premiers programmes

A Arbre de syntaxe

Il ne faut pas confondre une expression OCaml et sa valeur. Par exemple, `3 + 5` et `5 + 3` sont deux expressions différentes, mais ont la même valeur. On peut donc voir l'interpréteur OCaml comme une fonction qui, à chaque expression, associe une valeur.

Définition 1

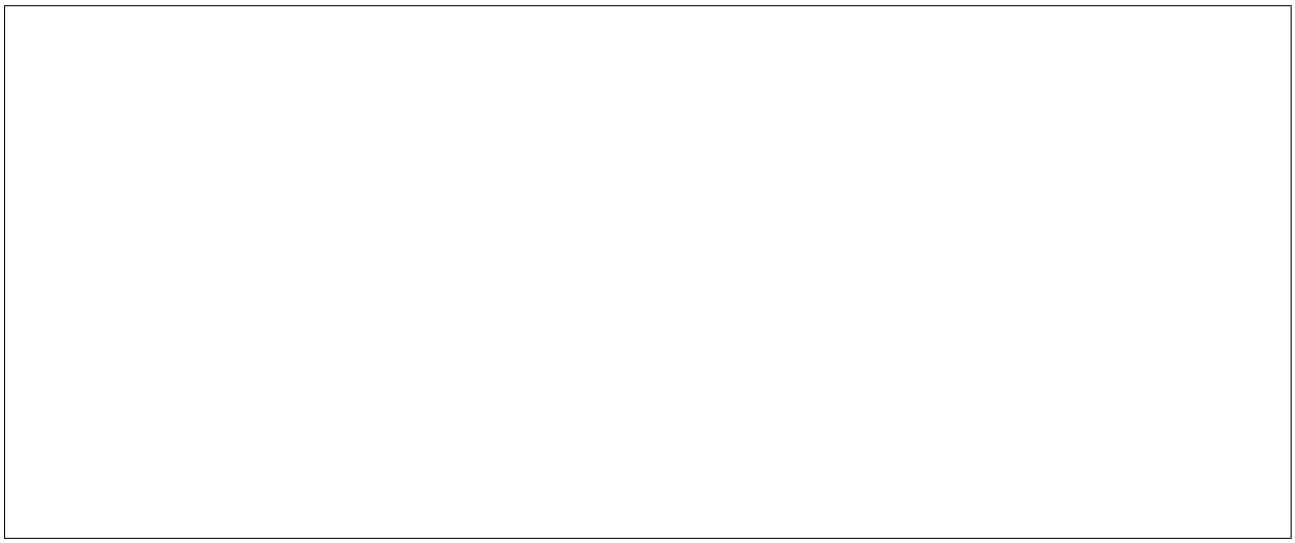
On définit l'ensemble $\mathcal{E}$ des expressions OCaml récursivement :

— Toute constante entière, flottante, booléenne, etc... est une expression valide ;
— Pour $e_1, e_2 \in \mathcal{E}$, on peut construire les expressions suivantes :
  — $e_1 + e_2$
  — $e_1 \times e_2$
  — $e_1 || e_2$
  — $e_1 < e_2$
  — $e_1 = e_2$
  — etc...
— Pour $e_1, e_2 \in \mathcal{E}$, on peut construire l'expression $e_1\ e_2$, lue "e_1 appliquée à e_2 ;
— Pour $e_1, e_2 \in \mathcal{E}$, on peut construire l'expression $\textbf{fun} x \to e_1$;

Cette définition n'est pas exhaustive : au fur et à mesure que l'on étudiera OCaml, on viendra étendre les possibilités pour construire des éléments de $\mathcal{E}$.

Un programme OCaml est donc une expression mathématique, que l'on peut représenter sous une forme graphique appelée "arbre de syntaxe". Voici quelques exemples d'expressions et leurs arbres de syntaxe :

1. Institut national de recherche en informatique et en automatique



(On note “Fun” les définitions de fonctions et “App” les applications de fonction.)

B Évaluation des expressions

Lorsque l'on exécute un programme Python, OCaml, ou dans n'importe quel autre langage de programmation, la première étape effectuée est de transformer le texte en un arbre de syntaxe. C'est cette représentation qui est ensuite utilisée pour analyser, évaluer, compiler le code.

Voyons, à la main, comment fait OCaml pour évaluer un arbre de syntaxe.

Pour `(1+7)*(7-3)` :

Prenons maintenant l'expression `(fun x -> x + 1)(5 + 3)`. Pour l'évaluer, OCaml commence par évaluer l'argument de la fonction : $5 + 3 = 8$. Ensuite, il doit évaluer le corps de la fonction, $x + 1$, en sachant que $x = 8$.

De manière générale, lorsque l'on plonge dans l'arbre de syntaxe pour en évaluer les différentes parties, il faut se rappeler des valeurs qu'ont les différentes variables. On appelle **contexte** l'ensemble des associations (variable $\mapsto$ valeur) enregistrées.

Exemple 1

Dessinons l'arbre de syntaxe de l'expression suivante, et tentons d'en trouver la valeur :

```
1 (fun x -> (fun y -> x + y) (x+1) - x) ((fun x -> x+1) 5)
```

C Fonctions d'ordre supérieur

Nous avons vu en TP qu'une fonction peut prendre en paramètre ou renvoyer une fonction, ou même les deux à la fois. Par exemple, considérons la fonction suivante :

```
1 fun f -> 2 * f 2
```

Si l'on tape l'expression précédente dans utop, l'interpréteur nous dit que le résultat est de type `(int -> int)-> int`. Cette fonction prend donc en paramètre une fonction, de type `int -> int`, et renvoie un entier. Par exemple :

```
1 (fun f -> 2 * f 2) (fun x-> x + 1) ;;
```

On a appliqué l'opération "Calculer $f(2)$ et doubler", en prenant f la fonction qui à x associe $x+1$, le résultat est donc logiquement $(2+1)\times 2 = 6$.

Définition 2

On dit qu'une fonction est ***d'ordre supérieur*** si elle prend en argument une fonction ou renvoie une fonction.

Exemples de vos autres cours :

- La dérivée : C'est la fonction $D : f \mapsto f'$.
- La transformée de Laplace, qui transforme une fonction dans le le domaine temporel en une fonction dans le domaine de Laplace.
- Le gradient, qui transforme un champ scalaire (i.e. une fonction de $\mathbb{R}^3$ dans $\mathbb{R}$)en un champ de vecteurs (i.e. une fonction de $\mathbb{R}^3$ dans $\mathbb{R}^3$).

Un exemple plus simple : Pour $k \in \mathbb{N}$, on peut considérer la fonction $f_k : x \mapsto x + k$. On a donc défini une famille $(f_k)_{k\in\mathbb{N}}$ des fonctions, et on peut alors considérer la fonction $G : k \mapsto f_k$: c'est une fonction d'ordre supérieur ! Par exemple, $G(2)$ est la fonction $f_2 : x \mapsto x + 2$, et donc $G(2)(5) = f_2(5) = 5 + 2 = 7$.
En OCaml, on pourrait définir G en écrivant :

```
1 fun k -> (fun x -> x+k);; (* G *)
2 (fun k -> (fun x -> x+k))(2)(5) (* G(2)(5) *)
```

En OCaml, lorsque l'on écrit `a b c`, c'est compris comme `(a b) c`. On peut donc juste écrire :

```
1 (fun k -> (fun x -> x+k)) 2 5
```

D Syntaxe "let in"

Il va vite être compliqué de construire des expressions lisibles. Afin de rendre le code plus digeste, on utilise en OCaml une syntaxe qui **ressemble** aux affectations de variable des langages impératifs. Cette syntaxe est construite avec les deux mots-clés `let` et `in`. Par exemple :

```
let x = 2 in
x + 3;;
```

Pour évaluer une expression de la forme `let x = e1 in e2`, on évalue `e1` en une valeur v_1 puis on évalue `e2` dans l'environnement (`x` $\mapsto v_1$).

Remarque 1

`let x = e1 in e2` a le même sens que `(fun x -> e2)e1;;`.

On peut imbriquer les ***let in*** comme on veut.

Q1. Taper les expressions suivantes :

```
let x = 3 in
let y = 5 in
x + y;;

let x =
  let t = 3.2 in
  t *. 5.0
in (x + 1.0);;
```

Dans les programmes OCaml et dans l'interpréteur, il existe un contexte global, auquel on peut ajouter des associations (variable $\mapsto$ valeur), de la manière suivante :

Q2. Taper le code suivant dans utop :

```
let plus_un = fun x -> (x + 1);;

let appliquer_deux_fois = fun f -> (fun x -> f (f x));;

let plus_deux = appliquer_deux_fois plus_un;;

let x = 3;;
let y = plus_un x;;
let z = plus_deux x;;
```

Remarque 2

let signifie "soit" en anglais, donc `let x = ... ;;` se dirait en français "soit x égal à ..." : cette syntaxe est donc inspirée du vocabulaire mathématique !

Pour les fonctions, il existe une deuxième simplification de la syntaxe, qui consiste à éliminer le mot clé `fun` et la flèche. Par exemple, les deux lignes suivantes sont **identiques** :

```
let f = fun x -> x * x ;;
let f x = x * x;;
```

La deuxième ligne sera généralement à privilégier, elle se lit de la même manière que l'énoncé mathématique (pas parfaitement formel) "Soit $f(x) = x^2$".

Regardons la fonction suivante :

```
let add x = fun y -> x + y;;
```

cette fonction est de type `int -> (int -> int)`. Autrement dit, elle prend en entrée un entier, et renvoie une fonction. Pour k entier, `add k` est la fonction qui ajoute k à son argument. Donc, on peut écrire :

```
let f = add 3;;
let x = f 5;;
```

ou même :

```
let x = add 3 5;; (*i.e. add(3)(5) *)
```

Avec cette écriture, on peut voir la fonction `add` comme prenant **deux** arguments x et y, et renvoyant $x + y$. En fait, on peut directement définir `add` avec la syntaxe suivante :

```
let add x y = x + y;;
```

Formellement, cela veut toujours dire que la fonction add prend un argument `x`, et renvoie une fonction qui prend elle même un argument `y`, et qui renvoie $x+y$. Informellement, on dira que `add` prend deux arguments, x et y. Cependant, c'est un ***abus*** de langage !

On peut alors appliquer **totalement** add, par exemple en écrivant `add 3 5` pour obtenir 8, ou bien l'appliquer **partiellement**, par exemple en écrivant `add 10` pour obtenir "la fonction qui ajoute 10 à son argument". Cette dualité de point de vue est un point essentiel de la programmation OCaml.

E Typage

Nous avons déjà mentionné qu'OCaml possède un système d'inférence de type, qui lui permet de déterminer le type des différentes expressions que l'on écrit, et de signaler les erreurs de typage.
Pour une fonction f définie par `let f x1 x2 ... xn = e`, OCaml donne un type à f, qui correspond aux types des paramètres et de la valeur calculée :

```
t1 -> t2 -> ... -> tn -> t
```

Ce type signifie que la fonction f prend en entrée un paramètre de type t1, un paramètre de type t2, ..., un paramètre de type tn, et renvoie une valeur de type t. Par exemple :

```
let ma_fonction f x y = f (x + y) + f(x - y);;
```

Cette ligne affiche :

```
val ma_fonction : (int -> int) -> int -> int -> int = <fun>
```

On peut lire cette ligne comme :

- `ma_fonction` prend 3 paramètres :
  1. Une fonction de type `int -> int`
  2. Une valeur de type `int`
  3. Une valeur de type `int`
- `ma_fonction` renvoie une valeur de type `int`

F Polymorphisme

Écrivons le code de la fonction ***identité***, qui renvoie directement son entrée :

```
let f x = x;;
```

Le type de cette fonction est `'a -> 'a`. Cette notation un peu particulière veut dire : "pour tout type α, on peut donner à la fonction le type $\alpha \to \alpha$".

Définition 3

On dit qu'une fonction est **polymorphe** si elle peut prendre en entrée et renvoyer plusieurs types.

On dit qu'OCaml possède du ***polymorphisme*** de types.
Un autre exemple, la composition de fonction :

```
let composee f g =
  fun x -> f (g x)
```

La fonction `composee` est de type `('a -> 'b)-> ('c -> 'a)-> ('c -> 'b)`, car elle prend deux fonctions $f : A \to B$ et $g : C \to A$, et renvoie la composée $f \circ g : C \to B$, où A, B, C peuvent être n'importe quels ensembles.

G Tuples

En OCaml, comme en Python, on peut manipuler des ***tuples***.

Q3. Taper les expressions suivantes et regarder leur type :

```
let p1 = (1, 2, 3) ;;
let p2 = (1, 'a', "toto", 4);;
let diago x = (x, x);;
```

Si $e_1, e_2, \dots, e_n$ sont des expressions de types respectifs `t1, t2, ..., tn`, alors le tuple $(e_1, e_2, \dots, e_n)$ est de type `t1 * t2 * ... * tn`. On dit que c'est un type ***produit***.

Pour utiliser un tuple, il est nécessaire de le ***déstructurer***. Cela signifie que l'on va extraire du tuple les différentes composantes. Par exemple :

```
let p1 = (1, 2, 3) ;;
let (x, y, z) = p1;;
```

On a ***déstructuré*** `p1` en utilisant un tuple de variables ayant la même ***structure*** : `(x, y, z)`. Lorsque l'on déstructure ainsi un tuple, ou n'importe quelle autre type (cf plus loin), on parle de ***let déstructurant***.

On peut également déstructurer des tuples directement dans les paramètres d'une fonction. Par exemple, les deux fonctions suivantes sont équivalentes :

```
let echange1 p =
  let (x, y) = p in (y, x) ;;
let echange2 (x, y) = (y, x) ;;
```

Attention, malgré les apparences, la fonction echange2 ne prend **qu'un seul** paramètre en entrée, mais ce paramètre est un couple. Il y a donc une différence fondamentale entre les deux fonctions suivantes :

```
let echange (x, y) = (y, x)

let faire_couple_inverse x y = (y, x)
```

En particulier, on pourrait écrire `faire_couple_inverse 2` pour appliquer partiellement la deuxième fonction, mais c'est impossible avec la première.

2 Éléments de base d'OCaml

A Typage et commentaires

Nous avons remarqué qu'OCaml détecte automatiquement le type des différentes expressions que l'on écrit, en particulier des fonctions. Néanmoins, la bonne pratique en OCaml est de préciser le type des fonctions que l'on écrit. Pour cela, on précise le type de chaque paramètre, ainsi que le type de retour. Par exemple :

```
let est_pair (x: int) : bool =
  x mod 2 = 0

let composee (f: 'a -> 'b) (g: 'c -> 'a) : 'c -> 'b =
  fun x -> f (g x)
```

On s'efforcera de **toujours** typer explicitement les fonctions que l'on définit, afin de rendre le code bien plus lisible. Dans ce même but, on veillera aussi toujours à fournir un **commentaire de documentation** pour chaque fonction, qui doit :

— Faire intervenir le nom de chaque paramètre de la fonction

— Décrire de manière formelle, presque mathématique, ce que renvoie la fonction selon les arguments qu'on lui donne. On appelle cela la **spécification** de la fonction.

Par exemple :

```
(** MAUVAIS COMMENTAIRE **)
(* Multiplie deux entiers *)
let mult a b = ...

(** BON COMMENTAIRE **)
(* Renvoie le produit a*b. a et b doivent être positifs *)
let mult (a: int) (b: int) : int = ...
```

B If-then else

OCaml dispose d'une construction ***if-then-else*** :

```
(* Renvoie le maximum entre x et y *)
let maximum (x: 'a) (y: 'a): 'a =
  if x < y then y
  else x

(* applique un ET logique sur a et b deux booléens*)
let logic_and (a: bool) (b: bool) : bool =
  if a then b else false
```

La syntaxe est la suivante : étant données E1, E2 et E3 des expressions, on peut construire l'expression E suivante :

```
if E1 then E2 else E3
```

Pour que cette expression être bien typée, l'expression E1 doit être du type `bool`, et les expressions E2 et E3 doivent être du même type T, et alors E est aussi de type T.
Enfin, pour évaluer `if E1 then E2 else E3` :

1. Évaluer E1 en une valeur v_1
2. Si v_1 est `true`, évaluer E2
3. Sinon, évaluer E3

C Matching

Les mots-clés ***match*** et ***with*** permettent de faire des disjonctions de cas plus puissantes qu'un simple if-else. Voyons quelques exemple :

```
(* Renvoie le nom de x si c'est un chiffre *)
let nom (x: int) : string =
  match x with
  | 0 -> "Zero"
  | 1 -> "Un"
  ...
  | 9 -> "Neuf"

(* renvoie true si x vaut 0, false sinon *)
let is_zero (x: int) : bool =
  match x with
  | 0 -> true (* si x vaut 0, alors true *)
  | y -> false (* si x vaut n'importe quoi d'autre, alors false *)

(* Première position d'un 0 du triplet t. -1 si t n'a aucun 0 *)
let zero_pos (t: float*float*float): int =
  match t with
  | (0, y, z) -> 0 (* si la 1ere composante vaut 0 *)
  | (x, 0, z) -> 1 (* si la 2eme composante vaut 0 *)
  | (x, y, 0) -> 2 (* si la 3eme composante vaut 0 *)
  | _ -> -1 (* Tous les autres cas *)

(* Renvoie le produit des composantes de p si aucune n'est nulle, sinon la somme *)
let prod_or_sum (p: int*int*int) : int =
  match p with
  | (0, y, z) -> y + z
  | (x, 0, z) -> x + z
  | (x, y, 0) -> x + y
  | _ -> x * y * z
```

La syntaxe précise de cette construction est :

```
match E with
| M1 -> E1
...
| Mn -> En
```

où E, E1, ... En sont des expressions, et où M1, ..., Mn est un **motif**.

Définition 4

Un ***motif*** est (définition temporaire) :

- Soit une constante
- soit une variable
- soit un tuple de motifs

Toutes les variables d'un motif doivent être distinctes.

Un motif représente un **squelette de valeur**. Par exemple, `x` est un motif qui veut dire "n'importe quelle valeur", et `(x, (y, 0), z, (t, u, 5))` est un motif signifiant "un quadruplet, dont :

1. le premier membre est n'importe quoi,
2. le deuxième est un couple dont la deuxième composante est nulle
3. le troisième est n'importe quoi
4. le quatrième est un triplet dont la troisième composante est 5

Ainsi, si l'on compare ce dernier motif avec `(3, (2, 0), 3, (7, 8, 5))`, le motif et la valeur correspondent, et alors `x` prend la valeur 3, `y` prend la valeur 2, etc...

Attention, les termes suivants ne sont pas des motifs :

```
| 1+x (* pas d'opérateur autorisé *)
|(x, x) (* pas de variable en double *)
```

Les règles de typage :

— E peut être de n'importe quel type, les motifs M1, ... Mn doivent avoir des types compatibles avec E ;

— Les E1, ..., En doivent être d'un même type T, et alors l'expression totale est de type T également ;

— Pour chaque couple Mk, Ek, il faut que les types des variables communes à Mk et Ek soient cohérents.

Exemple 2

On considère l'expression suivante :

```
match (1, (2, 1+2), (3+1, 5)) with
| (0, _, _) -> 0
| (x, (y, 0), z) -> (x - y)
| (x, (y, _), z) -> (x + y)
```

Pour l'évaluer :

— On forme l'arbre de syntaxe de l'expression entre le match et le with ;

— On évalue cet arbre de syntaxe, on obtient alors une valeur, $(1, (2, 3), 4)$, qui est aussi représentée sous la forme d'un arbre

— Chaque motif est alors comparé avec la valeur. Un motif est **également** un arbre, et il suffit alors de comparer l'arbre valeur avec l'arbre motif pour voir s'ils sont compatibles

— Lorsqu'un motif compatible est trouvé, on rajoute au contexte les variables du motif, et l'on évalue l'expression liée au motif.

La variable `_`, appelée "underscore", joue un rôle particulier : elle peut être présente plusieurs fois dans un motif, mais ne figurera pas dans le contexte. On appelle ce motif particulier un ***joker***, ou ***wildcard*** en anglais, il sert donc à ignorer ou à jeter à la poubelle des parties de la valeur matchée. Par exemple :

```
(* Calcule x*y *)
let mult x y = match x, y with
| 0, _ -> 0
| _, 0 -> 0
| _ -> x*y
```

Variables et motifs Un motif est un squelette, il décrit donc une **forme** mais ne contient pas de valeur. Par exemple :

```
let equal x y =
  match x with
  | y -> true
  | _ -> false
```

La fonction ci-dessus n'est pas correcte. En effet, le `y` dans le premier motif n'a aucun lien avec le `y` en paramètre. On pourrait remplacer le `y` du motif par n'importe quel identifiant, et même par un underscore :

```
let equal x y =
  match x with
  | _ -> true
  | _ -> false
```

Il est alors clair que cette fonction renvoie **toujours** true !

A retenir : on ne peut **pas** utiliser une variable préexistante dans un motif pour comparer la valeur du match à la valeur de cette variable.

Matching incomplet Lorsqu'un match with ne couvre pas tous les cas possibles du type concerné, on dit qu'il est incomplet, ou non-exhaustif. Par exemple :

```
let f x = match x with
| 0 -> 1
| 2 -> 3
```

Si l'on essaie d'évaluer cette expression, OCaml va raler, et dire que le matching n'est pas exhaustif. Il donnera même un exemple de valeur qui n'est pas matchée ! On doit ***toujours*** couvrir tous les cas dans un match with. Cependant, il se peut qu'un cas ne soit pas sensé arriver car on l'empêche dans le code. Dans ce cas, on peut utiliser la fonction `failwith`, qui affiche un message d'erreur et arrête le programme. Par exemple :

```
let parite x = x mod 2
let est_pair y = match parite y with
| 0 -> true
| 1 -> false
| _ -> failwith "Ne doit pas arriver"
```

Il ne faut **jamais** laisser un matching incomplet.

On peut aussi utiliser les motifs avec les let-in. On parle alors de ***let déstructurant***. On peut aussi utiliser le joker dans ce contexte :

```
let p = (2, "bla", true)
let a, b, c = p

let p = "important", "a jeter", "aussi a jeter"

let x, _, _ = p
```

D Fonctions récursives

En OCaml, on n'utilise pas (pour l'instant) de boucle for ou de boucle while. A la place, on utilise des ***fonctions récursives***, c'est à dire des fonctions qui s'appellent elles-même. La récursivité est au coeur de la programmation fonctionnelle.

En OCaml, il faudra trouver des formules permettant de définir tous les objets que l'on manipule ***récursivement***. Par exemple, écrivons une fonction qui calcule a^b pour $a, b \in \mathbb{N}$.

La définition de la puissance est :

$$\forall a \in \mathbb{N}, \forall b \in \mathbb{N}, a^b = \prod_{i=1}^{b} a$$

Cependant, on peut remarquer la chose suivante pour $a, b \in \mathbb{N}$:

— Si $b = 0$, $a^b = 1$

— Sinon, $a^b = a^{b-1} \times a$

Ces relations permettent de ***caractériser*** ou même de **définir** récursivement ce que signifie a^b. On veut donc écrire :

```
let puissance a b =
  if b = 0 then 1
  else a * puissance a (b-1)
```

mais cela n'est pas accepté. En effet, au moment où l'on écrit le corps de la fonction puissance, l'identifiant `puissance` ne fait pas partie de l'énvironnement. En OCaml, pour autoriser une fonction à utiliser son propre nom, i.e. pour préciser que l'on écrit une fonction récursive, on utilise le mot clé ***let rec*** :

```
let rec puissance a b =
  if b = 0 then 1
  else a * puissance a (b-1)
```

Le typage et la sémantique du let rec sont les mêmes que pour le ***let*** classique. Cependant, lorsque l'on évalue l'application d'une fonction récursive, la fonction elle-même sera dans le contexte.

Exercice 1

Q1. Écrire une fonction récursive qui, étant donnés $a, b \in \mathbb{N}$, calcule le quotient de la division euclidienne de a par b.

Q2. Écrire une fonction récursive qui, étant donnés $a, b \in \mathbb{N}$, calcule le reste de la division euclidienne de a par b.

Q3. Combiner le code des deux fonctions précédentes en une seule fonction :

```
(* renvoie le couple (quotient, reste) de la division euclidienne de a par b *)
let rec div (a: int) (b: int) : (int * int) = ...
```

E Listes

Un nouveau type : les listes. En OCaml, les listes sont définies récursivement, comme suit :

— La liste vide, notée `[]`, est une liste

— Si $E1$ est une expression de type `'a` et $E2$ une expression de type `'a list`, alors `E1::E2` est une liste, contenant $E1$ suivi des éléments de $E2$. On dit que $E1$ est la ***tête*** de liste, et que $E2$ est la ***queue*** de liste.

Exemple 3

Voyons des exemples de listes :

```
1  let l_vide = []
2  let l_simple = 3::6::8::[] (* comme 3 :: (6 :: (8 :: [])) *)
3
4  (* Renvoie une liste de taille n contenant
5     exclusivement des 1 *)
6  let rec ones n = match n with
7    | 0 -> []
8    | _ -> 1:: ones (n-1)
9
10 let cinq_uns = ones 5
```

Attention, la tête d'une liste est un élément, mais la queue d'une liste est une liste ! On ne peut pas agrandir une liste en mettant un élément à sa droite.

On remarque qu'OCaml affiche les listes sous la forme `[v1; v2; ... ; vn]`. On peut également utiliser cette syntaxe pour définir des listes :

```
1  let l_a = [1;2;3;4;5;6]
2  let l_b = 1::2::3::4::5::6::[]
```

La première version est plus simple à écrire mais est trompeuse : OCaml ne voit pas les listes comme des tableaux Python. On ne **pas** accéder à la i-ème case d'une liste avec une syntaxe $L[i]$: on n'a accès qu'à la tête de la liste.

Une liste doit contenir uniquement des expressions d'un même type T, et dans ce cas, la liste sera de type `T list`. Par exemple, `["blabla"; "bli"; "toto"]` est de type `string list`, mais `[1; 2.1; 0]` contient des entiers et des flottants, et n'est donc pas une expression bien typée.

Les listes forment un **type polymorphique**. Par exemple, la liste vide est typée `'a list` par OCaml. Cela signifie que c'est une liste de type `T list`, pour tout type T. Ainsi, lorsque l'on écrira des fonctions générales sur les listes, on pourra les appliquer sur des listes de n'importe quel type !

Les listes viennent également agrandir la définition des ***motifs*** :

— `[]` est un motif

— si `M` et `L` sont des motifs, alors `M::L` est un motif

Par exemple : `x::y::q` est un motif signifiant "deux éléments puis une liste". Ce motif sera compatible avec toutes les listes de taille au moins 2, et permettra de récupérer les deux premiers éléments, et la liste des éléments suivants. `(x, y)::q` est un motif signifiant "une liste de couples, d'au moins un élément". Il permettra de matcher par exemple `[("bla", 5); ("titi", 129)]`, et alors `x` vaudra `"bla"` et `y` vaudra `5`.

Utilisons ces motifs pour créer quelques fonctions :

```
(* Renvoie la somme des éléments de l *)
let rec somme_liste (l: int list) : int = match l with
  | [] -> 0
  | x::q -> x + somme_liste q

(* Renvoie true si la liste est de taille pair, false sinon *)
let rec taille_paire (l: 'a list): bool = match l with
  | [] -> true
  | x::[] -> false
  | x::y::q -> taille_paire q (* enlever deux éléments conserve la parité *)

(* Renvoie true si il existe un élément x dans l tel que f x = true *)
let rec il_existe (f: 'a -> bool) (l: 'a list) : bool =
  match l with
  | [] -> false
  | x :: q -> f x || il_existe f q
```

F Alias de type

En OCaml, on peut définir ses propres types. Par exemple, on peut redéfinir un type préexistant en lui donnant un autre nom :

```
type nombre = int
type vecteur3d = float * float * float

(* fonctions |N -> |B *)
type filtre_entier = int -> bool
```

Bien qu'OCaml devine tout seul le type des expressions que l'on écrit, il ne devine pas toujours par lui même les types définis ainsi :

```
let x = (0., 0., 0.) (* affiche float*float*float *)
let y = 5 (* affiche nombre plutôt que int *)
```

Cependant, on peut spécifier le type des variables avec " :" comme suit :

```
let x: int = 5

let f (x:int) : int = x + 1 (* prend en entrée un int et renvoie un int *)

let f: (int -> int) = fun x -> x - 1

let p: vecteur3d = (2., 3.2, -8.54)

let equal_5: filtre_entier = fun x -> x = 5

let produit_scalaire (a:vecteur3d) (b:vecteur3d) : float =
  let xa, ya, za = a in
  let xb, yb, zb = b in
  xa *. xb +. ya *. yb +. za *. zb


(* Ici, le type vecteur3d serait inféré automatiquement àcause du type
   de produit_scalaire *)
let norme_carree (a:vecteur3d) : float = produit_scalaire a a
```

G Type somme

Le type produit correspond au produit cartésien d'ensemble. Le type somme correspond plutôt à l'union disjointe.

Un type somme permet de représenter des catégories d'objets ayant plusieurs "cas". Un tel type est constitué de ***constructeurs***, qui sont les différents mots clés que l'on utilise pour construire des objets de ce type. Chaque constructeur correspond à un "cas" différent.

Exemple 4

On veut implémenter un type pour les fournitures scolaires. On veut pouvoir représenter :

— Les stylos BIC, qui peuvent être de couleurs différentes

— Les règles, qui peuvent être de tailles différentes et peuvent être centrées ou pas (i.e. 0 est au centre ou au bord)

— Les gommes

Créez un fichier "fourniture.ml". Vous pourrez l'exécuter dans l'interpréteur en tapant :

```
#use "fourniture.ml";;
```

On définit le type somme avec :

```
type fourniture =
  | Stylo of string (* couleur *)
  | Regle of int * bool (* taille en cm, centrée ou non *)
  | Gomme
```

Ce qui se lit : "Il y a trois types de fournitures : Les stylos, qui sont paramétrés par une chaîne de caractères, les règles, paramétrées par un entier et un booléen, et les gommes". Les commentaires précisent à quoi servent les paramètres.

On peut ensuite créer des éléments de ce type :

```
let x = Stylo "rouge"
let r1 = Regle (30, true)
let r2 = Regle (20, false)
let g = Gomme
```

Notons qu'OCaml peut automatiquement tester l'égalité entre deux objets d'un même type :

```
let x = Stylo "rouge"
let y = Stylo "rouge";;
x = y;; (* Vaut true *)
```

La manière principale de manipuler les types définis ainsi est le ***match with***. Par exemple, on suppose que le prix des fournitures est comme suit :

— Les gommes coûtent 1,50€ ;

— Les stylos bleus coûtent 1,20€, les autres coûtent 1€ ;

— Une règle de l centimètres coûte $1 + \frac{l}{15}$ euros.

Voici comment on implémenterait une fonction calculant le prix d'une fourniture en OCaml :

```
(* prix de fourn en euros *)
let prix (fourn: fourniture) : float =
  match fourn with
  | Gomme -> 1.5
  | Stylo "bleu" -> 1.2
  | Stylo _ -> 1.0
  | Regle(longueur, _) -> 1.0 +. float_of_int longueur /. 15.0
```

On peut représenter une trousse comme une liste de fournitures. Écrivons une fonction qui calcule le nombre de gommes dans une trousse :

```
type trousse = fourniture list;;

let rec nombre_gommes (t:trousse) : int =
  match t with
  | [] -> 0
  | Gomme::q -> 1 + nombre_gommes q
  | _::q -> nombre_gommes q
```

Exercice 2

On suppose qu'une trousse seule coûte 5€. Écrire une fonction calculant le prix total d'une trousse, en comptant tout ce qu'elle contient :

```
let rec prix_trousse (t: trousse) : float = ...
```

La syntaxe pour les type somme est donc ;

```
type nom_type =
     Constructeur1 of type1 (* ou Constructeur1 *)
   | Constructeur2 of type2 (* ou Constructeur2 *)
      ...
   | ConstructeurN of typeN (* ou ConstructeurN *)
```

On rajoute également à la définition des ***motifs*** :

— Si `Cons` est un constructeur d'un type somme, et `M` est un motif, alors `Cons M` est un motif.

Rien n'empêche un type de s'auto-référencer, on dit alors que c'est un type ***inductif***, ou récursif. Par exemple, si l'on veut créer un type représentant les expressions arithmétiques :

```
type expr =
  | Constante of int
  | Plus of expr * expr
  | Fois of expr * expr
```

Autrement dit : une expression est soit une constante entière, soit la somme de deux expressions soit le produit de deux expressions. L'expression $(1+2)$ s'écrira :

```
Plus (1, 2)
```

L'expression $(1+2)*(3+7*6)$ s'écrira :

```
let my_expr =
  Fois (
    Plus(Constante 1,Constante 2),
    Plus(
      Constante 3,
      Fois (Constante 7, Constante 6)
    )
  )
```

Pour manipuler des types inductifs, il faudra généralement utiliser des fonctions récursives. Sur l'exemple précédent, écrivons une fonction qui évalue une expression arithmétique :

```
let rec eval e = match e with
  | Constante n -> n
  | Plus (e1, e2) -> eval e1 + eval e2
  | Fois (e1, e2) -> eval e1 * eval e2
```

(Remarquez que cette fonction ressemble de près à l'algorithme d'évaluation des expressions OCaml décrit plus tôt dans le chapitre. En fait, à ce stade, vous pourriez créer un type pour les expressions OCaml, un type pour les environnement, et recoder OCaml en OCaml !)

Tentons de recoder le type des listes OCaml. On remarque que les listes sont polymorphes, autrement dit elles peuvent s'adapter à des types différents. On voudrait écrire :

```
type liste =
  | Vide (* liste vide *)
  | Cons of 'a * liste (* Constructeur: Cons (x, q) sera x suivi de q *);;
```

Cependant, OCaml affiche : `Error: The type variable 'a is unbound in this type declaration.`

Pour définir ce type, on doit le ***paramétrer*** :

```
type 'a liste =
  | Vide
  | Cons of 'a * ('a liste) ;;
```

La syntaxe générale des types paramétrés est :

```
type ('a1, 'a2, ..., 'an) nom_type =
  | Constructeur1 of type1
   ...
  | Constructeurk of typek
```

On peut ensuite manipuler ce type normalement, et OCaml gèrera le polymorphisme :

```
let ma_liste_1 = Cons (5, Cons(6, Vide)) ;; (* [5, 6] *)
let ma_liste_2 = Cons ("bla", Cons("blo", Cons ("bli", Vide))) ;; (* ["bla", "blo", "bli
    "] *)

let tete l = match l with
  | Vide -> failwith "liste vide"
  | Cons (x, q) -> x
```

Exercice 3

Écrire une fonction `vraie_liste: 'a liste -> 'a list` qui transforme une liste de notre type liste maison en une liste standard OCaml.