1 Introduction au langage OCaml

I Historique du langage

Le langage OCaml s'inscrit dans l'histoire des langages de programmation *fonctionnels*. Il trouve son origine dans le travail de Robert Milner, chercheur dans laboratoire d'informatique d'Édimbourg, en Écosse. Sa recherche porte notamment sur la création d'outils de preuve automatique. Pour cela, il va créer un nouveau langage de programmation appelé *ML* (pour *Meta-Language*), avec dans l'idée qu'il permette de garantir de construire des preuves correctes par le langage, alors que le langage alors utilisé pour construire des outils de preuve automatiques (Lisp, `https://lisp-lang.org`) laisse cette responsabilité au programmeur (qui se trompe souvent). Au fur et à mesure, ce langage s'est étendu pour devenir un langage de programmation à part entière. Il est "semi-confidentiel" : connu et respecté dans le milieu académique et de la recherche, utilisé dans des contextes spécifiques par des ingénieurs de haut niveau, influençant l'évolution du reste du domaine, mais peu utilisé dans le contexte général.

OCaml est né d'une divergence entre les chercheurs anglo-saxons et les chercheurs français : au Royaume-Uni et aux États-Unis, on a préféré continuer le développement de *Standard ML* (`https://smlfamily.github.io`), alors que la communauté française a construit le langage *CAML*, puis *OCaml*. CAML (*Categorical Abstract Machine Language*) a été conçu à partir de 1985 comme un langage de programmation fonctionnel, dont la syntaxe est proche d'une écriture mathématique en développant fortement la *récursivité* et le *filtrage par motif*, ou encore en le rendant fortement typé et en intégrant de l'inférence de type. CAML était aussi un langage multiparadigme, en permettant d'écrire en programmation impérative, où le langage décrit des séquences d'instructions atomiques qui modifie l'état du programme. Le langage CAML s'est ensuite incarné dans le langage OCaml à partir de 1996, qui ajoute un système de modules poussé et ajoute des éléments de programmation orientée objet (qui n'est pas au programme). Développé notamment par l'Inria (Institut National de Recherche en Informatique et en Automatique, `https://www.inria.fr`), OCaml s'est diffusé dans l'industrie dans des contextes où la correction et la sécurité des programmes est essentielle :

— dans la finance notamment dans l'entreprise *Jane Street* (`https://www.janestreet.com`, qui participe beaucoup au développement d'OCaml et de bibliothèques alternatives à la bibliothèque standard)
— dans le développement de systèmes critiques chez Airbus, Dassault, etc.
— dans des contextes généralistes, où OCaml est utilisé seul ou en tant que sous-programme : par exemple, Reason (`https://reasonml.github.io/`) développé par Facebook, est une réécriture de la syntaxe d'OCaml adaptée aux programmeurs Javascript et transpilable en Javascript. Ils l'utilisent notamment dans l'implémentation de Messenger.

D'autres exemples plus récents :

— un système d'exploitation pour l'informatique embarquée, *MirageOS* (`https://mirage.io`), qui permet de construire des systèmes légers et sécurisés pour l'informatique embarquée et le développement de services *Cloud* virtualisés. Une application récente : *SpaceOS*, un système d'exploitation pour satellite développé par *Tarides* (`https://tarides.com`) spécialisé dans l'informatique en périphérie de réseau, c'est-à-dire le traitement des masses de données générées par ces satellites sur place.
— La blockchain décentralisée Tezos est développée en OCaml pour sécuriser et mettre à jour son protocole.

Les langages ML, et OCaml en particulier, sont aussi source d'inspiration dans le développement des nouvelles méthodes et langages de programmation :

— Rust (`http://rust-lang.org`) est un langage qui s'inspire de langages fonctionnels tout en gardant une gestion mémoire manuelle (comme en C), des structures mutables par défaut, et un paradigme de programmation d'abord impératif,

— F# est un dérivé du langage OCaml, avec une syntaxe très similaire à OCaml, mais conçu spécifiquement par Microsoft pour fonctionner sous Windows.
— Dans les langages de programmation installés, les nouvelles fonctionnalités des langages de programmation empruntent souvent des concepts de programmation fonctionnelle en général, et d'OCaml en particulier : par exemple, le filtrage par motif en Python (`https://peps.python.org/pep-0635/`), les monades en Python ou Ruby, etc.

Les autres exemples de langages de programmation fonctionnels importants sont Scala, un langage de programmation plutôt impératif utilisant surtout les concepts de programmation fonctionnels pour rendre sa syntaxe plus élégante, et Haskell qui prend le parti d'être un langage de programmation "purement" fonctionnel, utilisant notamment l'évaluation paresseuse et implémentant des listes en compréhension.

OCaml vient de sortir sa version 5.0.0. Au programme : la prise en charge de l'exécution parallèle en mémoire partagée via les *domains* et un nouveau modèle d'exécution concurrente via les *effect handlers*.

I.A Présentation du langage

Le langage OCaml présente plusieurs caractéristiques à connaître :

— OCaml est un *langage fonctionnel* : le concept clé d'un tel langage est la fonction, dans le sens des fonctions mathématiques. Une fonction est un objet conceptuel qui associe à une entrée (ses *arguments*) une sortie. En OCaml, les fonctions sont des objets au même titre que les entiers, les flottants, les booléens, etc. Les langages fonctionnels privilégient aussi la récursion au lieu de l'itération et les variables non mutables plutôt que les variables mutables. Ainsi, les fonctions écrites en OCaml sont plus proches d'une définition mathématique qu'une séquence d'instructions à la machine, comme dans le langage C.
— Par défaut, les objets manipulés en OCaml sont *immuables* : il n'est pas possible de modifier la valeur d'une variable après sa création. Dans ce cadre immuable, l'intérêt principal est qu'une fonction appliquée au même argument renvoie le même résultat. Cependant, c'est un langage fonctionnel *impur* : il est possible de définir des variables mutables à l'aide d'un mot-clé spécifique.
— Il est *statiquement typé* : une variable possède un type qui est déterminé à la compilation. Ce n'est pas le cas en Python, par exemple, où le type d'une variable n'est détecté qu'à l'exécution. Il est aussi *fortement typé*, dans le sens où toutes les erreurs de type empêchent la compilation du programme, alors qu'en Python ces erreurs sont détectées au moment de l'exécution (et donc potentiellement après des heures de calcul qui seront perdues), et qu'en C ces erreurs ne sont pas détectées (et le résultat sera imprévisible). Enfin, OCaml intègre de l'*inférence de type* : le compilateur "devine" le type des objets manipulés dans le langage sans avoir besoin de le préciser explicitement dans le code source. Si besoin, OCaml attribue un *type paramétré* aux objets si plusieurs types sont possibles.
— Il est *strict* (en opposition à *paresseux*) : les arguments d'une fonction sont calculés *avant* l'appel à la fonction, et pas calculés au moment d'être utilisés. Haskell est l'exemple le plus connu de langage *paresseux*.
— Il possède deux modes d'exécution : *compilé*, où le fichier de code source est traduit en langage machine dans un fichier exécutable (comme C/C++), par la commande `ocamlopt` pour obtenir un fichier en langage machine ou `ocamlc` pour obtenir un fichier dans un format intermédiaire entre le code source et le code compilé, qui a besoin d'un programme intermédiaire pour être exécuté (comme Java avec JVM ou Python pour ses modules). L'alternative à un langage compilé est un langage *interprété* : c'est le cas de Python, du langage shell, ou encore de SQL. OCaml possède un mode interprété, accessible par la commande `ocaml` ou `utop`, qui permet d'exécuter des programmes sans avoir besoin de le compiler. On utilisera les deux modes cette année. Attention, le mode interprété se comporte différemment par rapport à celui de Python : par exemple, redéfinir une fonction ne redéfinit pas les fonctions l'utilisant comme fonction auxiliaire.
— Il est aussi *orienté objet* (c'est le O d'OCaml), mais on ne l'étudiera pas du tout (et ce n'est pas beaucoup utilisé en pratique).

Compétences attendues en fin de chapitre

- Connaître les caractéristiques principales du langage et être capable de les expliquer.
- Connaître la syntaxe présentée dans ce chapitre, savoir l'appliquer, à l'aide de la machine comme à l'écrit.
- *Après le* **??** *: connaître les complexités des opérations usuelles en OCaml, notamment identifier les opérations élémentaires, identifier les opérations non élémentaires et connaître leurs complexités.*

Il manque une partie de la syntaxe du langage, qui sera vue plus tard (Chapitre 2) : notamment, les tableaux, les valeurs modifiables, les boucles `for` et `while`. Tant qu'on ne les aura pas vus, il est évidemment interdit de les utiliser.

II Types de base, variables, fonctions, listes

II.A Types de base et opérateurs associés

On a vu que les *types* sont un concept fondamental en OCaml. Le type OCaml les types usuels en informatique : `int` qui correspond aux entiers (signés : qui peuvent être négatifs), `float` pour les nombres flottants, `bool` pour les booléens `true` et `false`, `char` pour les caractères et `string` pour les chaînes de caractères. Sur ces types, on dispose des opérateurs usuels. Pour les entiers, on les écrit simplement dans leur écriture décimale : `40`, `-5`[i]. L'ordre de priorité des opérateurs est défini comme d'habitude : `/ *` sont à la même priorité (et donc évalués de gauche à droite) et sont prioritaires sur `+ -`.

OCaml
```
# 25 + 12;;
- : int = 37
# 33 - 25;;
- : int = 8
# 2 * 52;;
- : int = 104
# 10 / 3;;
- : int = 3
# 10 mod 3;;
- : int = 1
# 5 + 3 * 2 / 6;;
- : int = 6
# (5 + 3 * 2) / 6;;
- : int = 1
```

Remarque 1 – Sur le code interprété dans le REPL

Première remarque : pour interpréter une expression OCaml dans le REPL (que ce soit `ocaml` ou `utop`), vous avez besoin d'ajouter `;;` à la fin. Ce n'est pas nécessaire quand on interprète le code dans une cellule Jupyter.

Remarque 2 – Sur la syntaxe du code présenté

Le code est présenté avec des caractères particuliers : # représente le début de l'invite de commande du REPL, qui attend que l'on écrive le code à interpréter. La ligne suivant le code commence par `- : int = 37` : le - représente le fait qu'on n'a pas donné de nom à la valeur, `int` est le type de la valeur et `37` est la valeur. Dans le REPL d'OCaml, il est aussi possible d'utiliser des directives n'appartenant pas au langage : par exemple, la directive `#use <nom_fichier>;;`, qui permet de charger

i. Il est aussi possible de les écrire en hexadécimal `0x10`, en binaire `0b010010` ou encore en octal `0o1075231`. On peut également utiliser des _ pour clarifier un grand nombre, par exemple en écrivant `1_000_000` au lieu de `1000000`.

et d'exécuter un fichier dans le REPL. Une liste complète de ces commandes est disponible avec la commande `#help;;`. Enfin, dans mes documents PDF, je présenterais des codes OCaml contenant le caractère ↪ désignant un retour à la ligne.
De même, il m'arrivera de présenter des commandes lancées dans un terminal, qui commenceront par $.

Attention !

Attention, les opérateurs mentionnés n'acceptent que des arguments `int` :

OCaml
```
# 1.0 * 25;;
Error: This expression has type float but an expression was expected of type
  ↪  int
```

Contrairement à Python ou C, les fonctions en OCaml ne sont donc pas *surchargées*. Cependant, comme vu plus tard, leur type peut être *paramétré*.

Ainsi, pour les flottants, les opérateurs usuels sont écrits différemment pour bien marquer leur différence :

OCaml
```
# 10.0 *. 3.15;;
- : float = 31.5
# 31.5 /. 10.0;;
- : float = 3.15
# 31.5 -. 10.0;;
- : float = 21.5
# 31.5 +. 10.0;;
- : float = 41.5
```

Pour les booléens, les opérateurs sont usuellement définis : `true && false` pour le "et" logique, `false || false` pour le "ou" logique [ii], `not true` pour la négation. Les opérateurs de comparaison ont l'air surchargés, mais en fait sont polymorphes et peuvent être utilisés non seulement sur des objets de type de base, mais aussi sur des types "structurés". Par contre, ils lèvent une exception quand l'un des arguments est une fonction (qui sont donc *non comparables*).

OCaml
```
# 1 < 2;;
- : bool = true
# 1.25 >= 2.36;;
- : bool = false
# (-1, 2) <= (-1, 3);; (* par l'ordre lexicographique *)
- : bool = true
# 4 <> 4;; (* Attention ! != n'a pas le même sens en OCaml *)
- : bool = false
# 4 = 4;; (* Attention ici aussi ! == n'a pas le même sens en OCaml *)
- : bool = true
```

Les booléens permettent de définir des expressions conditionnelles : elles sont de la forme `if` <cond> `then` <expr_vrai> `else` <expr_false>. Si la condition est évaluée à `true`, alors l'expression entière est évaluée à <expr_vrai>, et si elle est évaluée à `false`, l'expression entière est évaluée à <expr_faux>. Pour que cette expression ait un type, il faut donc que les deux expressions aient le même type :

ii. Comme dans le Chapitre 7, on entend "ou" dans un sens non exclusif : `vrai` ∨ `vrai` = `vrai`.

```OCaml
# let a = 0 and b = 0;;
val a : int = 0
val b : int = 0
# if a = b then 1.0 else 2.0;;
- : float = 1.
# if a = b then 1.0 else 25;;
Error: This expression has type int but an expression was expected of type
         float
  Hint: Did you mean `25.'?
```

Remarque 3 – Commentaires en OCaml

Les commentaires en OCaml sont délimités par `(* ... *)`, et peuvent délimiter un commentaire sur plusieurs lignes. On peut aussi utiliser `(** ... *)` : ces commentaires sont traités spécialement par le compilateur comme des commentaires de *documentation* : placés juste avant ou juste après une définition ou un argument, elles permettent aux outils de documentation d'OCaml d'en fournir la documentation.

On verra les opérations sur les chaînes de caractères en TP.

II.B Variables

La syntaxe générale pour définir une variable ou une fonction est de la forme `let <variable> = <valeur>`. Les variables (y compris les fonctions que l'on verra après) définies par cette syntaxe sont globales. Il est aussi possible de définir simultanément plusieurs variables (et fonctions) à l'aide de la syntaxe `let <variable> = <valeur> and <variableb> = <valeurb>`, ce qui sera notamment utile pour définir des fonctions simultanément récursives (qui appellent l'une dans l'autre et inversement).

```OCaml
# let x = 0;;
val x : int = 0
# x + 5;;
- : int = 5
# let y = 3.5 and z = 2.1;;
val y : float = 3.5
val z : float = 2.1
# y ** z;;
- : float = 13.8849043754660464
```

Remarquez qu'au lieu d'un `- :`, le REPL affiche `val <variable> : <type> = <valeur>`. Vérifier que le nom de la variable, son type et sa valeur sont celles attendues est une bonne pratique quand on utilise le REPL (quand on écrit un script avec Codium, le type est calculé automatiquement pendant la rédaction du code source).

Pour les variables locales, la syntaxe est `let <var_locale> = <valeur> in <expr>` : la variable locale est disponible jusqu'à la fin de l'expression. Si une variable avec une portée supérieure (déjà définie au moment du `let`, que ce soit une variable locale ou globale) a déjà le même nom, la variable locale masque la variable globale jusqu'à la fin de l'expression après le `in`. Attention, la variable locale peut avoir un type différent de la variable globale.

```OCaml
# let a = 25 in a*a;;
- : int = 625
# a + 2;; (* la variable a n'est plus accessible à cause du in *)
Error: Unbound value a
# let b = 12;;
val b : int = 12
```

```
# let b = 12.0 in b -. 2.0;; (* à l'intérieur du in, c'est le b = 12.0 qui est
  ↪ reconnu et pas la variable globale qui est masquée *)
- : float = 10.
# b;; (* par contre ici, c'est la variable globale qui n'est plus masquée *)
- : int = 12
# let c = b + 2;; (* la valeur de c est calculée ici, et ne changera plus... *)
val c : int = 14
# let b = 25.2;; (* ... *)
val b : float = 25.2
# c;; (* ... même si l'on redéfinit c entre temps *)
- : int = 14
```

Par défaut, une variable en OCaml est *immuable* : sa valeur ne change jamais après son initialisation. C'est comme dans une démonstration mathématique : on écrit "soit $x = \cdots$", et dans la suite de la preuve, la valeur de x ne change pas (sauf si on redéfinit une nouvelle variable du même nom x, mais c'est moche et à éviter).

II.C Définition de fonctions

La première syntaxe pour définir une fonction est `let <nom_fonction> <arg> = <expr>`. Par exemple :

OCaml
```
# let incremente x = x + 1;;
val incremente : int -> int = <fun>
```

Ici, la deuxième ligne est intéressante : ici, la fonction `incremente` est une variable comme une autre, donc la ligne commence par `val incremente :`. Ensuite, on voit que le type d'`incremente` est `int -> int`, c'est-à-dire une fonction qui prend un argument de type `int` et renvoie un `int`. On remarque donc que grâce au type de la valeur `1` et de la fonction `+`, OCaml a *inféré* le type que doit avoir l'argument `x`.

Pour appeler la fonction, la syntaxe est différente de la plupart des langages : il n'y a pas besoin de parenthèses autour des arguments, c'est même une erreur quand on a plus d'un argument. Attention aussi à l'ordre de priorité des opérations : l'appel à la fonction est prioritaire sur les autres opérations. Si l'on veut forcer l'ordre des opérations, comme pour les opérateurs, on utilise des parenthèses.

OCaml
```
# incremente 12;;
- : int = 13
# incremente 2 * 5;;
- : int = 15
# incremente (2 * 5);;
- : int = 11
```

L'étape suivante serait d'écrire des fonctions à plusieurs arguments, mais...**il n'y en a pas en OCaml**. Observez :

OCaml
```
# let ajoute_double x y = x + 2*y;;
val ajoute_double : int -> int -> int = <fun>
```

La fonction `ajoute_double` a pour type `int -> int -> int`. Le connecteur `->` est associatif à droite : ainsi, il faut lire le type fonctionnel ci-dessus comme `int -> (int -> int)`, c'est-à-dire une fonction à un argument `int` qui renvoie une fonction à un argument `int`, qui elle renvoie un `int`. Ainsi, si l'on n'écrit pas une fonction avec tous ces arguments, cela renvoie la fonction partielle.

OCaml
```
# ajoute_double 2 3;;
- : int = 8
# ajoute_double 2;;
```

```
- : int -> int = <fun>
# let f = (ajoute_double 2) in f 3;;
- : int = 8
(* Attention au type des arguments : puisque l'application de fonction
 * est prioritaire sur le reste, des parenthèses sont nécessaires pour
 * composer des fonctions. *)
# ajoute_double incremente 2 3;;
Error: This function has type int -> int -> int
       It is applied to too many arguments; maybe you forgot a `;'.
# ajoute_double (incremente 2) 3;;
- : int = 9
```

Ce processus, qui prend une fonction à plusieurs variables et l'interprète comme une fonction à un argument renvoyant une fonction sur le reste des arguments, est appelé la *curryfication*, en référence à Haskell Curry [iii].

Ici, `ajoute_double 2` est la fonction qui, appliquée sur `y`, renvoie `ajoute_double 2 y`. Les opérateurs vus en Section II.A sont aussi des fonctions : pour les utiliser avec la syntaxe des fonctions, on les met entre parenthèses :

OCaml
```
# ( + );;
- : int -> int -> int = <fun>
```

On peut définir des variables locales à l'intérieur de fonctions (c'est même tout l'intérêt des variables locales).

OCaml
```
# let f x y z =
    let a = ajoute_double x y
    and b = incremente z in
    ajoute_double (incremente a) (ajoute_double a b);;
val f : int -> int -> int -> int = <fun>
```

Le principal intérêt des variables locales est de ne pas calculer plusieurs fois la même valeur : ici, on a besoin deux fois de la variable locale `a` pour calculer le résultat de `f x y z`, mais on ne le calcule une seule fois au moment du calcul de la valeur de `a`. Au lieu d'utiliser un **and**, il est plus idiomatique d'enchaîner les définitions locales :

OCaml
```
let f x y z =
  let a = ajoute_double x y in
  let b = incremente z in
  ajoute_double (incremente a) (ajoute_double a b)
```

Remarque 4

L'indentation en OCaml est facultative : n'importe quel retour à la ligne, tabulation ou espace multiple peut être remplacé par un espace.

La syntaxe vue pour l'instant interdit les fonctions *récursives*, c'est-à-dire qui s'appellent elles-mêmes. Pour l'autoriser, il faut utiliser la syntaxe `let rec <fonction> = <expr>`. Appliquons-la sur la fonction factorielle, définie par récurrence sur $\mathbb{N}$ par :

— Si $n = 0$, alors $n! = 1$,
— Sinon, $n! = n(n-1)!$.

En OCaml, écrire cette fonction est très proche de la définition mathématique :

iii. Le langage Haskell fait référence à son prénom.

```OCaml
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)
```

Exercice 1

Pourquoi a-t-on besoin des parenthèses dans la fonction `fact` ?

II.D Premiers types structurés

En OCaml, on peut construire des types structurés (par oppposition aux types *de base* définis en Section II.A), dont les premiers sont prédéfinis en OCaml.

Listes D'abord, on va définir le type `list` : la liste vide est notée `[]`. Ensuite, il y a deux syntaxes différentes : on peut créer une nouvelle liste en ajoutant un élément en *tête* d'une autre liste par la syntaxe `::`, ou on peut la créer en la décrivant entièrement. Attention : tous les éléments de la liste doivent avoir le même type. Formellement :

Définition 5 – Type `'a list` en OCaml

Les listes d'éléments de type `'a` sont définies par induction :
- `[]` est une liste d'éléments de type `'a`, dite la *liste vide*
- pour tout élément `x` de type `'a` et toute liste `l` d'éléments de type `'a`, `x :: l` est une liste d'éléments de type `'a`, dite la *liste composée* de `x` et de `l`.

```OCaml
# let l = [];;
val l : 'a list = []
# let l1 = 1 :: l;;
val l1 : int list = [1]
# let l2 = 2 :: l;; (* notez bien que le calcul de l1 n'a pas modifié l,
  ↪ contrairement à un append en Python *)
val l2 : int list = [2]
```

Il existe une syntaxe alternative pour définir rapidement une liste : `[e1; e2; ...; en]` est équivalent à `e1 :: e2 :: ... :: en :: []`. Par exemple :

```OCaml
# let m = [3.2; 5.1; 0.0];;
val m : float list = [3.2; 5.1; 0.]
```

Cette syntaxe est plus lisible et on l'utilise quand on définit une liste de taille connue à l'avance. Attention, elle est aussi déceptive : contrairement aux listes pythons (qui ne sont pas des listes chaînées), on ne peut pas accéder au $k$ème élément d'une liste en temps constant. On est obligé de la parcourir jusqu'à l'élément voulu, ce qui prend un temps proportionnel à k.

À retenir sur les listes

- Il faut faire attention à l'ordre des opérandes de l'opérateur `(::)` : à gauche, on met la tête de la liste qui est de type `'a`, à droite on met la queue de la liste qui est de type `'a list` (éventuellement vide). Par exemple, on ne peut pas concaténer deux listes avec `l1 :: l2`, il faut utiliser l'opérateur `@` : `l1 @ l2`.
- Les listes sont immuables (on ne peut pas les modifier, seulement en créer des nouvelles).
- Les listes ne peuvent contenir qu'un seul type d'éléments : contrairement aux `list` de Python, on a forcément des `int list`, `float list`, `(bool * int list) list` ...Mais `[1;1.2;false]` est illégal.

— **Le type `'a list` n'a rien à voir avec le type `list` en Python : les listes en Python sont en fait des *tableaux dynamiques*, que nous allons voir un peu plus tard (il y a aussi d'autres différences, comme l'inconsistance des types).**

Les fonctions (dans le sens mathématique) définies sur des listes sont souvent définies par induction, avec une valeur par défaut quand la liste est vide (cas de base), et quand la liste est constituée d'une tête et d'une queue de la forme `tete :: queue`. OCaml dispose d'une syntaxe très proche, utilisant le *filtrage par motif* :

OCaml
```
let rec somme l = match l with
  | [] -> 0 (* dans le cas où la liste est vide, on renvoie 0 *)
  | tete :: queue -> tete + somme queue (* dans le cas où elle n'est pas vide, on
   ↪ déconstruit la liste en sa tête et sa queue *)
  ;;
val somme : int list -> int = <fun>
```

Dans cette fonction, le rôle du `match` est de déconstruire la liste `l` :
— soit elle est vide (et on renvoie `0`),
— soit on la décompose en sa tête `tete` et sa queue `queue`. On peut ensuite utiliser ces deux variables pour calculer la somme de la liste.

On peut aussi continuer la déconstruction de la liste et rajouter des cas : par exemple, pour calculer le maximum d'une liste.

OCaml
```
let rec max_liste l = match l with
  | [] -> failwith "une liste vide n'a pas de maximum"
  | [x] -> x
  | tete :: queue -> max tete (max_liste queue)
```

Ici, on a trois cas :
— Si la liste est vide, on lève une exception (on verra ça plus tard) car une liste vide n'a pas de maximum.
— Si la liste n'est constituée que d'un seul élément `x`, on le renvoie. *Un motif équivalent serait* `x :: []`.
— Sinon, la liste est composée d'une tête et d'une queue, et le maximum de la liste est le maximum entre la tête et le maximum de la queue.

Remarque 6

Quand une liste ne contient qu'un élément, elle respecte à la fois le second et le troisième motif. OCaml teste si la valeur dans le `match` correspond à chaque motif *dans l'ordre* : en l'occurence, cette fonction renvoie `x` sur l'entrée `[x]`. De plus, on sait que si l'on atteint le troisième motif, la liste contient au moins deux éléments (et `max_liste queue` ne lèvera pas d'exception).

Dernier exemple, pour vérifier qu'une liste est triée :

OCaml
```
let rec liste_est_triee l = match l with
  | [] | [_] -> true
  | tete :: cou :: queue -> tete <= cou && liste_est_triee (cou :: queue)
```

Plusieurs remarques sur ce filtrage :

Remarque 7 – Filtre _

Dans la fonction `est_triee`, la syntaxe `[_]` est équivalente à `[x]` : dans ce contexte, le tiret bas évite de nommer une variable sans l'utiliser. On peut aussi s'en servir comme dernier cas d'un filtrage avec `| _ -> <exn>`, qui est un motif acceptant n'importe quelle valeur.

Remarque 8 – Filtrage multiple

La syntaxe `match <exn> with | <motif_1> | <motif_2> ... -> <exn>` évalue sur toutes les entrées correspondant aux motifs `<motifs_k>` l'expression `<exn>`. Les motifs peuvent partager des noms de variables. Par contre, chaque nom de variable de `<exn>` n'étant pas dans déjà défini avant `match` doit être définie dans chaque motif.

En général, le filtrage par motif suit les règles suivantes : l'expression filtrée est testée contre les motifs dans l'ordre et le bloc *match* est évalué à l'expression associée au premier motif correspondant. Le compilateur émettra un avertissement en cas de filtrage par motif incomplet, c'est-à-dire pour lequel il existe une valeur ne correspondant à aucun motif. Cependant, il acceptera quand même la définition de fonction (tout en espérant que vous le corrigiez) :

OCaml
```
# let est_vide_incomplet l = match l with
    | [] -> true
    | [_] -> false;;
Lines 1-3, characters 27-14:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
_::_::_
val est_vide_incomplet : 'a list -> bool = <fun>
```

Dans le reste du cours, **on considérera tout filtrage incomplet comme une erreur à corriger immédiatement**. OCaml est trop gentil avec vous sur ce coup, il n'y a que des mauvaises raisons de faire un filtrage incomplet.

Exercice 2

Donner le type des fonctions `max_liste` et `liste_est_triee`.

Exercice 3

Est-ce que la fonction `liste_est_triee` va forcément parcourir toute la liste ?

Couples et tuples Si `'a` et `'b` sont des types, alors `'a * 'b` est le type *produit* de `'a` et `'b`, dont les éléments sont des couples écrits par la syntaxe `(x, y) : 'a * 'b`.

OCaml
```
# let couple = (2, [1.0; -4.2]);;
val couple : int * float list = (2, [1.; -4.2])
# fst couple;;
- : int = 2
# snd couple;;
- : float list = [1.; -4.2]
```

Comme pour les listes, on peut utiliser le filtrage par motif pour déconstruire un couple. Cependant, il n'y a souvent qu'un seul cas à considérer, comme pour :

OCaml
```
let somme_couple c = match c with
| (x, y) -> x + y
```

Exercice 4

Quel est le type de cette fonction ?

Quand un filtrage n'a qu'un seul cas, OCaml propose une syntaxe simplifiée pour déconstruire un argument ou une variable :

OCaml
```
# let somme_couple_2 (x, y) = x + y;;
val somme_couple_2 : int * int -> int = <fun>
# let (a, b) = (2, [1.0; 4.1]);;
val a : int = 2
val b : float list = [1.; 4.1]
```

Le cas des tuples (aussi appelés n-uplets) s'écrit de manière similaire, à l'exception des fonctions `fst` et `snd` qui n'existent que pour les couples.

OCaml
```
# let norme_3 (x, y, z) =
    sqrt (x *. x +. y *. y +. z *. z);;
val norme_3 : float * float * float -> float = <fun>
# let somme_vecteurs_3 v1 v2 =
    let (x1, y1, z1) = v1
    and (x2, y2, z2) = v2 in
    (x1 +. x2, y1 +. y2, z1 +. z2);;
val somme_vecteurs_3 :
  float * float * float -> float * float * float -> float * float * float =
  <fun>
```

Remarque 9 – Sur les fonctions `fst` et `snd`

Pour accéder aux deux composantes d'un couple, on dispose de deux fonctions `fst : 'a * 'b -> 'a` et `snd : 'a * 'b -> 'a`. Ces deux fonctions sont disponibles dans la bibliothèque standard, mais c'est une mauvaise habitude à ne surtout pas prendre de les utiliser : les cas où leur utilisation est plus élégante que le filtrage par motif est trop rare pour que je prenne le risque de vous l'apprendre. En plus, ce n'est pas au programme !

Attention, utiliser des tuples pour éviter d'avoir à gérer le fonctionnement des fonctions à plusieurs variables curryfiées est plutôt une mauvaise pratique en OCaml. Le type tuple doit être réservé aux moments où il est approprié, comme pour représenter des vecteurs de taille fixe (comme des coordonnées dans le plan / l'espace).

Remarque 10 – Sur les fonctions mathématiques

Des fonctions mathématiques sont incluses dans la bibliothèque standard d'OCaml, comme `sqrt`, `log`, les fonctions trigonométriques, etc. La liste complète des fonctions disponibles directement quand on lance OCaml est disponible dans la page de manuel du module **`Stdlib`** d'OCaml.

Remarque 11 – Sur la précédence des connecteurs de types de fonction

Si l'on observe le type de la fonction `somme_vecteurs_3`, on remarque que le connecteur `*` est prioritaire sur le connecteur `->`.

Types de fonction On a déjà discuté du connecteur `->` dans le cas des types fonctions. Pour information, il existe aussi deux syntaxes alternatives pour écrire une fonction : avec le mot-clé **`function`** et avec le mot-clé **`fun`**. Pour le premier, la syntaxe est **`function`** `| <motif1> -> <exn1> | ... | <motifk> -> <exnk>`, qui est une expression dont le type est `'a -> 'b`, avec `'a` le type des motifs et `'b` le type des expressions associées. Par exemple :

OCaml
```
let rec somme_modulo n = function
  | [] -> 0
  | t :: q -> (t + somme_modulo n q) mod n

let rec somme_modulo_2 n l = match l with (* écriture équivalente avec un match *)
  | [] -> 0
  | t :: q -> (t + somme_modulo_2 n q) mod n
```

Exercice 5

Quel est le type de cette fonction ?

La seconde syntaxe est de la forme `fun <arg1> ... <argk> = <exn>`, qui est une expression évaluée en une fonction de type `'a -> ... -> 'n -> 'z`, avec `'a` ... `'k` les types de `<arg1>` ... `<argk>` et `'z` le type de `<exn>`. Cette syntaxe est notamment utilisée pour définir des fonctions en une ligne, sans avoir besoin de la lier à un nom de fonction. On a déjà mentionné que les fonctions sont des valeurs comme les autres : ainsi, elles peuvent être utilisées comme arguments d'autres fonctions. Par exemple :

OCaml
```
# let compose f g =
    fun x -> g (f x);;
val compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

Observez bien le type de la fonction : on aurait pu s'attendre à obtenir une fonction de type `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)`, mais comme mentionné précédemment, l'ordre de priorité du connecteur `->` rend la dernière paire de parenthèses inutile. Par contre, les autres sont nécessaires : on ne demande pas un argument de type `'a` puis un autre de type `'b`, mais bien un argument de type `'a -> 'b`.

II.E Qu'est-ce qu'un code OCaml ?

Il n'existe que deux types de "morceaux de codes" licites, c'est-à-dire qu'OCaml accepte d'évaluer : les **expressions** et les **définitions**.

Expressions En OCaml, la brique de base du langage est l'expression (alors que dans les langages impératifs, c'est l'*instruction*). Notamment, parmi les expressions licites en OCaml on a :
- les expressions arithmétiques `1 + 2` ou constantes comme `true`, `[]` ou encore `"Hello world !"`,
- les constructeurs comme `25 :: t`, `([1;2;3], 0, "et l'induction évidemment")`,
- les conditionnelles `if x = 0 then 1 else 1 / x`,
- les définitions de fonctions par `function` et `fun`, par exemple `function | [] -> failwith "vide" | h :: _ -> h` ou encore `fun x -> 2*x + 1`,
- les définitions locales telles que `let x = fonction_tres_longue 1 in x * x * x`.
- les annotations de type `(<exn> : <type>)`

La documentation d'OCaml (contrairement à d'autres langages moins... *sérieux*) fournit une définition formelle des expressions licites en OCaml, c'est-à-dire celles que l'interpréteur/compilateur reconnaît et tente d'évaluer : `https://v2.ocaml.org/manual/expr.html`. Ensuite, il n'y a que trois possibilités :

1. Elle est évaluée en une valeur (pourquoi pas avec un avertissement, comme un filtrage non complet)
2. L'exécution lève une exception, ce qui en général arrête le programme (comme en Python, il est possible de "capturer les exceptions" avec la syntaxe `try <exn_avec_except> with <nom_exc> -> <exn>`).
3. L'exécution ne termine jamais : c'est le cas des boucles infinies et des appels récursifs infinis.

Quand on évalue une expression dans un interpréteur (et que son évaluation réussit), il affiche `- <type> = <valeur>`.

Définitions Si les expressions sont les briques de base de la syntaxe d'OCaml, que l'on combine pour obtenir des expressions de plus en plus complexes, les définitions sont sa clé de voûte. Un code OCaml correct est une suite de définitions.

OCaml
```
let rec fab n =
  if n <= 1 then 1 else fab (n-1) + fab (n-2)

let x = 1 + 2 * 3 and y = fab 25

let _ = fab (n-1) + 2
```

Remarque 12 – Utilisation du `let _ =`

Dans ce contexte, `let _` veut dire que l'on évalue l'expression à droite du = auquel on ne donne aucun nom. Quand on l'évalue dans un interpréteur, c'est comme si on évaluait l'expression seule. Quand on compile le code contenant cette définition, elle est évaluée et n'affiche rien (comme toutes les définitions).

Les définitions peuvent aussi prendre la forme `let <nom1> = <exn1> and <nom2> = <exn2>`. On verra dans la Section III.B qu'on peut aussi définir de nouveaux types avec la syntaxe `type <nom_type> = <exn_type>`.

III Types avancés en OCaml

III.A À propos des types de base

On a vu les types de base : `int`, `float`, `bool`, `list`. Quelques remarques supplémentaires :

— Les entiers en OCaml sont représentés sur 32 ou 64 bits selon la situation (dans la salle de TP : 64, mais dans un navigateur : 32). Cependant, l'un de ces bits est réservé par le compilateur, donc les entiers représentables en OCaml sont dans l'intervalle $[\![-2^{62}, 2^{62}-1]\!]$ ou $[\![-2^{30}, 2^{30}-1]\!]$, ce qui est différent des entiers en C par exemple. Les entiers maximums et minimums disponibles sont définis globalement par les constantes `max_int` et `min_int`. Comme on le verra plus tard dans l'année, les opérations sur les entiers sont donc faites modulo 2^{31} ou 2^{63}.

— Pour les flottants, il faut bien garder à l'esprit qu'ils *approximent* les réels, et qu'il y a donc des erreurs d'arrondi qui apparaissent naturellement dans les calculs. Par exemple, ça n'a presque jamais aucun sens de vérifier que deux flottants sont égaux, parce que les erreurs d'arrondi d'un côté ou de l'autre peuvent les rendre différents sans que ce soit vrai mathématiquement. On verra la représentation des flottants plus tard dans l'année.

— Contrairement à C ou Python, OCaml n'accepte pas de convertir des types pour accepter des expressions : si l'on a besoin de faire une opération entre un flottant et un entier, on peut utiliser la fonction `float_of_int : int -> float`. La même fonction existe dans le sens inverse `int_of_float : float -> int`, qui arrondi le flottant vers 0 (la partie entière pour les entiers positifs et la partie entière supérieure pour les négatifs). On verra que la syntaxe `type1_of_type2` revient souvent pour les autres types.

— L'évaluation des opérateurs booléens est séquentielle : pour `&&`, si le terme de gauche est évalué à `false`, OCaml ne calcule pas le terme de droite et évalue l'expression directement à `false`. C'est la même chose pour `||` et `true`. Par exemple :

OCaml
```
# (22 - 5 < 21) || (2 / 0 = 0);;
- : bool = true
```

— On a mentionné les opérateurs de comparaison en OCaml, et le fait qu'ils sont polymorphes. Les opérateurs = et <> évaluent l'égalité *structurelle* des deux arguments, ce qui vérifie que leurs structures sont les mêmes (et que les valeurs de base contenues dans ces structures sont les mêmes) : ainsi, `1 :: 2 :: [] = [1;2]` est évalué à `true`, même si les arguments correspondent à des objets différents

en mémoire et sont construits de manière différente. Les opérateurs == et != ne testent pas l'égalité structurelle, mais l'égalité physique, c'est-à-dire l'égalité en mémoire, où ces deux objets ne sont pas les mêmes. On ne les utilisera jamais.

— On a déjà vu la syntaxe des listes, et notamment l'opérateur `::`. D'autres fonctions sont définies sur les listes et sont disponibles dans le module `List`. Pour accéder à des fonctions dans un module, la syntaxe est `<Module>.<fonc>`. On utilisera les fonctions suivantes en TD : `List.hd`, `List.tl`, `List.rev`, `List.length`, `List.map`, `List.fold_left`, `List.fold_right`, `List.iter`. On les définira et on les utilisera en TP.

— Une remarque sur la syntaxe `<Module>.<fonction>` : elle ne correspond pas à la syntaxe `<objet>.<methode>` en Python, mais elle est simplement évaluée à la variable/fonction définie dans le module `<Module>` sous le nom `<fonction>`. OCaml dispose d'une syntaxe dédiée pour les objets et leurs méthodes / attributs, de la forme `<objet>#<attribut>`. On ne l'utilisera pas : vous aurez de très bon cours de programmation orientée objet en école d'ingénieur, où vous apprendrez les bonnes disciplines de programmation associées (en gros : ne pas trop en abuser).

III.B Définition de nouveaux types

OCaml permet de définir de nouveaux types de plusieurs façons, en utilisant la syntaxe `type <nom_type> = <exn_type>`.

Types produits et fonction La première façon de définir de nouveaux types est d'utiliser les connecteurs de types déjà vus, c'est-à-dire `*` et `->`. Ils correspondent au produit cartésien $\times$ et à l'exponentiation ensembliste. Par exemple :

OCaml
```
type vecteur_3d = float * float * float
type quotient = int * int
type base_R3 = vecteur_3d * vecteur_3d * vecteur_3d
type bijection_entier = int -> int
```

Quand on définit une fonction ou une variable, il est possible de "forcer" l'utilisation d'un type en utilisant une *annotation de type* : dans une définition de variable ou de fonction, on peut utiliser la syntaxe `(<variable/argument> : <type>)` en remplacement `<variable/argument>`. Il est aussi possible d'annoter le type de retour d'une fonction, par la syntaxe `let <fonction> <args> : <type_retour> = <exn>`. Par exemple :

OCaml
```
# let somme_3d ((x1, y1, z1) : vecteur_3d) ((x2, y2, z2) : vecteur_3d) :
  ↪  vecteur_3d =
    (x1 +. x2, y1 +. y2, z1 +. z2);;
val somme_3d : vecteur_3d -> vecteur_3d -> vecteur_3d = <fun>
```

La fonction `somme_3d` acceptera aussi les arguments de type `float * float * float` qui n'ont pas été annotés. En fait, elle acceptera tous les arguments d'un type *compatible* avec `vecteur_3d`, ce qui ici veut dire tous les types égaux à `float * float * float`. Ainsi, ce genre de définition de type est plus un *sucre syntaxique* qu'autre chose, mais qui peut être utile pour alléger des notations.

Types sommes Les types *sommes* (aussi appelés types *union*) correspondent à une autre opération sur les ensembles, qui est l'union disjointe $\sqcup$. D'abord, on peut définir des types en énumérant les valeurs possibles du type (quand cette énumération est finie), avec la syntaxe `type <nom_type> = | <Constructeur_1> | <Constructeur_2> | ... | <Constructeur_n>`. Ensuite, ces constructeurs peuvent être utilisés comme des valeurs de ce type, ou comme des motifs de filtrage. Par exemple :

OCaml
```
# type booleen_trivaluee = Vrai | Faux | NeSaisPas;;
type booleen_trivaluee = Vrai | Faux | NeSaisPas
# let et_trivaluee b1 b2 = match b1, b2 with
```

```
  | Vrai, Vrai -> Vrai
  | Faux, _ | _, Faux -> Faux
  | _ -> NeSaisPas;;
val et_trivaluee :
  booleen_trivaluee -> booleen_trivaluee -> booleen_trivaluee =
  <fun>
```

Remarque 13

La *logique trivaluée* est une alternative à la logique propositionnelle classique, où la notion de vérité admet une troisième valeur possible représentant le fait qu'on ne sait pas si un fait est vrai ou faux.

Exercice 6

Proposer un type énuméré correspondant au type `bool`, et écrire les fonctions reprenant les opérateurs `&& || not`.

Les constructeurs des types sommes peuvent aussi être paramétriques, c'est-à-dire accepter des paramètres ayant eux-mêmes un type. C'est comme cela que l'on définit les arbres binaires en OCaml :

OCaml
```
# type arbre =
  | Vide (* notez que le premier | est facultatif *)
  | Noeud of arbre * arbre;;
type arbre = Vide | Noeud of arbre * arbre
# let a = Noeud (Noeud (Vide, Vide), Vide);; (* un arbre avec une racine et un
 ↪  fils gauche *)
val a : arbre = Noeud (Noeud (Vide, Vide), Vide)
# let rec taille a = match a with
  | Vide -> 0
  | Noeud (gauche, droit) -> 1 + taille gauche + taille droit;;
val taille : arbre -> int = <fun>
```

Enfin, les types sommes eux-mêmes peuvent être paramétrés :

OCaml
```
# type 'a arbre_etiquete = Vide | Noeud of 'a arbre_etiquete * 'a * 'a
 ↪  arbre_etiquete;;
type 'a arbre_etiquete =
    Vide
  | Noeud of 'a arbre_etiquete * 'a * 'a arbre_etiquete
```

Exercice 7

Définir des fonctions `hauteur : arbre -> int` et `liste_sous_arbres : arbre -> arbre list` (on autorise les répétitions), et la fonction `max_arbre : 'a arbre_etiquete -> 'a`.

Exercice 8

Redéfinir le type `list` en OCaml. On remarquera ainsi que `::` n'est pas un *opérateur*, mais un *constructeur*.

Types option Un de ces types sommes est prédéfini en OCaml, qui est très utile quand on veut définir une fonction "restreinte" (dans le sens d'une fonction non définie sur tout le domaine d'entrée) : c'est le type paramétrique `'a option` qui peut être redéfini en `type 'a option = None | Some of 'a`. Ainsi, on peut définir une fonction "restreinte" comme une fonction évaluée à `Some` y quand $f(x) = y$ existe et `None` sinon. Par exemple, une fonction qui renvoie le premier index où un élément se trouve dans une liste :

```OCaml
let index x l =
  (* On utilise l'argument [n] comme un accumulateur qui stocke l'index courant
   ↪  dans la liste [l] *)
  let rec index_aux l_aux n = match l_aux with
  | [] -> None
  | tete :: queue ->
    if tete = x then Some n
    else index_aux queue (n+1)
  in index_aux l 0
```

Ici, on a utilisé une *fonction auxiliaire* `index_aux`, définie localement dans la fonction qui permet de passer l'index actuel en argument. Par contre, comme pour les autres constructeurs paramétrés, il faut filtrer l'option pour récupérer son argument éventuel. Par exemple, pour calculer le premier index où se trouve un élément dans l'une de deux listes :

```OCaml
let premier_index_deux_listes x l1 l2 =
  match (index x l1), (index x l2) with
  | Some n1, Some n2 -> Some (min n1 n2)
  | Some n1, _ -> Some n1
  | _, Some n2 -> Some n2
  | _ -> None
```

Remarque 14 – Constructeur None

Le constructeur `None` a un rôle similaire à la valeur `None` en Python. Il exprime le fait que le programmeur ne veut pas fournir une valeur et peut servir à fournir explicitement une valeur par défaut à la variable à l'intérieur de la fonction. Cependant, le compilateur/interpréteur impose/incite très fortement à écrire les deux cas `Some` et `None`, alors que Python accepte le fait de fournir `None` en argument à une fonction pas prévue pour, et le programmeur doit avoir la discipline de programmation de vérifier que ces cas n'arrivent pas dans son code. C'est le même souci qui arrive avec le pointeur `null` en C.

Remarque 15 – Les fonctions partielles avec des exceptions

L'alternative au type `option` est d'utiliser des *exceptions* : si une fonction n'est pas censée autoriser certaines entrées, ou si l'évaluation d'une expression retourne une valeur auquel on ne s'attendait pas, on lève une exception et le programme entier s'arrête. Pour cela, on a déjà vu la fonction `failwith : string -> 'a` qui, quand elle est évaluée, arrête le programme et affiche `Exception: Failure "Message d'erreur".`. Un autre moyen de lever une exception à connaître (bien qu'à la frontière du programme) l'assertion `assert` : on l'évalue sur un booléen `b`, et si `b` est évalué à `true` le programme renvoie `()`, sinon il lève une exception indiquant la ligne et la colonne de l'assertion fausse. Par exemple :

```OCaml
let rec fact n =
  assert (n >= 0);
  if n = 0 then 1 else n * fact (n-1)
```

Quand on écrit du code, on pensera à utiliser des assertions pour *tester* que les fonctions correspondent bien à leurs spécifications sur des exemples bien choisis. Par exemple :

```OCaml
let algorithme_de_tri_intelligent l =
  ...

let _ =
  assert (algorithme_de_tri_intelligent [5;2;3] = [2;3;5] )
let _ =
  assert (algorithme_de_tri_intelligent [] = [])
...
```

Ce genre de test est attendu dans vos productions en OCaml : **toutes les fonctions doivent être testées**. L'avantage des fonctions pures (purement fonctionnelles, qui n'utilisent que des variables immutables) est qu'il suffit de s'assurer que chaque fonction marche individuellement pour que le programme entier fonctionne. Pour cela, on réalise plusieurs tests sur des exemples différents, notamment sur les cas limites (listes vides ou à un élément, listes déjà triées, etc.). Pour OCaml, il existe plusieurs bibliothèques de tests unitaires, c'est-à-dire de tests exécutés avant la mise en production d'un code source : on peut citer `https://gildor478.github.io/ounit/ounit2/index.html` qui exécute des tests unitaires simples et hautement configurables, ou encore `https://github.com/c-cube/qcheck` qui génère aléatoirement des entrées, vérifient qu'elles passent une propriété et tente de réduire les éventuels contre-exemples en une entrée minimale rendant la fonction incorrecte.
Par exemple :

```OCaml
# (* fonction miroir incorrecte *)
  let miroir_incorrecte l = match l with
    | [] -> []
    | t :: q -> q @ [t]

  (* définition d'un test de QCheck *)
  let test fonction_miroir =
    QCheck.Test.make ~count:1000 ~name:"miroir est involutive"
    QCheck.(list small_nat)
    (fun l -> fonction_miroir (fonction_miroir l) = l);; (* on teste que le
     ↪  miroir du miroir de [l] est [l] *)

val miroir_incorrecte : 'a list -> 'a list = <fun>
val test : (int list -> int list) -> QCheck2.Test.t = <fun>

(* pour notre fonction incorrecte, QCheck donne un contre-exemple minimal *)
# let _ = QCheck.Test.check_exn (test miroir_incorrecte);;
  Exception:
  test `miroir est involutive` failed on >= 1 cases:
  [0; 0; 1] (after 7 shrink steps)

(* ici, la fonction est conforme au test, donc QCheck renvoie [()] *)
# let _ = QCheck.Test.check_exn (test List.rev);;
- : unit = () (* le test est passé, QCheck n'a pas trouvé d'erreurs *)
```

Types enregistrement Un type enregistrement correspond à un type produit, mais où les composantes sont nommées. La syntaxe pour les définir est `type <nom_type> = {<id1> : <type1>; ...; <idk> : <typek>}`. Pour définir une valeur de ce type, la syntaxe est `{<id1> = <exn1>; ... ; <idk> = <exnk>}`. Pour accéder à la valeur dans le champ `<id>`, la syntaxe est `<var>.<id>`. Par exemple, on va représenter un

jeu de cartes par un enregistrement avec un champ indiquant le signe de la carte et un champ indiquant sa valeur :

OCaml
```
type couleur = Pique | Coeur | Carreau | Trefle
type tete = Roi | Dame | Valet
type valeur = Tete of tete | Nombre of int
type carte = {co : couleur ; va : valeur}

# let c = {co = Pique; va = Nombre 2};;
val c : carte = {co = Pique; va = Nombre 2}
```

Exercice 9

Écrire une fonction évaluant si une carte est une Dame ou un Pique.

Remarques supplémentaires sur le filtrage En pratique, on utilisera surtout le filtrage sur des motifs constants `1` `-0.2` `"coucou !"` `None`, des constructeurs paramétrés par des variables (qui seront locales à l'expression associée) `Noeud (gauche, Noeud (droit1, droit2))`, `Some x`, ou des tuples `(1, 25.2, None)`. On peut omettre les parenthèses dans les motifs tuples. Ainsi, il est interdit d'inclure des fonctions dans les motifs, par exemple, on ne peut pas écrire :

OCaml
```
let est_pair_fausse n = match n with
| 2 * k -> true
| _ -> false

let est_pair n = match n mod 2 with
| 0 -> true
| _ -> false (* pourquoi ne pas utiliser 1 ici ? parce que [n mod 2] est négatif
 ↪  quand [n] l'est *)

let est_pair_mieux = n mod 2 = 0
```

Il est possible d'utiliser des motifs de filtre *gardés*, c'est-à-dire des filtres qui testent une expression booléenne en plus de la correspondance au motif. Par exemple :

OCaml
```
let rec some_positifs_liste l = match l with
| [] -> []
| x :: xs when x >= 0 -> Some x :: some_positifs_liste xs
| x :: xs -> None :: some_positifs_liste xs (* on arrive à ce motif ssi la
 ↪  précédente n'est pas vérifiée, i.e. [x < 0] *)

let rec some_positifs_liste_2 l = match l with (* écriture équivalente *)
| [] -> []
| x :: xs ->
  (if x >= 0 then Some x else None) :: some_positifs_liste_2 xs
```

On peut aussi filtrer sur des enregistrements en OCaml : il permet notamment de filtrer sur un sous-ensemble de champs, et les mettre dans le désordre, par exemple :

OCaml
```
let est_tete_ou_as_de_pique c = match c with
  | {va = Tete _} -> true
  | {va = Nombre 1; co = Pique} -> true
  | _ -> false

```

```
let est_tete_ou_as_de_pique_2 c = match c.co, c.va with
  | _, Tete _ -> true
  | Pique, Nombre 1 -> true
  | _ -> false
```

IV À retenir

Syntaxe à connaître et savoir manipuler

— `let`, `let rec`, `let ... and ...`, `let ... in` y compris déconstruction d'un tuple ;
— types `int float bool list option`, opérateurs de types `* ->` ;
— opérateurs `+ - * / mod`, `+. -. *. /.`, `&& || not` et leur évaluation paresseuse, `::` ;
— comparaisons `= <> < > <= >=` ;
— types tuples et listes ;
— syntaxe des fonctions : `let f ... = ...`, `fun ... -> ...`, `function | ... -> ... | ... -> ...` ;
— types sommes y compris récursifs, filtrage par motif, filtrage par motif gardé ;
— types enregistrement : syntaxe du type, des éléments et accès à un champ ;
— syntaxe des modules : pour l'instant, il n'y a que la fonction `List.length` à connaître ;
— notions d'expression et de définition, de portée d'une définition (définitions récursives, mutuellement récursives, locales, globales).

Erreurs à éviter

Les erreurs de type :

mélanger les types flottants et entier	~~`3.02 + 4`~~	`3.02 +. 4.`
comparer deux éléments de types différents	~~`5 > (3.2, false)`~~	`2.5 < 3.5`
mal typer le constructeur `::`	~~`[52; 43] :: [18]`~~	`52 :: 43 :: [18]`

Les erreurs de filtrage :

Ne pas écrire…	**Mais plutôt…**

Un filtrage incomplet :
OCaml
`match x with`
`| Some 3 -> ...`
`| None -> ...`
	OCaml
	`match x with`
	`| Some 3 -> ...`
	`| None -> ...`
	`| Some _ -> ...`

Un motif avec des noms de variable déjà utilisés :
OCaml
`let x = 0 in`
`  match y with`
`  | x -> ...`
`  | _ -> ...`
	OCaml
	`let x = 0 in`
	`match y with`
	`| x' when x' = x -> ...`
	`| _ -> ...`

Faire apparaître une variable plusieurs fois dans le même motif :
OCaml
`match l with`
`| x :: x :: _ -> ...`
`| _ -> ...`
	OCaml
	`match l1, l2 with`
	`| x :: x' :: _ when x = x' -> ...`
	`| _ -> ...`

Dans plusieurs motifs regroupés, ne faire apparaître une variable que dans l'une d'entre elles :
OCaml
`match l with`
`| [x] | _ :: y :: _ -> ...`
`| _ -> ...`
	OCaml
	`match l with`
	`| [x] -> ...`
	`| _ :: y :: _ -> ...`
	`| _ -> ...`

Et deux derniers pour la route : ne pas oublier les `Some` dans les motifs de filtrage sur les options :

OCaml
```
match x with
| None -> 0
| x -> x + 3

(* mais plutôt *)

match x with
| None -> 0
| Some x -> x + 3
```

Et ne pas oublier qu'utiliser == et != est strictement interdit en OCaml.