Premiers programmes en OCaml

Guillaume Rousseau
MPSI Lycée Pierre de Fermat
guillaume.rousseau@ac-toulouse.fr

12 février 2025

Introduction

OCaml est un langage de programmation développé dans les années 1980-1990, en France, par des chercheurs et chercheuses de l'INRIA [1].

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 Machine virtuelle et terminal

On peut programmer en OCaml de manière assez similaire à Python, en passant par un interpréteur, une console dans laquelle on peut rentrer des lignes de code qui sont exécutées. L'interpréteur OCaml que l'on utilisera est **utop**, il dispose d'un historique, de coloration syntaxique, et d'un système d'auto-complétion qui rendent la vie plus facile.

Les ordinateurs du lycée sont sous Windows, mais nous allons programmer sous un système d'exploitation de la famille **Linux**, qui est aussi celle utilisée lors des concours. Le logiciel **VMWare** permet d'utiliser des machines virtuelles, c'est à dire de simuler un ordinateur sous n'importe quel système d'exploitation. Nous allons utiliser ce logiciel dans Windows pour simuler un ordinateur sous Linux. Lorsque vous ouvrez VMWare, vous pouvez lancer la machine virtuelle "NonOS 2024". Vous vous connectez alors à l'unique session proposée, avec le mot de passe "concours".

Là où sous Windows on interagit avec l'ordinateur principalement via la souris, en cliquant sur les applications et les fichiers que l'on utilise, sous Linux il est très courant d'utiliser un **terminal**, une interface exclusivement textuelle dans laquelle on tape des commandes. L'interpréteur utop se lance via le terminal.

Question 1. Lancez un terminal en cliquant sur l'icône  de la barre en bas de l'écran.

Question 2. Tapez la commande `ls` : cette commande affiche le contenu du dossier où vous vous trouvez actuellement.

Question 3. Fermez le terminal, et ouvrez l'explorateur de fichiers classique (l'icône d'armoire à dossiers) : vous pouvez vous y déplacer comme sous Windows.

Question 4. Créez un dossier avec le clic-droit, et accédez-y. Faîtes un clic-droit dans l'explorateur de fichiers, et sélectionnez "ouvrir un terminal ici" : le terminal que vous venez d'ouvrir se situe directement dans le dossier que vous avez créé.

1. Institut national de recherche en informatique et en automatique

B Constantes

Question 5. Lancer utop avec la commande `utop`.

Remarque 1. Dans utop, à tout moment, on peut taper CTRL+C pour arrêter la ligne actuelle (en cas de boucle infinie par exemple). On peut aussi taper CTRL+D pour quitter utop.

On peut voir l'interpréteur utop comme une calculatrice, dans laquelle on tape des expressions qui sont évaluées. Pour signaler la fin d'une expression, il faut taper `;;`, appuyer sur entrée ne suffit pas (ce qui est pratique : on pourra taper du code sur plusieurs lignes).

Question 6. Lancer utop avec la commande `utop`.

Question 7. Taper les expressions suivantes (sans oublier `;;` à chaque fois) :

- `5`
- `5.67`
- `true`
- `'h'` (avec guillemets simples)
- `"bonjour"` (avec guillemets doubles)
- `()`

Lorsque l'on tape une valeur, OCaml l'évalue, et donc nous l'affiche à nouveau. On remarque qu'OCaml nous donne aussi le **type** des valeurs. Les types de base sont `int`, `float`, `bool`, `char`, `string` et `unit`.

C Opérateurs

Comme tous les langages de programmation, OCaml possède des opérateurs sur les types de base : additions, soustractions, tests d'égalité, d'inégalité, opérations booléennes, etc... La concaténation de chaînes de caractères se fait avec `^`.

Question 8. Taper les expressions suivantes :

- `3+8`
- `5 = 2+3`
- `9.36 < 1.2`
- `not true`
- `true && false`
- `true || false`
- `"bonjour "^ "tout le monde"`

Question 9. Taper les expressions suivantes :

- `3.65 + 9`
- `3.65 + 2.35`

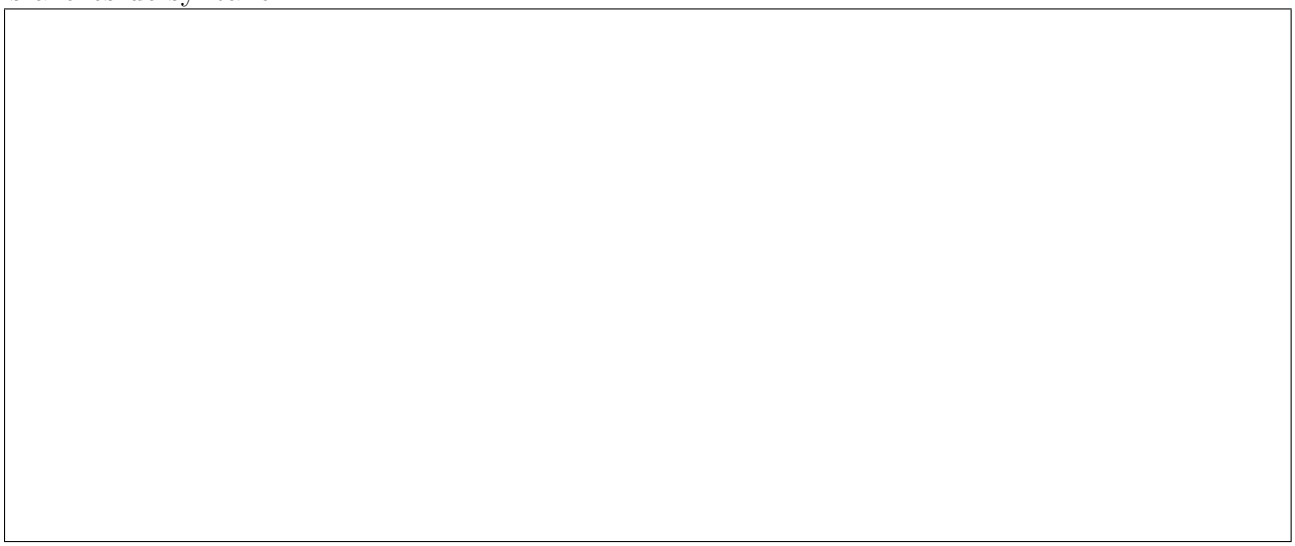
On constate que les règles de typage sont strictes. En particulier, les `int` et les `float` ne peuvent pas être mélangés : ils disposent même d'opérateurs distincts ! Pour additionner des flottants, on doit utiliser l'opérateur `+.`, idem pour la multiplication, la division, la soustraction.

Question 10. Taper les expressions suivantes :

- `2.0 +. 0.5`
- `5. /. 2.`

D Arbre de syntaxe

Un programme OCaml est 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 :



E Fonctions

Introduisons nos premières fonctions : `float_of_int` et `int_of_float`. Celles-ci permettent de transformer un entier en flottant, et inversement. En OCaml, l'application de fonction se fait en écrivant la fonction puis l'argument, séparés d'un espace, **sans besoin de parenthèses !**

Question 11. Taper :

- `float_of_int 3`
- `int_of_float 6.3`
- `float_of_int 3 +. 6.21`

La dernière expression est comprise comme `(float_of_int 3)+. 6.21` et pas `float_of_int (3 +. 6.21)`, ce qui montre que l'application de fonction est **prioritaire** sur l'addition. L'application de fonction est en réalité prioritaire sur (presque) tout.

En OCaml, **les parenthèses ne servent pas à faire des appels de fonction !** Les parenthèses servent uniquement à **regrouper une expression** pour la rendre "d'un seul tenant". Il est donc inutile d'écrire `float_of_int(5)` car 5 est déjà d'un seul tenant, on peut donc écrire `float_of_int 5` sans parenthèses. En revanche, lorsque l'on écrit `float_of_int (5 + 2)`, les parenthèses sont nécessaires.

OCaml étant un langage fonctionnel, les **fonctions** sont des valeurs comme les autres :

Question 12. Taper les expressions suivantes :

- `int_of_float`
- `float_of_int`

Pour la dernière expression, l'interpréteur affiche :

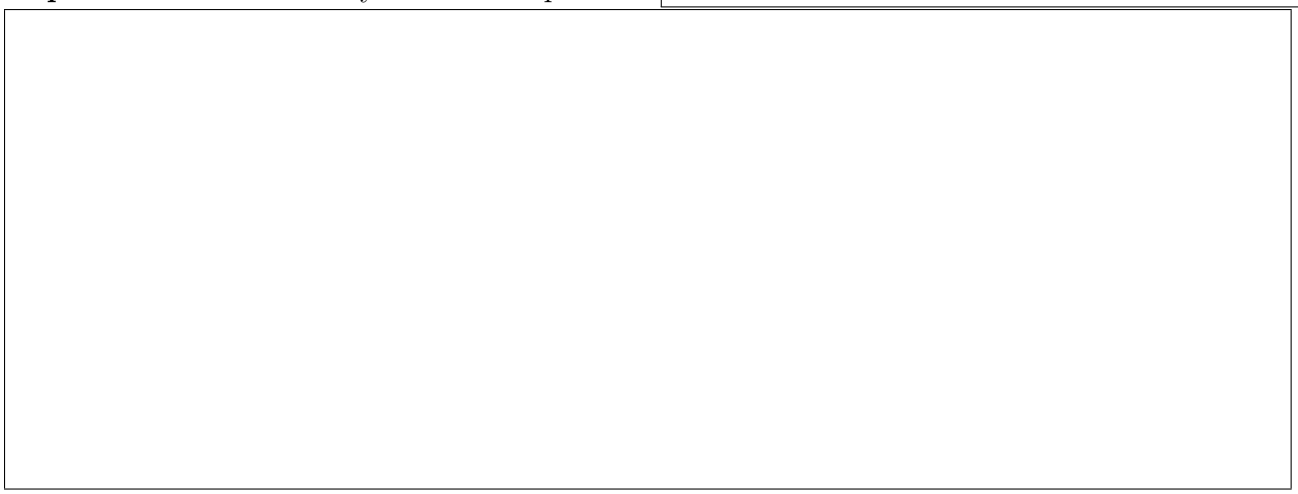
```
- : int -> float = <fun>
```

Dans un type, la flèche `->` se lit "flèche" ou "donne". On dit donc que `float_of_int` est une fonction de type "int donne float", ou "int flèche float". Une fonction de type `A -> B` prend

en entrée un élément de type A et calcule un élément de type B. Cela correspond à ce que vous pouvez noter en mathématiques $\mathcal{F}(A, B)$ ou B^A.

Dans les arbres de syntaxe, on pourra par exemple représenter les applications de fonction par le mot **App**, on peut donc traiter une application comme une sorte d'opérateur binaire.

Exemple 1. Voici l'arbre de syntaxe de l'expression `float_of_int (3 * 5 + int_of_float (2.5 *. 2.1))`



En OCaml, on peut créer ses propres fonctions. Par exemple, la fonction $f : x \mapsto x + 1$:

```
1 fun x -> x + 1
```

Notons qu'OCaml affiche :

```
- : int -> int = <fun>
```

L'interpréteur est capable de détecter automatiquement le type des objets, grâce à un système appelé l'**inférence de type**. OCaml sait donc que cette fonction doit être appliquée à un entier.

Pour appliquer cette fonction à une autre expression, par exemple $(5 + 3)$, on écrira :

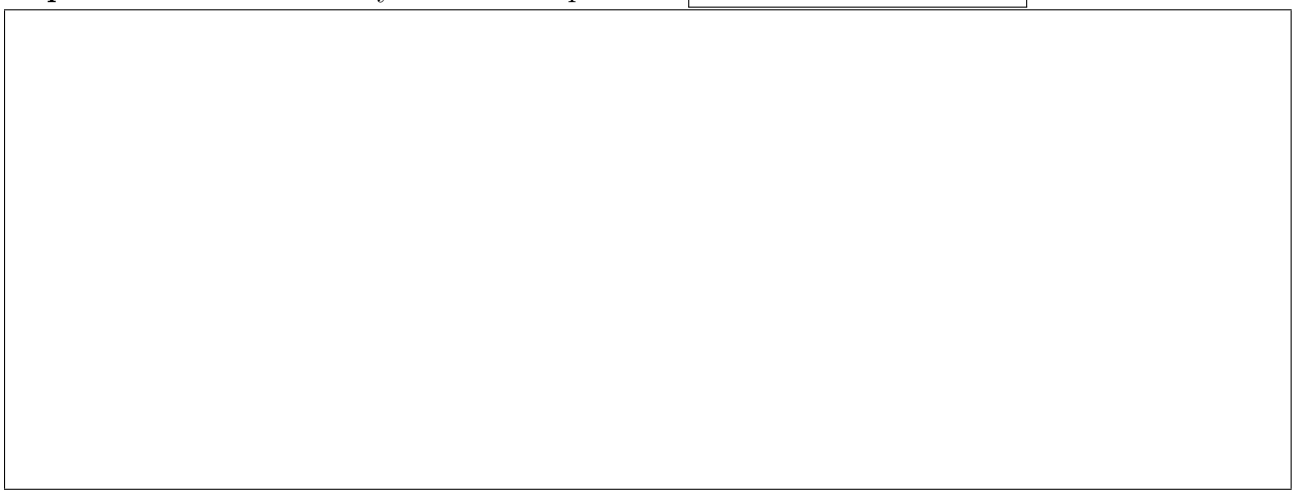
```
1 (fun x -> x + 1) (5 + 3)
```

ce qui vaut bien 9. L'expression suivante génère une erreur de typage :

```
1 (fun x -> x + 1) 0.5
```

Les fonctions créées avec le mot-clé **fun** sont appelées des **abstractions**. Dans les arbres de syntaxe, on pourra les représenter par le mot **Fun**.

Exemple 2. Voici l'arbre de syntaxe de l'expression `(fun x -> x + 1)(5 + 3)`



F Évaluation des expressions

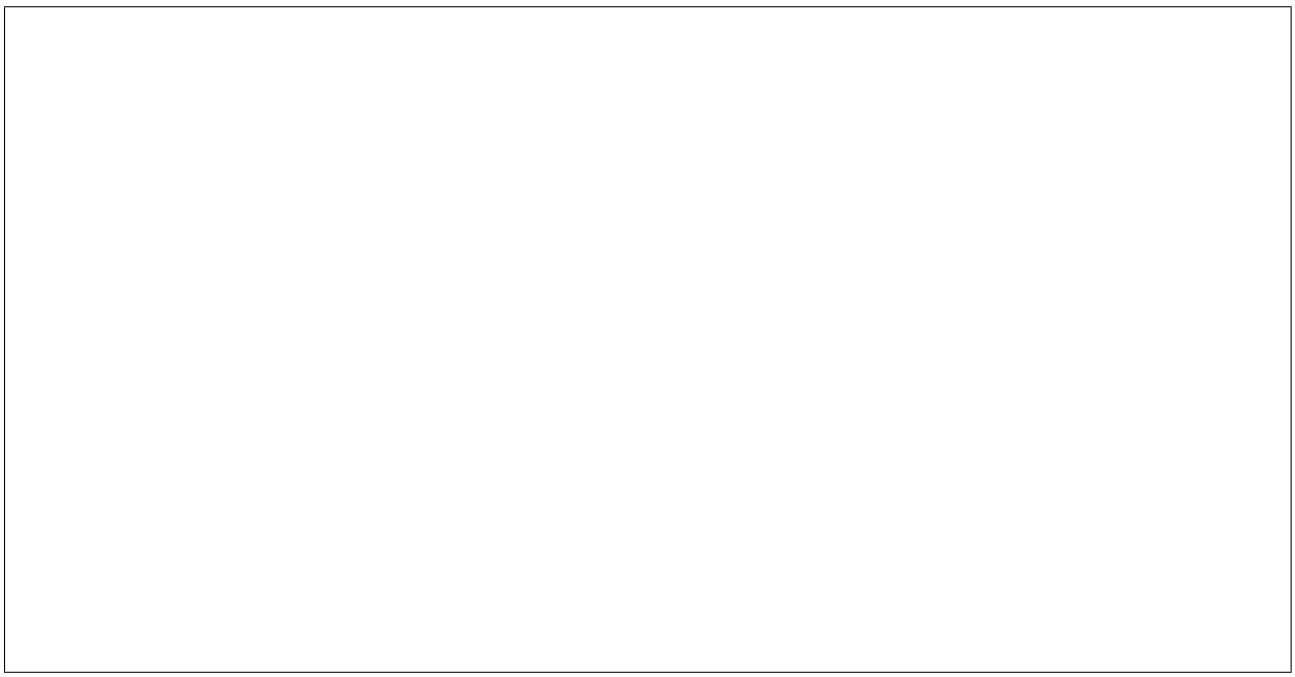
OCaml représente les expressions en mémoire par leur arbre de syntaxe, c'est donc à partir de cette représentation que la valeur est calculée. Voyons, à la main, comment fait OCaml pour évaluer un arbre de syntaxe.

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 3. 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)
```



G Fonctions d'ordre supérieur

En OCaml, les fonctions sont des valeurs comme les autres. En particulier, une fonction peut prendre en paramètre une fonction. 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) ;;
```

Définition 1. On dit qu'une fonction est ***d'ordre supérieur*** si elle prend en argument une fonction ou renvoie une fonction.

Vous avec déjà vu des fonctions d'ordre supérieur en mathématiques, en physique et en SI, même si vous avez pu utiliser d'autres termes (une "fonctionnelle", un "opérateur", etc...). Par exemple :

- 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
```

Exercice 1. Décrire les fonctions suivantes en français.

```
1 fun f -> (fun x -> f (f x)) ;;
2 fun f -> (fun x -> (f (x +. 0.00001) -. f x) /. 0.00001) ;;
3 fun k -> (fun y -> k*y) ;;
```

H 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 :

```
1 let x = 2 in
2 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 2. `let x = e1 in e2` est un raccourci pour dire `(fun x -> e2)e1;;`.

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

Question 13. 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).

Question 14. 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 3. ***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.

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 :

```
1 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.

I 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 :

```
1 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`

J Polymorphisme

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

```
1 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 2. 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.

K Tuples

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

Question 15. 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 $e_1, e_2, \ldots, e_n$ sont des expressions de types respectifs `t1, t2, ..., tn`, alors le tuple $(e_1, e_2, \ldots, e_n)$ 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.

L Récapitulatif

Pour faire des commentaires en OCaml, on les entoure par `(* *)`. Par exemple :

```
(* produit de x et y *)
let mul x y = x * y
```

Revoyons toutes les notions vues jusqu'ici :

— Les types de base : int, float, bool, char, string, unit

— Les opérations sur ces types : comme en C. Le ***non*** booléen se note `not` et les opérateurs flottants nécessitent un point : `+.` au lieu de `+` et idem pour les autres.

Type t	Opérateurs binaires pour t	Opérateurs unaires pour t	Type de l'expression composée
`int`	`+`, `-`, `*`, `/`, `mod`	`-`	`int`
`float`	`+.`, `-.`, `*.`, `/.`, `**` (puissance)	`-.`	`float`
Tous	`>`, `>=`, `=`, `<`, `<=`, `<>` (“différent de”)		`bool`
`bool`	`&&`, `||`	`not`	`bool`
`char`			
`string`	`^` (concaténation)		`string`

— On peut ajouter une variable au contexte globale avec `let x = ... ;;` et ajouter une variable dans un contexte local avec `let x = ... in ...`

— On peut définir des fonctions avec la syntaxe suivante :

```
let f x1 x2 ... xn =
  ...
```

— On applique une fonction avec :

```
f e1 e2 ... ek
```

où `e1, e2, ... ek` sont des expressions ayant des types compatibles avec la signature de la fonction `f`.

— Les parenthèses servent à encadrer les sous-expressions **et pas à appliquer les fonctions**

Voici un exemple de programme mettant en oeuvre ces différentes notions :

```
let x = 3;;
let y = 5;;
let z = 3 * 5 ;;

(* composée de f et g, i.e. la fonction h
   telle que h(x) = f(g(x)) *)
let composition f g =
  fun x -> f (g x);;

(* double de x *)
let double x = 2*x;;

let u = composition double int_of_string ;;
let a = u "12";; (* vaut 24 *)

let quadrupler = composition double double;;
let b = quadrupler 5;; (* vaut 20 *)
```

Quiz Pour chacune des expressions suivantes, prédire son type et sa valeur :

```
(* Q1 *)
let g x = 2*x in
let f x = g x + 1 in
f 5 + f 3;;


(* Q2 *)
let double x = 2*x in
let triple x = 3*x in
double (triple 5);;

(* Q3 *)
let s = "bonjour " in
let saluer x = s ^ x ^ " !!" in
let s = "salut " in
saluer "Jérémy";;

(* Q4 *)
let f x y z = (x z) (y z) in
let g x = let x = x - 1 in x * x in
let h x = fun y -> let x = y in x+1 in
f h g 3;;

(* Q5 *)
let u f (x, y) = f x y in
let g = u (fun x -> (fun y -> x y)) in
g ((fun a -> a+3), 5);;
```

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 :

```
1 let est_pair (x: int) : bool =
2   x mod 2 = 0
3
4 let composee (f: 'a -> 'b) (g: 'c -> 'a) : 'c -> 'b =
5   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. Un bon commentaire de documentation 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 :

```
1 (** MAUVAIS COMMENTAIRE **)
2 (* Multiplie deux entiers *)
3 let mult a b = ...
4
5 (** BON COMMENTAIRE **)
6 (* Renvoie le produit a*b. a et b doivent être positifs *)
7 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 :

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

où E1, E2 et E3 sont des expressions.

Pour le typage, l'expression $E1$ doit être du type `bool`, et les expressions $E2$ et $E3$ doivent être du même type T, et alors l'expression totale est de type T également.

Pour la sémantique, voici comment évaluer la valeur de `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

Comme les deux branches du if-else doivent avoir le même type, on ne peut pas avoir de if sans else. Une exception : le type `unit` :

```
if true then ();;
if x < y then print_int x;;
```

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 3. 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 4. 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

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 :

```
1 let puissance a b =
2   if b = 0 then 1
3   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*** :

```
1 let rec puissance a b =
2   if b = 0 then 1
3   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.

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.

Attention, la tête d'une liste est un élément, mais la queue d'une liste est une liste. On peut donc voir les listes OCaml comme des piles : on n'a accès qu'à l'élément au début d'une liste.

Exemple 5. Voyons des exemples de listes :

```
let l_vide = []
let l_simple = 3::6::8::[] (* comme 3 :: (6 :: (8 :: [])) *)

(* Renvoie une liste de taille n contenant
   exclusivement des 1 *)
let rec ones n = match n with
  | 0 -> []
  | _ -> 1:: ones (n-1)

let cinq_uns = ones 5
```

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

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

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 liste 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
```