OCaml, des fonctions et des types

A Pratiquer OCaml

Le plus simple Pour utiliser OCaml, il suffit de l'utiliser en ligne via les bacs à sable des sites OCaml ou TryOcaml.

Pour travailler localement Sur votre machine, le plus simple est d'utiliser l'interprète interactif OCaml. On peut facilement l'installer sur n'importe quel système d'exploitation en suivant ces instructions.

Pour travailler avec un éditeur de texte ăgrâce au mode Tuareg. Les commandes `M-x tuareg-mode` et `M-x run-ocaml` permettent d'activer ce mode. Le résumé (sic!) des commandes est accessible en ligne. Pour les puristes de la ligne de commande, ce mode est également disponible sous Vim.

Utiliser un IDE Enfin, il est possible d'utiliser OCaml avec la plupart des environnement de développement : Eclipse, Visual Studio ou IntelliJ (Jet Brains).

B Vocabulaire utile

■ **Définition 1 — Modèle de calcul.** Un modèle de calcul (MOC) est la description d'une manière de calculer une fonction mathématique étant donnée une entrée. Il existe des modèles de calcul séquentiels [a], fonctionnels [b] et concurrents [c].

a. les automates
b. le lambda calcul
c. les circuits logiques, les réseaux de Petri ou Synchronous Data Flow (cf. simulink)

■ **Définition 2 — Paradigme de programmation.** Un paradigme de programmation est un ensemble de formes et de figures qui constitue un modèle propre à un langage.

■ **Définition 3 — Paradigme impératif.** Le paradigme impératif s'attache à décrire des séquences d'instructions (ordres) qui agissent sur un état interne de la machine (contexte). L'impératif explicite le *comment procéder* pour exécuter un programme. Cette programmation se rapproche de la logique électronique des processeurs.

■ **Définition 4 — Paradigme procédural.** Ce paradigme est une déclinaison de l'impératif et propose de regrouper des éléments réutilisables de code dans des routines. Ces routines

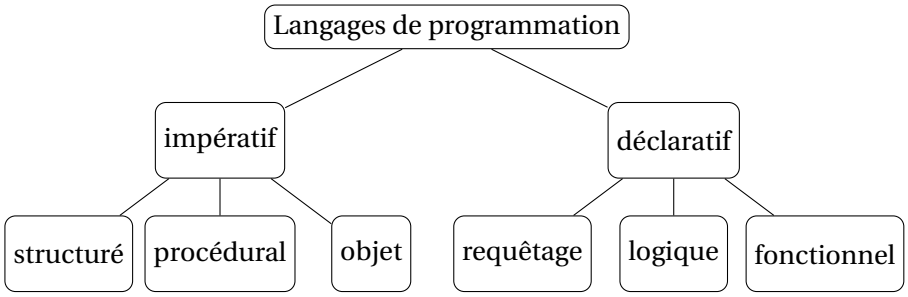

FIGURE 1 – Paradigmes des langages de programmation

sont appelées procédures (si elles ne renvoient rien) ou fonctions (si elles renvoient un résultat).

■ **Définition 5 — Paradigme objet.** Ce paradigme est une déclinaison de l'impératif et propose de décrire un programme comme l'interaction entre des objets à définir. Une classe est un type d'objet qui possède des attributs et des comportements. Ces caractéristiques sont encapsulées et peuvent être masquées à l'utilisateur d'un objet : cela permet de protéger l'intégrité de l'objet et de garantir une cohérence dans la manipulation des données.

■ **Définition 6 — Paradigme déclaratif.** Le paradigme déclaratif est une syntaxe qui s'attache à décrire le *quoi*, c'est à dire *ce que le programme doit faire*, non pas comment il doit le faire. Un langage déclaratif ne dépend pas de l'état interne d'une machine (contexte). Cette programmation se rapproche de la logique mathématique et délègue au compilateur la délicate question du *comment procéder*.

■ **Définition 7 — Paradigme fonctionnel.** Le paradigme fonctionnel est une déclinaison du déclaratif qui considère qu'un programme n'est qu'un calcul et qu'un calcul est le résultat d'une fonction. Le mot fonction est ici à prendre au sens mathématique du terme (lambda calcul) : une fonction appelée avec les mêmes paramètres produit le même résultat en toute circonstance.

C Description d'OCaml

Le langage OCaml est un langage qui s'appuie sur le modèle de calcul Categorical Abstract Machine (CAM) qui lui confère une syntaxe fonctionnelle très proche du langage mathématique. OCaml est un langage interprété et compilé dont le typage est fort et statique. Les paradigmes de programmation d'OCaml sont les paradigmes fonctionnel, impératif et orienté objet.

Les points forts d'OCaml sont :

le typage fort et implicite OCaml est **fortement typé** : toute expression possède un type. Le typage est implicite : l'utilisateur n'a pas à le préciser car le compilateur OCaml utilise

un algorithme d'**inférence de type** pour déterminer le type d'une expression. Le typage est statique et vérifié à la compilation : le couplage d'un typage fort et statique permet d'augmenter la performance du code et de rendre plus robuste le code face aux erreurs.

des structures de données muables et immuables OCaml propose des tableaux (Array), des structure de données mutables (tableaux, dictionnaires) mais aussi des structures immuables (listes). De nombreuses bibliothèques sont disponibles : files, tas, arbres…

un Garbage Collector un algorithme de gestion automatisée de la mémoire permet à OCaml de nettoyer les espaces mémoires qui ne sont plus utilisés par le programme en cours d'exécution. C'est important car les structures immuables engendrent en permanence la création et la destruction d'objets en mémoire.

la curryfication des fonctions OCaml permet de manipuler les fonctions comme des objets. Il permet également d'utiliser l'application partielle des fonctions (curryfication).

Dans la suite de ce chapitre, on pourra tester en même temps les expressions et les évaluer sur machine ou en ligne avec le bac à sable OCaml.

D Expressions et inférence de type

En tant que langage fonctionnel (et donc déclaratif), OCaml considère le calcul comme l'évaluation de fonctions mathématiques : OCaml traite donc des expressions qu'il évalue lorsqu'on fait suivre l'expression par `;;`.

Les types simples disponibles en OCaml sont

- `int` les entiers,
- `float` les flottants,
- `string` les chaînes de caractères,
- `bool` les booléens.

Voici un exemple d'évaluation d'un expression :

```
>>> 3 + 2*5 ;;
- : int = 13
```

Cet exemple montre bien qu'OCaml a inféré le type du résultat : il a écrit `int = 13`, c'est à dire que le résultat de l'évaluation de l'expression est un entier. Il faut noter qu'OCaml ne fait pas de transtypage implicite : pour lui, `3` est un entier et le restera, tout comme `3.5` est un flottant et le restera également. C'est pourquoi les opérateurs `+`,`*` et `/` qui additionne, multiplie ou divise les entiers ne sont pas les mêmes que ceux qui opèrent sur les flottants `+.`, `*.` ou `/.`.

D'ailleurs, l'inférence de type ne s'y trompe pas :

```
>>> 3.0 + 2.0*.5.0 ;;
Error : This expression has type float but an expression was expected of
    type int
```

Dans cet exemple, on a oublié d'utiliser l'opérateur `+.` pour additionner des flottants. Le compilateur s'en aperçoit car pour lui l'opérateur `+` n'accepte que deux opérandes entières et son

résultat est un entier. Or, les opérandes fournies sont des flottants. Il détecte donc l'incohérence, la signale et ne peut pas continuer l'exécution.

En OCaml, un chou est un chou et restera un chou. Parole de léonard.

E Expressions locales et globales

Le mot clef `let` permet de définir une variable **globale**. Cette définition est une déclaration-initialisation. Une variable est toujours initialisée : cela permet au mécanisme d'inférence de type de fonctionner. À droite du symbole = doit se trouver une expression (ci-dessous `21 * 2`). À gauche du symbole = doit se trouver un identifiant (ci-dessous `x`).

```
>>> let x = 21 * 2;;
val x : int = 42
>>> x / 2;;
- : int = 21
```

Les mots clefs `let ... in` permettent de définir des variables **locales**. La portée de la variable ainsi définie est l'expression qui suit le mot clef `in`.

```
>>> let recettes = 4000;;
val recettes : int = 4000
>>> let budget = let depenses = 3500 in recettes - depenses;;
val budget : int = 500
>>> depenses * 2;;
Error
: Unbound value depenses
>>> recettes
- : int = 4000
```

OCaml définit une expression conditionnelle à l'aide de l'opérateur ternaire `if t then e1 else e2`. À la différence de son homologue en programmation impérative, cette expression **renvoie** un résultat. C'est pourquoi elle exige que les expressions `e1` et `e2` soient du même type.

```
>>> let recettes = 4000;;
val recettes : int = 4000
>>> let budget = let depenses = 3500 in recettes - depenses;;
val budget : int = 500
>>> let beneficiaire = if budget > 0 then true else false;;
val beneficiaire : bool = true
>>> let result = if beneficiaire then "Excellent !" else "Va falloir aller
    à la banque !";;
val result : string = "Excellent !"
```

Il faut bien noter que le symbole = pour déclarer une expression est utilisé conjointement à `let`. Il ne s'agit pas d'une affectation. D'ailleurs, un dernier exemple nous montre que l'opérateur = utilisé seul est un test d'égalité :

```
>>> let recettes = 4000;;
val recettes : int = 4000
```

```
>>> recettes = recettes + 400;;
- : bool = false
```

Enfin, on ne peut pas modifier le contenu d'une variable :

```
>>> let a = 3;;
     val a : int = 3
>>> a = 4 + 4;;
     - : bool = false
>>> a = a + 4;;
     - : bool = false
>>> let a = a + 3;;
     val a : int = 6
```

Pour modifier une variable, il faut donc la redéfinir ou utiliser le mécanisme de références (cf. section K).

O En OCaml, les variables sont donc immuables... ce qui est déconcertant au premier abord, car le propre d'une variable n'est-il pas de varier? La réponse est qu'en OCaml, comme en mathématiques, les variables sont en fait des expressions. Si on prend l'équation $y = 2x + 3$, l'idée de modifier la variable x n'existe pas vraiment en tant que telle : x va pouvoir varier dans le sens où l'on peut l'initialiser à différentes valeurs et que l'équation reste valable.

F Fonctions

Une fonction OCaml est une fonction au sens mathématique du terme et une expression paramétrée ou non. C'est pourquoi OCaml permet de la déclarer via le mot clef `let`.

```
>>> let perimeter r = 2. *. 3.1415926 *. r;;
val perimeter : float -> float = <fun>
>>> perimeter 1.;;
- : float = 6.2831852
```

On observe que le résultat est bien une fonction marquée par le mot-clef `fun`. Lors de l'évaluation de la déclaration de la fonction, OCaml nous délivre la signature associée à la fonction : `perimeter : float -> float = <fun>`. Celle-ci signifie que la fonction `perimeter` prend un paramètre de type `float` et renvoie un paramètre de type `float`.

Si la fonction est récursive, il faut le mentionner pour qu'OCaml en tienne compte :

```
>>> let rec explode n = if n = 0 then "Boum !" else  ". " ^ (explode (n -
    1));;
val explode : int -> string = <fun>
>>> explode 3;;
- : string = ". . . Boum !"
```

O En ce qui concerne les fonctions OCaml :

- Les fonctions en OCaml renvoient la dernière valeur calculée. Il n'existe par de mot-clef `return` comme en Python.
- Les paramètres formels des fonctions ne sont pas délimités par des parenthèses.
- Les types des paramètres d'entrée et de sortie sont inférés automatiquement.

Enfin, OCaml permet également de curryfier les fonctions : l'application partielle d'une fonction est une fonction.

```
>>> let ajoute a b = a + b;;
val ajoute : int -> int -> int = <fun>
>>> let ajoute_deux = ajoute 2;;
val ajoute_deux : int -> int = <fun>
>>> ajoute_deux 4;;
- : int = 6
```

R La version curryfiée de la fonction est souvent un atout pratique en programmation fonctionnelle. On parle aussi d'application partielle. Dans l'exemple ci-dessus, l'expression `ajoute 2` est l'application partielle de la fonction `ajoute`.

Une fonction à n variables s'interprète donc soit :

1. comme une fonction à n variables,
2. une famille de fonctions à $n-1$ variables paramétrées par la première,
3. une famille de fonctions à $n-2$ variables paramétrées par les deux premières...

■ **Définition 8 — Curryfication.** Deux syntaxes sont possibles pour définir une fonction `f` de plusieurs variables :

1. la version non-curryfiée : `let f (x,y)= ... ;;`
2. la version curryfiée `let f x y = ... ;;`

G Effets

Vous aurez remarqué que cette introduction à OCaml n'a pas commencé par le traditionnel programme `"Hello Word!"` ! et il y a une bonne raison à cela : les langages fonctionnels n'aiment pas les effets de bords.

Vocabulary 1 — Side effect ⇜⇝ Effet secondaire mal traduit en français par effet de bord.

■ **Définition 9 — Effet (de bord) d'une expression.** Un effet de bord d'une expression est une action de celle-ci qui modifie l'état d'une variable en dehors de l'environnement local à la fonction : l'effet est donc quelque chose d'observable en dehors du fonctionnement

standard d'une fonction, c'est-à-dire en dehors de la valeur retournée par la fonction.

■ **Exemple 1 — Effets de bord.** Quelques exemples classiques d'effets de bord :

- la modification d'une variable définie en dehors de la fonction.
- la modification d'une variable muable passée en paramètre à la fonction.
- l'appel d'une fonction qui produit des effets,
- toute entrée/sortie : l'impression d'une chaîne de caractère sur la console d'un écran d'ordinateur, la réception d'un message sur un socket réseau, la lecture ou l'écriture dans un fichier, l'interaction avec un système d'exploitation.

■ **Définition 10 — Expression pure et impure.** On dit qu'une expression est pure si elle n'engendre aucun effet de bord. Dans le cas contraire, on la dit impure.

Dans l'esprit des concepteurs d'OCaml et des langages fonctionnels en général et pour des raisons de cohérence, une fonction doit toujours renvoyer exactement le même résultat si elle est invoquée avec les mêmes paramètres en entrée [1]. Or, si une fonction possède des effets de bords, il est possible que ce ne soit pas le cas.

■ **Exemple 2 — Fonction impure en Python.** Dans l'exemple ci-dessous, la fonction `setn` est appelée deux fois avec le même paramètre 3 mais produit deux résultats différents. C'est donc une fonction impure.

```
1  x = 0
2
3  def setn(n):
4      global x
5      n = n + x
6      return n
7
8  n = setn(3)
9  assert n == 3
10 x = 4
11 n = setn(3)
12 assert n == 3
```

Le programme `hello` ci-dessous est un exemple de fonction impure en OCaml. Écrire un message sur la console, c'est agir sur l'environnement d'exécution en dehors de la fonction. Le risque est, par exemple, que la console ne réponde pas aux ordres du système d'exploitation. Dans ce cas, le programme ne s'exécute pas correctement à cause de l'effet de bord.

```
1  >>> let hello name = print_string ("Hello " ^ name ^"\n");;
2  val hello : string -> unit = <fun>
3  >>> hello "Olivier";;
```

1. Dans le cadre de la programmation des systèmes parallèles (microprocesseurs à architectures multicœurs), cette vision fonctionnelle est très importante car elle permet d'éviter de nombreux problèmes de synchronisation mémoire.

```
4 Hello Olivier
5 - : unit = ()
```

La signature d'`hello` indique que la fonction renvoie un type `unit`.

■ **Définition 11 — Type unit.** Le type `unit = ()` représente le rien, le vide. Il est utilisé pour bien signifier qu'une fonction ne prend pas de paramètre ou ne renvoie rien. Elle peut par contre avoir un effet.

H Types algébriques

En plus des types simples, OCaml propose des mécanismes pour construire d'autres types éventuellement récursifs à partir des types simples :

- les types algébriques qui sont des composés
  - de types sommes qui sont des alternatives ou des énumérations,
  - et de types produits, des produits cartésiens de types,
- les record, ou enregistrements, qui permettent d'enregistrer une collection de types dans un même objet,
- les type optionnels qui par nature n'existent pas nécessairement.

■ **Définition 12 — Types sommes ou énumérations.** Un type somme est une alternative de types. Il est défini par l'alternance de ses constructeurs.

■ **Exemple 3 — Types sommes en OCaml.** Le type utilise l'alternative `|` et des constructeurs qui commencent par une majuscule.

```
1 >>>     type chess_piece = Pawn | Knight | Bishop | Rook | Queen | King;;
2 type chess_piece = Pawn | Knight | Bishop | Rook | Queen | King
3 >>> let p = Pawn;;
4 val p : chess_piece = Pawn
```

■ **Définition 13 — Types produits.** Un type produit est un produit cartésien de types. Il engendre des tuples. Généralement, on ne nomme pas un type produit.

■ **Exemple 4 — Types produits en OCaml.** Le produit de type est réalisé par l'opérateur `*`.

```
1 >>> type point3d = float * float * float;;
2 type point3d = float * float * float
3 >>> let zero = 0.0, 0.0, 0.0;;
4 val zero : float * float * float = (0., 0., 0.)
```

■ **Définition 14 — Types enregistrements.** Un enregistrement est une collection de types nommés et enregistrés dans une même structure.

■ **Exemple 5 — Types enregistrements en OCaml.** Dans le domaine des services réseaux, on a souvent besoin d'identifier un service par son socket réseau qui est la combinaison de son adresse IP ou nom d'hôte, d'un numéro de port et d'un protocole. Cela peut se faire via un type enregistrement. Attention à la syntaxe, aux points virgules, deux points et symbole égal.

```
>>> type service_info =
.. {    service_name : string;
    ..      port         : int;
    ..      protocol     : string;
    .. };;
type service_info = { service_name : string; port : int; protocol :
    string; }
>>> let http_service = {service_name = "www"; port = 80; protocol = "http
    " };;
val http_service : service_info =
{service_name = "www"; port = 80; protocol = "http"}
>>> http_service.port;;
- : int = 80
```

■ **Définition 15 — Types optionnels.** Un type optionnel est un type qui peut être typé et posséder une valeur ou ne pas être typé ni posséder de valeur.

■ **Exemple 6 — Types options en OCaml.** En OCaml, les mots-clefs pour le type `option` sont `None` et `Some`.

```
>>> type zip_code = None | Some of int;;
type zip_code = None | Some of int
>>> let brest_code = Some 29200;;
val brest_code : zip_code = Some 29200
>>> let lost_city_code = None;;
val lost_city_code : zip_code = None
```

■ **Définition 16 — Types algébriques.** Un type algébrique est un type éventuellement récursif qui est une alternative de types éventuellement produit.

■ **Exemple 7 — Types algébriques.** Dans cet exemple, on construit un jeu de carte. Une première fonction permet d'obtenir la liste des figures d'une couleur donnée. Une autre la liste des cartes numéros pour une couleur donnée (à la belote!). Le mot-clef `of` permet de préciser un type de donnée associé au constructeur.

```
>>> type couleur = Coeur | Carreaux | Pique | Trefle;;
```

```
type couleur = Coeur | Carreau | Pique | Trefle
>>> type carte = As of couleur | Roi of couleur | Dame of couleur |
    Valet of couleur | Numero of int * couleur;;
type carte =
As of couleur
| Roi of couleur
| Dame of couleur
| Valet of couleur
| Numero of int * couleur
>>> let figures_de c = [As c; Roi c; Dame c; Valet c];;
val figures_de : couleur -> carte list = <fun>
>>> figures_de Pique;;
- : carte list = [As Pique; Roi Pique; Dame Pique; Valet Pique]
>>> let numero_de c = [10,c;9,c;8,c;7,c];;
val numero_de : 'a -> (int * 'a) list = <fun>
>>> numero_de Coeur;;
- : (int * couleur) list = [(10, Coeur); (9, Coeur); (8, Coeur); (7,
    Coeur)]
```

I Listes

Le type liste OCaml est un type récursif immuable.

R **Immuable** signifie qu'on ne peut pas le modifier. Pour faire évoluer une liste, c'est à dire pour ajouter ou retirer des éléments, il est donc nécessaire de créer une autre liste.

O Une liste en OCaml ne contient qu'un seul type de données.

■ **Définition 17 — Définition inductive des listes.** Une liste est soit une liste vide soit un élément suivi d'une liste.

L'ensemble des listes $\mathcal{L}$ à valeur dans $\mathcal{E}$ est donc définie par :

Base la liste vide `[]` est une liste

Constructeur : : $\forall L \in \mathcal{L}, \forall e \in \mathcal{E}, e :: l \in \mathcal{L}$

Ce qui en OCaml donne :

```
type 'a list =
   | []
   | (::) of 'a * 'a list
```

L'expression `'a` désigne un type quelconque. Donc on peut construire une liste de n'importe quel type d'objet.

O Quelques remarques sur les opérations sur les listes en OCaml :

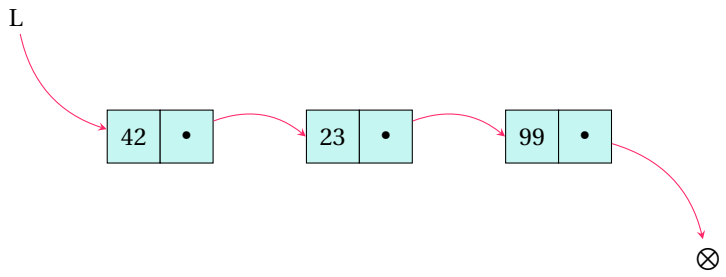

FIGURE 2 – Représentation d'une liste d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	L pointe sur le premier élément
Accès à un élément à la fin	$O(n)$	accès séquentiel
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	L pointe sur le premier élément
Ajout d'un élément à la fin	$O(n)$	accès séquentiel
Suppression d'un élément au début	$O(1)$	L pointe sur le premier élément
Suppression d'un élément à la fin	$O(n)$	accès séquentiel

TABLE 1 – Complexité des opérations associées à l'utilisation d'une liste simplement chaînée.

- pour supprimer un élément d'une liste, il faut en construire une autre.
- pour ajouter un élément à une liste, il faut en construire une autre.

Ces opérations ont donc un coût comme l'illustre le tableau 1.

■ **Exemple 8 — Liste en OCaml.** On peut construire une liste OCaml directement en initialisant les valeurs des éléments :

```
>>> let l = [1;2;3;4;5];;
val l : int list = [1; 2; 3; 4; 5]
```

Le constructeur `::` permet d'ajouter un élément en tête de liste. Par exemple :

```
>>> let l = 42::21::14::7::l;;
val l : int list = [42; 21; 14; 7; 1; 2; 3; 4; 5]
>>> let wrong = 0.2::l;;
Line 1, characters 17-18:
|1 | let wrong = 0.2::l;;
                     ^
Error: This expression has type int list but an expression was expected
     of type float list. Type int is not compatible with type float
```

Le type d'objet contenu dans une liste est nécessairement toujours le même : dans l'exemple ci-dessus, le type de la liste est `int list`, une liste d'entiers.

On ne peut accéder directement [a] qu'au premier élément. L'accès aux autres éléments nécessite de balayer la liste. Généralement, comme c'est une structure inductive, on procède par déconstruction.

```
>>>let head = List.hd l;;
val head : int = 42
```

a. en temps constant O(1)

J Filtrage de motif

Le filtrage de motif est une technique pour gérer les types algébriques et les ensembles inductifs.

Vocabulary 2 — Pattern matching ⇜⇝ Filtrage de motif

■ **Définition 18 — Filtrage de motif.** Le filtrage de motif est l'action de tester une expression pour détecter sa constitution dans le but de faire un calcule en fonction du motif détecté.

```
let f x =
    match x with
       | motif_1 - > expression 1 (* traitement de ce cas *)
       | motif_2 - > expression 2 (* traitement de ce cas *)
       .
       .
       .
       | motif_n - >  expression n (* traitement de ce cas *);;
```

L'expression `motif` est une constante ou un constructeur de types. L'ensemble des motifs décrits doit être **exhaustif**.

■ **Exemple 9 — Filtrage simple et exhaustif.** Considérons la fonction qui teste la parité d'un nombre entier. On l'écrirait, dans un langage impératif, avec une structure conditionnelle `if then else`. Il est possible de l'écrire ainsi en OCaml, mais on peut également utiliser le filtrage de motif comme suit :

```
let is_even n =
    match n mod 2 with
        | 0 ->  true
        | _ -> false ;;
```

Ce filtrage est bien exhaustif : le second cas _ englobe tous les cas possibles autre que 0.

■ **Exemple 10 — Filtrage de motif en OCaml.** Le code ci-dessous définit un type algébrique `shape` puis une fonction qui calcule l'aire en fonction du type `shape` passé en paramètre. L'aire du triangle est calculée avec la formule de Héron.

```
type shape = Circle of float | Square of float | Triangle of (float *
    float * float);;

let area s =
match s with
    | Circle r -> Float.pi *. (r ** 2.0)
    | Square s -> s ** 2.0
    | Triangle (a, b, c) -> let s = (a +. b +. c) /. 2.0 in sqrt (s *. (s
        -. a) *. (s -. b) *. (s -. c));;

area (Triangle (3.0, 4.0, 5.0));;
```

■ **Exemple 11 — Filtrage de motif et liste en OCaml.** La fonction ci-dessous est récursive et calcule la somme des éléments d'une liste. Elle déconstruit au fur et à mesure la liste pour additionner ses éléments.

```
let rec sum l =
    match l with
        | [] -> 0
        | head :: tail -> head + sum tail;;
let l = 42::21::14::7::1;;
sum l;;
```

Le motif `[]` représente une liste vide. Si ce motif est détecté, alors la fonction renvoie 0, car la somme des éléments d'une liste vide vaut 0.

Sinon, il y a au moins un élément en tête de liste `head` suivi `(::)` par une sous-liste `tail` éventuellement vide. Alors on renvoie la valeur `head` ajoutée à la somme du reste de la liste.

Dans cette exemple emblématique, la liste `l` n'est pas nécessairement détruite en mémoire, mais on la déconstruit d'un élément à chaque appel récursif en créant une autre liste `tail` qui est une partie de `l`. L'opération est de complexité linéaire.

■ **Exemple 12 — Filtrage de motif conditionnel.** Il est possible de distinguer des motifs identiques en construction mais différents dans les valeurs. On désigne cette opération par filtrage conditionnel.

```
let fcond n m b =
    match (n,b) with
        | (_,false) -> 0
        | (k, true) when k = m -> 1
        | (k, true) -> 2;;
```

■ **Exemple 13 — Filtrage non exhaustif.** Il n'est pas rare d'oublier de lister un motif possible. Dans ce cas, OCaml le signale lors de l'évaluation. Par exemple, le code suivant ne filtre pas tous les motifs de manière exhaustive :

```
let fcond n m b =
    match (n,b) with
        | (k, true) when k = m -> 1
        | (k, true) -> 2;;
```

C'est pourquoi l'avertissement `partial-match` suivante est émis par l'interprète OCaml :

```
  Warning 8 [partial-match]: this pattern-matching is not exhaustive.
        |Here is an example of a case that is not matched:
        |(_, false)
```

O Les objets immuables comme les listes nécessitent donc un espace mémoire adapté : il faut être capable de créer des listes à volonté. Le Garbage Collector, en cours d'exécution du programme, collecte les objets (comme les listes) non utilisés puis libère l'espace mémoire associé. Il garantit ainsi un exécution correcte, sans encombrement mémoire dû à la création multiple d'objets par les fonctions et les constructeurs.

K Références et programmation impérative en OCaml

OCaml est un langage mutliparadigme et on peut programmer de manière impérative avec effet. On peut regretter une certaine lourdeur de la syntaxe par rapport à Python mais les exigences des deux langages n'ont rien à voir non plus : Python est un langage laxiste [2], OCaml ne l'est pas [3].

O Le mécanisme pour pouvoir modifier une variable est nommé **référence**. La référence en OCaml est ce qui permet de manipuler une variable comme on l'entend dans les langages impératifs, c'est à dire une variable **muable**.

2. ce qui est pratique pour le prototypage d'applications et le calcul numérique
3. ce qui est important pour le développement d'applications fiables, robustes et performantes.

Pour référencer une variable, il suffit de le déclarer avec le mot-clef `ref` suivi de la valeur d'initialisation. Pour utiliser la valeur d'une référence, il faut la précéder d'un point d'exclamation !. Enfin, pour affecter une nouvelle valeur à une variable, il faut utiliser l'opérateur `:=`.

■ **Exemple 14 — Déclaration, initialisation et utilisation d'une référence en OCaml.** Lorsqu'on affecte une nouvelle valeur à une référence en OCaml, l'opération renvoie `unit`. **Une affectation est un effet de bord**.

```
>>> let x = ref 4;;
val x : int ref = {contents = 4}
>>> let y = 3 + !x;
val y : int = 7
>>> x := 0;;
- : unit = ()
>>> x
- : int ref = {contents = 0}
>>> !x
-: int = 0
>>> y
- : int = 7
```

OCaml propose également les structures itératives `for` et `while`. Les tableaux (Array) sont des structures impératives, tout comme les dictionnaires.

■ **Exemple 15 — Boucle et tableau en OCaml.** Dans cet exemple, un tableau initialisé à zéro est créé. Puis ses valeurs sont modifiées en utilisant une boucle `for`. On notera que l'accès à un élément su tableau se fait par `.(i)` et l'affectation de la nouvelle valeur par `<-`.

```
>>> a
- : int array = [|0; 1; 2; 3; 4|]
>>> let a = Array.make 5 0;
val a : int array = [|0; 0; 0; 0; 0|]
>>> for i = 0 to (Array.length a - 1) do a.(i) <- a.(i) + i done;;
- : unit = ()
>>> a
- : int array = [|0; 1; 2; 3; 4|]
```

O En OCaml, chaque type possède ses propres opérateurs et fonctions. Le compilateur peut générer facilement des messages compréhensibles. L'approche est rigoureuse.

O Même si les boucles existent en OCaml, si on veut s'en tenir au paradigme fonctionnel du langage, il est préférable de les éviter car celles-ci s'accompagnent le plus souvent de références et d'effets de bords. Dans ce cadre, on lui préfère une approche récursive.

L Synthèse

O En OCaml tout est expression à évaluer. Les types revêtent une importance capitale. On distingue :

- les types simples (`int`, `float`, `bool`, `char`, `string`)
- les types algébriques :
  - les types sommes `Pique | Trefle`
  - les types produits `int*char` et les enregistrrements `ey : int, value: floatkey : int , value: float`.

Ces types sont inférés automatiquement par l'interprète OCaml. Ils peuvent être récursifs. Les variables `let x ...` sont immuables. Le mécanisme des références permet de contourner cette limitation.

O En ce qui concerne les fonctions OCaml :

- Les fonctions en OCaml renvoient la dernière valeur calculée. Il n'existe par de mot-clef `return` comme en Python.
- Les paramètres formels des fonctions ne sont pas délimités par des parenthèses.
- Les types des paramètres d'entrée et de sortie sont inférés automatiquement.
- La signature de la fonction est systématiquement calculée par OCaml.

O Le filtrage de motif est un outil central qui couplé aux types algébriques et aux ensembles inductifs permet l'écriture de codes puissants, expressifs et lisibles.