Introduction à la programmation fonctionnelle avec Ocaml

M.V. Aponte

15 mars 2010

1 Introduction

Depuis les débuts de l'informatique une très grande quantité de langages de programmation (voir figure 1 [1]) sont apparus, si bien qu'il est parfois difficile de faire un choix, de les comparer ou tout simplement de les distinguer. Les *paradigmes de programmation* sont à cet égard un précieux outil. Il s'agit de styles de programmation communément acceptés par les programmeurs, avec la particularité de bien s'adapter à certains types de problèmes, et auxquels adhèrent plus ou moins la grande majorité des langages. Parmi les styles les plus connus on trouve la *programmation impérative*, la *programmation objet*, la *programmation logique*, la *programmation fonctionnelle*. Peu de langages de appartiennent à une seule de ces catégories. Le plus souvent, ils sont hybrides : ils se classent majoritairement dans une catégorie, tout en intégrant des capacités intéressantes appartenant à d'autres styles. La connaissance de plusieurs paradigmes permet au programmeur de discerner parmi la jungle des langages, et de comprendre et d'exploiter au mieux les avantages des différents styles.

Ocaml est un langage fonctionnel incorporant plusieurs paradigmes de programmation. Il est issu du langage ML, et a récemment influencé le langage F# de la plate-forme `.NET` développée par Microsoft. Il incorpore des traits impératifs et objets ainsi que de puissantes constructions modulaires. Il est développé à l'INRIA depuis les années 80. Ocaml est un langage intéressant à plusieurs égards :

- Ocaml est fortement typé avec des types inférés statiquement et du polymorphisme paramétrique. Cela en fait un langage sûr et facile à employer.
- Ocaml possède les traits habituels des langages objets : classes, instances, héritage multiple et liaison retardée. Il propose aussi des constructions plus évoluées : parmi celles-ci les classes abstraites et les classes paramétrées.
- Ocaml possède un système de modules très génériques, un mécanisme de programmation par filtrage puissant autorisant une écriture de fonctions concise et claire, des structures des données récursives expressives et faciles à décrire et à exploiter via la programmation par filtrage.
- La sémantique de Ocaml est bien définie et la partie *purement fonctionnelle* est particulièrement adapté au raisonnement mathématique sur les programmes que nous aborderons en deuxième partie de ce cours.
- Ocaml possède un glaneur de cellules (*garbage collector*), et des nombreuses bibliothèques. Il peut exécuter des processus légers (*threads*) et communique sur Internet (ouverture de canaux de communication, applications client-serveur, etc.). Il offre également la possibilité d'interagir avec le langage C.

1. Source : `http://www.oreilly.com/news/graphics/prog_lang_poster.pdf`

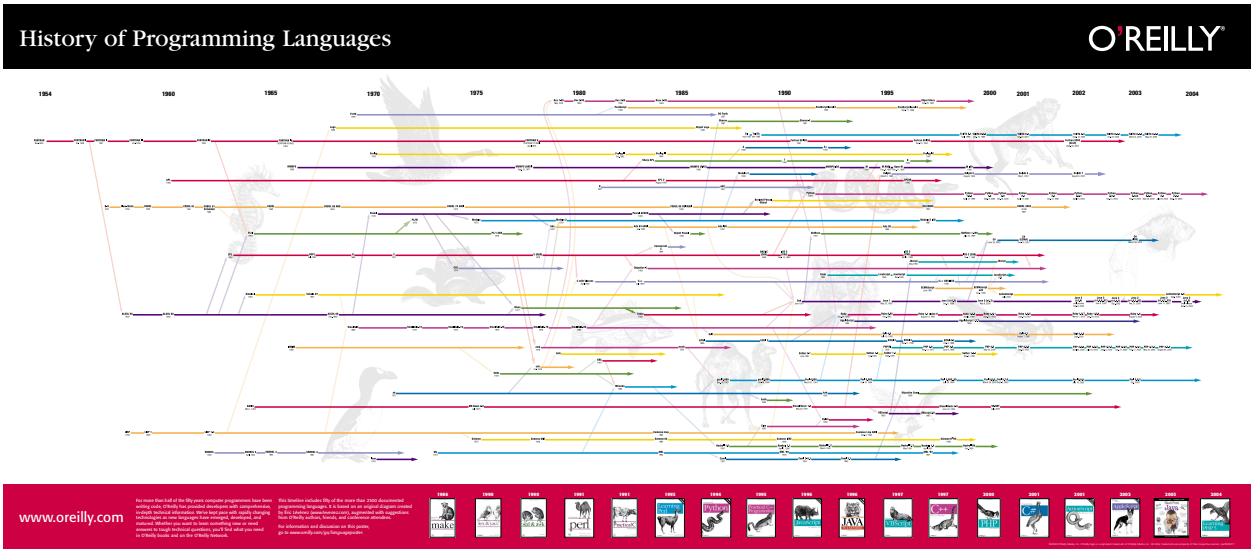


FIGURE 1 – La jungle des langages de programmation

– Ocaml est un langage compilé qui possède une boucle interactive : les tests y sont plus faciles. On peut également produire du code exécutable portable (*bytecode*) ou natif, dont l'efficacité est proche du code C dans certains cas.

Malgré ses origines académiques Ocaml est un langage dont l'utilisation dépasse le domaine de la recherche en langages et leurs outils associés. Il est utilisé pour l'enseignement dans des nombreuses universités en France, aux Etats Unis et au Japon, et dans les classes préparatoires françaises. Il est de plus en plus adopté en industrie : dans l'industrie aéronautique pour l'analyse des programmes, en vertu de sa sûreté de programmation (projet *Astrée* : Analyse Statique de logiciels Temps-RÉel Embarqués) ; pour le contrôle des systèmes critiques en avionique (Airbus) et dans le secteur nucléaire. Des grands groupes de l'industrie du logiciel utilisent Ocaml (`F#` :Microsoft, Intel, XenSource). Il est employé également pour écrire des applications dans le secteur financier (Jane Street Capital, Lexifi), par des projets libres comme MLDonkey (*peer-to-peer*), GeneWeb (logiciel de généalogie), Unison (logiciel de synchronisation de fichiers multi-plateforme dans certaines distributions Linux), par certains logiciels de l'environnement KDE, par des systèmes bio-informatiques pour l'analyse des protéines, et par beaucoup d'autres.

1.1 La programmation fonctionnelle

La *programmation fonctionnelle* est fondée sur la notion mathématique de fonction. Une fonction relie les éléments de deux ensembles : le *domaine* et le *co-domaine* de la fonction. Ainsi, à chaque valeur du *domaine*, la fonction fait correspond *une unique* valeur de son *co-domaine*.

$$\begin{array}{lccl} tri: & \text{suite d'articles} & \rightarrow & \text{suite d'articles triés} \\ & s & \mapsto & tri(s) \end{array}$$

La caractéristique essentielle des fonctions est d'associer une seule valeur image pour chaque valeur du domaine, de manière *stable et indépendante du contexte*. En mathématiques, une fonction appliquée sur un même argument, donne toujours le même résultat : elle ne change pas de résultat selon la valeur des

variables ou de "l'état du système". Les opérations des langages de programmation impératifs ne jouissent pas de cette stabilité. Voici par exemple une opération simple en langage C :

```
int i = 0;

int f (int j) {
  i = i+j;
  return i;
}
```

L'opération `f` ne correspond pas à une fonction : son résultat n'est pas seulement fonction de son argument `j` mais aussi de la valeur courante de la variable globale `i`, qui de plus est modifiée lors de chaque appel. Le résultat obtenu *dépend* donc de l'état des variables au moment de l'appel, et *modifie* directement cet état. L'opération `f` ne peut pas être étudiée en dehors de son contexte.

L'idée derrière la programmation fonctionnelle est de favoriser la réalisation des calculs à l'aide de fonctions au sens mathématique du terme. Ceci afin de tirer parti de leur séparation du contexte et de la stabilité qui en découle.

1.1.1 Les fonctions vues comme des valeurs

En programmation fonctionnelle les fonctions sont des valeurs au même titre que les entiers, les enregistrements ou les tableaux. Ainsi par exemple, il est possible de stocker une fonction dans une variable ou dans un tableau afin de l'appliquer plus tard à son argument, ou de passer une fonction en paramètre à une autre fonction. Cela élève les fonctions au statut des données. On parle alors de *pleine fonctionnalité* ou *ordre supérieur* ou encore des fonctions comme étant des entités de *première classe*. Il en découle un mécanisme de généralisation assez puissant. Considérons par exemple la fonction de tri d'un ensemble d'articles. Son fonctionnement dépend du critère d'ordonnancement à employer pour réaliser le tri. Plutôt que de fixer celui-ci dans la fonction de tri, on peut le lui passer en tant qu'argument. La fonction de tri devient générale à tout critère et peut être employée dans beaucoup de contextes.

$$\begin{array}{rccc} tri: & \text{critère de comparaison} \times \text{suite d'articles} & \rightarrow & \text{suite d'articles triés} \\ & (f_{comp}, s) & \mapsto & tri(f_{comp}, s) \end{array}$$

Cette généralisation en Ocaml prendra la forme :

```
let tri(f_compare, articles) = ...
```

Dans d'autres langages, ce genre de généralisation est plus lourde à implanter. Par exemple, en C, il faudra transmettre la fonction au moyen d'un pointeur, en Java il faudra transmettre la fonction à l'intérieur d'un objet dont on aura définit la classe ou l'interface auparavant.

```
interface Order {
   boolean compare(Object a, Object b);
}
Object[] tri (Object[] articles, Order o);
```

1.2 La programmation générique par les types

L'idée derrière la programmation générique ou *généricité* est de *ne pas restreindre inutilement* le type des données et des fonction, mais au contraire, de le rendre génériques à toute valeur compatible avec leur définition. En pratique, ceci est fait par le biais de *paramètres de types* qui jouent le rôle de type générique, pouvant être remplacé plus tard par un type spécifique. Par exemple, en Java 1.5 on peut définir le type d'une cellule (dans une liste chaînée) dont la nature du contenu dépend du paramètre de type A :

```
class Cellule<A> {A contenu; Cellule suivant};
```

En C++, on utilisera les partons (*templates*) pour généraliser le type d'une fonction. Par exemple, voici la fonction *identité* qui renvoie son arugment sans lui appliquer aucune opération :

```
template <typename A>
A identity (A x) {
  return x;
}
```

Ce mécanisme de généralisation est présent dans beaucoup de langages : C++, Java, Ada, Haskell, Eiffel, SML, Ocaml. Il permet de concilier flexibilité du typage avec la rigueur du typage statique ; il favorise fortement la réutilisation du code, et permet également le typage plus précis (avec moins d'erreurs) du code générique. Il s'agit d'un exemple réussi de mélange de styles de programmation dans la mesure où il a été adopté par différents langages de styles très divers.

1.3 Avantages et contraintes de la programmation fonctionnelle

La programmation fonctionnelle *pure* (sans effets de bord) présente beaucoup d'avantages :

1. *"Modularité" de la programmation, simplification du débogagge* : les fonctions au sens mathématique peuvent être considérées comme des *boîtes noires* au sein des programmes. Il devient plus facile de les concevoir séparément, de circonscrire leurs effets, et d'isoler les erreurs. Par ailleurs, la généralisation des fonctions par le biais de paramètres fonctionnels favorise la structuration des programmes et la réutilisation d'algorithmes.
2. *Formalisme mathématique sous-jacent* : cela permet la vérification du langage (compilateur, typage, sémantique formelle) mais aussi des programmes via diverses analyses, tests ou à l'aide de prouveurs automatiques.
3. *Généricité, flexibilité, réutilisation* : la programmation générique par les types favorise la réutilisation du code tout en assurant un typage précis. Les fonctions et les données deviennent plus génériques et donc potentiellement réutilisables.

Ces avantages ont un prix : les contraintes imposées par l'usage de la programmation fonctionnelle pure, parmi lesquelles on peut mentionner :

- Pas d'effets de bords : pas d'affectation, pas de variables modifiables. Seules les constantes sont admises ;
- Une fonction ne peut pas changer ce qui lui est extérieur ;
- Le comportement d'une fonction ne peut pas être modifié de l'extérieur ;
- Certaines constructions sont proscrites : boucles itératives, *goto*.
- La récursivité est à utiliser afin de coder les boucles.
- La gestion implicite de la mémoire devient indispensable.

2 Débuter avec Ocaml : quelques principes

Ocaml est un langage compilé. Au choix, il peut être compilé vers un langage intermédiaire (*bytecode*) ou vers du code natif avec des performances comparables à celles du code C. Il possède également un mode interactif où il analyse et répond à chaque phrase entrée par l'utilisateur.

2.1 Le mode interactif

Pour lancer Ocaml en mode interactif on tape la commande `ocaml` dans la fenêtre des commandes (sous Unix). Vous aurez la réponse :

```
% ocaml
        Objective Caml version 3.07

#
```

Le caractère `#` invite l'utilisateur à entrer une phrase écrite dans la syntaxe Ocaml , phrase qui par exemple nomme une valeur, explicite une définition ou décrit un algorithme. En mode interactif, chaque phrase Ocaml doit terminer par `;;` puis l'utilisateur valide sa frappe par un retour chariot. Dès lors, Ocaml analyse la phrase :

- calcule son type (*inférence des types*),
- la traduit en langage exécutable (*compilation*)
- et enfin l'*exécute* afin de fournir la réponse demandée.

En mode interactif, Ocaml *donne systématiquement une réponse* qui contient :

- Le *nom de la variable déclarée* s'il y en a.
- Le *type inféré* par le typage.
- La *valeur calculée* par l'exécution.

```
# let x = 4+2;;
val x : int = 6
```

La réponse d'Ocaml signale que l'identificateur `x` est déclaré `(val x)`, avec le type des entiers `(:int)` et la valeur calculée 6 `(=6)`. Un exemple qui utilise la valeur `x` déjà déclarée :

```
# x * 3;;
- : int = 18
```

2.2 Le mode compilé

Considérons un fichier source `hello.ml` avec le code suivant :

```
print_string "Hello World!\n";;
```

Sous une fenêtre de commande Unix nous pouvons compiler, lier et exécuter ce programme par la commande :

```
ocamlc -o hello hello.ml
./hello
Hello World!
```

2.3 Quelques principes

Valeurs de première classe Un valeur est dite de *première classe* si elle peut être reçue et/ou passée en argument à une fonction, ou si elle peut être stockée dans une variable. Les valeurs manipulées en Ocaml sont les objets, les valeurs de base (de type int, bool, string, etc.), les fonctions, les valeurs de types construits, prédéfinis (listes, tableaux, etc) ou définis par l'utilisateur. Toutes ces valeurs sont de première classe.

Pas de variables modifiables Dans la partie strictement fonctionnelle d'Ocaml il n'y a pas de variables modifiables. Il est seulement possible de déclarer des constantes nommées (identificateurs), mais leur valeur ne pourra changer au cours du programme. Par exemple

```
let a = 7;
val a: int = 7
```

permet de déclarer une nouvelle constante `a` *liée* à la valeur 7.

Environnement Ensemble de *liaisons* entre identificateurs *ayant été déclarés* et leurs valeurs. On distingue deux sortes d'environnement : *global* et *local* avec respectivement des liaisons globales et locales.

Typage fort et inférence de types Ocaml est un langage fortement typé : toute phrase du langage se voit attribuer un type. La compilation échoue si le typage du programme est incorrecte. Mais aucune annotation de types n'est nécessaire pour typer les valeurs : le typeur *infère* leurs types, y compris pour les fonctions (qui sont des valeurs comme les autres). Par exemple, un identificateur lié à un entier par un `let` se voit attribuer le type entier ; une fonction qui effectue une opération de concaténation sur son argument se voit attribuer un type où l'argument est de type `string`.

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

# let begaie x = x^"-"^x;;
  val begaie : string -> string
```

Les expressions Constructions du langage dont le but est de calculer une valeur dite *résultat* de l'expression. Une expression est constituée soit d'une constante, soit d'opérations et/ou appels de fonctions. On *évalue* une expression afin d'obtenir son résultat. Le *type d'une expression* est celui de sa valeur résultat. En supposant que `x` vaut -2, et que la fonction `f` teste si son argument est positif, voici quelques exemples d'expressions avec leurs valeurs et leurs types :

expression	valeur	type
`5`	5	`int`
`3 + x`	1	`int`
`f(3)`	true	`bool`
`f(x) || x=7`	false	`bool`
`if x>0 then 1 else 2`	2	`int`

Toute expression a un type Ocaml étant un langage fortement typé, toute expression se doit d'avoir un type. Dans le tableau précédent, notez par exemple que la conditionnelle `if x>0 then 1 else 2` est une expression, et en tant que telle permet de calculer une valeur (1 ou 2 selon la valeur de `x`). Par conséquent

son type est le type de cette valeur calculée, à savoir `int`.

Les programmes en Ocaml sont constitués uniquement de déclarations et d'expressions Ocaml est un langage *déclaratif*, autrement dit, le style d'écriture privilégié est celui de *la description de calculs à réaliser*.

3 Déclarations et types

3.1 Déclarations globales et locales

Déclarations globales Un identificateur est déclaré globalement par le mot clef `let` :

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

# x+2;;
- : int = 9

# let y = x+7;;
- : int = 14
```

L'identificateur ainsi déclaré est ajouté dans l'environnement global du programme. Dès lors, il peut être utilisé lors d'un calcul ultérieur.

Déclaration locale à une expression Un identificateur est déclaré localement **à une expression** par la construction `let ... in` :

```
#let y = 5 in x *y;
 - : int =45
```

Une déclaration locale permet de constituer un environnement local visible seulement dans l'expression qui suit le `in`. L'expression après le `in` est évaluée dans l'environnement global augmenté de cet environnement local déclaré avant le `in`. La construction `let ... in` **est une expression**, comme le montre le message donné par la boucle interactive : il n'y a pas d'indication `val` $nomVar$`:`, autrement dit, aucun identificateur n'est déclaré. En revanche, le résultat de l'expression après le `in` est calculé et renvoyé en résultat final.

```
#let y = 3 in x+y;;
- : int = 10
#y;;
Unbound value y
```

Redéfinitions et masquage : Les identificateur déclarés jusqu'ici sont constants : la liaison entre un identificateur `x` et sa valeur v, au moyen d'une déclaration `let x` $=v$, ne peut être modifiée. On pourra introduire une nouvelle liaison pour x, `let x` $= v_1$ qui masque la première dans la suite du programme, mais on ne pourra pas la modifier.

3.2 Les types

Comme la plupart des langages Ocaml offre des *types de base* prédéfinis et des types *composites*, dont certains sont prédéfinis et d'autres sont à déclarer par l'utilisateur.

– *types de base* prédéfinis : entiers, booléans, caractères, réels et le type *vide*. Le tableau suivant exhibe les types de base plus un type *composite* prédéfini : les chaînes de caractères. Pour chaque type nous donnons quelques exemples de constantes du type ainsi que les opérations primitives définies sur celui-ci. Le symbole ^ est l'opérateur de concaténation des chaînes de caractères.

Type de base	**Constantes**	**Primitives**
`unit`	`()`	pas d'opération !
`bool`	`true false`	`&& || not`
`char`	`'a' '\n' '\097'`	`code, chr`
`int`	`1 2 3`	`+ - * / max_int`
`float`	`1.0 2. 3.14`	`+. -. *. /.` cos
`string` (composite)	`"a\tb\010c\n"`	`^ s.[i] s.[i] <- c`

– *types composites* : nous indiquons s'il s'agit d'un type prédéfini ou à définir par l'utilisateur, et s'il s'agit ou non d'un type modifiable, nous donnons un exemple, et enfin le nom du type en Ocaml.

Types composites	**Caractéristiques**	**Exemple**	**Type Ocaml**
tableaux	prédéfini, modifiable	`[<1;2;3>]`	`int array`
chaînes	prédéfini, modifiable	`"bonjour"`	`string`
n-uplet	prédéfini	`(4,"abc",true)`	`int*string*bool`
enregistrement	à déclarer	`{nom="Dupond";num=017789}`	$< nomType >$
inductif	à déclarer	`Noeud(1, Vide, Vide)`	$< nomType >$
listes	prédéfinis	`['a'; 'b'; '7']`	`char list`
objets	à déclarer	non présenté	$< nomType >$

Primitives de comparaison : `=, > , < , >=, <=, <>`. Elles sont valables pour tous les types (y compris les types composites et les fonctions). Appliqués sur des types composites, la comparaison suit les règles d'une *comparaison lexicographique* : on compare les composantes de chaque valeur en tenant compte de leurs positions respectives.

Exemples :

```
# 1+ 2;;
- : int = 3

# 1.5 +. 2.3;;
- : float = 3.8

# let x = "cou" in x^x;;
- : string = "coucou"

# 2 > 7;;
- : bool = false

# "bonjour" > "bon";;
- : bool = true

# "durand" < "martin";;
- : bool = true
```

```
# "ab" = "ba";;
- : bool = false
```

3.2.1 Types n-uplets

Un n-uplet est un assemblage de n valeurs v_i de types hétérogènes, séparés par des virgules et possiblement entourés de parenthèses : $(v_1, v_2, \ldots v_n)$

```
# (1, true);;
 -: int * bool = (1, true)

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

Un n-uplet est semblable à un enregistrement à n champs *sans étiquettes*. L'avantage des n-uplets sur les enregistrements réside dans leur utilisation primitive, sans déclaration préalable. De son côté, un type enregistrement doit être déclaré avant toute utilisation.

Le type d'un n-uplet : Le type d'un n-uplet $(v_1, v_2, \ldots, v_n)$ est $t_1 * t_2 \ldots * t_n$, où t_i est le type de la composante v_i. Il s'agit du *produit cartésien* des types de chacune des composantes.

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

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

3.2.2 Le type enregistrement

Un enregistrement est une *collection de valeurs de types hétérogènes*, où chaque valeur est accompagnée d'une étiquette permettant de sélectionner la valeur qui l'accompagne. Les enregistrements sont des types à déclarer par le programmeur.

```
# type client = {numero: int; nom: string; solde: float};;
type client = { numero: int; nom: string; solde: float }
```

On *construit* une constante d'enregistrement en donnant des valeurs pour chacun de ses champs déclarés :

```
# let durand = { numero = 265; nom = "Durand";
                 solde=  0.0};;
val durand : client={numero=265; nom="Durand"; solde=0}
```

On *sélectionne* la valeur d'un champ `c`, dans l'enregistrement `e`, par la notation `e.c`

```
# durand.numero;;
- : int = 265
```

à moins de préciser le contraire, les champs d'un enregistrement ne sont pas modifiables : on ne peut pas changer les valeurs dans les champs, mais on peut les recopier. La fonction suivante prend un client et un nouveau solde et attribue ce nouveau solde au client. Pour cela, elle construit un nouvel enregistrement où sont recopiés tous les champs inchangés :

```
# let nouveau_solde c s = {numero = c.numero; nom = c.nom; solde = s};;
  val nouveau_solde: client -> float -> client = <fun>

# let durand = { numero = 265; nom = "Durand"; solde= 0.0};;
  val durand : client = {numero=265; nom="Durand"; solde=0}

# let durandBis = nouveau_solde durand 50.34;;
  val durandBis : client = {numero = 265; nom = "Durand"; solde = 50.34}
```

4 Les fonctions

Déclarer une fonction : Un identificateur de fonction est déclaré comme tout autre identificateur, à l'aide d'un `let`. La syntaxe d'une déclaration de fonction est :

```
let <nom-fonction> <param> = <corps-fonction>

let <nom-fonction> <(param_1, param_2, ... param_n)> = <corps-fonction>

let <nom-fonction> <param_1> <param_2> ... <param_n> = <corps-fonction>
```

La liste de paramètres peut être constitué d'un unique paramètre (entouré ou non de parenthèses), d'un *n-uplet* de paramètres [2] (entouré de parenthèses) ou de plusieurs paramètres séparés par des espaces. Voici un premier exemple de fonction avec un seul paramètre :

```
# let double (y) = y*2;;
val double : int -> int = < fun >
```

Le type d'une fonction : est noté `t` $\rightarrow$ `q`, où `t` est le type d l'argument et `q` celui du résultat de la fonction. Dans le cas de la fonction `double`, l'argument et le résultat sont de type entier.

Appliquer une fonction : On applique une fonction en la faisant suivre de son argument (éventuellement entre parenthèses) :

```
# double(9);;
- : int = 18

# double 9;;
- : int = 18

# let z = double(x) ;;
val z : int = 12
```

2. Autrement dit, d'une valeur de *type n-uplet*.

4.1 Les fonctions à plusieurs arguments

Une fonction peut prendre plusieurs arguments : soit sous la forme d'un n-uplet d'arguments, soit en séparant les noms des paramètres par des espaces, avant la définition du corps.

Fonction avec n-uplet d'arguments : Les arguments de la fonction sont encapsulés dans un n-uplet. Techniquement, la fonction prend *un seul argument* qui est constitué par un n-uplet de valeurs :

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

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

La fonction `somme` possède *un seul* argument, qui *est ici une paire*. Ceci est reflété par le type de la fonction : `int * int` $\rightarrow$ `int`, où `int * int` est le type de l'argument, et `int` est le type de son résultat.

Type d'un fonction avec n-uplet d'arguments : La fonction :

$$\texttt{let f}(x_1, x_2, \ldots, x_n) = corps$$

possède un argument, qui est un n-uplet de valeurs $(x_1, x_2, \ldots\ x_n)$ et un résultat calculé par *corps*. Le type du n-uplet $(x_1, x_2, \ldots, x_n)$ est $t_1 * t_2 \ldots * t_n$, où t_i est le type de chaque x_i. Donc, le type de f est :

$$t_1 * t_2 \ldots * t_n \rightarrow t_{corps}$$

Fonction avec plusieurs arguments séparés : Les arguments de la fonction sont énumères séparément, et sont séparés par des espaces :

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

Cette fonction prend deux arguments `x` et `y` de type `float` (les deux premiers `float` dans le type de la fonction), et renvoie un résultat de ce même type : `float` (le troisième `float` du type). L'application de la fonction se fait en faisant suivre le nom de la fonction de tous ses arguments donnés un par un et séparés par des espaces :

```
# moyenne 15.0 12.6;;
- : float = 13.8
```

Type d'une fonction avec argument séparés : Si f est une fonction à deux arguments, son type est alors de la forme : $t_1 \rightarrow t_2 \rightarrow t_{corps}$ où, t_1, t_2 sont les types des arguments, et t_{corps} est le type du résultat. On notera donc, que le type d'une fonction **contient autant de flèches qu'il y d'arguments séparés** dans la fonction.

4.2 Les fonctions récursives

En programmation impérative les processus répétitifs sont programmés à l'aide de *boucles itératives*, dont la condition d'arrêt dépend souvent de la valeur d'une variable qui est modifiée au sein de la boucle. En programmation fonctionnelle pure, les variables modifiables étant proscrites, on se sert de la *récursivité* pour écrire les boucles. Supposons qu'on ait à écrire une fonction qui calcule la factorielle $n!$ d'un entier $n \geq 0$, selon la formule :

$$\begin{aligned} 0! &= 1 \\ n! &= 1 \times 2 \times \ldots \times (n-1) \times n \end{aligned}$$

Par exemple, $4! = 1 \times 2 \times 3 \times 4 = 24$. On peut écrire cette fonction en Ocaml de manière récursive :

```
# let rec fact n =
    if n<2 then 1  else n*fact (n-1);;
val fact : int -> int = <fun>

# fact 5;;
- : int = 120
```

Examinons le comportement de `fact`. Si $n < 2$, à savoir, si $n = 1$ ou $n = 0$, la fonction retourne le résultat 1, qui est le bon résultat aussi bien pour 0! que pour 1!. Si $n \geq 2$ la fonction calcule $n \times fact(n-1)$. Cela revient à recommencer un nouvel appel à `fact` mais cette fois, pour un argument qui n'est plus n mais $(n-1)$. On pourrait résumer ce cas du calcul comme ceci : *pour calculer* $fact(n)$ *il suffit de calculer* $fact(n-1)$ *puis de multiplier ce résultat par* n. Comme le symbole `fact` apparaît dans la définition de la fonction, cette définition est *récursive* et nous devons utiliser le mot clé `rec`. Déroulons un appel à la fonction pour $n = 5$:

```
# fact(5) =  5 * fact (4)
          =  5 * 4 * fact(3)
          =  5 * 4 * 3 * fact(2)
          =  5 * 4 * 3 * 2 * fact(1)
          =  5 * 4 * 3 * 2 * 1
- : int = 120
```

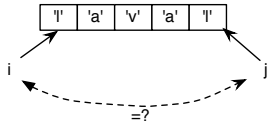
Correction de fonctions récursives : Un programme est *correcte* s'il calcule ce que nous voulons qu'il calcule. Comment nous convaincre de la correction d'une fonction récursive ? Notre raisonnement doit se pencher sur les résultats de la fonction pour *tous ses cas d'entrées possibles*. Plus tard dans ce cours nous utiliserons des techniques mathématiques simples pour ce faire, telles que le raisonnement par récurrence. Nous en donnons un bref aperçu ici. Dans un raisonnement par récurrence nous procédons en deux temps. En premier nous devons montrer que la fonction se comporte correctement dans son *cas le plus simple*. Ensuite, nous devons monter que *si la fonction est correcte pour un cas donné*, alors elle aussi correcte pour le *cas suivant*. Pour la fonction `fact` plus haut cela donne :

- `fact(0), fact(1)` sont les cas les plus simples. Dans ces deux cas $n < 2$ et la réponse est 1 qui est le résultat correct pour 0! et 1!. Donc `fact(0)` calcule bien 0! et `fact(1)` calcule 1!
- Supposons que `fact(n-1)` donne la réponse correcte, à savoir $(n-1)!$. Nous devons monter que le cas suivant `fact(n)` calcule $n!$. Comme $n \geq 2$ (autrement nous serions dans les cas précédents), la réponse calculée sera : `n*fact(n-1)`. Comme nous avons supposé que `fact(n-1)` = $(n-1)!$, il devient évident que `fact(n)` = `n*fact(n-1)` = $n * (n-1)! = n!$: la réponse correcte pour `fact(n)`.

Nous reviendrons en détail sur ce genre de raisonnement plus tard dans ce cours.

4.2.1 Fonctions récursives locales

L'itération en programmation impérative utilise l'affectation et les tests sur des variables modifiables afin de gérer les conditions d'arrêt. L'écriture récursive de l'itération nécessite parfois la définition de fonctions récursives prenant en paramètres supplémentaires les valeurs servant à gérer ces conditions d'arrêt. Considérons l'exemple de la fonction `palindrome` qui doit tester si une chaîne de caractères passée en paramètre est un palindrome. Un algorithme classique consiste à se munir de deux indices i, j initialisées respectivement avec la première et la dernière position de la chaîne :



- tant que $i < j$ et que les caractères à ces indices sont égaux, on incrémente i, on décrémente j, et on continue ;
- si les caractères sont différents, on arrête et on renvoie `false` ;
- si $i \geq j$ cela signifie qu'on a comparés tous les caractères sans tomber sur des caractères différents : on peut terminer avec `true`.

Dans cet algorithme, ce sont les valeurs de i, j qui changent à chaque itération, et qui déterminent la condition d'arrêt. Nous allons écrire cet algorithme récursivement à l'aide d'une fonction `palinRec` qui prend i, j en paramètres : elle testera les valeurs de i, j et au cas où l'on peut continuer, elle fera un nouvel appel récursif avec comme nouveaux paramètres $i + 1$ et $j - 1$. Cette fonction implante la boucle récursive. Elle est locale à la fonction `palindrome` que nous souhaitons écrire.

```
let palindrome s =
  let rec palinRec (i, j) =
     if (i<j) then
        if (s.[i] = s.[j])
            then palinRec ((i+1),(j-1))
        else false
     else true
  in palinRec (0, (String.length s - 1));;
val palindrome : string -> bool = <fun>
```

La fonction `palindrome` n'est pas récursive. Elle prend une chaîne s et se contente de faire un appel initial à sa fonction locale `palinRec`, qui elle est récursive, et qui implante l'itération de notre algorithme. Notez également que la fonction `palinRec` n'a pas besoin de prendre la chaîne `s` en argument : elle est visible en tant que paramètre de la fonction principale `palindrome`.

4.3 Les fonctions comme paramètres

Les fonctions peuvent recevoir en argument n'importe quelle valeur, y compris d'autres fonctions, puisque celles-ci sont des valeurs de première classe. Il s'agit d'un mécanisme fondamental de généralisation de programmes, au coeur de la programmation fonctionnelle.

Considérons l'exemple d'une fonction qui détermine, à partir de deux notes passées en argument, si un élève est reçu ou non ainsi que sa note finale. La fonction retourne une chaîne de caractères contenant ces deux résultats. Il y a deux paramètres à partir desquels le calcul peut s'articuler : le mode de calcul à appliquer aux deux notes (quel poids leur donner), et la note moyenne et maximale de référence afin de tester si l'élève est reçu (ex : 5 si la note maximale est 10, 10 si la note maximale est 20). Les arguments de cette fonction sont :

- `(n1,n2)` : une paire de notes ;
- `modeCalcul` : fonction à appliquer sur la paire des notes ;
- `(rMoy,rMax)` : notes moyenne et maximale de référence servant à déterminer si un élève est ou non reçu ;

```
let resEleve modeCalcul (rMoy,rMax) (n1,n2)=
  let noteFinale = modeCalcul (n1,n2) in
  let recu = if (noteFinale >= rMoy) then "Recu"
             else "Non recu" in
  let note = string_of_float(noteFinale) in
  let max =  string_of_float(rMax)
  in recu^" avec "^note^"/"^max;;
val resEleve : ('a * 'b -> float) -> float * float -> 'a * 'b -> string = <fun>
```

Examinons maintenant les calculs effectués dans le corps de cette fonction. Plusieurs résultats intermédiaires sont stockés dans des variables locales (tous les `let` imbriqués) et un calcul final est effectué (après le dernier `in`) et renvoyé par la fonction :

- `noteFinale` : variable locale qui contient le résultat d'appliquer le mode de calcul (fonction `modeCalcul`) sur la paire des notes ;
- `recu` : variable locale qui contient la chaîne "Recu" ou "Non recu" selon la valeur de `noteFinale` ;
- `note` : variable locale avec la note finale transformée en chaîne de caractères ;
- `max` : variable locale avec la note maximale transformée en chaîne de caractères ;
- ligne finale : c'est le résultat renvoyé par la fonction. Il s'agit d'une chaîne qui contient tous les résultats intermédiaires calculés.

Nous pouvons utiliser cette fonction afin de calculer le message à composer pour un élève dont on calcule la moyenne de deux notes avec 10/20 points de moyenne, ou encore pour un calcul sur la base d'une moyenne pondérée avec 40% pour la première note, et 60% pour la deuxième, et une moyenne de référence à 45 sur 100 points.

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

# let moyPonderee (x,y) = (x*.0.4+.y*.0.6);;
val moyPonderee : float * float -> float = <fun>

# let noteAnglais =
   resEleve moySimple (10.0,20.0) (11.0, 12.0);;
val noteAnglais : string = "Recu avec la note de 11.5/20."

# let noteMath =
    resEleve moyPonderee (45.0,100.0) (65.0, 42.0);;
val noteMath : string = "Recu avec la note de 51.2/100."
```

Type d'une fonction avec paramètres fonctionnels : Le type d'une fonction décrit les types de chacun de ses arguments. Dans le cas d'un argument fonctionnel, son type est celui d'une fonction et contient donc au moins une flèche. Afin de distinguer ce type fonctionnel des flèches et types des autres arguments éventuels de la fonction, le type d'un argument fonctionnel est entouré de parenthèses. Par exemple le type de la fonction `resEleve` est :

$$\texttt{resEleve} : (\texttt{'a*'b} \rightarrow \texttt{float}) \rightarrow \texttt{float*float} \rightarrow \texttt{'a*'b} \rightarrow \texttt{string}$$

où

- `modeCalcul` (type d'une fonction) : $(\texttt{'a*'b} \rightarrow \texttt{float})$
  Cette fonction est appliquée à une paire `(n1,n2)`, d'où le type `'a*'b` pour son argument. Son résultat est utilisé comme un `float`. En tant que type d'une fonction, ce type contient une flèche. Il est mis entre parenthèses pour ne pas confondre celle-ci avec celles correspondant aux autres paramètres de la fonction
- `(rMoy, rMax)` : type d'une paire `float*float`
- `(n1,n2)` : type d'une paire polymorphe `'a*'b`
- résultat de la fonction : `string`

4.4 Les fonctions avec résultat fonctionnel

Une fonction peut renvoyer des valeurs de n'importe quel type, y compris des valeurs fonctionnelles. Cela est possible si la fonction prend plusieurs arguments séparés, et qu'on ne lui applique qu'une partie de ses arguments. La fonction retourne alors une fonction qui "attend" les arguments qu'on ne lui pas encore passé. Considérons par exemple la fonction `resEleve` vue plus haut et que nous rappelons ici :

```
let resEleve modeCalcul (rMoy,rMax) (n1,n2)=
  let noteFinale = modeCalcul (n1,n2) in
  let recu = if (noteFinale >= rMoy) then "Recu"
             else "Non recu" in
  let note = string_of_float(noteFinale) in
  let max =  string_of_float(rMax)
  in recu^" avec "^note^"/"^max;;
val resEleve : ('a * 'b -> float) -> float * float -> 'a * 'b -> string = <fun>
```

Elle prend trois arguments. Si nous lui passons seulement son premier argument (le mode de calcul) nous obtenons une fonction qui attend ses deux autres arguments. La fonction est alors *spécialisé* pour un mode de calcul particulier (celui passé en paramètre), et attend une moyenne et note maximale de référence ainsi qu'une paire de notes sur qui s'appliquer.

```
 # let resMoyenneSimple =  resEleve moySimple;;
val resMoyenneSimple : float * float -> float * float -> string = <fun>

 # resMoyenneSimple (10.0,20.0) (5.0, 16.5);;
- : string = "Recu avec 10.75/20."
```

Ce mécanisme est connu sous le nom d'*application partielle*. Il permet de *spécialiser* des fonctions très génériques lorsque certains de ses paramètres sont connus, en attendant de les appliquer au reste de leurs données d'entrée.

4.5 Fonctions définies par cas ou par *filtrage*

Le *filtrage* est un mécanisme de *reconnaissance de motifs syntaxiques* que l'on peut appliquer sur une donnée structurée afin d'extraire quelques unes de ces composantes. C'est un mécanisme puissant et concis que nous utiliserons afin de définir des fonctions par cas sur la structure syntaxique de leurs arguments.

Motif : *Patron syntaxique* d'une valeur à reconstituer, composé de constantes et/ou d'identificateurs à lier. Quelques exemples de motifs :

Motif	il permet de reconnaître
`2`	constante 2
`(0,x)`	toute paire avec un 0 en 1ère composante
`y`	toute valeur
`{cle=k; contenu=c}`	enregistrement avec champs cle et contenu
`{cle=7}`	enregistrement avec au moins un champ cle égal à 7
`[0; _ ]`	liste avec 2 éléments dont le 1er est 0

Filtrage : Il s'agit de la *recherche de correspondance* entre une valeur et un motif ou patron pour cette valeur. Le filtrage réussit lorsqu'il y a correspondance entre la forme syntaxique du motif et celle de la valeur comparée. On dit alors que la valeur est *filtrée* (*matched*) par le motif. Si le motif contient des identificateurs, ceux-ci sont alors liés aux composantes auxquelles ils correspondent dans la valeur. Dès lors, ces identificateurs correspondent à des composantes de la valeur comparée, que l'on pourra utiliser pour calculer la valeur rendue par l'expression de filtrage. Voici quelques exemples de filtrage :

Motif	Valeur comparée	Réussite	Liaisons
1	1	oui	aucune
x	1	oui	x=1
`(0,x,y)`	`(0,1,6)`	oui	x=1, y=6
`{cle=k; contenu=c}`	`{cle=5; contenu="abc"}`	oui	k=5, c="abc"
0	1	échec	
`(1,x)`	`(1,2,3)`	échec	
`(_,x,_)`	`(0,1,6)`	oui	x=1

Il existe plusieurs manières de définir des fonctions par filtrage en Ocaml. Nous utiliserons les définitions à base de l'expression `match with`

```
match expr
 with motif_1 -> a_1
 |    motif_2 -> a_2
 |    ....
 |    _       -> tous_autres_cas
```

Cette expression compare la valeur correspondant à `expr` successivement à chacun des motifs dans l'ordre de leur définition. Si $\texttt{motif}_i$ est le premier à réussir, alors le résultat de l'expression est a_i, sinon, soit il y a un cas qui filtre tout _, et le résultat est l'expression qui l'accompagne, soit, si aucun motif ne filtre `expr` alors le résultat de la reconnaissance est un échec. Voici l'exemple de la fonction qui additionne une paire d'entiers :

```
# let somme x y =  match (x,y)
  with (0,n) -> n
```

```
  |     (n,0) -> n
  |     (a,b) -> a+b;;
val somme : int -> int -> int = < fun >

# somme 3 0;;
- : int = 3

# somme 2 4;;
- : int = 6
```

Examinons le comportement de l'appel `somme 3 0`

- on exécute `(match(3,0) with ...)` qui compare `(3,0)` avec chacun des motifs de la fonction :
  1. motif 1 : `(0,n)` comparé à `(3,0)` ⇒ échec,
  2. motif 2 : `(n,0)` comparé à `(3,0)` ⇒ réussit et on obtient la liaison `n=3`
     ⇒ on exécute la partie à droite de ce motif
     ⇒ le résultat renvoyé est `3`

Un filtrage est *incomplet* si tous les cas possibles d'une valeur ne sont pas répertoriés. Dans ce cas, un message d'avertissement est indiqué à la compilation.

```
let est_null x =
  match x
  with  0 -> true
   |    1 -> false
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
2
val est_null : int -> bool = <fun>

# est_null 1;;
- : bool = false

# est_null 3;;
Exception: Match_failure ("", 2, 2).
```

Pour compléter ce filtrage :

```
let est_null x =
  match x
  with  0 -> true
   |    _ -> false
val est_null : int -> bool = <fun>
```

Pour finir, deux fonctions écrites par filtrage sur un arugment de type enregistrements. Ce fonctions testent si si un client est débiteur ou créditeur. Notez qu'il n'est pas nécessaire de faire apparaître tous les champs du type lors d'un filtrage. La deuxième écriture est donc plus compacte.

```
type client = {numero :int; nom:string; solde:float; }
```

```
# let crediteur {numero=_;nom= _; solde=s}= s>0.0;;
val crediteur : client -> bool = <fun>

# let debiteur {solde = s} = s < 0.0;;
val debiteur : client -> bool = <fun>
```

4.6 Fonctions anonymes

Une fonction *anonyme* est une valeur fonctionnelle que l'on définit à la volée sans lui donner de nom, et donc sans passer par une déclaration. Par exemple, la fonction suivante permet de calculer la moyenne d'une paire d'arguments de type `float` :

```
# fun (x,y) -> (x+.y)/.2.0;;
- : float * float -> float = <fun>

# (fun (x,y) -> (x+.y)/.2.0) (3.5, 8.5);;
- : float = 6.
```

Ce genre de fonction est utile afin de construire des valeurs fonctionnelles (à passer en paramètre p.e.), mais dont il n'est pas nécessaire de garder une trace dans l'environnement. Par exemple, nous pouvons définir la fonction `resMoyenneSimple` que nous rappelons ici :

```
# let moySimple (x,y) = (x+.y)/.2.0;;

# let resMoyenneSimple =  resEleve moySimple;;
```

par une définition qui ne déclare pas la fonction `moyenneSimple` mais qui la construit de manière anonyme.

```
# let resMoyenneSimple = resEleve (fun (x,y) -> (x+.y)/.2.0);;
val resMoyenneSimple : float * float -> float * float -> string = <fun>
```

5 Programmes et expressions

Nous donnons ici la syntaxe des *expressions* et des *phrases* en Ocaml.

Programmes : Tout ou presque est expression dans un programme Ocaml. Ainsi, un programme est une suite de *phrases* (déclarations) et d'expressions. La plupart du temps ces dernières sont des appels de fonctions. Voici un exemple de programme :

```
# let baguette = 4.20;;                (* declaration valeur *)
val baguette : float = 4.2

# let euro x =  x /. 6.55957;;         (* declaration fonction *)
val euro : float -> float = <fun>

# euro baguette;;                      (* expression *)
- : float = 0.640285872397123645
```

Syntaxe des phrases :

définition de valeur	`let` x = e
définition de fonction	`let` f x1 ... xn = e
définition de fonctions	`let [ rec ]` f1 x1 ... = e1 ...
(mutuellement récursives)	`[ and` fn xn ... = en]
définition de type(s)	`type` q1 = t1... [`and` qn = tn]
expression	e

Syntaxe des expressions :

définition locale	`let` x = e1 `in` e2
fonction anonyme	`fun` x1 ... xn `->` e
appel de fonction	f e1 ... en
variable	`x`
	(`M.x` si x est défini dans le module M)
valeur construite	(e1, e2)
(dont les constantes)	1, 'c', "aa"
analyse par cas	`match` e `with` p1 `->` e1 ... `|` pn `->` en
boucle for	`for` i = e0 `to` ef `do` e `done`
boucle while	`while` e0 `do` e `done`
conditionnelle	`if` e1 `then` e2 `else` e3
une séquence	e ; e '
parenthèses	(e) `begin` e `end`

Expression conditionnelle La construction `if...then...else` permet d'exprimer l'alternative. Il s'agit d'une expression dont le résultat est celui de l'expression de l'une de ses deux branches.

```
# if 3>5 then "trois superieur a cinq" else "trois inferieur a cinq";;
- : string = "trois inferieur a cinq"
```

Pour que le typage soit correcte, les expressions dans chacune des branches doivent avoir le même type.

```
 if 3>5 then "trois superieur a cinq" else 5;;
Characters 43-44:
This expression has type int but is here used with type string
```

6 La généricité ou polymorphisme paramétrique

La *généricité* permet de rendre génériques les *types* de certaines fonctions ou de certaines structures des données. Avec un typage générique, ces fonctions ou structures deviennent utilisables sur des données de *n'importe quel type* tout en étant soumises aux contraintes de type décrites par ailleurs par leur typage. Un exemple classique est donnée par la fonction `premier` permettant d'extraire la première composante d'une paire.

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

Idéalement, cette fonction devrait pouvoir s'appliquer sur n'importe quelle paire de valeurs, et renvoyer en résultat sa première composante :

```
premier(1, true) ⇒ 1
premier("abc", 's') ⇒ "abc"
premier(2.4, 0) ⇒ 2.4
```

Mais quel type lui donner ? Un typage trop spécifique aurait pour effet de contraindre la fonction à un seul type de paires. Un type trop laxiste (par exemple `Object` en Java) aboutirait à un manque de précision du typage. Avec la généricité, on peut typer cette fonction *pour toutes les types de paires de type* $A * B$ où A et B sont des *paramètres de types*, à savoir, des types *inconnus* au moment de la définition de la fonction, mais qui seront à préciser ou *instancier* plus tard. Nous avons déjà vu en 1.2 que ce mécanisme est présent en C++ et en Java. En Ocaml, la fonction `premier` est typée par :

```
# let premier (x,y) = x;;
val premier: 'a * 'b -> 'a

# premier (1, true);;
- : int = 1

# premier("abc", 's');;
- : string = "abc"
```

Le type donné à `premier` contient deux variables de type notées `'a` et `'b`. En Ocaml, la généricité est un mécanisme de typage *utilisé par défaut* lors de l'inférence de types. Ainsi, toute expression est typée avec le type le plus *générique possible* qui est compatible avec sa définition, et au cas où cela est nécessaire, des paramètres de types sont introduits dans son typage. La généricité est connue également sous le nom de *polymorphisme paramétrique*. Une fonction est *polymorphe* si elle possède plusieurs types. Le polymorphisme est *paramétrique* lorsque les types font intervenir des paramètres de types.

Le type `'a * 'b -> 'a` donné à `premier` n'est pas simplement très générique. Il est également précis : il indique qu'il existe un lien entre le types de la première composante de son paramètre et son résultat. Par exemple, si la fonction est appliquée sur une paire de type `int*bool`, son résultat sera de type `int`. Ceci permet au typeur de trouver des erreurs de typage quand ces contraintes ne sont pas respectées :

```
# let x = premier (1, true) + premier(2,"ab");;
val x : int = 3

# let y = premier (1, true) + premier("ab",2);;
Error: This expression has type string but an
expression was expected of type  int
```

Voici pour finir quelques exemples de primitives polymorphes en Ocaml :

```
# fst;;
- : 'a * 'b -> 'a = <fun>

# snd;;
- : 'a * 'b -> 'b = <fun>
```

```
# (=);;
- : 'a -> 'a -> bool = <fun>

# (>);;
- : 'a -> 'a -> bool = <fun>
```

6.1 Structures des données polymorphes

En Ocaml il est possible de déclarer des types correspondant à des structures des données polymorphes paramétriques. Les données stockées pourront alors être de n'importe quel type, à condition bien sûr de respecter les contraintes trouvées par le typage. Dans une telle déclaration

- on *déclare* le paramètre de type *devant le nom du type* à déclarer ;
- ce paramètre doit apparaître dans le corps de la déclaration.

Considérons l'exemple des cellules avec une clé entière, et deux champs de contenu, un pour le contenu courant, un autre pour celui du mois précédent. En Java 1.5 on pourra écrire le type `class Cellule<A>` :

```
class Cellule<A> {int cle; A contenu; A moisPrec};
```

En Ocaml, on déclare le type `'a cellule` :

```
type 'a cellule = {cle: int; contenu: 'a; moisPrec: 'a};;
```

Lors du stockage des données de ce type, le typeur infère l'instantiation des paramètres de type qui convient à chaque cas : `'a` $\mapsto$ `int` pour `c1` ; `'a` $\mapsto$ `string` pour `c2` ; une erreur de typage pour `c3` (car `'a` ne peut pas être instancié à la fois en `int` et `string`) :

```
#   let c1 = {cle = 1; contenu = 3; moisPrec = 7};;
val c1 : int cellule = {cle = 1; contenu = 3; moisPrec = 7}

#  let c2 = {cle = 2; contenu = "Paris"; moisPrec = "Lyon"};;
val c2 : string cellule = {cle = 2; contenu = "Paris"; moisPrec ="Lyon"}

# let c3 = {cle = 17; contenu = 3; moisPrec = "Lyon"};;
Error: This expression has type string but an expression was
 expected of type  int
```

Une fonction qui prend un argument de type `'a cellule` pourra les cas échéant, déduire une instantiation de la variable de type `'a` :

```
# let sommeContenus c = c.contenu + c.moisPrec;;
val sommeContenus : int cellule -> int = <fun>

#  let donneMoisPrec c = c.moisPrec;;
val donneMoisPrec : 'a cellule -> 'a = <fun>
```

Le type de la fonction `sommeContenus` est contraint aux arguments de type `int cellule`, alors que `donneMoisPrec` reste polymorphe.

6.2 L'inférence de types

Nous finissons cette partie avec un exemple d'inférence de type *monomorphe*. Cet exemple permet d'illustrer la différence entre fonctions polymorphes et monomorphes : elle tient aux contraintes de type que les opérateurs ou expressions dans la définition de la fonction imposent afin d'obtenir un typage correct. Considérons la fonction :

```
let f x = x+1
```

Sachant que + est un opérateur définit uniquement sur les entiers, et dont le type est + : `int` $*$ `int` $\rightarrow$ `int`, le typeur déduit les contraintes suivantes :

- le type de `f` est de la forme :

$$\texttt{f} : t_x \rightarrow t_{corps}$$

  où t_x est le type de x (argument de la fonction), et t_{corps} est le type du corps.
- pour que `x+1` soit bien typé, `x` doit être de type `int` $\Rightarrow \boxed{t_x = \texttt{int}}$
- si ces contraintes sont respectées, alors le résultat de l'opération `x + 1` est de type `int`, $\boxed{t_{corps} = \texttt{int}}$.

Conclusion : `f` est de type `int` $\rightarrow$ `int`

Le typeur infère des types polymorphes en absence de telles contraintes.

7 Les types liste : un type inductif prédéfini

En Ocaml, une *liste* est une séquence de valeurs de même type, ce type pouvant être quelconque. Il s'agit donc d'un type polymorphe, prédéfini, et dont le nom est `'a list`. Selon les éléments stockées, une liste pourra être de type `int list`, `string list`, etc. Voici quelques éléments de syntaxe :

- `[]` : la liste vide. Son type est `'a list`
- `@` : opérateur de concaténation.
- `[a; b; c]` : liste formée de trois éléments a,b,c. Les éléments de la liste sont séparés par des points-virgules. Cette notation est utilisée pour les listes données en extension (on donne tous leurs éléments) ;
- `a::l` liste formée d'un premier élément a suivi d'une liste l. Cette notation est employée dans les fonctions sur les listes écrites par filtrage.

Quelques exemples :

```
# [1;2;3];;
- : int list = [1; 2; 3]

# ["a";"bc"] @ ["bonjour"];;
- : string list = ["a"; "bc"; "bonjour"]

# [];;
- : 'a list = []

# [1]@[];;
- : int list = [1]

# ["ab";"cd"]@[];;
- : string list = ["ab"; "cd"]
```

Liste de paires et liste de listes :

```
# [(1,'a'); (23, 'c')];;
- : (int * char) list = [(1, 'a'); (23, 'c')]

# [[1;2];[3]];;
- : int list list = [[1; 2]; [3]]
```

7.1 Construction de listes

Le type *liste* en Ocaml correspond à une structure des données *récursive*. En effet, toute liste est :

- `[]`, soit vide,
- `a::l` , soit formée d'un élément a, suivi d'une *liste* l

Le symbole `::` est le *constructeur* des listes. Il s'agit d'une sorte d'opérateur qui prend deux arguments a et l afin de construire une liste par la notation $a :: l$, où

- a (de type T) est le *premier élément* de la liste à construire `a::l` ;
- l (de type T `list`) est une *liste* est la *suite* de la liste à construire `a::l` ;

Ainsi, la construction `a::l` fabrique une nouvelle liste, de type T `list`, dont le premier élément est a et qui est suivie par tous les éléments dans l. Toutes les listes non vides en Ocaml sont construites à l'aide de cet opérateur (les autres notations étant du *sucre syntaxique*). Il doit être appliqué de manière emboîtée, autant de fois que d'éléments on veut inclure dans la liste. Cet emboîtement se termine toujours par une liste vide. Par exemple :

```
# 1::[];;
- : int list = [1]

# 1::[3;5];;
- : int list = [1; 3; 5]

# 1::(2::(3::[]));;
- : int list = [1; 2; 3]
```

7.2 Fonctions sur les listes

Elles sont souvent définies par filtrage sur les deux cas *structurellement* possibles d'une liste :

(a) la liste vide `[]` ;
(b) la liste non vide `a::l`, où a est son premier élément et l la liste qui le suit.

Par exemple, voici la fonction `premier` qui extrait le premier élément d'une liste, compare sa liste argument avec ces deux cas possibles :

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

# premier [3;4];;
- : int = 3
```

```
# premier ['a';'g'];;
- : char = 'a'

# premier [];;
Exception: Failure "premier".
```

La construction prédéfinie `failwith`, lève l'exception `Failure` suivie du nom de la fonction qui a échoué. Cela arrête l' exécution du programme dans les cas où la fonction n'est pas définie. La fonction `premier` est dans la bibliothèque (module) `List` sous le nom `List.hd`.

Examinons maintenant le comportement de l'appel `(premier [3;4;5])` :

1. `[3;4;5]` est comparé au premier motif `[]` ⇒ échec
2. `[3;4;5]` est comparé au motif `e::reste` ⇒ réussite avec les liaisons `e=3`, `reste=[4;5]` le résultat renvoyé est `e` ⇒ 3

La fonction qui teste si une liste est vide :

```
# let vide l =
match l
with [] -> true
  | _ -> false;;
val vide : 'a list -> bool = <fun>

# vide [1;2];;
- : bool = false

# vide [];;
- : bool = true
```

Voici maintenant quelques fonctions récursives sur les listes : `longueur` calcule la longueur d'une liste ; `appartient` teste si un élément appartient à 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

# let rec appartient e l =
match l
with [] -> false
  | a::reste -> e=a || appartient e reste;;
 val appartient : 'a -> 'a list -> bool = <fun>
```

```
# appartient 1 [5;6;3];;
- : bool = false

# appartient 1 [5;3;1;6];;
- : bool = true
```

Ces fonctions se nomment `List.length` et `List.mem` dans la librairie. La fonction `longueur` se comporte de la manière suivante :

- elle accumule 1 dans la somme du nombre d'éléments pour chaque élément rencontré en début de liste ;
- et recommence par un appel récursif sur le reste de la liste ;
- renvoie 0 si la liste est vide.

Voici le déroulement de l'appel `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
⇒ 3
```

Voici la fonction qui à partir d'une liste l et d'un élément e construit une nouvelle liste où la première occurrence de e est remplacé par un élément x.

```
# let rec remplace e x l =
match l
with [] -> []
  | a::reste -> if e=a then x::reste else a::(remplace e x reste)
val remplace : 'a -> 'a -> 'a list -> 'a list = <fun>

# remplace 'a' 'b' ['e'; '2'; 'a'; '?'];;
- : char list = ['e'; '2'; 'b'; '?']

# remplace 1 6 [2;3;4];;
- : int list = [2; 3; 4]
```

La fonction `remplace` construit une nouvelle liste à l'aide du constructeur `::` en recopiant tous les éléments de sa liste argument, sauf la première occurrence de e qui est changée en x.

```
remplace 'a' 'b' ['e'; '2'; 'a'; '?';'h'] ⇒
'e'::(remplace 'a' 'b' ['2'; 'a'; '?';'h']) ⇒
'e'::('2'::(remplace 'a' 'b' ['a'; '?';'h'])) ⇒
'e'::('2'::('b'::['?';'h'])) ⇒
['e';'2';'b';'?';'h']
```

7.3 Le module `List` de la librairie

Il existe des nombreuses fonctions dans la librairie Ocaml. Elles sont organisées en *modules*, et sont accessibles avec la *notation pointée*. Les des utilitaires sur les listes sont contenus dans le module `List`. Par exemple, pour utiliser la fonction `length` qui calcule la longueur d'une liste, on écrira :

```
# List.length ["ab"; "bonjour"];;
- : int = 2
```

Pour finir, voici quelques fonctions du module `List` :

- `val length : 'a list -> int` Renvoie la longueur (nombre d'éléments) d'une liste donnée.
- `val hd : 'a list -> 'a` Renvoie le premier élément d'une liste donnée. Lève l'exception `Failure "hd"` si la liste est vide.
- `val tl : 'a list -> 'a list` Renvoie une liste donnée sans son premier élément. Lève l'exception `Failure "tl"` si la liste est vide.
- `val nth : 'a list -> int -> 'a` Renvoie le n-ième élément d'une liste donnée. Le premier élément est à la position 0.
- `val rev : 'a list -> 'a list` Inversion d'une liste donnée.

7.4 Les fonctionnelles sur les listes

Une *fonctionnelle* est une fonction qui *prend des fonctions en argument*. Dans le module `List` il existe un bon nombre de fonctionnelles fort utiles sur les listes. Par exemple, la fonctionnelle `map`, prend en argument une fonction f et une liste l, et construit une *nouvelle liste*, résultat d'appliquer f à chaque argument de la liste d'entrée l :

$$\texttt{map}\ f\ \ [a_1; a_2; \ldots a_n] \Rightarrow [(fa_1); (fa_2); \ldots (fa_n)]$$

La fonctionnelle `map` fait partie du module `List`. Pour l'utiliser nous pouvons écrire :

```
# let succ x = x+1;;

# List.map succ [1;2;3];;
- : int list = [2; 3; 4]
```

Voici une utilisation de `map` afin d'extraire les numéros de vol d'une liste de vols :

```
type vol = {dest:string; num:int; prix:float; disp:int};;

let ccs = {dest="Caracas"; num=221; prix=800.0; disp=10};;
let mxc = {dest="Mexico"; num=121; prix=900.0; disp=6};;
let mx1 = {dest="Mexico"; num=654; prix=750.0; disp=10};;
let mx2 = {dest="Mexico"; num=680; prix=750.0; disp=5};;
let ny = {dest="New York"; num=110; prix=400.0; disp=56};;

let lvols = [ccs; mxc;mx1;mx2; ny];;

# let donneNumVol v = v.num;;
val donneNumVol : vol -> int = <fun>
```

```
# let listeNumVols l = List.map donneNumVol l;;
val listeNumVols : vol list -> int list = <fun>

# listeNumVols  lvols;;
- : int list = [221; 121; 654; 680; 110]
```

Une implantation possible de `map` est donnée par :

```
# let rec map f l =
  match l
  with []     -> []
  |  a::reste -> (f a)::(map f reste);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Dans le type de cette fonction nous avons :

- `('a -> 'b)` : fonction à appliquer sur chaque élément de la liste, de type `'a`, pour obtenir un résultat de type `'b`
- `'a list` : type de la liste donnée en entrée
- `'b list` : type de la liste renvoyée en résultat

Une autre fonctionnelle très utile est `filter`. Elle prend en argument une fonction booléenne `test` et une liste l, et construit une nouvelle liste composée de tous les éléments de l qui rendent vrai le test :

$$\texttt{filter test } [a_1; a_2; \ldots a_n] \Rightarrow [a_i \mid \texttt{test}(a_i) = \texttt{true}]$$

L'utilisation ci-dessous permet d'extraire les nombres pairs d'une liste d'entiers :

```
# let est_pair x = x mod 2 = 0;;
val est_pair : int -> bool = <fun>

# filter est_pair [1;2;5;6;80];;
- : int list = [2; 6; 80]
```

Voici une implantation possible de `filter` :

```
# let rec filter cond l =
  match l
  with []     -> []
  |  a::reste -> if (cond a) then a::(filter cond reste)
                 else filter cond reste;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Nous utilisons `filter` pour extraire la liste de vols d'une destination donnée. La fonction `filtreDest` prend une chaîne d et une liste de vols lv et extrait de la liste tous les vols dont la destination est égale à d. Elle s'écrit tout simplement par un appel à `filter` où la fonction de test passée consiste à vérifier que le champs destination d'un vol est égal à une chaîne donnée :

```
# let filtreDest d lv = filter (fun v -> v.dest = d) lv;;
val filtreDest : string -> vol list -> vol list = <fun>

# filtreDest "Mexico" liste_vols;;
```

```
- : vol list =
[{dest = "Mexico"; num = 121; prix = 900.; dispo = 6};
 {dest = "Mexico"; num = 680; prix = 750.; dispo = 5};
 {dest = "Mexico"; num = 654; prix = 750.; dispo = 10}]
```

Pour finir, voici quelques fonctionnelles du module `List` (extraits du manuel) :

- `map: ('a -> 'b) -> 'a list -> 'b list`
- `for_all : ('a -> bool) -> 'a list -> bool` Teste si une condition est vrai pour tous les éléments de la liste.
- `exists: ('a -> bool) -> 'a list -> bool` Teste si une condition est vrai pour au moins pour un élément dans la liste.
- `mem: 'a -> 'a list -> bool` Equivalent de la fonction `appartient`.
- `filter: ('a -> bool) -> 'a list -> 'a list`
- `find: ('a -> bool) -> 'a list -> 'a`
  `find p l` renvoie le premier élément qui satisfait la condition p et lève une exception s'il n'est pas trouvé.
- `partition:('a->bool)->'a list->'a list* 'a list`
  `partition p l` renvoie une paire de listes (l1, l2), où l1 contient les élément qui satisfont p, et l2 ceux qui ne la satisfont pas.