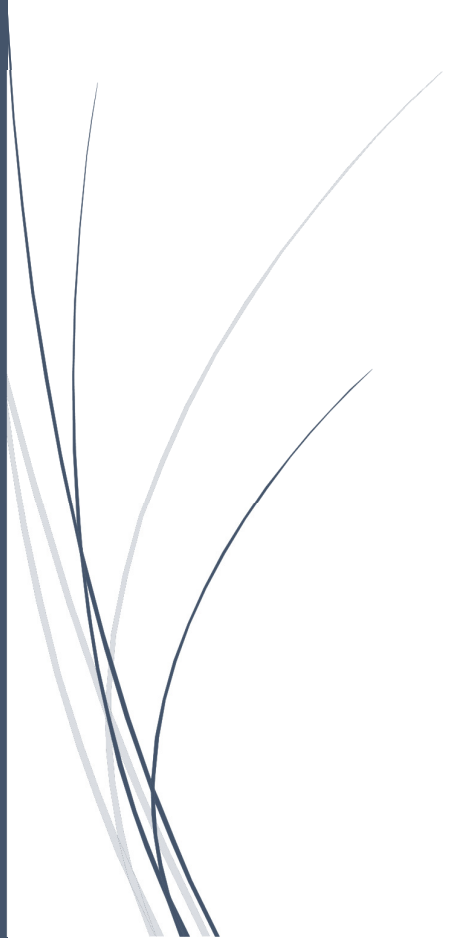
Réalisé par :
Chadli Abdelhafid

Support de cours :

Programmation Fonctionnelle



UNIVERSITE IBN KHALDOUN - TIARET
FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE
2022 - 2023

Introduction

Ce manuscrit est un nouveau polycopié d'introduction à la programmation fonctionnelle et aux fondements du lambda calcul. Il s'inscrit dans l'évolution et l'enrichissement continuel de ce sujet et il témoigne de sa vitalité et de son mûrissement.

La **programmation fonctionnelle** est un *paradigme* de programmation, c'est-à-dire un *mode de programmation* et aussi une démarche indépendante de tout langage de programmation, permettant de construire un programme résolvant un problème donné. Ce mode de programmation est fondé sur une théorie mathématique éprouvée, celle des fonctions, du *lambda-calcul* et des *systèmes de type*. Les concepts présentés sont donc fondamentaux et peuvent resservir pour répondre à d'autres questions liées à la programmation (certification de programmes, sémantique des langages, compilation, parallélisme, etc.).
La programmation fonctionnelle est de plus en plus présente et facilitée dans les langages. Même les langages traditionnels sont étendus pour permettre le mode fonctionnel comme en témoigne l'introduction es *fonctions anonymes* (ou *lambda-expressions)* dans C++ (depuis la norme C++ 11) et Java 8. Les langages fonctionnels *purs* s'imposent de plus en plus comme une alternative sérieuse aux langages traditionnels non purs.

Les parties composant ce manuscrit sont issues du programme du module "Programmation fonctionnelle" de la promotion : 1ère année Master "Génie Logiciel". Dans le chapitre 1, il s'agit d'introduire les préliminaires et concepts du domaine, ensuite nous définissons les notions de fonction simple et de fonctions d'ordre supérieur. Dans le chapitre 2, nous présentons la récursivité qui est un des points forts de la programmation fonctionnelle. Le Filtrage de Motif et la déclaration de types font l'objet du chapitre 3. Dans le chapitre 4, nous présentons les schémas de programmes pour l'optimisation de la programmation. Finalement, dans le chapitre 5, nous entamons le lambda calcul. Les exercices sont accompagnés de solutions qui sont la seule garantie que l'exercice est faisable.

Sommaire

Chapitre 1 : Introduction à la programmation fonctionnelle

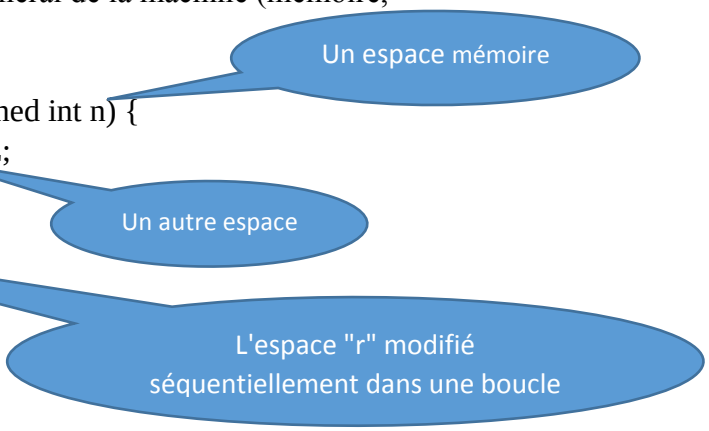
Les modèles de calcul connus jusqu'à aujourd'hui pour décrire les algorithmes sont :

a. Programmation impérative : principe

- On modifie des *variables* par des *instructions*
- Les variables sont des espaces mémoire
- Les instructions sont organisées séquentiellement
- La machine se programme par effets de bords
  (= modifications de l'état général de la machine (mémoire, vidéo, etc.))

Exemple

```
unsigned long int fact(unsigned int n) {
        unsigned long int r =1L;
          for ( ;  n>0 ;  --n)
            r *= n;
return r;
}
```



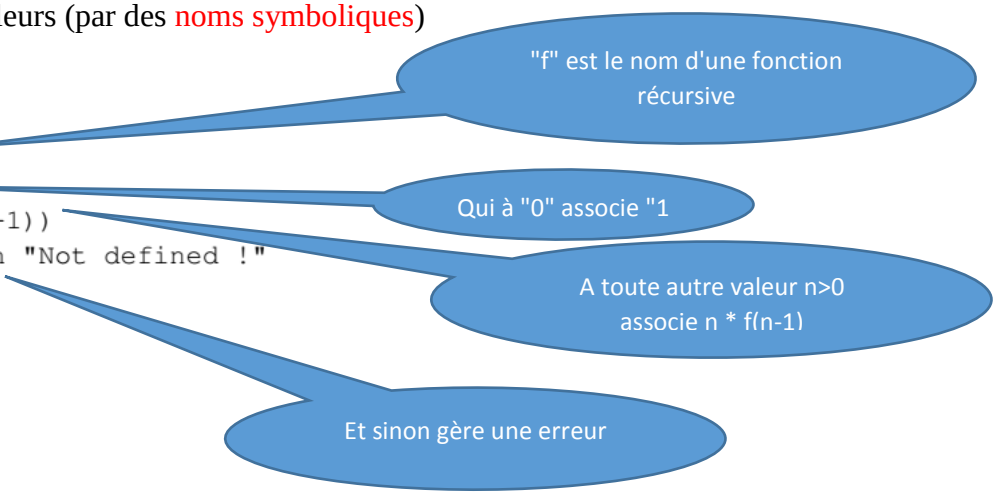
Les noms des variables sont des noms d'espaces mémoire

b. Programmation fonctionnelle : principe

- Plus d'instructions, plus d'espaces mémoire, plus d'effets de bord
- Uniquement des expressions
- Les expressions se *réduisent* en des *valeurs :* entiers, réels, booléens, fonctions
- Les fonctions sont des valeurs comme les autres
- On peut nommer les valeurs (par des noms symboliques)

Exemple:

```
let rec f = function
  0           -> 1
| n when n>0 -> n*(f (n-1))
| _           -> failwith "Not defined !"
;;
```



L'affectation n'existe plus !!

1- Les types de base

OCaml	Haskell	Scheme	Type
`-1234` `: int`	`-1234` `: Integer`	`-1234` `: integer`	entier relatif
		`1/3` `: rational`	nombre rationnel
`-12.34e-45` `: float`	`-12.34e-45` `: Double`	`-12.34e-45` `: real`	nombre réel
		`-1+2i` `: complex`	nombre complexe
`'A'` `: char`	`'A'` `: Char`	`#\A` `char`	caractère
`"hello"` `: string`	`"hello"` `: String`	`"hello"` `: string`	chaîne de caractères
`true` `: bool`	`True` `: Bool`	`#t` `: boolean`	booléen

Les entiers

Opérateurs arithmétiques : +, −, ∗, /, mod , − (préfixe)

- La division par 0 génère une exception
- Les Opérateurs de comparaisons : =,< , >, <=, >=

```
# 1;;
- : int = 1
# 1+2;;
- : int = 3
# 4/0;;
Exception: Division_by_zero.
# 4 mod 0;;
Exception: Division_by_zero.
# 99999999999* 9999999999;;
- : int = 3875819909684212737
# 4<5;;
- : bool = true
```

Les réels

Opérateurs arithmétiques : +., -., ∗., /., ∗∗, -. (préfixe)

- attention aux problèmes de précision
- sécurisés : valeurs spéciales (infinity (+∞), neg_infinity (-∞), NaN (*Not A Number*))
- la division de $x > 0$ par 0 produit infinity
- la division de $x < 0$ par 0 produit neg_infinity
- la division de 0 par 0 produit NaN

Opérateurs de comparaison : =, <, >, <=, >=

```
# 1.0;;
- : float = 1.
# 3.14/.0.;;
- : float = infinity
# -1.0/.0.;;
- : float = neg_infinity
# 0./.0.;;
- : float = nan
# -1.0/0.;;
Error: This expression has type float but an expression
         was expected of type int
# 6e34;;
- : float = 6e+34
```

Fonctions sur les réels et les entiers

- Fonctions sur les réels `float → float`
  `ceil`, `floor`, `sqrt`, `exp`, `log`, `log10`
- Fonctions trigonométriques sur les réels `float → float`
  `cos`, `sin`, `tan`, `acos`, `asin`, `atan`
- Fonctions de conversion
  - `float_of_int` : `int → float`
  - `int_of_float` : `float → int`

```
# int_of_float 3.14;;
- : int = 3
# int_of_float -3.14;;
Error: This expression has type float -> int
       but an expression was expected of type int
# int_of_float (-3.14);;
- : int = -3
# int_of_float nan;;
- : int = 0
# int_of_float neg_infinity;;
- : int = 0
```

Les caractères

```
# 'a';;
- : char = 'a'
# int_of_char;;
- : char -> int = <fun>
# int_of_char 'a';;
- : int = 97
# char_of_int;;
- : int -> char = <fun>
# char_of_int 97;;
- : char = 'a'
# '\097';;
- : char = 'a';;
# '\n';;
- : char = '\n';;
```

Le module `Char` contient des fonctions qui permettent de manipuler des caractères :

- `Char.lowercase_ascii: char → char`
  équivalent minuscule d'un caractère ;
- `Char.uppercase_ascii: char → char`
  équivalent majuscule d'un caractère ;
- et d'autres...

L'ordre sur les caractères est l'ordre ASCII-7.

Les chaînes de caractères

```
# "Bonjour monde !";;
- : string = "Bonjour monde !"
# string_of_float;
- : float -> string = <fun>
# (string_of_float 3.14)^" est un nombre formidable !";;
- : string = "3.14 est un nombre formidable !"
# int_of_string;;
- : string -> int = <fun>
# int_of_string "18";;
- : int = 18
# "Bonjour".[5];;
- : char = 'u'
```

Il existe aussi des conversions à partir/vers les autres types élémentaires :

- Chaînes ↔ entiers
  - `string_of_int` : `int` → `string`
  - `int_of_string` : `string` → `int`
- Chaînes ↔ réels
  - `string_of_float` : `float` → `string`
  - `float_of_string` : `string` → `float`
- Chaînes ↔ booléens
  - `string_of_bool`
  - `bool_of_string`

Le module `String` contient de nombreuses autres fonctions de manipulation de chaînes.
Les caractères sont indicés à partir de 0. Une exception est générée en cas d'erreur.

- longueur :
  `String.length` : `string` → `int`

```
# String.length "J'aime les poires";;
- : int = 17
```

- sous-chaine :
  `String.sub` : `string` → `int` → `int` → `string`

```
# String.sub "J'aime les poires" 7 3;;
- : string = "les"
```

- suppression des espacements au début et à la fin
  `String.trim` : `string` → `string`

```
# String.trim "  J'aime les poires   ";;
- : string = "J'aime les poires"
```

- position de la première occurrence d'un caractère : `String.index` : `string` → `char` → `int`

```
# String.index "J'aime les poires" 'i';;
- : int = 3
#  String.index "J'aime les poires" 'x';;
Exception: Not_found.
```

- position de la dernière occurrence d'un caractère : `String.rindex` : `string` → `char` → `int`
- `String.index_from` et `String.rindex_from` : même chose à partir d'une position donnée

- `String.contains`, `String.contains_from`, `String.rcontains_from` :
  même chose mais retourne un booléen plutôt qu'une position
- `String.uppercase_ascii`, `String.lowercase_ascii` : `string → string`
  mise en majuscule/minuscule

```
# String.uppercase_ascii "J'aime les poires";;
- : string = "J'AIME LES POIRES"
```

- `String.capitalize_ascii`, `String.uncapitalize_ascii` : `string → string`
  mise en majuscule/minuscule de la première lettre

```
# String.capitalize_ascii "J'aime les poires";;
- : string = "J'aime les poires"
# String.uncapitalize_ascii "J'aime les poires";;
- : string = "j'aime les poires"
```

Les Booléens

- Valeurs `true` et `false`
- Opérateurs `not`, `&&`, `||`
- Opérateurs de comparaison : polymorphes, mais les opérandes doivent avoir le même type
  - = : égalité structurelle,
  - == : égalité physique (les opérandes occupent le même espace mémoire),
  - <> (négation de =), != (négation de ==),
  - <, >, <=, >=.

```
# "bonjour"=="bonjour";;
- : bool = false
# "bonjour"="bonjour";;
- : bool = true
# 3.14=3.14;;
- : bool = true
# 3.14==3.14;;
- : bool = false
# 3==3;;
- : bool = true
# 3=3;;
- : bool = true
```

Le type unit

Le type *unit* est particulièrement simple, puisqu'il ne comporte qu'une seule valeur, notée (). Pour comprendre sa raison d'être, imaginons que l'on veuille afficher une chaîne de caractère à l'écran : nous n'avons rien à calculer, seulement modifier l'environnement (c'est à dire réaliser ce qu'on appelle un *effet de bord*). Or un langage fonctionnel ne peut que définir et appliquer des fonctions. Voilà pourquoi la fonction **print_string** est de type *string -> unit* : elle renvoie le résultat *void*, tout en réalisant au passage un effet de bord.

```
# print_string "Hello World" ;;
Hello World- : unit = ()
```

Autre exemple, la fonction **print_newline** est de type *unit -> unit* et a pour effet de passer à la ligne. Pour l'utiliser, il ne faut donc pas oublier son argument (qui ne peut qu'être le *void*) :

```
# print_newline () ;;

- : unit = ()
```

2- Les opérateurs

Le tableau suivant regroupe les opérateurs arithmétiques de chaque langage. Dans ce tableau, n1 et n2 sont des entiers, f1 et f2 sont des réels, x1 et x2 sont des entiers ou des réels.

OPÉRATEURS ARITHMÉTIQUES.

OCaml	Haskell	Scheme	Opération
n1+n2 f1+.f2	x1+x2	(+ x1 x2)	Addition
n1-n2 f1-.f2	x1-x2	(- x1 x2)	Soustraction
n1*n2 f1*.f2	x1*x2	(* x1 x2)	Multiplication
f1/.f2	x1/x2	(/ x1 x2)	Division
n1 mod n2	n1 `mod` n2	(modulo n1 n2)	Modulo
n1/n2	n1 `div` n2	(quotient n1 n2)	Division entière
-n1 -.f1	-x1	(- x1)	Opposé
n1+1 f1+.1	x1+1	(1+ x1)	Incrémentation
n1-1 f1-.1	x1-1	(-1+ x1)	Décrémentation
f1**f2	x1^x2	(expt x1 x2)	Élévation à la puissance

Le tableau Suivant contient les autres opérateurs. Dans ce tableau, b1 et b2 sont des booléens, s1 et s2 sont des chaînes, l1 et l2 sont des listes, o1 et o2 sont des objets quelconques.

AUTRES OPÉRATEURS.

OCaml	Haskell	Scheme	Opération
l1@l2	l1++l2	(append l1 l2)	Concaténation de listes
s1^s2	s1++s2	(string-append s1 s2)	Concaténation de chaînes
o1=o2	o1==o2	(equal? o1 o2)	Égalité structurelle
o1<>o2	o1/=o2	(not (equal? o1 o2))	Inégalité structurelle
o1<o2	o1<o2	(< x1 x2)	"Plus petit que"
o1<=o2	o1<=o2	(<= x1 x2)	"Plus petit ou égal à"
o1>o2	o1>o2	(> x1 x2)	"Plus grand que"
o1>=o2	o1>=o2	(>= x1 x2)	"Plus grand ou égal à"
not b1	not b1	(not b1)	"Non" logique
b1 && b2	b1 && b2	(and b1 b2)	"Et" logique
b1 || b2	b1 || b2	(or b1 b2)	"Ou" logique

3- Déclarations globales

Un identificateur est déclaré globalement par le mot clef "let":

```
# let x = 7;;
val x : int = 7

# x+2;;
- : int = 9
```

4- Déclarations locales

Un identificateur est déclaré localement par la construction
let ... in:

```
#let y = 5 in 3 *y;
 - : int =15

#y;;
Unbound value y
```

Une déclaration locale est visible seulement dans l'expression qui suit le "in".

Remarque : Il y a une différence de taille entre ces deux constructions :

- la première est une déclaration et ne retourne pas de valeur.
- la deuxième est une expression et retourne donc une valeur.

```
# 2 + (let y = 5 in 3 *y);;
- : int = 17

# 2 + (let y = 15);;
Syntax error
```

5- Expression conditionnelle

Expression dont le résultat dépend de la valeur de la condition (booléenne).
Une expression conditionnelle a toujours deux branches: les expressions dans ces branches doivent être de même type.

```
# if true then "vrai" else "faux";;
- : string = "vrai"


# let x = 7 in if x>2 then 1 else true;;
Characters 32-36:
  let x = 7 in if x>2 then 1 else true;;
                               ^^^^
This expression has type bool but is here used
 with type int
```

6- Notion de fonction

En mathématiques, une fonction "*f*" fait correspondre à chaque élément *x* d'un ensemble *A*, appelé **domaine** de la fonction, un élément unique, appelé **image** de *x* par *f*, dans un autre ensemble *B*. On appelle **codomaine** l'ensemble des images (qui est inclus dans *B*). L'image de *x* par *f* est notée *f* (*x*), et l'on note :

$$f : A \to B$$
$$x \mapsto f(x)$$

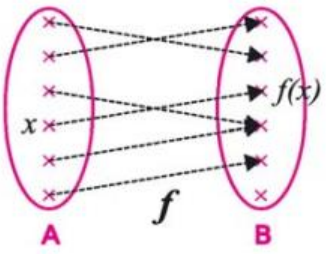


Figure 1: Une fonction *f* de *A* dans *B*

Exemple: La fonction "*suiv*" qui à tout nombre entier naturel associe son successeur. Son domaine est l'ensemble des entiers naturels **N** et son codomaine est **N***.

Règle de correspondance ou équation
Une manière élégante de définir une fonction consiste à utiliser une variable et une **équation** pour établir une règle de correspondance entre tout élément du domaine et son image dans le codomaine.

Exemple Une telle équation est :

$$suiv(n) = n + 1$$

où :

– *suiv* est le nom de la fonction,

– *n* est une variable c.-à-d. un paramètre formel pour désigner un élément quelconque du domaine de la fonction,

– *suiv*(*n*) note l'image de *n* par la fonction *suiv*.

Cette équation signifie que, pour tout élément *n* appartenant au domaine de la fonction, l'image de *n* par la fonction est égale au résultat de l'addition de *n* et 1.

Mon premier programme fonctionnel
La plupart des langages fonctionnels permettent de définir une fonction en utilisant une équation. Voici une définition de la fonction *suiv* dans les différents langages :

Une définition de la fonction *suiv*

OCaml	Haskell	Scheme
`let suiv(n) = n+1;;`	`suiv(n) = n+1`	`(define (suiv n) (+ n 1))`

En OCaml, une définition commence par le *mot-clé* "let" et toutes les phrases se terminent par ;;

Exemple En OCaml, le programmeur peut exécuter le programme précédent avec l'interpréteur, en tapant la définition de la fonction *suiv*, puis l'expression de l'application de cette fonction à un argument.

```
# let suiv(n) = n+1;;
val suiv : int -> int = <fun>
# suiv(3);;
- : int = 4
```

6-1 Fonctions partielles

En mathématiques, une *fonction partielle* sur un ensemble *E* est une fonction dont le domaine est une partie de *E*. En programmation fonctionnelle, la plupart des fonctions que le programmeur est amené à définir sont seulement partielles (ne sont pas totales) sur l'ensemble correspondant au type de l'argument.

Exemple : La fonction inverse "1/x" a pour domaine **R***. Elle est partielle sur **R** qui en OCaml est représenté par le type float.

Exemple La fonction inverse peut être définie ainsi en OCaml :

```
let inverse(x) = if x<>0.0 then 1.0/.x else failwith "bottom";;
```

6-2 Fonctions plus complexes

On peut définir des fonctions plus complexes par composition de fonctions ou en utilisant des *n-uplets*.

Composition de fonctions

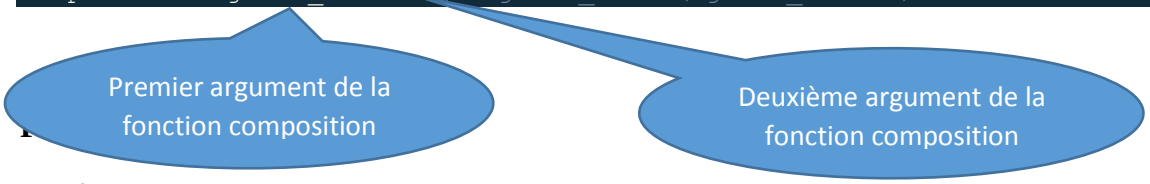
La composition de fonctions est un procédé qui consiste à définir une fonction en se servant d'une ou plusieurs autres fonctions. Par exemple, le résultat d'une fonction peut s'exprimer en utilisant l'application d'une autre fonction à un argument.

Exemple :

```
let composition (f : int -> int) (x : int) : int = f (f x)
```

composition est une fonction d'ordre supérieur : elle prend une autre fonction comme premier argument. On peut l'utiliser de cette façon par exemple :

```
let ajoute_deux (x : int) : int = x + 2

composition ajoute_deux 4 (* ajoute_deux (ajoute_deux 4) = 2 + 2 + 4 = 8 *)
```



Une fonction peut prendre plusieurs arguments:

Exemple :

```
#  let moyenne x y = (x+.y)/. 2.0;;
val moyenne : float -> float -> float = <fun>
```

La fonction:

let f x_1 x_2 . . . x_n = *corps*

possède n arguments x_1, x_2, . . . x_n et un résultat calculé par *corps*. Son type est de la forme:

$$t_1 \rightarrow t_2 \ldots \rightarrow t_n \rightarrow t_{res}$$

où t_i est le type de l'argument x_i, et t_{res} est le type du résultat calculé par *corps*.

Exemple:

```
# let somme x y = x + y;;
val somme : int -> int -> int = <fun>

# somme 3 4;;
- : int = 7
```

N-uplets

Un n-uplet est un “paquet” de n valeurs *v*1, *v*2, . . . *vn* separées par des virgules.
Exemple: (1,true) est un 2-uplet; ("bonjour", 5, ’c’) est un triplet.

```
# let a = (1, true);;
val a : int * bool = (1, true)

# let livre = ("Paroles", "Prevert, Jacques", 1932);;
val livre : string * string * int =("Paroles", "Prevert, Jaques " , 1932)
```

Un n-uplet permet de mettre dans un “paquet” autant de valeurs que l’on veut. Cela est pratique, si une fonction doit renvoyer “plusieurs” résultats:

```
# let f x y = (x+y, x*y);;
val f : int -> int -> int * int = <fun>

# f 2 3;;
- : int * int = (5, 6)

# let division_euclidienne x y = (x/y, x mod y);;
val division_euclidienne : int -> int -> int * int = <fu

# division_euclidienne 5 2;;
- : int * int = (2, 1)
```

6-3 Fonctions avec n-uplet d’arguments

Une autre manière de définir une fonction à plusieurs arguments est de les mettre tous dans un n-uplet.

```
# let somme (x,y) = x+y;;
 val somme : int * int -> int = <fun>
```

Attention: cette fonction ne possède *qu’un seul* argument, qui *est ici une paire*. Parmi les appels suivants, le premier est correct, alors que le deuxième ne l’est pas:

```
# somme (2,3);;
- : int = 5

# somme 2 3;;
This function is applied to too many arguments
```

6-4 Fonctions anonymes

Une fonction est donc un objet typé, qui en tant que tel peut être calculé, passé en argument, retourné en résultat. Pour ce faire, on peut construire une valeur fonctionnelle anonyme en suivant la syntaxe **function x → expr**.

Exemple :

```
# function x -> 2*x+1;;
- : int -> int = <fun>
```

6-5 Polymorphisme

On a pu constater que Caml est pourvu d'une reconnaissance automatique de type : par exemple, c'est la présence de l'entier **1** et de l'opérateur entier **+** qui permet d'associer à la fonction **function n -> n + 1** le type *int -> int*.
Mais il peut arriver qu'une fonction puisse s'appliquer indifféremment à tous les types ; on dit dans ce cas que cette fonction est *polymorphe*. On utilise alors les symboles *'a*, *'b*, *'c*,. . . pour désigner des types quelconques.
Revenons par exemple sur les fonctions **fst** et **snd** qui agissent sur les paires. Leur définition est évidente, et n'imposent aucune contrainte de typage :

```
# let fst (x, y) = x ;;
fst : 'a * 'b -> 'a = <fun>
# let snd (x, y) = y ;;
snd : 'a * 'b -> 'b = <fun>
```

Nous avons vu que les opérations algébriques, l'addition par exemple, ne sont pas polymorphes : l'opérateur infixe **+** est de type *int -> int* alors que **+.** est de type *float -> float*. En revanche, les opérations de comparaison telle l'égalité (notée **=**) le sont 4. On peut utiliser le même opérateur pour comparer entre eux des entiers, des nombres complexes ou des chaînes de caractères (et plus généralement tout objet pour lesquels la comparaison a un sens), car c'est leur représentation machine qui est comparée.

```
# 1 = 1 ;;
- : bool = true
# 1.0 = 2.0 ;;
- : bool = false
# prefix = ;;
- : 'a -> 'a -> bool = <fun>
```

Les autres opérateurs de comparaison polymorphes sont **<>** (la négation de l'égalité), **<**, **>**, **<=** (inférieur ou égal), **>=** (supérieur ou égal).

Exercice 1

Prévoir le résultat fourni par l'interpréteur OCaml après chacune des commandes suivantes :

```
#let x = 2;;
#let x = 3
 in  let y = x + 1
      in x + y;;
#let x = 3 and y = x + 1
 in  x + y;;
```

Exercice 2

Pourquoi la deuxième et la troisième commande ne fournissent-elles pas le même résultat ?
Expliquer à présent le comportement suivant :

```
#let x = 3;;
x : int = 3
#let f y = y + x;;
f : int -> int = <fun>
#f 2;;
- : int = 5
#let x = 0;;
x : int = 0
#f 2;;
- : int = 5
```

Exercice 3

Ajouter les parenthèses n´nécessaires pour que le code ci-dessous compile :

```
let somme x y = x + y;;
somme  somme somme 2 3 4 somme 2 somme 3 4  ;;
```

Exercice 4

Ecrire une fonction qui teste si son argument est pair. On indique que l'expression *a mod b* retourne le reste dans la division entière de *a* par *b*.
Remarque : **qui teste** signifie la plupart du temps **qui retourne true si la condition est vraie et false sinon**.

Exercice 5

Ecrire une fonction qui retourne -1 si son argument est négatif, 0 si c'est 0 et 1 si l'argument est positif.

Exercice 6

Ecrire une fonction qui calcule le volume d'une sphère

Exercice 7

Ecrire une fonction qui prend en argument 3 entiers et retourne le plus grand de ces entiers.

Exercice 8

Ecrire une fonction qui ajoute de part et d'autre d'une chaine de caractères quelconque la chaine \!!" appelée cadre.
Modifier la fonction de manière `a ce que le cadre devienne aussi un argument de la fonction (on dit que l'on *abstrait* le cadre).
On veut maintenant encadrer dissymétriquement. Introduire les paramètres n´nécessaires et écrire la nouvelle fonction

Chapitre 2 : Récursivité

1-- Définition Récursive

Considérons par exemple la fonction *somme* qui à tout nombre entier naturel n, associe la somme $0 + 1 + 2 + ... + n$ des entiers de 0 jusqu'à n.
Nous pouvons définir cette fonction cas par cas en considérant deux cas :
le cas où $n = 0$ et alors le résultat est 0,
et le cas où $n > 0$, et alors on peut remarquer que $somme(n) = 0 + 1 + 2 + ... + n$ peut être définie à partir de $somme\ (n-1) = 0 + 1 + 2 + ... + (n-1)$: c'est le résultat de l'addition de $somme\ (n-1)$ avec n. Ce raisonnement mène à ces équations :

$$\begin{cases} somme(n) = 0 & \text{si } n = 0 \\ somme(n) = somme(n-1) + n & \text{si } n > 0 \end{cases}$$

qui constituent une définition de la fonction *somme.* Cette définition a ceci de remarquable qu'elle fait appel à la fonction qu'elle définit. On parle de *définition récursive.*

Question : comment peut-on définir la fonction somme explicitement ?

En Ocaml, la construction **let rec** permet des définitions récursives, la syntaxe est :
let rec nomfonction p1 p2 ... = expression ;;

Exemple : La fonction somme peut s'écrire de cette façon en OCaml

```
let rec somme(n) = if n=0 then 0
                   else if n>0 then somme(n-1)+n
                   else failwith "bottom";;
```

Une fonction récursive peut être écrite de différentes façons. C'est une fonction qui fait appel à elle-même.
Dans la définition des fonctions récursives, on fait usage de la structure de contrôle conditionnelle : **if** condition **then** expr2 **else** expr3.

Exemple :
La fonction factorielle peut être définie comme une fonction récursive :
Factorielle(x)= x * factorielle(x-1), on arrête le calcul lorsque x est égal à 0.

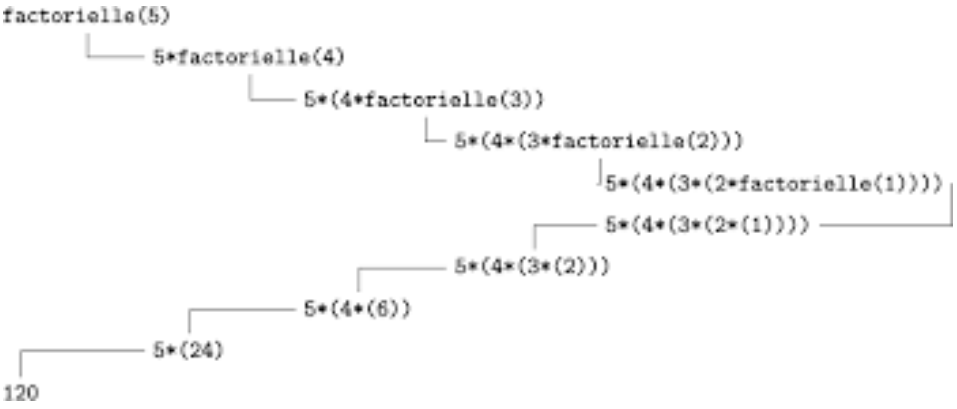
```
# let rec factorielle x =
if x = 0                    (*test d'arrêt*)
then 1                      (*partie non récursive*)
else x * factorielle(x-1)   (*partie récursive*);;

val factorielle : int -> int = <fun>
# factorielle 10 ;;
- : int = 3628800
```

On peut remarquer que cette fonction (factorielle) risque de ne pas terminer si son argument est strictement négatif. De même, les déclarations locales peuvent être récursives. Cette nouvelle définition de la fonction fact teste la validité de l'argument avant d'effectuer le calcul du factoriel définie par une fonction locale factorielle.

```
# let fact n =
let rec factorielle x =
if x = 0 then 1 else x * factorielle(x-1) in
if n >= 0 then factorielle n else 0 ;;
val fact : int -> int = <fun>
# fact 8 ;;
- : int = 40320
# fact (-4) ;;
- : int = 0
```

Ou bien

```
#  let rec factorielle x = if x = 0 then 1 else if x > 0 then x * factorielle (x-1) else 0
# factorielle 5 ;;
-: int = 120
```



À chaque appel récursif, il faut empiler l'ensemble des paramètres. Dans certains cas (récursivité terminale), les appels récursifs peuvent être transformés en une boucle.

2 -- Récursivité Terminale

Malgré leurs progrès constants, les compilateurs ont des limites, et la récursivité entraîne souvent des opérations supplémentaires (inhérentes à la gestion de la récursivité et non au problème à résoudre) que le compilateur a des difficultés à éliminer. Il existe cependant une forme de récursivité qui n'entraîne pas d'opérations supplémentaires et qui permet au compilateur d'atteindre l'optimal. Cette forme est reconnaissable lorsqu'il n'y a aucune opération qui suit un appel récursif.

Définition

Une fonction est récursive terminale (en anglais : tail-recursive) si elle est récursive et si l'appel récursif est le dernier terme à évaluer.

Les fonctions récursives terminales sont considérées comme meilleures que les fonctions non terminales puisqu'une récursion terminale peut être optimisé par le

compilateur.
L'idée utilisée par les compilateurs pour optimiser les fonctions récursives est simple, puisque l'appel récursif est la dernière instruction, il n'y a plus rien à faire dans la fonction courante, donc sauvegarder le cadre de la pile de la fonction courante n'est d'aucune utilité.

Considérons la fonction suivante pour calculer la factorielle de N.
C'est une fonction récursive non-terminale. Bien qu'elle ressemble à une fonction récursive terminale à première vue. Si on regarde de plus près, nous pouvons voir que la valeur retournée par factorielle (n-1) est utilisée dans factorielle(n), de sorte que l'appel de factorielle(n-1) n'est pas la dernière chose à faire par factorielle(n)

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

La fonction ci-dessus peut être écrite comme une fonction récursive terminale. L'idée est d'utiliser un argument de plus et d'accumuler la valeur factorielle dans le second argument. Lorsque **n** atteint 1, retournez la valeur accumulée.

```
let fact' n =
   let rec f acc n = if n=0 then acc
                     else f (acc*n) (n-1) in f 1 n;;
```

3 -- Les fonctions récursives croisées
Deux fonctions sont mutuellement récursives si chacune d'elles fait appel à l'autre.
La construction **and** permet de définir des fonctions mutuellement récursives.

Exemple
let rec pair x =
if x = 0 then true else impair (x - 1)
and impair x =
if x = 0 then false else pair (x - 1) ;;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>
pair 853 ;;
- : bool = false
impair 51 ;;
- : bool = true

# pair 5 ;;
pair <-- 5
impair <-- 4
pair <-- 3
impair <-- 2
pair <-- 1
impair <-- 0

```
impair --> false
pair --> false
impair --> false
pair --> false
impair --> false
pair --> false
- : bool = false
```

4- Le type list

OCaml intègre un type **list** pour les listes.

- la liste vide est notée []
- les listes sont *polymorphes* : les éléments peuvent être de n'importe quel type
- mais elles sont *homogènes* : tous les éléments doivent avoir le même type
- comme les autres valeurs, elles sont non mutables

```
# [];;
- : 'a list = []
# [1;4;2];;
- : int list = [1; 4; 2]
# ['a';'g';'p'];
  ;;
- : char list = ['a'; 'g'; 'p']
# [1;'a'];;
Error: This expression has type char but an
         expression was expected of type int
# [[];[[]]];;
- : 'a list list list = [[]; [[]]]
```

- Opérateurs (de construction de nouvelles listes) : ": :" (*cons*) est l'opérateur (associatif à droite) de mise en tête d'un élément.

```
# 1::[2;3];;
- : int list = [1; 2; 3]
# 'o'::'c'::'a'::'m'::'l'::[];;
- : char list = ['o'; 'c'; 'a'; 'm'; 'l']
# 1::[];;
- : int list = [1]
# 1::['a'];;
Error: This expression has type char but an
         expression was expected of type int
```

- Opérateurs (de construction de nouvelles listes) : "@" est l'opérateur de concaténation de deux listes de même type

```
# [1;2;3]@[4;5];;
- : int list = [1; 2; 3; 4; 5]
# [1;2]@[];;
- : int list = [1; 2]
# [1]@['a'];;
Error: This expression has type char but an
         expression was expected of type int
```

- Fonctions de manipulation du module List

- longueur : `List.length: 'a list -> int`

```
# List.length ['a';'b';'c'];;
- : int = 3
```

- tête : `List.hd: 'a list -> 'a`

```
# List.hd ["bonjour";"salut";"hello"];;
- : string = "bonjour"
# List.hd [] ;;
Exception: Failure "hd".
```

- queue : `List.tl: 'a list -> 'a list`

```
# List.tl ["bonjour";"salut";"hello"];;
- : string list = ["salut"; "hello"]
# List.tl ['a'];;
- : char list = []
# List.tl [] ;;
Exception: Failure "tl".
```

- nième élément : `List.nth: 'a list -> int -> 'a`

```
# List.nth [3;5;7] 2;;
- : int = 7
# List.nth [3;5;7] 4;;
Exception: Failure "nth".
```

- appartenance : `List.mem: 'a -> 'a list -> bool`

```
# List.mem 5 [3;5;7];;
- : bool = true
# List.mem 12 [3;5;7];;
- : bool = false
```

- renversement : `List.rev: 'a list -> 'a list`

```
# List.rev [3;5;7];;
- : int list = [7; 5; 3]
```

- appliquer une fonction à chaque élément :

```
List.map: ('a -> 'b) -> 'a list -> 'b list
# List.map (function x->x*x) [3;5;7];;
- : int list = [9; 25; 49]
```

- appliquer une fonction sur tous les éléments :

```
List.fold_left: ('a -> 'b -> 'a) -> 'a
                                -> 'b list -> 'a
# List.fold_left (fun x y -> x+y) 0 [3;5;7];;
- : int = 15
```

- appliquer une fonction sur tous les éléments :

```
List.fold_right: ('a -> 'b -> 'b) -> 'a list
                                   -> 'b -> 'b
# List.fold_right (fun x y -> x+y) [3;5;7] 0;;
- : int = 15
```

- et bien d'autres (voir documentation)

5- Fonctions récursives sur les listes

Les fonctions qui examinent "en profondeur" les listes sont le plus souvent récursives et définies par filtrage.

Exemple : Compter le nombre d'éléments d'une liste :

```
# let rec longueur l =
match l
with [] -> 0
  |  _::reste -> 1 +  longueur reste;;
val longueur : 'a list -> int = <fun>

# longueur [];;
- : int = 0

# longueur ["a"; "salut"];;
- : int = 2
```

Cette fonction est prédéfinie sous le nom de List.lenght

```
# let rec longueur l =
match l
with [] -> 0
  |  _::reste -> 1 +  longueur reste;;
val longueur : 'a list -> int = <fun>


longueur [1;5;2] ⇒ longueur 1::[5;2]
⇒ 1 + (longueur [5;2])
⇒ 1 + (longueur 5::[2])
⇒ 1 + 1 + (longueur [2])
⇒ 1 + 1 + (longueur 2::[])
⇒ 1 + 1 + 1 + longueur []
⇒ 1 + 1 + 1 + 0
```

La version fonction récursive terminale de la fonction longueur

```
let rec length_inner l n =
  match l with
    [] -> n
  | h::t -> length_inner t (n + 1)

let length l = length_inner l 0
```

Un exemple d'évaluation de cette fonction est comme suit :

```
       length [5; 5; 5]
⟹     length_inner [5; 5; 5] 0
⟹     length_inner [5; 5] 1
⟹     length_inner [5] 2
⟹     length_inner [] 3          base case
⟹     3
```

Une autre fonction pour calculer la somme des éléments d'une liste d'entiers

```
sum : int list → int

let rec sum l =
  match l with
    [] -> 0
  | h::t -> h + sum t
```

Une autre fonction qui renvoie la liste constituée seulement des éléments qui se trouvent dans les positions impairs (1er, 3ème, 5ème, …) d'une liste initiale

```
let rec odd_elements l =
  match l with
    [] -> []
  | [a] -> [a]
  | a::_::t -> a :: odd_elements t
```

Un exemple d'évaluation de cette fonction est comme suit :

```
        odd_elements [2; 4; 2; 4; 2]
⟹      2 :: odd_elements [2; 4; 2]
⟹      2 :: 2 :: odd_elements [2]
⟹      2 :: 2 :: [2]
⟹      [2; 2; 2]
```

Exercice 1

Soit la fonction **app** définit de la façon suivante

```
# let app(f,y) = (f y ) + y;;
val app : (int -> int) * int -> int = <fun>
```

1- Définir une fonction **plus5** qui associe à x la valeur x+5
2- Définir une fonction **fois8** qui associe à x la valeur x*8
3- En déduire l'appel qui donne les résultats suivants :

```
# app(plus5, ? );;
- : int = 29
# app(fois8, ? );;
- : int = 108
```

Exercice 2

Soit la fonction ***app_couple*** définit comme suit :

```
#  let app_couple(f,c,a) =
  let (x1,x2)=c in f (x1+a) * f (x2+a) ;;
```

Donnez les réponses de l'interpréteur OCaml

1- après la définition de ***app_couple***
2- après cet appel `# app_couple(plus5,(1,1),3);;`
3- et après cet appel `# app_couple((fun y -> y * y),(1,1),3);;`

Exercice 3

Soit la fonction **decalage** définit comme suit :

```
# let decalage(f,a) = fun y -> f (y+a) ;;
val decalage : (int -> int) * int -> int -> int = <fun>
```

Donnez les réponses de l'interpréteur après l'exécution des commandes suivantes :

1- `# decalage(plus5,8);;`

2- `# decalage(plus5,8) 1 ;;`

Exercice 4

Donner la réponse de l'interpréteur à la définition suivante (polymorphisme)

```
# let composition (f,g) x = f(g x);;
# composition((fun x -> x+4),(fun y -> y*5)) 8 ;;
# composition((fun y -> y*5),(fun x -> x+4)) 8;;
```

Exercice 5

Définir une fonction polymorphe qui donne le maximum de ces deux arguments x et y et affichez la réponse de l'interpréteur Caml.

Exercice 6

Définir la fonction récursive `sigma` qui calcule la somme des entiers (positifs) compris entre `0` et la valeur de son argument.

Exercice 7

Définir une nouvelle fonction `sigma` qui teste la validité de l'argument avant d'effectuer le calcul de la somme définie par une fonction locale `sigma_rec` qui affiche la chaîne "Erreur : argument négatif " si l'argument est négatif et affiche le résultat dans le cas contraire.

Exercice 8

Ecrire la fonction récursive longueur qui calcule à partir de son argument (une liste) la longueur de cette liste (le nombre de ses éléments).

Exercice 9

a) Ecrire une fonction square qui affiche sous forme chaîne de caractère le carré de son argument de type entier.

b) appliquez la fonction square à une liste d'entiers et donnez la réponse de l'interpréteur OCaml.

Chapitre 3 : Filtrage de Motif et déclaration de types

Le filtrage Permet de programmer pas cas. On examine la structure d'une valeur et l'on choisit les actions à effectuer selon chaque cas.
La structure de chaque cas est décrite à l'aide d'un *filtre* ou *motif*.

Syntaxe:

```
match expr
 with filtre_1 -> action_1
 |    filtre_2 -> action_2
 |    ....
 |    _        -> action_tous_autres_cas
```

A gauche de chaque flèche : un *motif* (en anglais, *pattern*), qui décrit une forme possible de la valeur à comparer.
A droite de chaque flèche : l'action à effectuer dans ce cas.
Le symbole "_" est un motif lu "dans tous les autres cas".
La valeur filtrée est comparée successivement à chacun des filtres selon l'ordre de leur définition

Exemple : Définition de la fonction logique "implication"

```
# let imply v = match v with
      (true,true)   -> true
    | (true,false)  -> false
    | (false,true)  -> true
    | (false,false) -> true;;
val imply : bool * bool ->
bool = <fun>
```

```
# let imply v = match v with
      (true,x)  -> x
    | (false,x) -> true;;
val imply : bool * bool ->
bool = <fun>
```

Ces deux versions de imply calculent la même fonction. C'est-à-dire qu'elles retournent les mêmes valeurs pour les mêmes entrées.

1- **Motif linéaire** : Un motif doit être obligatoirement *linéaire*, c'est-à-dire qu'une variable donnée ne peut y figurer au plus qu'une fois. Ainsi, on aurait pu espérer pouvoir écrire :

```
# let equal c = match c with
    (x,x) -> true
  | (x,y) -> false;;
Characters 35-36:
This variable is bound several times in this matching
```

2- **Motif universel** : Le symbole "_" filtre toutes les valeurs possibles. Il est appelé motif universel. Il peut être utilisé pour filtrer des types complexes. On l'utilise, par exemple, pour simplifier encore la définition de la fonction imply :

```
# let imply v = match v with
        (true,false) -> false
      | _            -> true;;
val imply : bool * bool -> bool = <fun>
```

3- **Combinaison de motifs** : La combinaison de plusieurs motifs permet d'obtenir un nouveau motif qui pourra déstructurer une valeur selon l'un ou l'autre de ses motifs originaux. La forme syntaxique est la suivante : $p_1 \mid ... \mid p_n$

L'exemple suivant montre comment vérifier qu'un caractère est une voyelle.

```
# let est_une_voyelle c = match c with
    'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
  | _ -> false ;;
val est_une_voyelle : char -> bool = <fun>
# est_une_voyelle 'i' ;;
- : bool = true
# est_une_voyelle 'j' ;;
- : bool = false
```

4- **Nommage d'une valeur filtrée** : Lors d'un filtrage de motif, il est parfois pratique de nommer tout ou partie du motif. La forme syntaxique suivante introduit le mot clé as qui associe un nom à un motif : **(** *p* **as** *nom* **)**

Dans l'exemple suivant, la fonction min_rat rend le plus petit rationnel d'un couple de rationnels. Ces derniers sont représentés par un couple numérateur et dénominateur.

```
# let min_rat cr = match cr with
   ((_,0),c2) ->  c2
 | (c1,(_,0)) ->  c1
 | (((n1,d1) as r1), ((n2,d2) as r2)) ->
        if (n1 *  d2 ) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <fun>
```

5- **Filtrage avec garde** : Le filtrage avec gardes correspond à l'évaluation d'une expression conditionnelle immédiatement après le filtrage d'un motif. Si cette expression renvoie *true*, alors l'expression associée au motif est évaluée, sinon le filtrage continue avec le motif suivant.

match *expr* with
  | p_i when $cond_i$ → $expr_i$

L'exemple suivant utilise deux gardes pour tester l'égalité entre deux rationnels.

```
# let eq_rat cr = match cr with
   ((_,0),(_,0)) ->  true
 | ((_,0),_) ->  false
 | (_,(_,0)) ->  false
 | ((n1,1), (n2,1)) when n1 = n2 -> true
 | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) -> true
 | _ -> false;;
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

Dans le filtrage du quatrième motif, si la garde échoue, le filtrage se poursuit sur le cinquième motif.

6- **Filtrage d'intervalles de caractères** : Dans le cadre du filtrage sur des caractères, il est fastidieux de construire la combinaison de tous les motifs correspondant à un intervalle de caractères. En effet, si l'on désire tester si un caractère est bien une lettre, il faudra au minimum écrire 26 motifs simples et les combiner. Objective CAML autorise pour les caractères, l'écriture de motifs de la forme : 'c_1' .. 'c_n'
Par exemple le motif *'0' .. '9'* correspond au motif *'0'* | *'1'* | *'2'* | *'3'* | *'4'* | *'5'* | *'6'* | *'7'* | *'8'* | *'9'*. La première forme est plus agréable à lire et plus rapide à écrire.

```
# let char_discriminate c = match c with
     'a' | 'e' | 'i' | 'o' | 'u' | 'y'
   | 'A' | 'E' | 'I' | 'O' | 'U' | 'Y'  -> "Voyelle"
   | 'a'..'z' | 'A'..'Z' -> "Consonne"
   | '0'..'9' -> "Chiffre"
   |   _ -> "Autre" ;;
val char_discriminate : char -> string = <fun>
```

7- **Filtrage des listes** : Comme nous l'avons vu, une liste peut être :

- soit vide (la liste est de la forme []),
- soit composée d'un premier élément (sa tête) et d'une sous-liste (sa queue). La liste est alors de la forme t**::**q.

Ces deux écritures possibles d'une liste sont utilisables comme des motifs et permettent de filtrer une liste.

```
# let rec size x = match x with
     [] -> 0
   | _::queue_x -> 1 + (size queue_x) ;;
val size : 'a list -> int = <fun>
# size [];;
- : int = 0
# size [7;9;2;6];;
- : int = 4
```

```
# let premier l =
match l
with []       -> failwith "premier"
  |  e::reste -> e;;
val premier : 'a list -> 'a = <fun>
```

8- Declaration de types

Les déclarations de type sont un autre constituant possible d'une phrase Objective CAML. Elles permettent de définir de nouveaux types correspondant aux structures de données originales utilisées dans un programme. Il y a deux grandes familles de types : les types produit pour les n-uplets ou les enregistrements ; les types somme pour les unions.
Syntaxe:

```
# type  nouveau_nom = typedef ; ;
```

Le nom du type "*nouveau_nom*" doit commencer par une minuscule et le "*typedef*" doit commencer par une majuscule

8.1 Enregistrements

Les enregistrements sont des n-uplets dont chaque champ est nommé à la manière des *record* de Pascal ou des *struct* de C. Un enregistrement correspond toujours à la déclaration d'un nouveau type. Un type enregistrement est défini par la déclaration de son nom, du nom de chacun de ses champs et de leur type.

Syntaxe :

type $nom = \{ nom_1 : t_1; \ldots; nom_n : t_n \}$;;

Exemple : définition d'un type représentant les nombres complexes

```
# type complex = { re:float; im:float } ;;
type complex = { re: float; im: float }
```

La création d'une valeur de type enregistrement s'effectue en donnant une valeur à chacun des champs (dans un ordre quelconque):

$\{ nom_{i1} = expr_{i1}; \ldots; nom_{in} = expr_{in} \}$;;

```
# let c = {re=2.;im=3.} ;;
val c : complex = {re=2; im=3}
# c = {im=3.;re=2.} ;;
- : bool = true
```

L'accès à un champ est possible de deux façons : par la notation pointée ou par le filtrage de certains champs :

- $expr.nom$
- $\{ nom_i = p_i ; \ldots; nom_j = p_j \}$

Exemple : La fonction `add_complex` accède aux champs par la notation pointée alors que la fonction `mult_complex` y accède par filtrage.

```
# let add_complex c1 c2 =  {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re=4; im=6}
# let mult_complex c1 c2 = match (c1,c2) with
   ({re=x1;im=y1},{re=x2;im=y2}) -> {re=x1*.x2-
.y1*.y2;im=x1*.y2+.x2*.y1} ;;
val mult complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re=-5; im=12}
```

8.2- Types Somme

À la différence des n-uplets ou des enregistrements, qui correspondent à un produit cartésien, la déclaration d'un type somme correspond à une union d'ensembles. On regroupe dans un même type des types différents (par exemple des entiers et des chaînes de caractères). Les différents membres de la somme sont discriminés par des constructeurs qui permettent d'une part, comme leur nom l'indique,

de *construire* les valeurs de ce type et d'autre part d'*accéder* aux composantes de ces valeurs grâce au filtrage de motif. Appliquer un constructeur à un argument, c'est indiquer que la valeur retournée appartient à ce nouveau type.

8.2.1- Constructeurs constants

On déclare un type somme en donnant le nom de ses constructeurs et le type de leur éventuel argument.

Syntaxe :

```
# type  nouveau_nom = Const1 | Const2 | ... | Constn ; ;
```

Remarque : Les noms des constructeurs commencent toujours par une majuscule.

Exemple:

```
# type couleur = Pique | Coeur | Carreau | Trefle;;
type couleur = Pique | Coeur | Carreau | Trefle

# let est_rouge x = match x with
      Pique   -> false
    | Coeur   -> true
    | Carreau -> true
    | Trefle  -> false;;
val est_rouge : couleur -> bool = <fun>

# type figure = As | Roi | Dame | Cavalier | Valet |
                Dix | Neuf | Huit | Sept;;
type figure = As | Roi | Dame | Cavalier | Valet |
                Dix | Neuf | Huit | Sept
```

8.2.2- Constructeurs avec arguments

Les constructeurs peuvent avoir des arguments. Le mot clé **of** indique le type des arguments du constructeur. Cela permet de regrouper sous un même type des objets de types différents, chacun étant introduit avec un constructeur particulier.

Syntaxe:

```
# type  nouveau_nom = etiquette of type1 * ... * typen ; ;
```

Voici un exemple classique de définition d'un type de données pour représenter les cartes dans un jeu, ici le Tarot. On définit les types couleur et carte de la manière suivante :

```
# type couleur = Pique | Coeur | Carreau | Trefle ;;
# type carte =
      Roi of couleur
    | Dame of couleur
    | Cavalier of couleur
    | Valet of couleur
    | Petite_carte of couleur * int
    | Atout of int
    | Excuse ;;
```

La création d'une valeur du type carte s'effectue par application d'un constructeur à une valeur de son type.

```
# Roi Pique ;;
- : carte = Roi Pique
# Petite_carte(Coeur, 10) ;;
- : carte = Petite_carte (Coeur, 10)
# Atout 21 ;;
- : carte = Atout 21
```

Et voici, par exemple, la fonction *toutes_les_cartes* qui construit la liste de toutes les cartes d'une couleur passée en argument.

```
# let rec interval a b =  if a = b then [b] else a::(interval (a+1) b) ;;
val interval : int -> int -> int list = <fun>
# let toutes_les_cartes s =
    let les_figures = [ Valet s; Cavalier s; Dame s; Roi s ]
    and les_autres  = List.map (function n -> Petite_carte(s,n))
(interval 1 10)
    in les_figures @ les_autres ;;
val toutes_les_cartes : couleur -> carte list = <fun>
# toutes_les_cartes Coeur ;;
- : carte list =
[Valet Coeur; Cavalier Coeur; Dame Coeur; Roi Coeur; Petite_carte (Coeur,
1);
 Petite_carte (Coeur, 2); Petite_carte (Coeur, 3); Petite_carte (Coeur,
...);
 ...]
```

8.3- Types avec paramètre unique :

Syntaxe :

```
# type 'a nouveau_nom = typedef ; ;
```

Exemple :

```
# type 'a tri_meme_type = TripletM of 'a * 'a * 'a;;
type 'a tri_meme_type = TripletM of 'a * 'a * 'a

# TripletM("bb", "cc", "aa");;
- : string tri_meme_type = TripletM ("bb", "cc", "aa")
```

8.4- Types avec plusieurs paramètres :

Syntaxe :

```
# type ('a1, ...,'an) nouveau_nom = typedef ; ;
```

Exemple :

```
# type ('a, 'b, 'c) tri_diff_type = TripletD of 'a * 'b * 'c;;
type ('a, 'b, 'c) tri_diff_type = TripletD of 'a * 'b * 'c

# TripletD(3,4,5);;
- : (int, int, int) tri_diff_type = TripletD (3, 4, 5)

# TripletD(3,"bonjour",5.4);;
- : (int, string, float) tri_diff_type =
    TripletD (3, "bonjour", 5.4)
```

8.5- Types récursifs :

Un type récursif, un peu comme une fonction récursive, est un type qui se contient lui-même dans sa définition. Ce sont donc des types sommes, dont un (ou plusieurs) constructeur a une donnée associée du même type. Et là aussi, on définit un cas de base non récursif.

Exemple 1:

```
type t =
     C of char (* constructeur non recursif *)
    |S of int * t   (* constructeur recursif *)
```

Exemple de valeurs dy type `t` ?

`C('x')` `S(5, C('x'))` `S(12, S(5, C('x')))` etc.

Exemple 2:

```
# type liste_entier =
        | Vide
        | Element of int * liste_entier

# let ma_liste = Element ( 1, Element ( 2, Element ( 3, vide) ) )
```

Mais on peut imaginer un exemple plus complexe, par exemple une liste qui contient soit un **int** soit un **float** :

```
# type liste_nombres =
        | vide
        | Element_Entier of int * liste_nombres
        | Element_Reel of float * liste_nombres
```

Les types récursifs peuvent être parcourus avec une fonction récursive qui fait un **match** sur les constructeurs :

```
# let somme_elements liste = match liste with
        | Vide -> 0
        | Element (x , reste) -> x + somme_elements reste
```

Exercice 1 :
Ecrire une fonction récursive pour concaténer deux listes en utilisant le filtrage.

Exercice 2 :
Ecrire une fonction récursive "rev" qui permet de donner la liste miroir d'une liste donnée en utilisant le filtrage.

Exercice 3 :
Soit la fonction suivante qui renvoie la liste constituée seulement des éléments qui se trouvent dans les positions impairs (1er, 3ème, 5ème, …) d'une liste initiale. Optimiser cette fonction.

```
let rec odd_elements l =
  match l with
    [] -> []
  | [a] -> [a]
  | a::_::t -> a :: odd_elements t
```

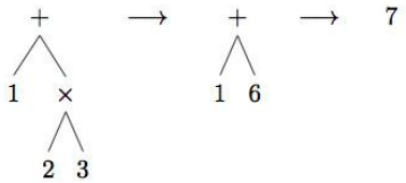
Exercice 4 :
Ecrire la fonction récursive "Take" qui reçoit 2 arguments un entier "n" et une liste "l" et qui renvoie la liste composée du nombre d'éléments "n" en utilisant le filtrage.

Exercice 5 :
Soit la fonction suivante qui qui détermine le nombre de solutions réelles d'une équation du second degré (0 quand pas de solution réelle, 1 si racine double, 2 sinon)

```
let solutions (a,b,c) =
     if a = 0.0 then failwith "solutions : premier degre"
     else  let delta = b*. b -. 4.0 *. a *. c in
             if delta < 0.0 then 0
             else if delta=0.0 then 1
                  else 2;;
```

Réécrire cette fonction en utilisant le filtrage avec gardes.

Exercice 6 :
1) Définir un type récursif pour les expressions mathématiques telle que :



Par exemple l'expression : 1 + 2*3 est représentée comme suit par ce type :

```
Add (Num 1, Mul (Num 2, Num 3))
```

2) Ecrire la fonction récursive "evaluate" pour évaluer ces expressions mathématiques.

Exercice 7 :
a) Concevez un nouveau type "rect" pour représenter les rectangles. Traitez les carrés comme un cas spécial.
b) Ecrire une fonction "area r" de type "rect -> int" pour calculer la surface d'un "rect" donné.

Chapitre 4 : Schémas de programmes

Introduction

Exercice : Écrivez un programme qui calcule la somme des éléments d'une liste

```
let rec somme l = match l with
  [] -> 0 | t::r -> t+(somme r);;
```

Exercice : Écrivez un programme qui calcule le produit des éléments d'une liste

```
let rec produit l = match l with
  [] -> 0 | t::r -> t*(produit r);;
```

Plusieurs programmes sur les listes se ressemblent. Nous pouvons dire que ces programmes suivent un même schéma. Une idée intéressante est donc de les exprimer une fois pour toutes. Un autre intérêt d'un schéma de programmes est qu'il correspond à un algorithme. Puisque plusieurs algorithmes sont possibles pour résoudre un même problème, nous pouvons changer d'algorithme en remplaçant un schéma par un autre.

1. Schémas de programme

Les définitions de fonctions f précédentes, qui prennent une liste l en argument, suivent le même schéma récursif suivant.

- Si l est vide alors le résultat est une valeur a qui ne dépend pas de l (cas de base).

- Sinon, soit l de la forme h : : t, alors le résultat est h op f(t), où op est une opération binaire (cas récursif).

```
let rec f l =
 match l with
  [] -> a
| t::r -> op t (f r)
;;
```

Exemple La fonction *somme* qui calcule la somme des éléments d'une liste peut être définie de la manière suivante (en OCaml) :

```
let rec somme l = match l with
  [] -> 0 | t::r -> t+(somme r);;
```

On peut remplacer dans cette définition `t+(somme r)` par `(+) t (somme r)` puisque ces expressions sont *sémantiquement égales*. On obtient la définition suivante, équivalente à la précédente et conforme au schéma en remplaçant `a` par `0` et `op` par `(+)` :

```
let rec somme l = match l with
  [] -> 0 | t::r -> (+) t (somme r);;
```

2. CAPTURER UN SCHÉMA DE PROGRAMMES

On peut capturer le schéma précédent ou tout autre schéma de programmes en définissant une fonction d'ordre supérieur. Le schéma précédent correspond à une fonction d'ordre supérieur dont la définition s'obtient à partir de la définition récursive en remplaçant les expressions *a* et *op* par des variables : a et op, et en ajoutant ces variables en argument de la fonction. Ainsi, les valeurs *a* et *op* utilisées pour définir une fonction qui suit ce schéma, deviennent des paramètres formels : on dit que l'on « abstrait » ces valeurs et on appelle parfois cette technique « *abstraction* ». La même technique peut être utilisée pour définir tous les schémas de programmes.

3. SCHÉMA *REDUCE*

Le schéma précédent est appelé **réduction**. Il correspond à une fonction d'ordre supérieur que l'on peut définir de la manière suivante dans le langage Ocaml :

Une définition du schéma *reduce*

En OCaml :

```
let rec reduce op a l = match l with
   [] -> a | t::r -> op t (reduce op a r);;
(* reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

Exemple Voici quelques exemples d'utilisation de ce schéma en OCaml :

```
# reduce (+) 0[1;2;3];;(* calcul de 1+2+3 = 0+(1+(2+3)) *)
- : int = 6
# reduce (*) 1[1;2;3];;(* calcul de 1*2*3=1*(1*(2*3)) *)
- : int = 6
# reduce (@) [][[1;2];[3]];;(*[1;2]@[3]=[]@([1;2]@[3]) *)
- : int list = [1; 2; 3]
```

4. CACHER LA RÉCURSIVITÉ

Un schéma de programmes permet de définir les fonctions de façon beaucoup plus concise en reportant la récursivité dans la définition de la fonction d'ordre supérieur.

Exemple : Les fonctions précédentes sur les listes s'écrivent de façon plus concise en utilisant le schéma ***REDUCE***.

```
# let somme = reduce (+) 0;;
val somme : int list -> int = <fun>
# let produit = reduce ( * ) 1;;
val produit : int list -> int = <fun>
# let tri l = reduce insere [] l;;
val tri : 'a list -> 'a list = <fun>
# let concat l1 l2 = reduce (fun x y -> x::y) l2 l1;;
val concat : 'a list -> 'a list -> 'a list = <fun>
```

5. AUTRES SCHÉMAS CLASSIQUES

Le schéma REDUCE est donc apparenté à plusieurs schémas dit d'« accumulation » (en anglais « fold » qui se traduit littéralement par « pliage ») qui décrivent différents ordres possibles pour accumuler les résultats partiels.

5.1. Schémas *fold_left* et *fold_right*

Le résultat du schéma réduction sur une liste $l = [e_1, e_2, \ldots, e_n]$ est :

$$e_1 \; op \; (e_2 \; op \; \ldots (e_n \; op \; a) \ldots)$$

Nous calculons donc d'abord $e_n \; op \; a$, ensuite $(e_{n-1} \; op \; (e_n \; op \; a))$, etc.

Mais nous pourrions aussi faire :

$$(\ldots ((a \; op \; e_1) \; op \; e_2) \ldots) \; op \; e_n$$

C'est-à-dire, calculer d'abord $a \; op \; e_1$, ensuite $(a \; op \; e_1) \; op \; e_2$, etc.

- Le schéma *fold_right*

Le premier schéma, où nous utilisons les éléments dans l'ordre inverse, s'appelle aussi plier à droite et correspond exactement au schéma réduction que nous avons déjà vu.

```
fold_right op a l =
   match l with
   | [ ] -> a
   | h : : t -> op  h  ( fold_right   op  a   t )
```

Par exemple :

```
fold_right (+) 0 [1, 2, 3]
= 1 + fold_right (+) 0 [2, 3]
= 1 + (2 + fold_right (+) 0 [3])
= 1 + (2 + (3 + fold_right (+) 0 []))
= 1 + (2 + (3 + 0))
```

- Le schéma *fold_left*

L'autre schéma, où on utilise les éléments dans l'ordre s'appelle plier à gauche et correspond à un schéma différent du schéma réduction.

Exemple : Écrivez une fonction d'ordre supérieur qui correspond au schéma ***fold_left***.

```
fold_left  op  a  l =
  match l with
  | [ ] -> a
  | h : : t ->  fold_left   op  ( op  a  h  )  t
```

Par exemple :

```
fold_left (+) 0 [1, 2, 3]
= fold_left (+) (0 + 1) [2, 3]
= fold_left (+) ((0 + 1) + 2) [3]
= fold_left (+) (((0 + 1) + 2) + 3) []
= (((0 + 1) + 2) + 3)
```

La fonction **fold_left** est récursive terminale et donc plus efficace que **fold_right**.
Exercice : Écrivez la fonction longueur en utilisant **fold_left**.
Nous voulons le comportement suivant :

```
fold_left 0 [1, 2, 3]
= fold_left (0 + 1) [2, 3]
= fold_left ((0 + 1) + 1) [3]
= fold_left (((0 + 1) + 1) + 1) []
= (((0 + 1) + 1) + 1)
```

Donc, nous devons trouver la fonction **op** correspondante. Notez que l'accumulation est maintenant à gauche. Donc :
Op = fun a h -> a+1

5.2. Schéma map

Un autre schéma très courant consiste à appliquer une même fonction à tous les éléments d'une liste. Ce schéma est nommé **map**. Par exemple, pour une liste $e_1, e_2, \ldots, e_n$, et une fonction **f**, le résultat de **map** est la liste $f(e_1), f(e_2), \ldots, f(e_n)$.
Exemple : Écrivez la fonction d'ordre supérieur qui correspond au schéma **map** :

```
# let rec map f l =
        match l with
        | [ ] -> [ ]
        | h::t -> f h::(map f t)
```

Exercice : Écrivez la fonction qui prend une liste d'entiers en argument et retourne la même liste où les entiers sont multipliés par 2.

```
# let fois_2 = map (fun x -> x*2)
# fois_2 [ 1 ; 2 ; 3 ]
  - : [ 2 ; 4 ; 6 ]
```

```
map (fun x->x*2) [1, 2, 3]
= 2 ::(map (fun x->x*2) [2, 3])
= 2 ::(4 :: (map (fun x->x*2) [3]))
= 2 ::(4 ::(6 ::map (fun x->x*2) []))
= 2 ::(4 ::(6 ::[]))
```

A quelle fonction ressemble la fonction map ?

Exercice : Réécrivez la fonction qui correspond au schéma **map** en utilisant le schéma **fold_right**.

let map f = fold_right (fun x y -> f(x) : : y) [ ]

5.3. Schéma map2

Le schéma **map2** généralise le schéma map en utilisant deux listes comme argument. Soit les listes [a_1; . . . ; a_n] et [b_1; . . . ; b_n] et la fonction f. Le résultat de **map2** est la liste [f(a_1; b_1); . . . ; f(a_n; b_n)].

Exemple : Écrivez une fonction d'ordre supérieur qui correspond à **map2** :

```
# let rec map2  f  l1  l2  =
        match ( l1 , l2 ) with
        | ( [ ] , _ ) -> [ ]
        | ( _ , [ ] ) -> [ ]
        | ( h1 : : t1 , h2 : : t2 ) ->  f h1 h2 : : (map2 f  t1  t2)
```

Le schéma **map2** est aussi appelé **zip**.

Evidemment, il est aussi possible de généraliser le schéma **map** à n listes.

Annexe au cours

Découpage de l'algorithme

L'algorithme du tri par insertion est conservé : on découpe toujours le code en deux fonctions, la fonction "insérer" qui insère un élément à la bonne position dans la liste, et la fonction "tri_insertion", qui réutilise la fonction insérer pour trier le tableau, en insérant chaque élément successivement à la bonne place.

Insertion

Voici le code d'insertion. Il insère un élément dans une liste triée en ordre croissant, de manière à renvoyer une liste toujours triée.

```
let rec insere elem liste = match liste with
|  [] -> elem::[]
|  tete::queue ->
   if elem < tete then elem :: liste
   else tete :: insere elem queue
```

Le code se lit de lui-même : les deux arguments de la fonction **insere** sont **elem** et liste.

- si c'est la liste **vide** (premier |, premier cas), on renvoie l'élément suivi de la liste vide ;
- si la liste est une **tête** suivie d'une **queue**, on compare l'élément et la tête : si l'élément est plus petit, on a trouvé la bonne place pour le mettre : on renvoie donc l'élément, suivi de la liste, sinon (si l'élément est plus grand), on renvoie la tête, suivie de la queue dans laquelle on a inséré **elem**.

Tri

Maintenant qu'on a fait le gros du boulot, la fonction de tri vient toute seule :

```
let rec tri_insertion = function
|  [] -> []
|  tete::queue -> insere tete (tri_insertion queue)
```

- si la liste est vide, on la renvoie ;
- si la liste est une tête suivie d'une queue, on trie la queue, et on y insère la tête.

Exercice 1 :

Une *matrice creuse* est une matrice qui comporte un grand nombre de coefficients nuls. Par exemple:

$$\begin{pmatrix} 0 & 0 & 7 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 \end{pmatrix}$$

On choisit de représenter une matrice creuse $A = (a_{ij})$ de dimensions $n \times m$ par le couple $((n, m), l)$ où l est une liste des couples $((i, j), a_{ij})$ tels que a_{ij} est un coefficient non nul. Par exemple, la matrice creuse précédente est représentée par le couple dont le premier élément est (4, 6) et le second la liste [((1, 3), 7), ((1, 6), 4), ((4, 3), 5)].

1) Écrire une fonction *transpose* qui calcule la *transposée* d'une matrice creuse. La transposée d'une matrice A de dimensions $n \times m$ est la matrice A_t de dimensions $m \times n$, telle que $A_t = (a_{ji})$.

Indication : 1- Ecrire d'abord une fonction **swap** qui utilise List.map pour transposer les éléments.
2- Ecrire la fonction ***transpose*** qui utilise la fonction swap.

2) Écrire une fonction *addition* qui, étant donné deux matrices creuses de mêmes dimensions, calcule une matrice creuse qui représente la somme des matrices (on rappelle que $A + B = (a_{ij} + b_{ij})$) (la liste du résultat ne doit pas contenir deux éléments ayant les mêmes indices i et j).

Indication : 1- Ecrire une fonction *meme_indice* qui vérifie si les couples ont le même indice. 2- Ecrire la fonction *some_elt* pour ajouter des éléments ayant le même indice. 3- Ecrire la fonction *ajoute_elt* pour ajouter un élément eltB à son homologue d'une liste eltsA. 4- Ecrire une fonction *ajoute_elts* qui généralise la fonction *ajoute_elt* pour tous les éléments d'une liste. 5- Ecrire la fonction *addition* pour sommer les éléments de 2 matrices A et B.

Exercice 2:

1) Ecrire une fonction *concat* récursive qui permet de concaténer les éléments chaines de caractère d'une même liste.
2) Réécrire la fonction *concat* avec le schéma *REDUCE* et donner une application de ce schéma.
3) Réécrire la fonction *concat* avec le schéma *fold_left.*

Exercice 3:

1) Ecrire une fonction *conclist* récursive pour concaténer les éléments d'une liste de listes (on obtient une seule liste)
2) Récrire *conclist* avec le schéma fold_right.

Exercice 4 :

Soit la fonction suivante qui calcule la somme des éléments d'une liste en utilisant un accumulateur

```
#let rec somme_accu accu l = match l with
      []  -> accu
   | t::r -> somme_accu (accu + t) r;;
val somme_accu : int -> int list -> int = <fun>
```

Donnez le schéma qui décrit la fonctionnelle générale avec accumulateur.

Exercice 5 :

1) Ecrire une fonction récursive *min_2_listes* qui étant données deux listes quelconques de même type, renvoie une liste composée d'éléments minimum de chaque position.

Exp : l1 = [4; 8 ;5; 9; 14; 3] et l2 = [45; 2; 9; 23 ;52 ; 11; 32; 28]

Resultat = [4; 2; 5; 9; 14; 3]

2) Récrire cette fonction en utilisant le schéma **map2**.

Chapitre 5 : Lambda calcul

1- Introduction :

Le lambda-calcul est un langage et un système de réécriture imaginé par le mathématicien Alonzo Church en 1932.
Le lambda-calcul est un langage qui est :

- très petit : il ne comporte que deux constructions syntaxiques.
- très expressif : il est capable d'exprimer toutes les fonctions calculables.

2- Syntaxe :

Les expressions du lambda-calcul sont appelée **lambda-expressions** ou **lambda-termes**.
Une lambda-expression est :

- soit une **constante** : 4, π, +,. . . (un nombre, un symbole ou un opérateur).
- soit une **variable** : *x*, *y*, *z*, . . . (typiquement une seule lettre minuscule).
- soit une **abstraction** de la forme : **(λ*x*.*M*)**

  Où **x** est une variable et **M** est une lambda-expression.

- soit une application de la forme : **(M N)**
  où **M** et **N** sont des **lambda-expressions.**

3- Conventions de parenthésage :

Les lambda-expressions ainsi définies sont non ambigües. Mais le grand nombre de parenthèses rend la lecture des expressions difficile :
(((λx.(λy.((+ x) y))) 1) 2).

Conventions:

1. Les parenthèses du début et de la fin sont optionnelles.
2. L'application est plus prioritaire que l'abstraction.
3. L'application est associative à gauche.

Conventions de parenthésage des lambda-expressions

$$
\begin{aligned}
(M) &\equiv M \\
\lambda x.(M\,N) &\equiv \lambda x.M\,N \\
\lambda x.(\lambda y.M) &\equiv \lambda x.\lambda y.M \\
(M\,N)\,O &\equiv M\,N\,O
\end{aligned}
$$

Exemple :

Enlever les parenthèses inutiles de la lambda-expression : (((λx.(λy.((+ x) y))) 1) 2)

Règle (1) : ((λx.(λy.((+ x) y))) 1) 2
Règle (2) : ((λx.(λy.(+ x) y)) 1) 2
Règle (3) : ((λx.λy.(+ x) y) 1) 2
Règle (4) (deux fois) : (λx.λy. + x y) 1 2

4- Signification des lambda-expressions

4.1- Signification de l'abstraction

Une abstraction de la forme **λx.M** exprime la fonction anonyme qui à tout **x** associe **M** (**M** est l'image de **x** par cette fonction).
Dans ce cas, la lambda-expression M est appelée **corps de l'abstraction**

Exemple : L'abstraction **λx.x** exprime la fonction qui à tout **x** associe **x** (fonction identité).
(L'abstraction **λy.y** exprime la même fonction.)

Exemple : La fonction **λx.λy.y** exprime la fonction d'ordre 2 qui à tout **x** associe la fonction identité.

4.2- Signification de l'application

Une application de la forme **M N** exprime l'image de **N par M**.
Exemple : L'application **(λx.x) 1** exprime l'image du nombre **1** par **la fonction identité**.

Exemple : En lambda-calcul, la constante + exprime la fonction d'ordre 2 qui à tout **x** associe **la fonction** qui à tout **y** associe **x + y**. Donc, l'expression **+ 1 2 ≡ (+ 1) 2** exprime le résultat de l'application de cette fonction d'ordre supérieur à **1** puis à **2**.

5- Réduction

Intuitivement, une réduction en lambda-calcul est l'action de transformer une lambda-expression en une autre plus simple et ayant la même signification, et de répéter cette opération jusqu'à ce que la lambda-expression ne puisse plus être réduite.

Exemple : **(λx. ∗ 2 x) 3 → ∗ 2 3 → 6**

De manière générale, la réduction d'une lambda-expression s'interprète comme le calcul du résultat.
Une réduction peut comporter plusieurs étapes, ces étapes décrivent le déroulement du calcul.

Exemple : **(λx.λy. + x y) 1 2 → (λy. + 1 y) 2 → + 1 2 → 3**

5.1- Bêta-réduction

Une expression de la forme **(λx.M) N** se réduit en **M** où les occurrences de la variable **x** sont remplacées par **N**. C'est à dire :

$$(\lambda x.M)\ N \xrightarrow{\beta} M[x := N]$$

Exemple :

$$\begin{aligned}
&(\lambda x.\lambda y.+\ x\ y)\ 1\ 2 \equiv ((\lambda x.\lambda y.+\ x\ y)\ 1)\ 2 \\
\xrightarrow{\beta}\ &((\lambda y.+\ x\ y)[x := 1])\ 2 \equiv (\lambda y.\ 1\ y)\ 2 \\
\xrightarrow{\beta}\ &(\lambda y.+\ 1\ y)[y := 2] \equiv +\ 1\ 2
\end{aligned}$$

5.2- Delta-réduction

La δ-réduction modélise les opérations mathématiques classiques. C'est-à-dire :

$$op\ c_1\ c_2 \ldots c_n \xrightarrow{\delta} c_0$$

Où **op** est un opérateur qui exprime une opération **n-aire**, chaque c_i est une constante qui exprime un argument, et c_0 est une constante qui exprime le résultat de l'opération.

Exemple :

$$+\ 1(*\ 2\ 3) \xrightarrow{\delta} +\ 1\ 6 \xrightarrow{\delta} 7$$

5.3- Problème de capture de variable

Soit la lambda-expression **(λf.λa.f a)**. Il s'agit de la fonction d'ordre supérieur qui reçoit une fonction et un argument et applique la fonction à l'argument.
Nous avons :

$$\begin{aligned}
&((\lambda f.\lambda a.f\ a)\ g)\ x) \\
\xrightarrow{\beta}&(\lambda a.g\ a)\ x \\
\xrightarrow{\beta}&g\ x
\end{aligned}$$

Pourtant, en remplaçant **g** par **a** nous avons :

$$\begin{aligned}
&((\lambda f.\lambda a.f\ a)\ a)\ x) \\
\xrightarrow{\beta}&(\lambda a.a\ a)\ x \\
\xrightarrow{\beta}&x\ x
\end{aligned}$$

C'est-à-dire, le résultat est incorrect.

La variable "**a**" qui apparaît dans le corps de l'abstraction exprime un autre objet que celui exprimé par la variable "**a**" en argument.

On dit que la variable "**a**" a été capturée dans le corps de l'abstraction après la **β-réduction**.

En conséquence, la **β-réduction** ne peut pas toujours être appliquée.

Il faut que les variables libres dans l'argument ne soient pas liées dans le corps de l'abstraction.

5.4- Variables libres et liées

Une occurrence d'une variable **x** est **liée** dans une lambda-expression **M** si elle apparaît dans **M** à l'intérieur d'une sous-expression de la forme **λx.E**.

Une occurrence d'une variable **x** est **libre** dans une lambda-expression **M** si elle n'est pas liée dans **M**.

Exemple :

$$(\lambda f.\lambda a.f\ a)\ b$$

La deuxième occurrence de **"a"** est liée dans l'expression. Cela veut dire que les deux occurrences de **"a"** désignent le même objet.
L'occurrence de **"b"** est libre dans l'expression, elle désigne un autre objet.

5.5- Retour à la bêta-réduction

Une expression de la forme **(λx.M) N** se réduit en **M** où les occurrences de la variable **x** sont remplacées par **N**. C'est à dire :

$$(\lambda x.M)\ N \xrightarrow{\beta} M[x := N]$$

S'il n'existe pas de variable dont une occurrence est libre dans **N** et une autre occurrence est liée dans **M**.

5.6- Alpha-réduction

La **α-réduction** est le renommage des variables dans une abstraction :

$$\lambda x.M \xrightarrow{\alpha} \lambda y.(M[x := y])$$

où **y** est une variable qui n'apparaît pas dans **M**.

Exemple :

$$\lambda a.f\ a \xrightarrow{\alpha} \lambda b.(f\ a)[a := b] \equiv \lambda b.f\ b$$

Voici en résumé l'ensemble des règles de réduction :

Les règles de réduction

$(\lambda x.M)\ N \xrightarrow[\beta]{} M\,[x := N]$	s'il n'existe pas de variable dont une occurrence est libre dans N et une autre liée dans M.
$\lambda x.M \xrightarrow[\alpha]{} \lambda y.(M\,[x := y])$	où y est une variable qui n'apparaît pas dans M.
$op\ c_1\ c_2 \ldots c_n \xrightarrow[\delta]{} c_0$	où op est un opérateur, les c_i $(i = 0..n)$ sont des constantes et c_0 exprime le résultat de l'opération.

5.7- Réduction généralisée

Notez que nous pouvons appliquer les réductions à des sous-expressions.

Exemple : Nous pouvons faire :

$$(\lambda x.(\lambda y.\lambda z.+\ z\ y)\ 4\ x)\ 3$$
$$\xrightarrow{\beta}(\lambda y.\lambda z.+\ z\ y)\ 4\ 3)$$

Ou bien :

$$(\lambda x.(\lambda y.\lambda z.+\ z\ y)\ 4\ x)\ 3$$
$$\xrightarrow{\beta}(\lambda x.(\lambda z.+\ z\ 4)\ x)\ 3$$

La vraie règle de calcul est la β-réduction ; on s'intéresse donc particulièrement aux sous-expressions que l'on peut réduire en utilisant cette règle.

Redex

Une sous-expression que l'on peut choisir de réduire par β-réduction (donc de la forme **(λx.M) N))** est appelée **redex**.
On peut indiquer l'endroit dans une lambda-expression où se trouve un redex en mettant un accent circonflexe en dessous de l'application, comme ceci : $(\lambda x.M)_{\wedge}N.$

Exemple : La lambda-expression de l'exemple précédent comporte donc deux redex :

$$(\lambda x.(\lambda y.\lambda z.+\ z\ y)_{\wedge}4\ x)_{\wedge}3$$

Le premier **redex** est plus à l'intérieur de l'expression que le deuxième.

Des réductions distinctes, c'est-à-dire des façons différentes de réduire une lambda-expression, peuvent a priori conduire à des résultats différents, ce qui nous amène à définir la notion de **forme normale** qui est commune à tous les systèmes de réécriture.

Forme normale

Lorsqu'une lambda-expression ne peut plus se réduire autrement que par **la α-réduction**, alors elle est en forme normale.
Lorsqu'une lambda-expression **M** se réduit en une lambda-expression **N** et que **N** est en forme normale, alors **N** est la forme normale de M.

Théorème de Church-Rosser : Si une même lambda-expression **M** se réduit en une lambda-expression **M1** (en choisissant certains redex) et en une autre lambda-expression **M2** (en choisissant d'autres redex), alors il existe une autre lambda-expression **N** telle que **M1 et M2** se réduisent en **N**.
Autrement dit, la réduction est **confluente**.

Exemple : Considérons à nouveau la lambda-expression **(λx.(λy. λx.+ x y) 4 x) 3** et deux réductions distinctes de cette lambda-expression.
– En choisissant toujours le redex le plus à l'intérieur :

$$(\lambda x.(\lambda y.\lambda x. + x\ y)_{\curlywedge} 4\ x)\ 3 \underset{\beta}{\rightarrow} (\lambda x.(\lambda x. + x\ 4)_{\curlywedge} x)\ 3$$
$$\underset{\alpha}{\rightarrow} (\lambda x.(\lambda z.+ z\ 4)_{\curlywedge} x)\ 3 \underset{\beta}{\rightarrow} (\lambda x.+ x\ 4)_{\curlywedge} 3 \underset{\beta}{\rightarrow} + 3\ 4 \underset{\delta}{\rightarrow} 7$$

On obtient la forme normale 7.
– En choisissant toujours le redex le plus à l'extérieur :

$$(\lambda x.(\lambda y.\lambda x. + x\ y)\ 4\ x)_{\curlywedge} 3 \underset{\beta}{\rightarrow} (\lambda y.\lambda x. + x\ y)_{\curlywedge} 4\ 3$$
$$\underset{\beta}{\rightarrow} (\lambda x.+ x\ 4)_{\curlywedge} 3 \underset{\beta}{\rightarrow} + 3\ 4 \underset{\delta}{\rightarrow} 7$$

On obtient bien la même forme normale.

Donc, toutes les réductions d'une même lambda-expression aboutissent à une même forme normale (à des **α-réductions** près) **si elles terminent**.
Cependant, une réduction **peut ne pas terminer** :

Exemple : En choisissant toujours le redex le plus à l'intérieur :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_{\wedge}(\lambda z.z\ z)) \overset{\beta}{\rightarrow} (\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_{\wedge}(\lambda z.z\ z)) \overset{\beta}{\rightarrow} \ldots$$

la réduction ne termine pas.
En choisissant toujours le redex le plus à l'extérieur :

$$(\lambda x.\lambda y.y)_{\wedge}((\lambda z.z\ z)\ (\lambda z.z\ z)) \overset{\beta}{\rightarrow} \lambda y.y$$

la réduction termine en une étape.

6. Stratégies de réduction

Une stratégie de réduction définie l'ordre dans lequel les redex sont utilisés.
La plupart des langages fonctionnels utilisent l'une de ces deux stratégies (avec quelques variantes) :

- **Ordre applicatif de réduction (AOR) :** consiste à choisir toujours le redex **interne.**
- **Ordre normal de réduction (NOR) :** consiste à choisir toujours le redex **externe.**

Exemple : Considérons l'expression :

$$(\lambda x.(\lambda a.*\ a\ a)\ x)\ ((\lambda y.y)\ 2)$$

Stratégie AOR :

$$(\lambda x.(\lambda a.*\ a\ a)_{\wedge}x)\ ((\lambda y.y)\ 2) \xrightarrow{\beta} (\lambda x.*\ x\ x)\ ((\lambda y.y)_{\wedge}2) \xrightarrow{\beta}$$
$$(\lambda x.*\ x\ x)_{\wedge}2) \xrightarrow{\beta} *\ 2\ 2) \xrightarrow{\delta} 4$$

Stratégie NOR :

$$(\lambda x.(\lambda a.*\ a\ a)\ x)_{\wedge}((\lambda y.y)\ 2) \xrightarrow{\beta} (\lambda a.*\ a\ a)_{\wedge}((\lambda y.y)\ 2) \xrightarrow{\beta}$$
$$*\ ((\lambda y.y)_{\wedge}2)\ ((\lambda y.y)\ 2) \xrightarrow{\beta} *\ 2\ ((\lambda y.y)_{\wedge}2) \xrightarrow{\beta} *\ 2\ 2 \xrightarrow{\delta} 4$$

Passage des arguments

- Avec **AOR**, l'argument est évalué avant l'application de la fonction. Ceci correspond au passage par valeur (le mode utilisé par le langage C, par exemple).
- Avec **NOR**, la fonction est appliquée avant l'évaluation de l'argument. Ceci correspond au passage par nom (le mode utilisé par le langage FORTRAN, par exemple).

Évaluation

- Avec **AOR** l'évaluation des arguments est faite dès que possible. Ceci correspond à l'évaluation affairée (**eager evaluation**).
- Avec **NOR** l'évaluation des arguments est faite le plus tard possible. Ceci correspond à l'évaluation paresseuse (**lazy evaluation**).

AOR vs. NOR

AOR est généralement plus rapide que **NOR**.
La raison est qu'il arrive fréquemment la situation où nous devons réduire une expression de la forme **(λx.M) N** où **M** est sous forme normale et contient plusieurs occurrences libres de **x**.
Avec **AOR**, l'expression **N** est réduite en premier et donc l'argument est évalué une seule fois.

Exemple : Considérez l'expression :

$$(\lambda a.*\ a\ a)\ ((\lambda y.y)\ 2)$$

Avec AOR, la sous-expression **((λy.y) 2)** sera évalué une seule fois.

Pourtant, AOR ne garantit pas la terminaison :
Exemple : Rappelez-vous de l'expression :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$$

Nous avons vu que la réduction ne termine pas alors qu'une forme normale existe : **λy.y**.

NOR est généralement moins efficace que **AOR**.
Pourtant, elle garantit la terminaison quand une **forme normale** existe.

Théorème de normalisation de Curry : L'ordre normal de réduction conduit à coup sûr à la forme normale lorsqu'elle existe.

Exemple : Nous avons vu que la forme normale de l'expression ci-dessous peut être trouvé avec **NOR** :

$$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$$

Il existe certains cas où **NOR** est plus efficace que **AOR**.

Exemple : Soit **(λx.M) N** où **M** est sous forme normale et il n'y a aucune occurrence libre de **x** dans **M**, alors **NOR** permet d'attendre la forme normale en une étape.

Stratégie du langage OCaml

Le langage OCaml utilise **AOR** avec une variation.

Exemple : Voici les étapes d'évaluation d'une expression OCaml qui correspond à la lambda-expression suivante :

$$(\lambda x.(\lambda a.*\ a\ a)\ x)\ ((\lambda y.y)\ 2)$$

```
(function x -> (function a -> a * a) x) ((function y -> y) 2)
(function x -> x * x) ((function y -> y) 2) =
(function x -> x * x) 2 = 2 * 2 = 4
```

Pourtant, OCaml associe à certaines constructions syntaxiques une évaluation spécifique. L'évaluation de l'alternative **if**, de la disjonction et conjonction logique sont dites **non-strictes** (ou court-circuitée).

Cependant, la définition avec les opérateurs **&& et ||** est dangereuse, car elle repose implicitement sur **l'ordre** dans lequel les opérandes apparaissent dans le programme.

Exemple : Par exemple, les deux définitions récursives suivantes (d'une fonction qui teste si un entier donné est un nombre impair positif ne donnent pas le même résultat)

```
let rec f n = n = 1 || (n > 1 && f (n - 2))
```

est correcte et termine.

Pourtant, celle-ci ne termine pas :

```
let rec f n = n = 1 || (f (n - 2) && n > 1)
```

Exercice 1 :

Répondre aux questions suivantes pour chacune de ces lambda-expressions :

a) $\lambda x.(x\ 3)$
b) $\lambda x.\lambda y.+\ (x\ 3)$
c) $\lambda a.a\ \lambda b.(b\ a)$

1) Indiquez si l'expression est une constante, une variable, une abstraction ou une application ? Même question pour toutes ses sous-expressions.
2) Quelle expression obtient-on en supprimant les parenthèses inutiles ?

Exercice 2:

Donner la signification en termes de fonctions de chacune de ces lambda-expressions :

a) $\lambda x.*\ 2\ x$
b) $\lambda x.\lambda y.+\ x\ y$
c) $\lambda x.+\ x$
d) $\lambda f.f\ 3$

Exercice 3:

Réduire les lambda-expressions suivantes en utilisant la règle de β-réduction autant que possible (ce sont des lambda-expressions pour lesquelles le problème de capture de variable ne se pose pas et où il n'y a qu'un redex à chaque étape) :

a) $(\lambda a.\lambda b.+\ a\ b)\ 5$
b) $(\lambda x.+\ x\ x)\ \lambda y.y\ z$
c) $(\lambda c.\lambda v.\lambda f.c\ v\ f)\ \lambda x.\lambda y.x$

Exercice 4 :

Pour chacune des lambda-expressions suivantes et pour chaque occurrence de variable, indiquer si cette occurrence est libre ou liée dans cette lambda-expression.

a) $\lambda x.\lambda y.(\lambda z.y)\ \lambda y.x$
b) $(\lambda x.\lambda y.\lambda z.y)\ \lambda y.x$
c) $(\lambda a.\lambda b.a)\ \lambda b.(\lambda a.a)\ b$
d) $\lambda free.bound\ (\lambda bound.\lambda free.free)\ bound$
e) $(\lambda x.x)\ y\ \lambda y.y$
f) $(\lambda x.x\ y)\ \lambda y.y$
g) $(\lambda x.\lambda y.x\ z\ (y\ z))\ \lambda x.y\ \lambda y.y$

Exercice 5 :

Pour chacune des lambda-expressions suivantes, indiquer avec le symbole ^ tous les redex qu'elle contient.

a) $(\lambda x.((\lambda z.z)\ x)\ y)\ \lambda y.y$
b) $\lambda x.\lambda y.z\ \lambda z.z\ \lambda x.y$
c) $(\lambda y.+\ ((\lambda x.x)\ y)\ y)\ ((\lambda y.*\ 2\ y)\ 1)$
d) $(((\lambda f.\lambda x.\ f\ x)\ \lambda x.\lambda a.x\ a)\ \lambda x.x)\ \lambda y.y$

Exercice 6 :

Réduire chacune des lambda-expressions de l'exercice précédent jusqu'à atteindre la forme normale si elle existe :
1) en utilisant l'ordre normal de réduction (NOR).
2) en utilisant l'ordre applicatif de réduction (AOR).
Indiquer le redex choisi avec le symbole ^ et utiliser la α-réduction seulement lorsque le problème de capture de variable se pose.

Solution des exercices

Solution des exercices du chapitre 1

Exercice 1 :

```
# let x = 2;;
val x : int = 2
# let x = 3
  in let y = x + 1
     in x + y;;
- : int = 7
# let x = 3 and y = x + 1
  in x + y;;
- : int = 6
```

Exercice 2 :

Correction : La définition de f utilise la première définition de x. Redéfinir x dans la suite n'a aucun effet sur la définition de f qui est déjà fixée.

Il faut bien comprendre que les définitions de variables en OCaml ne "modifient" pas une variable, il s'agit uniquement de donner un nom à une valeur (ce qui peut cacher une définition précédente de ce nom). C'est comme en mathématiques où on peut dire "soit x le plus petit réel tel que...", et longtemps après de la fonction qui "à x associe $3x+2$". On ne modifie pas x en l'utilisant pour deux usages différents, on se contente d'oublier l'ancienne définition.

Le concept de mémoire modifiable existe en OCaml (les références), mais il est complètement séparé la notion de variable – contrairement à d'autres langages comme Java qui mélangent les deux aspects. Cela peut sembler plus lourd pour les débutants, mais ça clarifie les concepts et permet de raisonner plus facilement sur le code que l'on écrit.

Exercice 3

Correction : let somme x y = x + y ;; somme (somme (somme 2 3) 4) (somme 2 (somme 3 4)) ;;

Exercice 4

```
let  est_pair x = x mod 2 = 0;;
```

Ou

```
let  est_pair x = if x mod 2 = 0 then true else false;;
```

Exercice 5

```
let signe x = if x < 0 then (-1)
              else  if x=0 then 0 else  1 ;;
```

Exercice 6

On peut envisager plusieurs façons de définir la fonction selon que l'on déclare la constante pi localement ou globalement. En tout cas on insiste sur l'intéret de nommer cette valeur. Ci dessous 3 solutions : préférer les 2 dernières qui encapsulent la constante pi qui a priori n'est nécessaire que pour calculer le volume de la sphère. Remarquons que dans la 3ème écriture, cube est une fonction locale.

```
let pi = 3.14;;

(* Interface sphere
type float -> float
argument : r le rayon de la spère
precondition : r positif ou nul
postcondition : (sphere r)= volume de la sphere de rayon r
Tests : sphere 3.4
*)
let sphere r =  4. /. 3. *. pi *. r *. r *. r;;

let sphere r = let pi = 3.14 in
                  4. /. 3. *. pi *. r *. r *. r;;

let sphere r = let pi = 3.14 in
                  let cube x = x *. x *. x in
                  4. /. 3. *. pi *. (cube r);;
```

Exercice 7 :

```
*(Interface max3 :
type 'a *'a *'a -> 'a
arguments : (a,b,c)
postcondition : max(a,b,c) = le plus grand des 3 entiers
Tests : (*place 3*)
         max3 (3,4,5); max3 (4,3,5);
        (*place 2*)
         max3 (2,5,4); max3 (4,5,3);
        (*place 1*)
         max3 (5,4,2); max3 (5, 2,4);
        (* 2 égaux *)
        max3 (3,3,1)
        (3 égaux)
        max3 (-6, -6, -6)
*)
let max3 (a,b,c) = if a>= b then if a >= c then a else c
                            else if b >= c then b else c;;
```

Autre solution : Proposer la solution qui consiste à écrire une fonction max2 (attention la fonction max prédéfinie est currifyée) qui calcule le plus grand de 2 nombres puis l'utiliser pour écrire max3

```
let max2 (a,b) = if a >= b then a else b;;

let max3 (a,b,c) = max2  (a, max2 (b,c));;
```

Pour les tests, envisagez les différents ordres possibles pour les nombres et compléter pour être exhaustif avc 3 différents, 2 différents et tous égaux.

Dernière chose : remarquons le type de la fonction : `('a * 'a * 'a* -> 'a`. *L'opérateur de comparaison est polymorphe : on peut appliquer cet opérateur sur tous les types.*

Exercice 8

```
let encadrer s = let cadre = "!!" in cadre ^ s ^ cadre;;

let encadrer2 (s,cadre) =  cadre ^ s ^ cadre;;

let encadre3 (gauche, s, droit) =  gauche ^ s ^ droit;;
```

Solution des exercices du chapitre 2

Exercice 1 :

```
# let app(f,y) = (f y ) + y;;
val app : (int -> int) * int -> int = <fun>
# let plus5 = fun x -> x+5;;
val plus5 : int -> int = <fun>
# let fois8 = fun x -> x*8;;
val fois8 : int -> int = <fun>
# app(plus5,12);;
- : int = 29
# app(fois8,12);;
- : int = 108
```

Exercice 2 :

```
#  let app_couple(f,c,a) =
  let (x1,x2)=c in f (x1+a) * f (x2+a) ;;

val app_couple : (int -> int) * (int * int) * int -> int = <fun>

# app_couple(plus5,(1,1),3);;
- : int = 81

# app_couple((fun y -> y * y),(1,1),3);;
- : int = 256
```

Exercice 3

```
# let decalage(f,a) = fun y -> f (y+a) ;;
val decalage : (int -> int) * int -> int -> int = <fun>

# decalage(plus5,8);;
- : int -> int = <fun>

# decalage(plus5,8) 1 ;;
- : int = 14
```

Exercice 4

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

# composition((fun x -> x+4),(fun y -> y*5)) 8 ;;
- : int = 44

# composition((fun y -> y*5),(fun x -> x+4)) 8;;
- : int = 60
```

Exercice 5

let maximum x y = if x > y then x else y ;;

val maximum : 'a -> 'a -> 'a = <fun>

Exercice 6

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

Exercice 7

```
# let sigma x =
    let rec sigma_rec x =  if x = 0 then 0 else x + sigma_rec (x-1) in
     if (x<0) then "erreur : argument négatif"
     else "sigma = " ^ (string_of_int (sigma_rec x)) ;;
val sigma : int -> string = <fun>
```

Exercice 8 :

```
# let rec longueur l =
# match l with
# | [] -> 0
# | x::r -> 1+(longueur r);;
val longueur : 'a list -> int = <fun>

# let rec size l =
    if null l then 0
    else 1 + (size (List.tl l)) ;;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int =

# let rec sum l =
# if l=[] then 0 else (List.hd l)+(sum (List.tl l));;
val sum : int list -> int = <fun>
# sum [2;6;9;3];;
- : int = 20
```

Exercice 9

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# List.map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

Solution des exercices du chapitre 3

Exercice 1 :

```
append : α list → α list → α list

let rec append a b =
  match a with
    [] -> b
  | h::t -> h :: append t b
```

```
        append [1; 2; 3] [4; 5; 6]
⟹      1 :: append [2; 3] [4; 5; 6]
⟹      1 :: 2 :: append [3] [4; 5; 6]
⟹      1 :: 2 :: 3 :: append [] [4; 5; 6]
⟹      1 :: 2 :: 3 :: [4; 5; 6]
⟹*     [1; 2; 3; 4; 5; 6]
```

Exercice 2 :

```
rev : α list → α list

let rec rev l =
  match l with
    [] -> []
  | h::t -> rev t @ [h]
```

```
        rev [1; 2; 3; 4]
⟹      rev [2; 3; 4] @ [1]
⟹      rev [3; 4] @ [2] @ [1]
⟹      rev [4] @ [3] @ [2] @ [1]
⟹      rev [] @ [4] @ [3] @ [2] @ [1]
⟹      [] @ [4] @ [3] @ [2] @ [1]
⟹*     [4; 3; 2; 1]
```

Exercice 3 :

```
odd_elements : α list → α list

let rec odd_elements l =
  match l with
    a::_::t -> a :: odd_elements t        there is something to skip over
  | _ -> l                                there is nothing to skip over
```

Exercice 4 :

```
let rec take n l =
  if n = 0 then [] else
    match l with
      h::t -> h :: take (n - 1) t
```

Exercice 5 :

```
let sol coeff = match coeff with
(0.,_,_) -> failwith "solutions : premier degre"
|(a,b,c) when b*. b -. 4.0 *. a *. c < 0. -> 0
|(a,b,c) when b*. b -. 4.0 *. a *. c = 0. -> 1
|(a,b,c) when b*. b -. 4.0 *. a *. c > 0. -> 2;;
```

Exercice 6 :

```
type expr =
  Num of int
| Add of expr * expr
| Subtract of expr * expr
| Multiply of expr * expr
| Divide of expr * expr
```

```
evaluate : expr → int

let rec evaluate e =
  match e with
    Num x -> x
  | Add (e, e') -> evaluate e + evaluate e'
  | Subtract (e, e') -> evaluate e - evaluate e'
  | Multiply (e, e') -> evaluate e * evaluate e'
  | Divide (e, e') -> evaluate e / evaluate e'
```

Exercice 7 :

a) We need two constructors – one for squares, which needs just a single integer (the length of a side), and one for rectangles.

```
type rect =
  Square of int
| Rectangle of int * int
```

b) On filtre sur l'argument

```
area : rect → int

let area r =
  match r with
    Square s -> s * s
  | Rectangle (w, h) -> w * h
```

Solution des exercices du chapitre : 4

Exercice 1 :

```
(* Question 1) *)

let swap elts = List.map (fun ((i,j),a) -> ((j,i),a)) elts;;

let transpose matA = match matA with
    ((n,m),elts) -> ((m,n),(swap elts));;
(* Exemple d'application : *)

transpose mat;;
(*- : (int * int) * ((int * int) * int) list = ((6, 4), [((3, 1),
7); ((6, 1), 4); ((3, 4), 5)]) *)

(* Question 2) *)

let  meme_indice ((i,j),_) ((i',j'),_) = i=i' && j=j';;

let somme_elt ((i,j),a) (_,b) = ((i,j),a+b);;

let rec ajoute_elt eltsA eltB = match eltsA with
  [] -> [eltB]
| eltA::r -> if meme_indice eltA eltB
             then (somme_elt eltA eltB)::r
             else eltA::(ajoute_elt r eltB);;

let ajoute_elts = List.fold_left ajoute_elt;;

let addition matA matB = match (matA, matB) with
  ((nA,mA),eltsA), ((nB,mB),eltsB) ->
      if nA=nB && mA=mB
      then ((nA,mA), (ajoute_elts eltsA eltsB))
      else failwith "bottom";;

(* Exemple d'application : *)

addition mat ((4,6),[((1,1),1);((2,2),1);((4,3),1);((4,4),1)]);;
(* - : (int * int) * ((int * int) * int) list =
((4, 6), [((1, 3), 7); ((1, 6), 4); ((4, 3), 6); ((1, 1), 1); ((2,
2), 1); ((4, 4), 1)])
*)
```

Exercice 2:

1)

Concaténation de toutes les chaînes d'une liste

```
#let rec implose l = match l with
      [] -> ""
   | t::r -> t^(implose r);;
val implose : string list -> string = <fun>
```

2)

Une définition du schéma *reduce*

En OCaml :

```
let rec reduce op a l = match l with
   [] -> a | t::r -> op t (reduce op a r);;
(* reduce : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

```
let rec reduce ( ^ ) "" [ "ab"; "cd"; "ef) ]
```

3)

```
let rec fold_left  op a l =
          match l with
          | [ ] -> a
          | h : : t ->  fold_left  op ( op a h ) t
```

Fold_left (^) "" ["ab"; "cd"; "ef"]

Exercice 3 :

1)

Concaténation des éléments d'une liste de listes

```
#let rec concatene_listes l = match l with
     []    -> []
   | t::r -> t @ concatene_listes r;;
val concatene_listes : 'a list list -> 'a list = <fun>
```

2)

```
# let rec fold_right op a l =
          match l with
          | [ ] -> a
          | h : : t -> op  h  ( fold_right   op a  t )
```

fold_right (@) [ ] [[1;2] ; [3;4] ; [5;6]]

Exercice 4 :

```
#let rec accumulateur_sur_listes f accu l  = match l with
     []  -> accu
   | t::r -> accumulateur_sur_listes f (f t accu) r;;
val accumulateur_sur_listes : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

Exercice 5 :

1)

```
# let rec min_2_listes   l1  l2 =
        match ( l1 , l2 ) with
        | ( [ ] , _ ) -> [ ]
        | ( _ , [ ] ) -> [ ]
        | ( h1 : : t1 , h2 : : t2 ) -> let res = if h1 <= h2 then h1 else h2 in res : : ( min_2_listes t1 t2 )
```

2)

```
# let rec map2  f  l1  l2  =
         match ( l1 , l2 ) with
         | ( [ ] , _ ) -> [ ]
         | ( _ , [ ] ) -> [ ]
         | ( h1 : : t1 , h2 : : t2 ) ->  f h1 h2 : : (map2 f  t1  t2)
```

```
#  map2 (fun (x,y) -> if x <= y then x else y) [1; 5;8;7;14;69] [18;0;6;45;12;21;11]
```

Solution des exercices du chapitre : 5

Exercice 1 :

a) **1)** $\lambda x.(x\ 3)$ est une abstraction de variable x et de corps $(x\ 3)$; la sous-expression $(x\ 3)$ est une application de x (qui est une variable) à 3 (qui est une constante).
2) $\lambda x.x\ 3$ (ici les parenthèses sont inutiles, car l'application est plus prioritaire que l'abstraction).

b) **1)** $\lambda x.\lambda y.+\ (x\ 3)$ est une abstraction de variable x et de corps $\lambda y.+\ (x\ 3)$; la sous-expression $\lambda y.+\ (x\ 3)$ est une abstraction de variable y et de corps $+\ (x\ 3)$; la sous-expression $+\ (x\ 3)$ est une application de $+$ (qui est une constante) à $x\ 3$ (qui est une application de x à 3).
2) $\lambda x.\lambda y.+\ (x\ 3)$ (on ne peut pas enlever les parenthèses, car l'application n'est pas associative à droite : elle est seulement associative à gauche).

c) **1)** $\lambda a.a\ \lambda b.(b\ a)$ est une abstraction de variable a et de corps $a\ \lambda b.(b\ a)$; la sous-expression $a\ \lambda b.(b\ a)$ est une application de a (qui est une variable) à la sous-expression $\lambda b.(b\ a)$; la sous-expression $\lambda b.(b\ a)$ est une abstraction de variable b et de corps $b\ a$; la sous-expression $b\ a$ est une application de b (qui est une variable) à a (qui est une variable).
2) $\lambda a.a\ \lambda b.b\ a$ (même remarque que pour l'expression a)).

Exercice 2:

a) $\lambda x.*\ 2\ x$ exprime la fonction anonyme qui à tout x associe $2x$.
b) $\lambda x.\lambda y.+\ x\ y$ exprime la fonction curryfiée à deux arguments (x et y) qui retourne la somme $x + y$.
c) $\lambda x.+\ x$ exprime la même fonction que l'expression b). On peut encore simplifier l'expression de cette fonction en lambda-calcul, en écrivant seulement +.
d) $\lambda f.\ f\ 3$ exprime une fonctionnelle qui à toute fonction f associe $f\ (3)$.

Exercice 3 :

a) $(\lambda a.\lambda b.+\ b\ a)\curlywedge 5\ 2 \underset{\beta}{\rightarrow} ((\lambda b.+\ b\ a)[a := 5])\ 2 \equiv \underset{\beta}{\rightarrow} (\lambda b.+\ b\ 5)\curlywedge 2 \underset{\beta}{\rightarrow} (+\ 2\ 5)$
On peut encore réduire cette lambda-expression en 7 par δ-réduction.

b) $(\lambda x.+\ x\ x)\curlywedge \lambda y.y\ z \underset{\beta}{\rightarrow} +\ (\lambda y.y\ z)\ (\lambda y.y\ z)$
C'est déjà fini ! Il n'y a plus de redex. D'après les conventions de parenthésage, on a : $+\ (\lambda y.y\ z)\ (\lambda y.y\ z) \equiv ((+\ (\lambda y.(y\ z)))\ (\lambda y.y\ z))$

c) $(\lambda c.\lambda v.\lambda f.c\ v\ f)\curlywedge \lambda x.\lambda y.x \underset{\beta}{\rightarrow} \lambda v.\lambda f.(\lambda x.\lambda y.x)\curlywedge v\ f \underset{\beta}{\rightarrow} \lambda v.\lambda f.(\lambda y.v)\curlywedge f \underset{\beta}{\rightarrow} \lambda v.\lambda f.v$

La dernière réduction est un cas particulier où il n'y a pas d'occurrence de la variable (ici y) dans le corps de l'abstraction (ici v) donc l'argument (f) disparaît, il ne reste que v.

Exercice 4 :

Les réponses sont données ci-dessous en colorant en rouge les occurrences de variable libres dans la lambda-expression ; les autres occurrences sont liées :

a) $\lambda x.\lambda y.(\lambda z.y)\ \lambda y.x$

b) $(\lambda x.\lambda y.\lambda z.y)\ \lambda y.x$

c) $(\lambda a.\lambda b.a)\ \lambda b.(\lambda a.a)\ b$

d) $\lambda free.bound\ (\lambda bound.\lambda free.free)\ bound$

e) $(\lambda x.x)\ y\ \lambda y.y$

f) $(\lambda x.x\ y)\ \lambda y.y$

g) $(\lambda x.\lambda y.x\ z\ (y\ z))\ \lambda x.y\ \lambda y.y$

Exercice 5 :

Les redex et le nombre de redex dans chaque lambda-expression sont indiqués ci-dessous :

a) $(\lambda x.((\lambda z.z)\curlywedge x)\ y)\curlywedge\lambda y.y$ (2 redex)

b) $\lambda x.\lambda y.z\ \lambda z.z\ \lambda x.y$ (0 redex !)

c) $(\lambda y.+\ ((\lambda x.x)\curlywedge y)\ y)\curlywedge((\lambda y.*\ 2\ y)\curlywedge 1)$ (3 redex)

d) $(((\lambda f.\lambda x.\ f\ x)\curlywedge\lambda x.\lambda a.x\ a)\ \lambda x.x)\ \lambda y.y$ (1 redex)

Exercice 6 :

1) Avec la stratégie NOR :

a) $(\lambda x.((\lambda z.z)\ x)\ y)\curlywedge\lambda y.y \xrightarrow[\beta]{} ((\lambda z.z)\curlywedge\lambda y.y)\ y \xrightarrow[\beta]{} (\lambda y.y)\ \curlywedge\ y \xrightarrow[\beta]{} y$

b) La lambda-expression $\lambda x.\lambda y.z\ \lambda z.z\ \lambda x.y$ est déjà sous forme normale (elle ne contient pas de redex).

c) $(\lambda y.+\ ((\lambda x.x)\ y)\ y)\curlywedge((\lambda y.*\ 2\ y)\ 1)$

$\xrightarrow[\beta]{} +\ ((\lambda x.x)\curlywedge\ ((\lambda y.*\ 2\ y)\ 1))\ ((\lambda y.*\ 2\ y)\ 1)$

$\xrightarrow[\beta]{} +\ ((\lambda y.*\ 2\ y)\curlywedge 1)\ ((\lambda y.*\ 2\ y)\ 1)$

$\xrightarrow[\beta]{} +\ (*\ 2\ 1)\ ((\lambda y.*\ 2\ y)\ 1) \xrightarrow[\delta]{} +\ 2\ ((\lambda y.*\ 2\ y)\curlywedge 1)$

$\xrightarrow[\beta]{} +\ 2\ (*\ 2\ 1) \xrightarrow[\delta]{} +\ 2\ 2 \xrightarrow[\delta]{} 4$

d) $(((\lambda f.\lambda x.f\ x)\curlywedge\lambda x.\lambda a.x\ a)\ \lambda x.x)\ \lambda y.y$

$\xrightarrow[\beta]{} ((\lambda x.(\lambda x.\lambda a.x\ a)\ x)\curlywedge\lambda x.x)\ \lambda y.y$

$\xrightarrow[\beta]{} ((\lambda x.\lambda a.x\ a)\curlywedge\lambda x.x)\ \lambda y.y \xrightarrow[\beta]{} (\lambda a.(\lambda x.x)\ a)\curlywedge\lambda y.y$

$\xrightarrow[\beta]{} (\lambda x.x)\curlywedge\lambda y.y \xrightarrow[\beta]{} \lambda y.y$

2) Avec la stratégie AOR :

a) $(\lambda x.((\lambda z.z)\curlywedge x)\ y)\ \lambda y.y \underset{\beta}{\rightarrow} (\lambda x.x\ y)\curlywedge\lambda y.y \underset{\beta}{\rightarrow} (\lambda y.y)\curlywedge y \underset{\beta}{\rightarrow} y$

b) Même remarque qu'avec la stratégie NOR.

c) $(\lambda y.+\ ((\lambda x.x)\curlywedge y)\ y)\ ((\lambda y.*\ 2\ y)\ 1)$

$\underset{\beta}{\rightarrow} (\lambda y.+\ y\ y)\ ((\lambda y.*\ 2\ y)\curlywedge 1)$

$\underset{\beta}{\rightarrow} (\lambda y.+\ y\ y)\ (*\ 2\ 1) \underset{\delta}{\rightarrow} (\lambda y.+\ y\ y)\curlywedge 2 \underset{\beta}{\rightarrow} +\ 2\ 2 \underset{\delta}{\rightarrow} 4$

d) $(((\lambda f.\lambda x.\ f\ x)\curlywedge\lambda x.\lambda a.x\ a)\ \lambda x.x)\ \lambda y.y$

$\underset{\beta}{\rightarrow} ((\lambda x.(\lambda x.\lambda a.x\ a)\curlywedge x)\ \lambda x.x)\ \lambda y.y$

$\underset{\beta}{\rightarrow} ((\lambda x.\lambda a.x\ a)\curlywedge\lambda x.x)\ \lambda y.y$

$\underset{\beta}{\rightarrow} (\lambda a.(\lambda x.x)\curlywedge a)\ \lambda y.y \underset{\beta}{\rightarrow} (\lambda a.a)\curlywedge\lambda y.y \underset{\beta}{\rightarrow} \lambda y.y$

Références

Conception, evolution, and application of functional programming languages », ACM Computing Surveys, vol. 21, no 3, septembre 1989, p. 359-411

Manuel M. T. Chakravarty, Gabriele Keller, « The risks and benefits of teaching purely functional programming in first year. », J. Funct. Program, vol. 14(1), 2004, p. 113-123

Chris Okasaki, « Purely functional data structures », Cambridge University Press New York, NY, USA, 1998 (ISBN 0-521-63124-6)

PETER HENDERSON, Functional Programming Application and Implementation,Prentice Hall, 1980.

ANTHONY J. FIELD, PETER G. HARRISSON, Functional Programming, Addison-Wesley, 1988.

SIMON PEYTON-JONES, Mise en œuvre des langages fonctionnels de programmation,Masson, 1990.

JEAN-LOUIS KRIVINE, Lambda-calcul, types et modèles, Elsevier Masson, 1990.

RICHARD BIRD, Introduction to Functional Programming using Haskell, 2e édition,Prentice Hall, 1998.

PIERRE WEIS et XAVIER LEROY, Le langage Caml, 2e édition, Dunod, 1999.

ANNE BRYGOO, TITOU DURAND, MARYSE PELLETIER, CHRISTIAN QUEIN-NEC, ET Programmation récursive (en Scheme), 2e édition, Dunod, 2004.