Université Paris Sud

Master Informatique M1 2006–2007

Initiation à la programmation fonctionnelle

Jean-Christophe Filliâtre

Table des matières

Introduction

Ce cours est une introduction au langage de programmation OCAML et plus généralement à la programmation fonctionnelle. Cette introduction est nécessairement succincte — il s'agit d'un cours de quatre heures seulement — et ne peut donc ni entrer dans les détails ni même être exhaustive. Pour une introduction détaillée au langage OCAML on se référera au polycopié de C. Marché et R. Treinen *Formation au langage Caml* dont une copie accompagne le présent document.

Le but de cette initiation est plutôt d'introduire un certain nombre d'*idées clés* concernant la programmation fonctionnelle, ainsi que de démonter d'autres idées, fausses, sur celle-ci. Il ne s'agit nullement d'affirmer qu'OCAML est le meilleur des langages de programmation ; une telle affirmation n'aurait d'ailleurs aucun sens, les langages de programmation ne pouvant être comparés de manière absolue. Comme on a coutume de le dire,

> *il n'y a pas de bon langage de programmation,*
> *il n'y a que de bons programmeurs*

Un certain nombre de concepts exposés dans ce cours se retrouvent d'ailleurs dans d'autres langages de programmation (exceptions, GC) et d'autres encore sont valables quel que soit le langage utilisé (notion de persistance).

Pour de plus amples informations sur le langage OCAML (comment le récupérer, quels sont les livres ou les bibliothèques de programmes existants, les listes de discussion, etc.) on consultera le site web officiel : `caml.inria.fr`

Ce cours a été écrit en utilisant la version 3.08 d'OCAML mais la majorité — sinon la totalité — des exemples fonctionnent avec des versions antérieures et en toute probabilité avec les versions futures du langage. Nous supposerons également un environnement UNIX, sans que cela ne dépasse l'aspect anecdotique.

Coordonnées de l'auteur :

Jean-Christophe Filliâtre

LRI bâtiment 490
Université Paris Sud
91405 ORSAY Cedex
France

Tél : 01-69-15-65-76
Bureau 144
Email : `filliatr@lri.fr`
Web : `http://www.lri.fr/~filliatr/`

Chapitre 1

Fondamentaux

Le terme de *programmation fonctionnelle* désigne une famille de langages de programmation ayant un certain nombre de traits en commun, dont un rôle central donné aux fonctions, d'où le nom. Les représentants les plus connus de cette famille se nomment HASKELL, STANDARD ML et OCAML, parmi de nombreux autres.

La programmation fonctionnelle est très souvent opposée à la programmation *impérative*, famille regroupant des langages tels que C ou PASCAL mais aussi JAVA ou C++, où l'on procède essentiellement par effets de bord i.e. modification en place du contenu des variables. Mais cette opposition n'a pas vraiment lieu d'être. En effet, seul le langage HASKELL exclut complètement les effets de bord et est dit pour cela *purement fonctionnel*. Les langages tels que STANDARD ML ou OCAML autorise parfaitement les effets de bords et comportent d'ailleurs toutes les structures usuelles de la programmation impérative (variables modifiables en place, tableaux, boucles `for` et `while`, entrées-sorties, etc.)

Malheureusement, comme nous le verrons à plusieurs reprises, le terme *fonctionnel* évoque souvent l'*absence d'effets de bord* ou le caractère *immuable*, comme dans l'expression malheureuse « structure de données (purement) fonctionnelle ». Mais en réalité, les langages de programmation fonctionnels partagent beaucoup de points avec ceux dits impératifs. Et ils ont également beaucoup de points forts en dehors du caractère immuable de certaines de leurs valeurs ou de leur gestion des fonctions. C'est ce que nous allons essayer d'illustrer dans ce cours.

Dans ce qui suit, il nous arrivera parfois de faire un parallèle avec un morceau de code C ou JAVA. Que le lecteur ne connaissant pas l'un ou l'autre ne s'inquiète pas et se contente de l'ignorer.

1.1 Premiers pas

1.1.1 Le premier programme

Le programme le plus célèbre, parce que le « premier », est celui qui se contente d'afficher `hello world !` sur la sortie standard. En OCAML il s'écrit ainsi

```
print_string "hello world!\n"
```

Si le texte ci-dessus est contenu dans un fichier `hello.ml`, il est compilé en un exécutable `hello` avec la commande

```
% ocamlc -o hello hello.ml
```

exactement comme avec un compilateur C, puis cet exécutable peut être lancé, donnant le résultat escompté :

```
% ./hello
hello world!
%
```

On constate tout de suite deux différences importantes avec le même programme écrit en C ou en Java.

D'une part, l'application d'une fonction s'écrit en Ocaml par simple juxtaposition de la fonction et de son argument. À la différence de la plupart des langages où l'application de `f` à `x` doit s'écrire `f(x)`, on se contente ici d'écrire `f x`. Comme dans les autres langages, les parenthèses peuvent — et doivent — être utilisées lorsque les priorités des opérateurs l'exigent, comme dans l'expression `2*(1+3)`. Rien n'interdit donc en particulier d'écrire

```
print_string("hello world!\n")
```

mais les parenthèses autour de la chaîne de caractères sont tout simplement inutiles.

On constate d'autre part qu'il n'est nul besoin de définir une fonction principale `main` contenant le code à exécuter. Le programme Ocaml est constitué ici d'une simple expression à évaluer. Cette expression est l'application d'une fonction, `print_string`, à un argument, la chaîne de caractères `"hello world!\n"`. On aurait très bien pu écrire un programme se réduisant à

```
1+2
```

qui aurait eu pour effet de calculer le résultat de `1+2`. Mais rien n'aurait été affiché. Pour cela, il faudrait écrire par exemple le programme `print_int (1+2)`.

1.1.2 Déclarations

Plus généralement, un programme Ocaml est constitué d'une suite quelconque d'expressions à évaluer et de déclarations, séparées par un double point-virgule `;;`. Une déclaration affecte le résultat de l'évaluation d'une expression à une variable, et est introduite par le mot clé `let`. Ainsi le programme suivant

```
let x = 1 + 2;;
print_int x;;
let y = x * x;;
print_int y;;
```

calcule le résultat de `1+2`, l'affecte à la variable `x`, affiche la valeur de `x`, puis calcule le carré de `x`, l'affecte à la variable `y` et enfin affiche la valeur de `y`.

Une déclaration telle que `let x = 1+2` peut être vue comme l'introduction d'une variable globale `x`. Mais il y a là beaucoup de différences avec la notion « usuelle » de variable globale :

1. La variable est *nécessairement initialisée*, ici par le résultat de `1+2` (en C une variable non initialisée peut contenir n'importe quelle valeur ; en Java une variable non initialisée se voit donner une valeur par défaut, qui sera toujours la même, mais ce n'est pas la même chose qu'exiger une initialisation de la part du programmeur).
2. Le type de la variable n'a pas besoin d'être déclaré, il est *inféré* par le compilateur (nous verrons comment dans la section 1.3) ; ici le type inféré est `int`, le type des entiers relatifs. Comme tout autre compilateur, le compilateur Ocaml vérifie que les expressions sont bien typées (pour rejeter des expressions telles que `1+true`) mais l'utilisateur n'a pas besoin d'indiquer de types dans son programme.
3. Le contenu de la variable n'est *pas modifiable* ; en d'autre termes, la variable `x` contiendra la valeur `3` jusqu'à la fin du programme (nous verrons dans un instant qu'il existe aussi des variables modifiables en place).

1.1.3 Références

Si l'on souhaite utiliser une variable modifiable en place, il faut l'introduire à l'aide du mot clé supplémentaire `ref` :

```
let x = ref 1;;
```

Une telle variable est appelée une *référence*. De la même manière que pour une variable immuable, elle doit être nécessairement initialisée et son type est automatiquement inféré par le compilateur. On peut alors modifier le contenu de `x` avec la même syntaxe qu'en Pascal :

```
x := 2;;
```

En revanche l'accès à la valeur de `x` doit s'écrire `!x`. Voici un exemple de programme utilisant une référence :

```
let x = ref 1;;
print_int !x;;
x := !x + 1;;
print_int !x;;
```

Cette syntaxe peut paraître lourde mais elle sera justifiée plus loin. Rien n'empêche d'utiliser la variable `x` directement, mais elle désigne alors la référence elle-même et non son contenu, pour un passage par référence par exemple. Le typage explicite cette distinction, en donnant le type `int ref` à une référence contenant un entier. C'est exactement la même situation qu'avec les pointeurs du C, où l'on distingue le pointeur `x` et son contenu `*x`, si ce n'est que l'arithmétique de pointeurs n'est pas possible et qu'il n'existe pas d'équivalent au pointeur `null`.

On voit donc qu'Ocaml propose deux sortes de variables : des variables modifiables en place, les références, semblables à ce que l'on trouve dans les langages impératifs, mais également des variables *immuables* dont le contenu ne peut être modifié. On ne trouve pas vraiment d'équivalent dans les langages C ou Java, où seule la notion de variable modifiable en place existe — même si les mots clés `const` et `final` y permettent respectivement de déclarer une variable comme non modifiable.

1.1.4 Expressions et instructions

Une autre spécificité de la programmation fonctionnelle, assez déroutante pour le débutant, est l'absence de distinction entre expressions et instructions. Dans les langages impératifs, ce sont deux catégories syntaxiques bien distinctes : une conditionnelle `if-then-else` ou une boucle `for` n'est pas acceptée en position d'expression, et inversement certaines expressions ne sont pas autorisées en position d'instruction. Ainsi on ne peut pas écrire en JAVA une expression telle que

```
1 + (if (x == 0) f(); else g();)
```

ou bien une instruction telle que

```
2 * { int s = 0; for (int i = 0; i < 10; i++) s += i; return s; };
```

Certaines constructions seulement peuvent apparaître autant comme expression que comme instruction, telles que l'affectation ou l'appel de fonction.

En OCAML, il n'y a pas de telle distinction expression/instruction : il n'y a que des expressions. Ainsi on peut écrire

```
1 + (if x = 0 then 2 else 3)
```

car la construction `if-then-else` est une expression comme une autre. Elle s'évalue de manière évidente : sa première opérande est évaluée et si le résultat vaut `true` la deuxième opérande est évaluée et son résultat est celui de toute l'expression `if`; sinon c'est la troisième opérande qui est évaluée et donne le résultat de toute l'expression.

On retrouve toutes les constructions usuelles de la programmation impérative comme autant d'expressions OCAML. La séquence s'écrit avec le traditionnel point-virgule, comme dans

```
x := 1; 2 + !x
```

mais n'est autre qu'une expression qui évalue sa première opérande, ignore son résultat et évalue sa seconde opérande dont le résultat est celui de la séquence en tant qu'expression. On peut donc écrire par exemple :

```
3 * (x := 1; 2 + !x)
```

même si ce style ne doit pas être encouragé car difficile à lire. La notion usuelle de *bloc* (les accolades en C et JAVA) est ici réalisée par une simple paire de parenthèses. Pour plus de lisibilité on peut également utiliser les mots clés `begin`/`end` en lieu et place d'une paire de parenthèses.

De même, une boucle `for` est une expression comme une autre, ayant la syntaxe suivante :

```
for i = 1 to 10 do x := !x + i done
```

où la variable indice `i` est immuable et de portée limitée au corps de la boucle. Une telle expression doit avoir un type et une valeur, comme toute autre expression, mais une boucle `for` n'a pas lieu de retourner une valeur particulière. Pour cela, OCAML introduit un type prédéfini appelé `unit` possédant une unique valeur notée `()`. C'est ce type qui est donné à une boucle `for`, ainsi qu'à une affectation — à la différence de C et JAVA, en effet, OCAML ne donne pas à une affectation la valeur affectée.

En particulier, la valeur `()` et le type `unit` sont automatiquement donnés à la branche `else` d'une construction `if` lorsque celle-ci est absente. On peut ainsi écrire

```
if !x > 0 then x := 0
```

mais en revanche on ne pourra pas écrire

```
2 + (if !x > 0 then 1)
```

car une telle expression est mal typée : la branche `then` est un expression de type `int` alors que la branche `else` est une expression de type `unit` (le message d'erreur correspondant peut être parfois déroutant pour le débutant). Il est parfaitement logique qu'une telle expression soit rejetée : sinon, quelle valeur le compilateur pourrait-il bien donner à cette expression lorsque le test `!x > 0` se révèle faux ?

1.1.5 Variables locales

Comme dans tout autre langage, il existe en OCAML une notion de variable locale. En C ou JAVA la localité d'une variable est définie par le bloc dans lequel elle est introduite. On écrira ainsi

```
{
  int x = 1;
  ...
}
```

et la portée de la variable `x` s'étend jusqu'à la fin du bloc. En C, les variables locales ne peuvent être introduites qu'au début du bloc, alors qu'en Java elles peuvent être introduites n'importe où dans le bloc, mais la règle de portée reste la même.

En OCAML, la notion de variable locale n'est pas liée à la notion de bloc (qui ici n'existe pas). Elle est introduite par la construction `let in` qui introduit une variable *localement à une expression*, comme dans

```
let x = 10 in 2 * x
```

Comme pour la déclaration d'une variable globale, la variable est nécessairement initialisée, immuable et son type est inféré. Sa portée est exactement l'ensemble de l'expression qui suit le mot clé `in`. La construction `let in` est une expression comme une autre, et on peut ainsi écrire par exemple

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

Bien entendu on peut introduire une variable modifiable en place avec l'adjonction du mot clé `ref` comme pour une déclaration globale. Voici en parallèle un programme JAVA et son équivalent OCAML :

```
{ int x = 1;
  x = x + 1;
  int y = x * x;
  System.out.print(y); }
```

```
let x = ref 1 in
x := !x + 1;
let y = !x * !x in
print_int y
```

Deux remarques sont nécessaires. D'une part on constate que la précédence de la construction `let in` est plus faible que celle de la séquence, permettant ainsi un style très comparable à celui de Java (ou de C). D'autre part la variable locale `y` n'est pas une référence mais une variable immuable : n'étant pas modifiée par le programme, elle n'a pas lieu d'être une référence. Cela allège le programme et évite les erreurs. D'une manière générale, il est judicieux de préférer l'utilisation de variables immuables autant que possible, car cela améliore la lisibilité du programme, sa correction et son efficacité. Cela ne veut pas dire qu'il faut tomber dans l'excès : les références sont parfois très utiles, et il serait regrettable de chercher à s'en passer systématiquement (ce que font les programmeurs Haskell).

Récapitulation

Pour résumer rapidement cette section, nous avons vu

- qu'un programme est une suite d'expressions et de déclarations ;
- que les variables introduites par le mot clé `let` ne sont pas modifiables ;
- qu'il n'y a pas de distinction entre expressions et instructions.

1.2 La boucle d'interaction `ocaml`

Avant d'aller plus loin, nous pouvons nous arrêter sur une spécificité du langage Ocaml : sa boucle d'interaction. À côté du compilateur `ocamlc` dont nous avons déjà illustré l'usage, il existe un autre « compilateur » du langage, interactif celui-ci. On le lance avec la commande `ocaml` :

```
% ocaml
      Objective Caml version 3.08.0

#
```

et l'on se retrouve invité (par le signe `#` appelé justement *invite* en français, *prompt* en anglais) à entrer une expression ou une déclaration Ocaml à l'aide du clavier. Le cas échéant, la phrase est analysée syntaxiquement, typée, évaluée et le résultat est affiché.

```
# let x = 1 in x + 2;;
- : int = 3
#
```

Ici le système indique que l'expression est de type `int` et que sa valeur est `3`. On retrouve alors l'invite, et ce indéfiniment. C'est pourquoi on parle de *boucle* d'interaction (*toplevel* en anglais). Si l'on entre une déclaration, celle-ci est typée et évaluée de la même manière :

```
# let y = 1 + 2;;
val y : int = 3
```

Ici le système indique de plus qu'il s'agit d'une variable nommée `y`. Les déclarations et expressions se font suite exactement comme dans le texte d'un programme. La variable `y` peut donc être maintenant utilisée :

```
# y * y;;
- : int = 9
```

Cette boucle d'interaction est très utile pour apprendre le langage ou écrire de courts morceaux de code et les tester immédiatement, avec la possibilité d'examiner types et valeurs de manière immédiate.

Dans la suite de ce cours, nous utiliserons souvent le résultat d'une évaluation dans le toplevel `ocaml`, de manière à visualiser immédiatement le type et/ou la valeur.

1.3 Fonctions

Le lecteur était sans doute impatient d'en arriver aux fonctions, puisqu'il s'agit de programmation fonctionnelle. Dans un premier temps, il n'y a pas de différence avec les autres langages de programmation : les fonctions servent à découper le code de manière logique, en éléments de taille raisonnable, et à éviter la duplication de code.

1.3.1 Syntaxe

La syntaxe d'une déclaration de fonction est conforme à la syntaxe de son utilisation. Ainsi

```
let f x = x * x
```

définit une fonction `f` ayant un unique argument `x` et retournant son carré. Comme on le voit ici, le corps d'une fonction n'est autre qu'une expression, qui sera évaluée lorsque la fonction sera appelée. Il n'y a pas de `return` comme en C ou en JAVA, ce qui est une fois encore cohérent avec l'absence de distinction expression/instruction. D'autre part, on note que le système infère automatiquement le type de l'argument `x`, ainsi que le type du résultat (et donc le type de la fonction `f`).

Si l'on entre la déclaration ci-dessus dans le toplevel OCAML, on obtient

```
# let f x = x * x;;
val f : int -> int = <fun>
```

Autrement dit le système indique que l'on vient de déclarer une variable appelée `f`, que son type est `int -> int` c'est-à-dire le type d'une fonction prenant un entier en argument et retournant un entier, et enfin que sa valeur est `<fun>` c'est-à-dire une fonction. Une telle valeur ne saurait être affichée autrement, car le code n'est pas une valeur de première classe (contrairement à LISP par exemple).

Comme nous l'avons déjà vu, l'application s'écrit par simple juxtaposition. Ainsi l'application de `f` à 4 donne

```
# f 4;;
- : int = 16
```

Procédures et fonctions sans argument

En OCAML, une procédure n'est rien d'autre qu'une fonction dont le résultat est de type `unit`. On peut ainsi introduire une référence `x` et une fonction `set` pour en fixer la valeur :

```
# let x = ref 0;;
# let set v = x := v;;
val set : int -> unit = <fun>
```

et la fonction `set` ne retourne pas (vraiment) de valeur, ainsi que l'indique son type. Elle procède par effet de bord. On peut ainsi l'appeler sur la valeur `3` et constater l'effet sur le contenu de la référence `x` :

```
# set 3;;
- : unit = ()
# !x;;
- : int = 3
```

Inversement une fonction sans argument va s'écrire comme prenant un unique argument de type `unit`, celui-ci se notant alors `()`. On peut ainsi définir une fonction `reset` remettant le contenu de la référence `x` à `0` :

```
# let reset () = x := 0;;
val reset : unit -> unit = <fun>
```

et l'appel à `reset` s'écrit `reset ()`. Sur ce dernier point on constate que la syntaxe de la valeur `()` n'a pas été choisie au hasard : on retrouve la syntaxe usuelle des fonctions C ou JAVA sans argument.

Fonctions à plusieurs arguments

Une fonction ayant plusieurs arguments se déclare avec toujours la même syntaxe consistant à juxtaposer fonction et arguments :

```
# let f x y z = if x > 0 then y + x else z - x;;
val f : int -> int -> int -> int = <fun>
```

On voit sur cet exemple que le type d'une fonction prenant trois entiers en argument et retournant un entier s'écrit `int -> int -> int -> int`. L'application d'une telle fonction emprunte toujours la même syntaxe :

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

Fonctions locales

Comme nous le verrons plus en détail dans la section suivante, une fonction est en OCAML une valeur comme une autre. Dès lors elle peut être déclarée *localement* comme une variable de n'importe quel autre type. Ainsi on peut écrire

```
# let carre x = x * x in carre 3 + carre 4 = carre 5;;
- : bool = true
```

ou encore une fonction locale à une autre fonction

```
# let pythagore x y z = let carre n = n*n in carre x + carre y = carre z;;
val pythagore : int -> int -> int -> bool = <fun>
```

On notera que la notion de fonction locale existe dans certains langages tel que PASCAL ou certaines extensions non-ANSI du C, même si la syntaxe y est un peu plus lourde qu'en OCAML.

1.3.2 Fonctions comme valeurs de première classe

Jusqu'à présent, les fonctions d'OCAML ne diffèrent pas vraiment des fonctions du C ou des méthodes statiques de JAVA. Mais elles vont en réalité beaucoup plus loin, justifiant le nom de programmation fonctionnelle. En effet, elles sont des *valeurs de première classe*, c'est-à-dire des valeurs pouvant être créées par des calculs, passées en argument à des fonctions ou retournées, comme n'importe quelles autres valeurs.

Une fonction peut être une expression comme une autre, alors anonyme, et introduite par le mot clé `fun`. Ainsi

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

est la fonction qui à un entier `x` associe son successeur. C'est bien entendu une expression ayant pour type `int -> int` et pouvant donc être appliquée à un entier :

```
# (fun x -> x+1) 3;;
- : int = 4
```

Une déclaration de fonction de la forme

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

n'est en réalité rien d'autre que du sucre syntaxique pour la déclaration

```
let f = fun x -> x + 1;;
```

Il n'a donc pas une déclaration `let` pour les variables (ici au sens usuel de variable contenant une valeur d'un type de base tel que `int`) et une déclaration `let` pour les fonctions, mais une unique déclaration `let` pour introduire des variables pouvant contenir des valeurs de type quelconque, que ce soient des entiers, des booléens, des fonctions, etc.

Application partielle

Les fonctions anonymes peuvent avoir plusieurs arguments, avec la syntaxe suivante

```
fun x y -> x * x + y * y
```

De manière rigoureusement équivalente on peut écrire

```
fun x -> fun y -> x * x + y * y
```

Cette dernière écriture suggère que l'on peut appliquer une telle fonction à un seul argument. On peut effectivement le faire, et le résultat est alors une fonction. On parle alors d'*application partielle*. Ainsi on peut définir une fonction `f` prenant deux entiers en arguments

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

puis construire une seconde fonction `g` en appliquant `f` partiellement :

```
# let g = f 3;;
val g : int -> int = <fun>
```

Le type de `g` est bien celui d'une fonction prenant un argument entier, et si l'on applique `g` à `4` on obtient bien la même chose qu'en appliquant directement `f` à `3` et `4` :

```
# g 4;;
- : int = 25
```

La fonction `g` est comparable à la fonction

```
fun y -> 3 * 3 + y * y
```

c'est-à-dire que son corps est le même que celui de `f` dans lequel la variable formelle `x` a été substituée par la valeur de l'argument effectif (`3` ici). Il est important de noter que si l'on avait partiellement appliqué `f` à une expression plus complexe, comme `1+2`, alors cette expression n'aurait été évaluée qu'une seule fois, comme si l'on avait écrit

```
let x = 1 + 2 in fun y -> x * x + y * y
```

et non pas substituée textuellement (ce qui équivaudrait alors à `fun y -> (1+2)*(1+2)+y*y`). D'une manière générale, OCAML évalue toujours le ou les arguments d'une fonction avant de procéder à l'appel : on dit que c'est un langage *strict* (par opposition aux langages dits *paresseux* où l'évaluation d'un argument est retardée jusqu'au moment de sa première utilisation).

L'application partielle d'une fonction est une expression qui est encore une fonction. C'est donc une manière de retourner une fonction. Mais on peut procéder à un calcul avant de retourner un résultat fonctionnel, comme dans

```
# let f x = let x2 = x * x in fun y -> x2 + y * y;;
val f : int -> int -> int = <fun>
```

On obtient ici une fonction `f` prenant deux entiers et retournant un entier, qui se comporte comme si l'on avait écrit

```
# let f x y = x * x + y * y;;
```

mais la première version est plus efficace si elle est appliquée partiellement. En effet, `x * x` est alors calculé *une seule fois* dans la variable `x2`, dont la valeur sera ensuite directement consultée pour chaque appel à la fonction `fun y -> x2 + y * y`. Alors qu'une application partielle de la seconde version n'apportera aucun bénéfice (si ce n'est peut-être dans l'écriture du code) car `x * x` sera recalculé à chaque fois.

Un exemple subtil mais néanmoins typique est celui d'un compteur utilisant une référence. La fonction `compteur_depuis` ci-dessous prend en argument une valeur entière et retourne un compteur (une fonction de type `unit -> int` produisant un nouvel entier à chaque appel) démarrant à cette valeur. Pour cela une référence est créée localement à la fonction servant de compteur :

```
# let compteur_depuis n = let r = ref (n-1) in fun () -> incr r; !r;;
val compteur_depuis : int -> unit -> int = <fun>
```

(`incr` est une fonction prédéfinie incrémentant la valeur d'une référence entière). On obtient alors un nouveau compteur chaque fois que l'on applique partiellement la fonction `compteur_depuis` :

```
# let compteur = compteur_depuis 0;;
val compteur : unit -> int = <fun>
# compteur ();;
- : int = 0
# compteur ();;
- : int = 1
# compteur ();;
- : int = 2
...
```

Ordre supérieur

De même qu'une fonction peut retourner une autre fonction comme résultat, elle peut également prendre une ou plusieurs fonctions en argument. Cette capacité de manipuler les fonctions comme des valeurs de première classe est appelée *ordre supérieur*.

Ainsi on peut écrire une fonction prenant en argument deux fonctions `f` et `g` et recherchant le premier entier naturel où elles diffèrent d'au moins deux unités :

```
# let diff f g =
    let n = ref 0 in while abs (f !n - g !n) < 1 do incr n done; !n;;
val diff : (int -> int) -> (int -> int) -> int = <fun>
```

Le type est bien celui d'une fonction prenant deux arguments, chacun de type `int -> int` donc des fonctions des entiers vers les entiers, et retournant un entier. On peut alors appliquer `diff` à deux fonctions :

```
# diff (fun x -> x) (fun x -> x*x);;
- : int = 2
```

Un exemple très courant de fonction d'ordre supérieur est celui d'un *itérateur*. Dès que l'on a une structure de données de type collection (ensemble, dictionnaire, file, etc.), on l'équipe naturellement d'une manière d'itérer sur tous ses éléments (par exemple pour les afficher ou les compter). En Java cette itération se présente généralement sous la forme d'une méthode `elements` retournant une énumération, elle-même équipée de deux méthodes `hasMoreElements` et `nextElement`. L'itération est alors réalisée par une boucle de la forme

```
for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
    ... on traite e.nextElement() ...
}
```

En Ocaml l'itérateur est habituellement réalisé par une fonction d'ordre supérieur prenant en argument la fonction effectuant le traitement sur chaque élément. Ainsi une table associant des chaînes de caractères à d'autres chaînes de caractères fournira une fonction de profil

```
val iter : (string -> string -> unit) -> table -> unit
```

Le premier argument de `iter` est la fonction qui sera appliquée à chaque couple de chaînes présent dans la table, et le second est la table proprement dite (d'un type `table` supposé défini). Si l'on souhaite compter le nombre d'associations dans une telle table `t` il suffira d'écrire

```
let n = ref 0 in iter (fun x y -> incr n) t; !n
```

et si l'on souhaite afficher toutes les associations on pourra écrire

```
iter (fun x y -> Printf.printf "%s -> %s\n" x y) t
```

La plupart des structures de données Ocaml fournissent de tels itérateurs, souvent avec plusieurs variantes, y compris les structures impératives usuelles telles que les tableaux, les files, les tables de hachage, etc. Et pour cette raison en particulier il est fréquent d'utiliser des fonctions anonymes.

Différence avec les pointeurs de fonctions

Il est très important de bien comprendre la différence avec les pointeurs de fonctions du C, car l'on entend bien souvent « en C aussi une fonction peut recevoir ou retourner une fonction par l'intermédiaire d'un pointeur de fonction ».

Pour une certaine catégorie de fonctions d'ordre supérieur suffisamment simples il n'y a en effet pas de différence. Mais dès lors que les fonctions construites font référence à des calculs locaux il n'y a plus d'analogie possible. Reprenons l'exemple de la fonction

```
let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

Lorsque l'on applique partiellement `f` on obtient une fonction qui fait référence à la valeur de `x2`. Plus précisément, chaque application partielle donnera une *nouvelle* fonction faisant référence à une variable `x2` *différente*. On ne saurait réaliser la même chose avec des pointeurs de fonctions, car la fonction C en question devrait faire référence à des variables locales qui ne survivent pas à l'appel à `f` (de manière générale, il est incorrect de retourner un pointeur sur une fonction C locale). Pour obtenir le même effet qu'en

OCAML il faudrait construire en C une structure de données plus complexe, contenant non seulement un pointeur de fonction mais également les valeurs susceptibles d'être utilisées par cette fonction. C'est ce que l'on appelle une *clôture* et c'est d'ailleurs ainsi qu'OCAML représente les fonctions (la référence à un morceau de code *et* un environnement dans lequel évaluer celui-ci).

1.3.3 Fonctions récursives

Dans les langages de programmation impératifs, l'utilisation de fonctions récursives est traditionnellement négligée, voire méprisée, au profit de boucles. Il y a à cela des raisons historiques (le coût d'un appel de fonction a longtemps été prohibitif) et des raisons techniques (peu de langages compilent efficacement les fonctions récursives terminales, d'où une utilisation excessive, voire fatale, de l'espace de pile).

Dans les langages fonctionnels, l'utilisation de fonctions récursives est au contraire privilégiée, pour les raisons inverses : d'une part un appel de fonction coûte très peu cher, et d'autre part la récursivité terminale est correctement compilée en une boucle. En OCAML la définition d'une fonction récursive est introduite par l'adjonction du mot clé `rec` au mot clé `let`.

Ainsi, une manière de réécrire la fonction `diff` ci-dessus est d'utiliser une fonction récursive locale au lieu d'une boucle `while` :

```
let diff f g =
  let rec boucle n = if abs (f n - g n) < 1 then boucle (n+1) else n in
  boucle 0
```

On voit tout de suite que cette écriture, à peine plus longue, évite l'utilisation d'une référence. L'argument `n` de la fonction `boucle` n'a en effet aucune raison d'être modifié. Cette version récursive est même plus efficace que celle utilisant une boucle `while` (car l'accès à `n` et son incrémentation sont plus rapides que lorsqu'il s'agit d'une référence).

D'une manière générale, l'écriture à l'aide d'une fonction récursive donne souvent un code plus lisible et plus susceptible d'être correct (car d'invariant plus simple) que son équivalent impératif utilisant une boucle. Pour s'en convaincre il suffit de comparer ces deux versions de la fonction factorielle :

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

et

```
let fact n =
  let f = ref 1 in
  let i = ref n in
  while !i > 0 do f := !f * !i; decr i done;
  !f
```

L'argument justifiant la correction de la seconde version est nettement plus complexe que pour la première version.

1.3.4 Polymorphisme

Au point où nous en sommes, le lecteur attentif peut s'être légitimement demandé si une déclaration telle que

```
let f x = x;;
```

est acceptée par le compilateur OCAML et le cas échéant quel est le type donné à cette fonction. Il s'avère qu'une telle déclaration est en effet acceptée et qu'OCAML lui donne le type suivant :

```
# let f x = x;;
val f : 'a -> 'a = <fun>
```

Ici `'a` ne désigne pas vraiment un type mais une *variable de type*, pouvant prendre n'importe quelle valeur parmi tous les types possibles. Le type de `f` est donc celui d'une fonction prenant un argument d'un type quelconque et retournant une valeur du même type. Une telle fonction est dite *polymorphe*. On peut ainsi l'appliquer à un entier :

```
# f 3;;
- : int = 3
```

mais aussi à un booléen :

```
# f true;;
- : bool = true
```

ou encore à une fonction :

```
# f print_int;;
- : int -> unit = <fun>
```

Un autre exemple de fonction polymorphe est celui d'une fonction choisissant une valeur parmi deux en fonction d'un troisième argument booléen :

```
# let choix b x y = if b then x else y;;
val choix : bool -> 'a -> 'a -> 'a = <fun>
```

ou encore l'exemple typique de la fonction réalisant la composition de deux fonctions :

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

Sur ce dernier exemple on constate qu'OCAML infère bien le type attendu : la composition d'une fonction de A vers B et d'une fonction de C vers A est une fonction de C vers B. D'une manière générale, OCAML infère toujours le type le plus général possible. D'autre part, on remarque que l'on aurait pu aussi bien écrire :

```
let compose f g x = f (g x);;
```

La vision naturelle de `compose` comme prenant deux fonctions et retournant une fonction n'est qu'une manière de voir cette fonction, comme partiellement appliquée. Mais on peut très bien la voir comme prenant trois arguments `f`, `g` et `x` et retournant `f (g x)`.

Récapitulation

Dans cette section, nous avons vu que
- les fonctions sont des valeurs comme les autres : elles peuvent être locales, anonymes, arguments d'autres fonctions, etc. ;
- les fonctions peuvent être partiellement appliquées ;
- les fonctions peuvent être polymorphes ;
- l'appel de fonction ne coûte pas cher.

1.4 Allocation mémoire

Jusqu'à présent nous avons manipulé des valeurs simples (entiers et booléens) et des fonctions. Dans cette section, nous allons aborder les types de données complexes tels que tableaux, enregistrements, etc., c'est-à-dire ceux qui impliquent une allocation mémoire[1]. En OCAML, l'allocation mémoire est réalisée par un *garbage collector* — ramasse-miettes ou glaneur de cellules en français, et GC pour faire court. Le principal intérêt d'un GC est la récupération automatique de la mémoire qui n'est plus utilisée. Les GC ne sont pas propres aux langages de programmation fonctionnelle ; JAVA en possède un également.

Outre la récupération automatique, l'intérêt d'un GC se situe également dans l'*efficacité* de l'allocation mémoire. Celle-ci est bien supérieure à l'utilisation d'un `malloc` en C. Le GC d'OCAML est particulièrement efficace, si bien qu'il faut perdre le réflexe « allouer coûte cher » qu'ont certains programmeurs, même s'il faut bien entendu continuer à se soucier de la complexité en espace des programmes que l'on écrit.

Nous commençons par présenter les structures usuelles de tableaux et d'enregistrements, avant d'introduire l'un des points forts d'OCAML, les types construits.

1.4.1 Tableaux

Mis à part une syntaxe un peu déroutante pour le débutant, les tableaux d'OCAML sont très semblables aux tableaux de C ou de PASCAL. On alloue un tableau avec la fonction `Array.create`[2]. Le premier argument est la taille du tableau et le second la valeur initiale de ses éléments :

```
# let a = Array.create 10 0;;
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Le type `int array` est celui d'un tableau dont les éléments sont de type `int`. Comme pour une déclaration de variable, le tableau doit être initialisé. Cela mérite une parenthèse. De manière générale, OCAML n'autorise pas l'introduction d'une valeur qui serait incomplète ou mal formée, et ceci permet de garantir une propriété très forte : toute expression bien typée s'évalue en une valeur de ce type, dès lors que l'évaluation termine et ne lève pas d'exception. Dit autrement, un programme OCAML ne peut pas produire de `segmentation fault` comme en C ou de `NullPointerException` comme en JAVA.

[1]Les valeurs fonctionnelles impliquent également une allocation mémoire, mais nous avons passé cet aspect sous silence.

[2]Le point séparateur dans `Array.create` sera expliqué au chapitre 2.

On remarque ensuite que le toplevel OCAML utilise une syntaxe particulière pour afficher la valeur du tableau. Cette syntaxe peut être utilisée en entrée pour allouer un nouveau tableau :

```
let a = [| 1; 2; 3; 4 |]
```

On accède à l'élément d'indice i du tableau `a` avec la syntaxe `a.(`i`)` et on le modifie (en place) avec la syntaxe `a.(`i`) <-` v. On obtient la taille d'un tableau (en temps constant) avec la fonction `Array.length`. Voici par exemple comment on écrit le tri par insertion en OCAML :

```
let tri_insertion a =
  let swap i j = let t = a.(i) in a.(i) <- a.(j); a.(j) <- t in
  for i = 1 to Array.length a - 1 do (* insérer l'élément i dans 0..i-1 *)
    let j = ref (i - 1) in
    while !j >= 0 && a.(!j) > a.(!j + 1) do swap !j (!j + 1); decr j done
  done
```

ou encore, en utilisant plutôt une fonction récursive pour la boucle interne :

```
let tri_insertion a =
  let swap i j = let t = a.(i) in a.(i) <- a.(j); a.(j) <- t in
  for i = 1 to Array.length a - 1 do (* insérer l'élément i dans 0..i-1 *)
    let rec insère j =
      if j >= 0 && a.(j) > a.(j+1) then begin swap j (j+1); insère (j-1) end
    in
    insère (i-1)
  done
```

1.4.2 Enregistrements

Les enregistrements en OCAML sont comparables aux records du PASCAL ou aux structures du C. Comme dans ces langages-là, il faut commencer par déclarer le type enregistrement et ses champs :

```
type complexe = { re : float; im : float }
```

On peut alors définir un enregistrement avec la syntaxe suivante :

```
# let x = { re = 1.0; im = -1.0 };;
val x : complexe = {re = 1.; im = -1.}
```

Comme toujours, on note que l'enregistrement doit être totalement initialisé, et que son type est inféré (ici le type `complexe`). On accède au champ d'un enregistrement avec la notation usuelle :

```
# x.im;;
- : float = -1.
```

En revanche, les champs d'un enregistrement ne sont pas modifiables par défaut. Pour cela il faut le déclarer *explicitement* à l'aide du mot clé `mutable` :

```
type personne = { nom : string; mutable age : int }
```

On peut alors modifier le champ correspondant avec l'opérateur `<-` :

```
# let p =  { nom = "Martin"; age = 23 };;
val p : personne = {nom = "Martin"; age = 23}
# p.age <- p.age + 1;;
- : unit = ()
# p.age;;
- : int = 24
```

Retour sur les références

On peut maintenant expliquer ce que sont les références d'OCAML. Ce ne sont rien d'autre que des enregistrements du type polymorphe prédéfini suivant :

```
type 'a ref = { mutable contents : 'a }
```

c'est-à-dire un enregistrement avec un unique champ mutable appelé `contents`. On s'en aperçoit en inspectant la valeur d'une référence dans le toplevel :

```
# ref 1;;
- : int ref = {contents = 1}
```

Création, accès et modification d'une référence ne sont en réalité que des opérations cachées sur des enregistrements du type `ref`.

1.4.3 N-uplets

Il existe en OCAML une notion primitive de n-uplets. Un n-uplet est introduit avec la notation mathématique usuelle :

```
# (1,2,3);;
- : int * int * int = (1, 2, 3)
```

et son type est formé à partir de l'opérateur `*` et des types de ses différents éléments. Le n-uplet peut être d'arité quelconque et ses éléments peuvent être de types différents :

```
# let v = (1, true, "bonjour", 'a');;
val v : int * bool * string * char = (1, true, "bonjour", 'a')
```

On peut accéder aux différents éléments d'un n-uplet à l'aide d'une déclaration `let` *déstructurante* :

```
# let (a,b,c,d) = v;;
val a : int = 1
val b : bool = true
val c : string = "bonjour"
val d : char = 'a'
```

Une telle déclaration `let` peut être globale (comme ici) ou locale, et associe à autant de variables qu'il y a d'éléments dans le n-uplet les valeurs correspondantes (ici `a`, `b`, `c` et `d`). Les éléments d'un n-uplet ne sont pas modifiables en place — et il n'y a pas moyen de changer cet état de fait, contrairement aux enregistrements.

Les n-uplets sont utiles lorsqu'une fonction doit retourner plusieurs valeurs (on pourrait utiliser un type enregistrement mais un n-uplet est plus immédiat à utiliser). Ainsi on peut écrire une fonction de division par soustraction retournant quotient et reste sous la forme d'une paire :

```
# let rec division n m =
    if n < m then (0, n)
    else let (q,r) = division (n - m) m in (q + 1, r);;
val division : int -> int -> int * int = <fun>
```

Note : Rien n'empêche d'écrire une fonction prenant plusieurs arguments sous la forme d'une fonction prenant un unique n-uplet en argument. OCAML offre d'ailleurs une syntaxe agréable pour cela, comparable à celle du `let` destructurant :

```
# let f (x,y) = x + y;;
val f : int * int -> int = <fun>
# f (1,2);;
- : int = 3
```

Si l'on retrouve alors la syntaxe usuelle des fonctions C ou JAVA il ne faut pas s'y tromper : d'une part, il s'agit bien d'une fonction OCAML prenant un *unique* argument, rendant en particulier l'application partielle impossible ; et d'autre part elle peut être moins efficace qu'une fonction à plusieurs arguments, à cause de la construction d'un n-uplet lors de son appel[3].

1.4.4 Listes

Il existe en OCAML un type prédéfini de *listes*. Ces listes sont immuables (non modifiables en place) et homogènes (tous les éléments d'une liste sont du même type). Si α désigne le type des éléments d'une liste, celle-ci a le type α `list`. On construit des listes à partir de la liste vide `[]` et de l'adjonction d'un élément en tête d'une liste avec l'opérateur infixe `::`. Ainsi la liste contenant les entiers `1`, `2` et `3` peut être définie par

```
# let l = 1 :: 2 :: 3 :: [];;
val l : int list = [1; 2; 3]
```

Comme on le voit sur cet affichage, OCAML propose une syntaxe plus concise pour construire directement une liste étant donnés tous ses éléments :

```
# let l = [1; 2; 3];;
val l : int list = [1; 2; 3]
```

[3]En réalité, OCAML évite souvent de construire un n-uplet lors de l'appel à une telle fonction.

De nombreuses fonctions sur les listes sont prédéfinies : pour accéder au premier élément, à tous les autres, calculer la longueur, etc. La puissance des listes d'OCAML vient de la possibilité de construction par cas sur la forme d'une liste, appelée *filtrage*. Une liste est en effet soit vide, soit formée d'un premier élément et d'une autre liste, et la construction `match with` permet de raisonner par cas selon la forme d'une liste. On peut écrire une fonction calculant la somme des éléments d'une liste d'entiers comme ceci :

```
# let rec somme l =
    match l with
      [] -> 0
    | x :: r -> x + somme r;;
val somme : int list -> int = <fun>
```

La construction `match with` est constituée d'une expression à examiner (entre les mots clés `match` et `with`, ici la liste `l`) et d'un ou plusieurs cas de filtrage (séparés par une barre verticale `|`). Un cas de filtrage est constitué d'un motif et d'une expression séparés par une flèche. Le motif est un constructeur (ici `[]` ou `::`) et ses arguments peuvent être nommés (ici les deux arguments de `::` sont nommés `x` et `r`). La sémantique est intuitive : l'expression examinée est comparée aux différents motifs (selon leur ordre d'apparition) et lorsqu'il y a correspondance pour un motif, l'expression associée à ce motif est évaluée, dans un environnement où les variables du motif sont associées aux valeurs correspondantes dans l'expression filtrée.

Ainsi, si l'on applique la fonction `somme` à la liste `1::2::3::[]`, le cas de filtrage qui s'applique est le second, `x` prenant la valeur `1` et `r` la valeur `2::3::[]`. On évalue alors l'expression `x+somme r` dans cet environnement. Ce cas de filtrage va encore s'appliquer deux fois, avant que l'on parvienne finalement à la liste `[]` pour laquelle le premier cas s'appliquera, retournant la valeur `0`. On aura au final le résultat attendu :

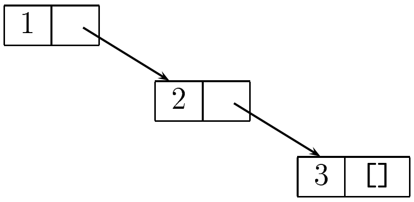
```
# somme [1;2;3];;
- : int = 6
```

On comprend vite d'où vient la puissance du filtrage : il agit comme une série de tests et de définitions de variables locales en même temps, le tout avec une syntaxe extrêmement concise. Il existe même un raccourci syntaxique pour faire du filtrage sur le dernier argument d'une fonction, avec le mot clé `function`. On peut ainsi réécrire la fonction `somme` aussi simplement que

```
let rec somme = function
  | [] -> 0
  | x :: r -> x + somme r
```

(Il est possible d'introduire une barre verticale avant le premier cas de filtrage, dans un souci purement esthétique de symétrie.)

Il est très important de comprendre que les listes d'OCAML ne sont fondamentalement pas différentes des listes chaînées que l'on utiliserait en C ou en JAVA. La liste vide est représentée en interne par l'entier 0 (comme le pointeur `null` en C ou en JAVA) et une liste non vide est représentée par un *pointeur* vers un bloc mémoire contenant deux valeurs, à savoir l'élément de la liste et le reste de la liste (à son tour soit 0 pour `[]`, soit un pointeur). La liste `[1 ;2 ;3]` correspond donc à une allocation mémoire de la forme :



Lorsqu'un programme OCAML manipule des listes (les passer en arguments à des fonctions, les retourner, etc.), il ne fait que manipuler des pointeurs, exactement comme le ferait un bon programme C ou JAVA. Mais la différence essentielle est qu'OCAML ne permet de construire que des listes bien formées, en particulier parce que les pointeurs ne sont pas explicités. Et là où un programmeur C ou JAVA doit *penser* à tester si un pointeur est `null`, le programmeur OCAML utilisera une construction de filtrage qui l'*obligera* à considérer ce cas, mais avec une grande concision syntaxique.

1.4.5 Types construits

Les listes ne sont qu'un cas particulier de types *construits* (encore appelés types *algébriques*). Un type construit regroupe des valeurs formées à partir d'un ou plusieurs *constructeurs*. Comme pour les enregistrements, les types construits doivent être déclarés, afin d'introduire les noms des constructeurs et leur arité. Ainsi la déclaration suivante

```
type formule = Vrai | Faux | Conjonction of formule * formule
```

introduit un nouveau type construit `formule`, ayant deux constructeurs constants (`Vrai` et `Faux`) et un constructeur `Conjonction` ayant deux arguments de type `formule`. Les constructeurs constants sont directement des valeurs du type construit :

```
# Vrai;;
- : formule = Vrai
```

Les constructeurs non constants doivent être appliqués à des arguments de types compatibles avec leur déclaration :

```
# Conjonction (Vrai, Faux);;
- : formule = Conjonction (Vrai, Faux)
```

Les parenthèses sont obligatoires. Un constructeur est nécessairement complètement appliqué ; ce n'est pas une fonction.

Le type prédéfini des listes a la définition suivante, à la syntaxe de ses constructeurs prêt :

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

On peut voir un type construit grossièrement comme une union de structures en C, mais avec une représentation mémoire plus efficace. En effet, chaque constructeur est représenté soit par un entier s'il est constant (comme la liste vide), soit par un pointeur vers un bloc mémoire de la taille adéquate sinon.

La notion de filtrage introduite avec les listes se généralise à tout type construit. Ainsi on peut écrire une fonction d'évaluation des formules logiques ci-dessus de la façon suivante :

```
# let rec evalue = function
  | Vrai -> true
  | Faux -> false
  | Conjonction (f1, f2) -> evalue f1 && evalue f2;;
val evalue : formule -> bool = <fun>
```

Le filtrage sur les types construits peut être *imbriqué*, i.e. les arguments des constructeurs dans les motifs peuvent être à leur tour des motifs. Ainsi on peut raffiner la fonction `evalue` en

```
let rec evalue = function
  | Vrai -> true
  | Faux -> false
  | Conjonction (Faux, f2) -> false
  | Conjonction (f1, Faux) -> false
  | Conjonction (f1, f2) -> evalue f1 && evalue f2;;
```

On a multiplié les motifs pour faire apparaître des cas particuliers (lorsqu'un argument de `Conjonction` est `Faux` on retourne `false` directement). Le compilateur OCAML vérifie d'une part l'exhaustivité du filtrage (tous les cas sont couverts), et d'autre part l'absence de motif couvert par un motif précédent ; il émet un avertissement en cas de problème.

Lorsqu'un argument de constructeur n'a pas besoin d'être nommé, il peut être remplacé par `_` (le motif universel). Lorsque deux motifs sont associés à la même expression, ils peuvent être regroupés par une barre verticale. Ainsi la fonction ci-dessus devient-elle encore plus lisible :

```
let rec evalue = function
  | Vrai -> true
  | Faux -> false
  | Conjonction (Faux, _) | Conjonction (_, Faux) -> false
  | Conjonction (f1, f2) -> evalue f1 && evalue f2;;
```

Le filtrage n'est pas limité aux types construits. Il peut être utilisé sur des valeurs de tout type, avec la syntaxe habituelle. Ainsi on peut multiplier les éléments d'une liste d'entiers en combinant filtrage sur les listes et sur les entiers :

```
let rec mult = function
  | [] -> 1
  | 0 :: _ -> 0
  | x :: l -> x * mult l
```

Enfin, lorsque le filtrage est composé d'un seul motif, il peut être écrit de manière plus concise à l'aide d'une construction `let`, sous la forme `let` *motif* `=` *expr*. La syntaxe utilisée plus haut pour déstructurer les n-uplets n'en est qu'un cas particulier.

Récapitulation

Dans cette section, nous avons vu que

– l'allocation mémoire ne coûte pas cher, la libération se fait automatiquement ;

– les valeurs allouées sont nécessairement initialisées ;
– la majorité des valeurs construites ne sont pas modifiables en place : seuls le sont les tableaux et les champs d'enregistrements explicitement déclarés `mutable` ;
– la représentation mémoire des valeurs construites est efficace ;
– le filtrage permet un examen par cas sur les valeurs construites.

1.5 Exceptions

Il nous reste un aspect du langage « de base » à décrire : les exceptions. Elles sont comparables aux exceptions qui existent dans d'autres langages tels que JAVA. À tout instant une exception peut être *levée* pour signaler un comportement exceptionnel, une erreur le plus souvent. Une exception est levée à l'aide de la construction `raise` :

```
let division n m =
  if m = 0 then raise Division_by_zero else ...
```

La construction `raise` est une expression comme une autre, mais son typage est particulier : elle peut prendre n'importe quel type, selon son contexte d'utilisation. Ainsi dans l'expression

```
if x >= 0 then 2 * x else raise Negatif
```

l'expression `raise Negatif` a le type `int`, de manière à ce que l'expression complète soit bien typée, alors que dans l'expression

```
if x < 0 then raise Negatif; 2 * x
```

la même expression `raise Negatif` aura le type `unit`.

On rattrape les exceptions à l'aide de la construction `try with` dont la syntaxe est identique (au mot clé près) à celle de la construction `match with`. Ainsi on peut écrire

```
try division x y with Division_by_zero -> (0,0)
```

L'intégralité de la construction `try with` est une expression OCAML, ici de type `int * int`. L'évaluation de cette expression se fait ainsi : l'expression `division x y` est évaluée ; si elle donne un résultat, celui-ci est retourné comme le résultat de l'expression toute entière ; si l'exception `Division_by_zero` est levée, alors l'expression `(0,0)` est évaluée et son résultat est celui de l'expression toute entière ; si une autre exception est levée, elle est propagée jusqu'au premier `try with` à même de la rattraper.

On peut déclarer de nouvelles exceptions avec la déclaration `exception` et les exceptions, comme les constructeurs, peuvent avoir des arguments (non polymorphes) :

```
exception Error
exception Unix_error of string
```

En réalité, les exceptions *sont* des constructeurs, du type prédéfini `exn`. Comme en JAVA les exceptions sont des valeurs de première classe. On peut ainsi écrire

```
# let vérifie e = if e = Not_found then raise e;;
val vérifie : exn -> unit = <fun>
```

Il existe quelques exceptions prédéfinies, dont la plus courante est `Not_found`. Elle est utilisée notamment dans les structures de données implantant des tables d'association, pour signaler l'absence de valeur associée. Ainsi la fonction de recherche dans une table de hachage, `Hashtbl.find`, lève l'exception `Not_found` pour signaler une recherche infructueuse. On rencontrera donc souvent la structure de code

```
try Hashtbl.find table clé
with Not_found -> ...
```

Si les exceptions servent principalement à la gestion des erreurs, elles peuvent également être utilisées à d'autres fins, par exemple pour modifier le flot de contrôle. On peut ainsi utiliser une exception pour sortir d'une boucle, comme dans

```
try
  while true do
    let key = read_key () in
    if key = 'q' then raise Exit;
    ...
  done
with Exit ->
  close_graph (); exit 0
```

Chapitre 2

Modules

Lorsque les programmes deviennent gros, il est important qu'un langage de programmation apporte de bons outils de *génie logiciel*, en particulier pour découper les programmes en unités de taille raisonnable (*modularité*), occulter la représentation concrète de certaines données (*encapsulation*), et éviter au mieux la duplication du code. Il existe de nombreuses manières de parvenir à ces objectifs. Dans la programmation orientée objets, ce sont les classes et les concepts associés qui en sont la clé. En OCAML, ces fonctionnalités sont apportées par le système de *modules*.

2.1 Modules

Comme le nom de *module* le suggère, un module est avant tout une manière d'introduire de la modularité dans un programme, i.e. de le découper en *unités* de taille raisonnable.

2.1.1 Fichiers et modules

L'unité de programme la plus simple est le fichier. En OCAML, chaque fichier constitue un module différent. Si l'on place un certain nombre de déclarations dans un fichier, disons `arith.ml` :

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

alors la compilation de ce fichier en tant que module se fait en invoquant une compilation sans édition de lien grâce à l'option `-c` du compilateur (même option que `gcc`) :

```
% ocamlc -c arith.ml
```

Le résultat de cette compilation est composé de deux fichiers : `arith.cmo` contenant le code et `arith.cmi` contenant son interface (ici une constant `pi` de type `float` et une fonction `round` de type `float -> float`). Le nom du module constitué par un fichier est obtenu en capitalisant la première lettre de son nom (si nécessaire). Ici le module s'appelle donc `Arith`.

On peut alors faire référence aux éléments de ce module dans un autre fichier OCAML, par l'intermédiaire de la notation `Arith.`x. Ainsi on peut utiliser le module `Arith` dans un autre fichier `main.ml` :

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

et ensuite compiler ce second fichier :

```
% ocamlc -c main.ml
```

Pour que cette compilation réussisse, il est nécessaire que le module `Arith` ait été préalablement compilé : en effet, lorsqu'il est fait référence à un élément de module, comme ici `Arith.pi`, le compilateur recherche un fichier d'interface `arith.cmi`. On peut alors réaliser l'édition de liens, en fournissant les deux fichiers de code au compilateur OCAML :

```
% ocamlc arith.cmo main.cmo
```

Remarque : on aurait pu également compiler `main.ml` et réaliser l'édition de liens en une seule commande :

```
% ocamlc arith.cmo main.ml
```

ou même compiler les deux fichiers et réaliser l'édition de liens en une unique commande :

```
% ocamlc arith.ml main.ml
```

Dans ce dernier cas, les fichiers sont compilés dans l'ordre de leur occurrence sur la ligne de commande, ce qui a l'effet escompté.

2.1.2 Encapsulation

Les modules en tant que fichiers permettent donc le *découpage* du code. Mais les modules permettent bien d'autres choses, dont l'*encapsulation*, i.e. la possibilité d'occulter certains détails de codage. On a en effet la possibilité de fournir une *interface* aux modules que l'on définit, et seuls les éléments présents dans cette interface seront visibles, de même qu'en JAVA le mot clé `private` limite la visibilité d'attributs ou de méthodes. Pour cela on place l'interface dans un fichier de suffixe `.mli` à côté du fichier de suffixe `.ml` contenant le code. Ainsi on peut n'exporter que la fonction `round` du module `Arith` ci-dessus en créant un fichier `arith.mli` de contenu :

```
val round : float -> float
```

On constate que la syntaxe est la même que celle utilisée par le toplevel OCAML dans ses réponses. Ce fichier d'interface doit être compilé *avant* le fichier de code correspondant :

```
% ocamlc -c arith.mli
% ocamlc -c arith.ml
```

Lors de la compilation du code, le compilateur vérifie l'adéquation de types entre les éléments déclarés dans l'interface et les éléments effectivement définis dans le code. Si l'on cherche à recompiler le module `main.ml` on obtient maintenant une erreur :

```
% ocamlc -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

En revanche la définition de `pi` reste visible dans l'intégralité du fichier `arith.ml`.

L'interface ne se limite pas à restreindre les valeurs exportées. Elle permet également de restreindre les types exportés et mieux encore leur seule définition. Supposons par exemple que l'on souhaite coder une mini-bibliothèque d'ensembles d'entiers représentés par des listes. Le code pourrait constituer le fichier `ensemble.ml` suivant

```
type t = int list
let vide = []
let ajoute x l = x :: l
let appartient = List.mem
```

et une interface possible serait la suivante :

```
type t = int list
val vide : t
val ajoute : int -> t -> t
val appartient : int -> t -> bool
```

Ici la définition du type `t` ne sert à rien, sinon à améliorer la lecture de l'interface en identifiant les ensembles (sans le type `t`, le compilateur aurait inféré des types plus généraux mais compatibles pour `vide`, `ajoute` et `appartient`).

Mais on peut aussi cacher la représentation du type `t`, en donnant à l'interface le contenu suivant :

```
type t
val vide : t
val ajoute : int -> t -> t
val appartient : int -> t -> bool
```

Un tel type s'appelle un *type abstrait* ; c'est une notion essentielle sur laquelle nous reviendrons au chapitre 3. Faire du type `t` ci-dessus un type abstrait cache complètement la manière dont les valeurs de ce type sont formées. Si l'on cherche à traiter une valeur du type `Ensemble.t` comme une liste d'entiers (en dehors du fichier `ensemble.ml`), on obtient une erreur. On peut donc modifier à loisir le codage de ces ensembles sans perturber le reste du programme.

Compilation séparée

Le langage OCAML permet en effet la *compilation séparée*. Cela signifie que la compilation d'un fichier ne dépend *que* des interfaces des modules qu'il mentionne, et non de leur code. Ainsi si un module `B` utilise un autre module `A`, une modification dans le code de `A` n'implique pas de recompiler `B`. Seule une modification de l'*interface* de `A` le justifierait, car il faut alors vérifier que les changements éventuels des types des valeurs exportées par `A` préservent le typage de `B`.

La compilation séparée offre un gain de temps considérable au programmeur. Sur un projet composé de cent fichiers OCAML, une modification de code n'impliquera qu'*une seule* compilation et une édition de liens pour refaire l'exécutable.

2.1.3 Langage de modules

L'une des forces du système de modules d'OCAML réside dans le fait qu'il ne se limite pas aux fichiers. On peut en effet définir un nouveau module de même que l'on définit un type, une constante, une fonction ou une exception. Ainsi on peut écrire

```
module M = struct
  let c = 100
  let f x = c * x
end
```

Le mot clé `module` introduit ici la déclaration d'un nouveau module, `M`, dont la définition est le bloc encadré par `struct` et `end`. Un tel bloc est aussi appelé une *structure* (d'où le mot clé `struct`) et est composé d'une suite de déclarations et/ou expressions, *exactement* comme un texte de programme. De telles déclarations de modules peuvent être mêlées aux autres déclarations, et contenir elles-mêmes d'autres déclarations de modules. On peut ainsi écrire des choses comme :

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

De tels modules « locaux » prennent tout leur sens avec les foncteurs, qui sont exposés dans la section suivante, mais on constate déjà qu'ils permettent une organisation de l'*espace de nommage*. Ainsi dans l'exemple ci-dessus, on a pu définir deux fonctions appelées `f`, car celle définie à l'intérieur du module `B` n'est pas immédiatement visible dans le corps du module `A` ; il faut qualifier avec la notation `B.f` pour y avoir accès. On peut donc se servir des modules locaux pour regrouper un certain nombre de types et fonctions ayant un rapport entre eux, et leur donner des noms génériques (comme `add`, `find`, etc.) car la qualification par le nom du module permet de les distinguer d'autres fonctions de mêmes noms.

Lorsque l'on fait très souvent référence à des valeurs d'un module, on peut éviter la qualification systématique par le nom du module en *ouvrant* le module avec la déclaration `open`. Ceci a pour effet de rendre « visibles » tous les éléments de ce module. Ainsi, plutôt que d'écrire systématiquement `Printf.printf`, on ouvrira le module `Printf` en début de fichier :

```
open Printf
...
let print x y = printf "%d + %d = %d\n" x y (x+y)
...
```

Signatures

À l'instar de toutes les autres déclarations, les déclarations de modules sont typées. Le type d'un module est appelé une *signature*. La syntaxe d'une signature est identique à celles des interfaces (les fichiers `.mli`) et encadrée par les mots clés `sig end`. On le constate en déclarant un module dans le toplevel :

```
# module M = struct let a = 2 let f x = a * x end;;
module M : sig val a : int val f : int -> int end
```

Il est logique que la syntaxe soit identique à celle d'une interface car une interface n'est rien d'autre que le type d'un module-fichier. De même que l'on peut définir des modules locaux, on peut définir des signatures :

```
# module type S = sig val f : int -> int end;;
module type S = sig val f : int -> int end
```

On peut alors se servir de telles signatures pour typer explicitement des déclarations de modules :

```
# module M : S = struct let a = 2 let f x = a * x end;;
module M : S
```

L'intérêt d'une telle déclaration est le même que celui des interfaces : restreindre les valeurs ou types exportées (ce que nous avons appelé l'encapsulation). Ici la valeur `M.a` n'est plus visible :

```
# M.a;;
Unbound value M.a
```

Récapitulation

Pour résumer, le système de modules d'OCAML permet
- la modularité, par le découpage du code en unités appelées *modules* ;
- l'encapsulation de types et de valeurs, et en particulier la définition de *types abstraits* ;
- une vraie compilation séparée ;
- une organisation de l'espace de nommage.

2.2 Foncteurs

De même qu'une fonction OCAML peut être générique vis-à-vis de types (polymorphisme) ou d'autres fonctions (ordre supérieur), un module peut être paramétré par un ou plusieurs autres modules. Un tel module s'appelle un *foncteur*.

Pour bien justifier l'utilité des foncteurs, le plus simple est de prendre un exemple. Supposons que l'on veuille coder une table de hachage en OCAML, de la façon la plus générique possible de manière à ce que ce code soit réutilisable dans toute circonstance où une table de hachage est requise. Pour cela il faut paramétrer le code vis-à-vis d'une fonction de hachage (pour l'insertion dans la table et la recherche) et d'une fonction d'égalité (pour la recherche)[1]. On pourrait imaginer utiliser l'ordre supérieur et paramétrer toutes les fonctions de notre module par ces deux fonctions, ce qui donnerait une interface de la forme

```
type 'a t
  (* le type abstrait des tables de hachage contenant des éléments
     de type "'a" *)
val create : int -> 'a t
  (* "create n" crée une nouvelle table de taille initiale "n" *)
val add : ('a -> int) -> 'a t -> 'a -> unit
  (* "add h t x" ajoute la donnée "x" dans la table "t" à l'aide
     de la fonction de hachage "h" *)
val mem : ('a -> int) -> ('a -> 'a -> bool) -> 'a t -> 'a -> bool
  (* "mem h eq t x" teste l'occurrence de "x" dans la table "t" pour
     la fonction de hachage "h" et l'égalité "eq" *)
```

Au delà de sa lourdeur, une telle interface est dangereuse : en effet, on peut ajouter un élément dans la table avec une certaine fonction de hachage, puis le rechercher avec une autre fonction de hachage. Le résultat sera alors erroné, mais le système de types d'OCAML n'aura pas permis d'exclure cette utilisation incorrecte (elle reste bien typée).

Une solution un peu plus satisfaisante consisterait à fixer les fonctions de hachage et d'égalité à la création de la table. Elles seraient alors stockées dans la structure de données et utilisées par les opérations chaque fois que nécessaire. L'interface deviendrait alors plus raisonnable :

```
type 'a t
  (* le type abstrait des tables de hachage contenant des éléments
     de type "'a" *)
val create : ('a -> int) -> ('a -> 'a -> bool) -> int -> 'a t
  (* "create h eq n" crée une nouvelle table de taille initiale "n"
     avec "h" pour fonction de hachage et "eq" pour égalité *)
val add : 'a t -> 'a -> unit
  (* "add t x" ajoute la donnée "x" dans la table "t" *)
val mem : 'a t -> 'a -> bool
  (* "mem t x" teste l'occurrence de "x" dans la table "t" *)
```

De manière interne, le codage pourrait ressembler à quelque chose comme

```
type 'a t = { hash : 'a -> int; eq : 'a -> 'a -> bool; data : 'a list array }
let create h eq n = { hash = h; eq = eq; data = Array.create n [] }
...
```

[1] Il se trouve qu'OCAML prédéfinit une égalité et une fonction de hachage polymorphes, mais il arrive que l'on souhaite utiliser des fonctions différentes.

Cependant, un tel codage a encore des défauts : d'une part, l'accès aux fonctions de hachage et d'égalité se fait forcément par l'intermédiaire de clôtures et pénalise l'exécution par rapport à des fonctions statiquement connues ; d'autre part, la structure de données codant la table contient maintenant des fonctions et pour cette raison il devient difficile de l'écrire sur le disque et de la relire ultérieurement (c'est techniquement possible mais avec d'importantes restrictions).

Heureusement, les foncteurs apportent ici une solution très satisfaisante. Puisque l'on souhaite paramétrer le codage de nos tables de hachage par un type (le type des éléments) et deux fonctions, on va écrire un module paramétré par un autre module contenant ces trois éléments. Un tel foncteur `F` paramétré par un module `X` de signature `S` s'introduit avec la syntaxe

```
module F(X : S) = struct ... end
```

Dans notre cas la signature `S` est la suivante

```
module type S = sig
  type elt
  val hash : elt -> int
  val eq : elt -> elt -> bool
end
```

et le codage des tables de hachage peut donc s'écrire ainsi :

```
module F(X : S) = struct
  type t = X.elt list array
  let create n = Array.create n []
  let add t x =
    let i = (X.hash x) mod (Array.length t) in t.(i) <- x :: t.(i)
  let mem t x =
    let i = (X.hash x) mod (Array.length t) in List.exists (X.eq x) t.(i)
end
```

On constate que dans le corps du foncteur, on accède aux éléments du module-paramètre `X` exactement comme s'il s'agissait d'un module défini plus haut. Le type du foncteur `F` explicite le paramètre `X` avec la même syntaxe :

```
module F(X : S) : sig
  type t
    (* le type abstrait des tables de hachage contenant des éléments
       de type "X.elt" *)
  val create : int -> t
    (* "create h eq n" crée une nouvelle table de taille initiale "n" *)
  val add : t -> X.elt -> unit
    (* "add t x" ajoute la donnée "x" dans la table "t" *)
  val mem : t -> X.elt -> bool
    (* "mem t x" teste l'occurrence de "x" dans la table "t" *)
end
```

On peut alors instancier le foncteur `F` sur le module de son choix. Ainsi, si l'on souhaite une table de hachage pour stocker des entiers on pourra commencer par définir un module `Entiers` ayant la signature `S` :

```
module Entiers = struct
  type elt = int
  let hash x = abs x
  let eq x y = x=y
end
```

puis appliquer le foncteur `F` sur ce module :

```
module Hentiers = F(Entiers)
```

On constate donc que la définition d'un module n'est pas limitée à une structure : il peut s'agir aussi d'une application de foncteur. On peut alors utiliser le module `Hentiers` comme tout autre module :

```
# let t = Hentiers.create 17;;
val t : Hentiers.t = <abstr>
# Hentiers.add t 13;;
- : unit = ()
# Hentiers.add t 173;;
- : unit = ()
# Hentiers.mem t 13;;
- : bool = true
# Hentiers.mem t 14;;
- : bool = false
...
```

Applications

Les applications des foncteurs sont multiples. On les utilise en particulier pour définir

1. *des structures de données paramétrées par d'autres structures de données*
   OCAML offre trois tels foncteurs dans sa bibliothèque standard :
   - `Hashtbl.Make` : des tables de hachage semblables à celles que nous venons de voir ;
   - `Set.Make` : des ensembles finis codés par des arbres équilibrés ;
   - `Map.Make` : des tables d'associations codées par des arbres équilibrés.
2. *des algorithmes paramétrés par des structures de données*
   Ainsi on peut écrire l'algorithme de Dijkstra de recherche du plus court chemin dans un graphe sous forme d'un foncteur paramétré par la structure de donnée représentant le graphe. Le type d'un tel foncteur pourrait être

   ```
   module Dijkstra
     (G : sig
            type graph (* type des graphes *)
            type sommet (* type des sommets *)
   ```

```
            val voisins : graph -> sommet -> (sommet * int) list
              (* ensemble des voisins avec le poids de l'arête *)
          end) :
sig
  val plus_court_chemin :
    G.graph -> G.sommet -> G.sommet -> G.sommet list * int
    (* "plus_court_chemin g a b" recherche le plus court chemin de
       "a" à "b" dans "g" *)
end
```

On voit donc que les foncteurs sont un moyen puissant de réutiliser le code, en l'écrivant de la manière la plus générique possible. Les foncteurs peuvent être rapprochés des *templates* de C++, bien que les différences soient nombreuses.

Note : Le système de modules d'OCAML est en réalité *indépendant* du langage de base. Il forme un langage à part entière, qui pourrait être appliqué à tout autre langage. En particulier, il pourrait être complètement déplié statiquement pour donner un code sans module ni foncteur, même si ce n'est pas ce qui est fait en pratique.

Chapitre 3

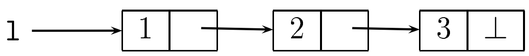
Persistance

Dans ce chapitre nous allons revenir sur un aspect essentiel des structures de données, la *persistance*. Cette notion n'est pas propre à OCAML, ni même à la programmation fonctionnelle, mais est de fait naturelle dans cette famille de langages.

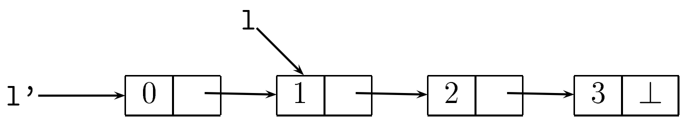
3.1 Structures de données immuables

Comme nous l'avons fait remarquer à de nombreuses reprises, les structures de données OCAML sont pour l'essentiel *immuables* c'est-à-dire non modifiables une fois construites. Ceci a une conséquence très importante : une valeur d'un tel type n'est pas affectée par une opération sur cette valeur ; seules de *nouvelles* valeurs sont retournées. Le plus simple est de l'illustrer avec des listes.

Si on définit la liste `l` par `let l = [1 ; 2 ; 3]` alors `l` est, en termes de représentation mémoire, un pointeur vers un premier bloc contenant `1` et un pointeur vers un second bloc, etc. :



Nous l'avons déjà expliqué. Si on définit maintenant la liste `l'` comme l'adjonction d'un autre élément à la liste `l`, avec la déclaration `let l' = 0 :: l`, on a maintenant la situation suivante :



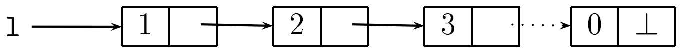
c'est-à-dire que l'application du constructeur `::` a eu pour effet d'allouer un nouveau bloc, dont le premier élément est `0` et le second un pointeur ayant la même valeur que `l`. La variable `l` continue de pointer sur les mêmes blocs qu'auparavant. D'une manière générale, n'importe quelle fonction que l'on pourra écrire sur les listes aura cette propriété de ne pas modifier les listes qui lui sont passées en arguments. C'est cette propriété de la structure de données que l'on appelle *persistance*.

Il est très important de comprendre qu'il y a ici *partage*. La déclaration de `l'` n'alloue pas plus qu'un seul bloc (puisqu'un seul constructeur est appliqué), les blocs formant `l` étant en quelque sorte réutilisés mais non modifiés. On a bien deux listes de 3 et 4 éléments respectivement, à savoir `[1 ;2 ;3]` et `[0 ;1 ;2 ;3]`, mais seulement quatre blocs mémoire. En particulier il n'y a *pas eu de copie*. D'une manière générale, OCAML ne copie jamais de valeurs, sauf si l'on écrit explicitement une fonction de copie, telle que

```
let rec copie = function [] -> [] | x :: l -> x :: copie l
```

mais une telle fonction est complètement inutile car une liste ne peut être modifiée en place. Les fonctions de copie ne sont utiles que lorsque les structures de données sont susceptibles d'être modifiées en place.

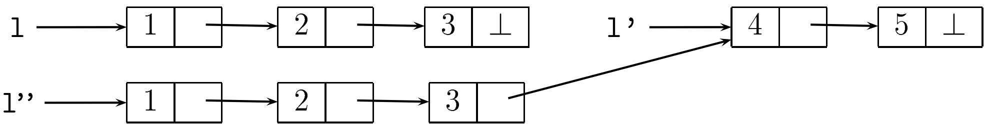
On comprend maintenant qu'il n'y a pas de possibilité d'ajouter un élément en queue de liste aussi facilement qu'en tête, car cela signifierait une modification en place de la liste `l` :



Pour rajouter un élément en queue de liste, il faut copier tous les blocs de la liste. C'est ce que fait en particulier la fonction `append` suivante qui construit la concaténation de deux listes :

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

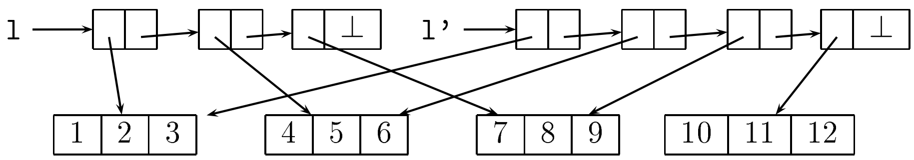
On constate que cette fonction recrée autant de blocs qu'il y en a dans `l1`, pour ne partager que ceux de `l2`. Ainsi, si l'on déclare `let l’ = [4 ; 5]` puis que l'on réalise la concaténation de `l` et de `l’` avec `let l’’ = append l l’`, on aura la situation suivante :



Les blocs de `l` ont été copiés et ceux de `l’` partagés.

Pour cette raison les listes doivent être utilisées lorsque les opérations naturelles sont l'ajout et le retrait en tête (structure de *pile*). Lorsque les accès et/ou modifications doivent se faire à des positions arbitraires, il vaut mieux préférer une autre structure de données.

Note importante : les *éléments* de la liste eux-mêmes, en revanche, ne sont pas copiés par la fonction `append`. En effet, `x` désigne un élément de type quelconque et aucune copie n'est effectuée sur `x` lui-même. Sur des listes d'entiers, ce n'était pas significatif. Mais si l'on a une liste `l` contenant trois éléments d'un type plus complexe, par exemple des triplets d'entiers comme dans `let l =[(1,2,3) ; (4,5,6) ; (7,8,9)]`, alors ceux-ci resteront partagés entre `l` et `append l [(10,11,12)]` :



Tout ceci peut paraître inutilement coûteux lorsque l'on a l'habitude d'utiliser des listes modifiées en place, ce qui est la manière traditionnelle de faire dans le contexte de langages impératifs. Mais ce serait sous-estimer l'intérêt pratique de la persistance. Il est d'ailleurs important de noter que le concept de persistance peut être facilement mis en œuvre dans un langage impératif : il suffit de manipuler les listes chaînées exactement comme le compilateur OCAML le fait. Inversement, on peut tout à fait manipuler des listes modifiables en place en OCAML, par exemple en définissant le type suivant :

```
type 'a liste = Vide | Element of 'a element
and 'a element = { valeur : 'a; mutable suivant : 'a liste };;
```

(On peut même rendre le champ `valeur` mutable si on le souhaite.) Mais à l'inverse des langages impératifs, OCAML offre la possibilité de définir des structures de données immuables de manière naturelle et sûre (car même si l'on code une structure persistante en C ou JAVA, le système de types ne peut empêcher sa modification en place, les variables étant par nature modifiables).

Enfin, il ne faut pas oublier que la mémoire inutilisée est automatiquement récupérée par le GC d'OCAML. Ainsi dans une expression telle que

```
let l = [1;2;3] in append l [4;5;6]
```

les trois blocs de `l` sont effectivement copiés lors de la construction de la liste `[1 ;2 ;3 ;4 ;5 ;6]` mais immédiatement récupérés par le GC car nulle part référencés.

3.2 Intérêts pratiques de la persistance

Les intérêts pratiques de la persistance sont multiples. De manière immédiate, on comprend que la persistance facilite la lecture du code et sa correction. En effet, on peut alors raisonner sur les valeurs manipulées par le programme en termes « mathématiques », puisqu'elles sont immuables, de manière équationnelle et sans même se soucier de l'ordre d'évaluation. Ainsi est-il facile de se persuader de la correction de la fonction `append` ci-dessus une fois que l'on a énoncé ce qu'elle est censé faire (i.e. `append l1 l2` construit la liste formée des éléments de `l1` suivis des éléments de `l2`) : une simple récurrence sur la structure de `l1` suffit. Avec des listes modifiables en place et une fonction `append` allant modifier le dernier pointeur de `l1` pour le faire pointer sur `l2` l'argument de correction est nettement plus difficile. L'exemple est encore plus flagrant avec le retournement d'une liste.

La correction d'un programme n'est pas un aspect négligeable et doit toujours l'emporter sur son efficacité : qui se soucie en effet d'un programme rapide mais incorrect ?

La persistance n'est pas seulement utile pour augmenter la correction des programmes, elle est également un outil puissant dans les contextes où le *backtracking* (retour en arrière) est nécessaire. Supposons par exemple que l'on écrive un programme cherchant la sortie dans un labyrinthe, sous la forme d'une fonction `cherche` prenant en argument un état, persistant, et retournant un booléen indiquant une recherche réussie. Les déplacements possibles à partir d'un état sont donnés sous forme d'une liste par une fonction `déplacements` et une autre fonction `déplace` calcule le résultat d'un déplacement à partir d'un état, sous la forme d'un nouvel état puisqu'il s'agit d'un type de données persistant.

On suppose qu'une fonction booléenne `sortie` indique si un état correspond à la sortie. On écrit alors trivialement la fonction `cherche` sous la forme suivante :

```
let rec cherche e =
  sortie e || essaye e (déplacements e)
and essaye e = function
  | [] -> false
  | d :: r -> cherche (déplace d e) || essaye e r
```

où `essaye` est une fonction testant un par un les déplacements possibles d'une liste de déplacements. C'est la persistance de la structure de données codant les états qui permet une telle concision de code. En effet, si l'état devait être modifié en place il faudrait effectuer le déplacement avant d'appeler récursivement `cherche` dans `essaye` mais aussi *annuler* ce déplacement en cas d'échec avant de passer aux autres déplacements possibles. Le code ressemblerait alors à quelque chose comme :

```
let rec cherche () =
  sortie () || essaye (déplacements ())
and essaye = function
  | [] -> false
  | d :: r -> (déplace d; cherche ()) || (revient d; essaye r)
```

ce qui est indubitablement moins clair et plus propice aux erreurs. Cet exemple n'est pas artificiel : le *backtracking* est une technique couramment utilisée en informatique (parcours de graphes, coloriage, dénombrement de solutions, etc.)

Un autre exemple typique est celui de la portée dans les manipulations symboliques de programmes (ou de formules). Supposons par exemple que l'on manipule des programmes Java triviaux composés uniquement de blocs, de variables locales entières, de tests d'égalité entre deux variables et de `return`, c'est-à-dire des programmes comme :

```
{
  int i = 0;
  int j = 1;
  if (i == j) {
    int k = 0;
    if (k == i) { return j; } else { int i = 2; return i; }
  } else {
    int k = 2;
    if (k == j) { int j = 3; return j; } else { return k; }
  }
}
```

De tels programmes peuvent être représentés en Ocaml par le type `instr list` où le type construit `instr` est défini par

```
type instr =
  | Return of string
  | Var of string * int
  | If of string * string * instr list * instr list
```

Supposons maintenant que l'on souhaite vérifier que les variables manipulées dans de tels programmes ont bien toujours été déclarées au préalable. Pour cela on peut écrire deux fonctions mutuellement récursives `vérifie_instr` et `vérifie_prog` vérifiant respectivement qu'une instruction et qu'une liste d'instructions sont bien formées. Pour cela ces deux fonctions prennent en argument la liste des variables dans la portée desquelles on se trouve (i.e. visibles). Le code est presque immédiat :

```
let rec vérifie_instr vars = function
  | Return x ->
      List.mem x vars
  | If (x, y, p1, p2) ->
      List.mem x vars && List.mem y vars &&
      vérifie_prog vars p1 && vérifie_prog vars p2
  | Var _ ->
      true
and vérifie_prog vars = function
  | [] ->
      true
  | Var (x, _) :: p ->
      vérifie_prog (x :: vars) p
  | i :: p ->
      vérifie_instr vars i && vérifie_prog vars p
```

La simplicité de ce code tient à l'utilisation d'une structure de données persistante pour l'ensemble des variables, à savoir une liste. Si l'on avait utilisé une table impérative pour cela, il aurait fallu revenir en arrière entre la première branche d'un `if` et la seconde, de manière à « oublier » les variables locales à la première branche. Le code n'aurait pas été aussi simple.

Cet exemple est relativement simple, mais on en trouve beaucoup d'instances en pratique, dès que l'on fait des manipulations symboliques et que des phénomènes de portées interviennent (table de symboles, environnement de typage, etc.). Il devient alors très agréable d'utiliser une structure de données persistante.

Donnons un dernier exemple de l'utilité de la persistance. Supposons un programme manipulant une base de données. Il n'y a qu'une seule instance de cette base à chaque instant et donc *a priori* il n'y a pas lieu d'utiliser une structure persistante pour cette base. Mais supposons que les mises à jour effectuées dans cette base soient complexes, i.e. impliquent chacune un grand nombre d'opérations dont certaines peuvent échouer. On se retrouve alors dans une situation difficile où il faut savoir *annuler* les effets du début de la mise à jour. Schématiquement, le code pourrait ressembler à ceci :

```
try
  ... effectuer l'opération de mise à jour ...
with e ->
  ... rétablir la base dans un état cohérent ...
  ... traiter ensuite l'erreur ...
```

Si l'on utilise une structure persistante pour la base de données, il suffit de stocker la base dans une référence, soit `bd`, et l'opération de mise à jour devient une mise à jour de cette référence :

```
let bd = ref (... base initiale ...)
...
try
  bd := (... opération de mise à jour de !bd ...)
with e ->
  ... traiter l'erreur ...
```

Dès lors, il n'y a pas lieu d'annuler quoi que ce soit. En effet, l'opération de mise à jour, si complexe qu'elle soit, ne fait que construire une nouvelle base de données et une fois seulement cette construction terminée, la référence `bd` est modifiée pour pointer sur cette nouvelle base. Cette toute dernière modification est atomique et ne peut échouer. S'il y a une quelconque exception levée pendant l'opération de mise à jour proprement dite, alors la référence `bd` restera inchangée.

3.3 Interface et persistance

Le type de données des listes est persistant d'une manière évidente, car c'est un type construit dont on connaît la définition, i.e. concret et immuable. Lorsqu'un module OCAML implante une structure de données sous la forme d'un type *abstrait*, son caractère persistant ou non n'est pas immédiat. Bien entendu, un commentaire approprié dans l'interface peut renseigner le programmeur sur cet état de fait. Mais en pratique ce sont les types des opérations qui fournissent cette information. Prenons l'exemple d'une structure de données persistante représentant des ensembles finis d'entiers. L'interface d'un tel module ressemblera à ceci :

```
type t                        (* le type abstrait des ensembles *)
val empty : t                 (* l'ensemble vide *)
val add : int -> t -> t       (* l'ajout d'un élément *)
val remove : int -> t -> t    (* la suppression d'un élément *)
...
```

Le caractère persistant des ensembles est implicite dans l'interface. En effet, l'opération `add` retourne un élément de type `t`, i.e. un nouvel ensemble ; de même pour la suppression. Plus subtilement, l'ensemble vide `empty` est une valeur et non une fonction, et toutes les occurrences de `empty` seront donc partagées quelque soit sa représentation.

En revanche une structure de données modifiable en place pour des ensembles d'entiers (par exemple sous la forme de tables de hachage) présentera une interface de la forme :

```
type t                           (* le type abstrait des ensembles *)
val create : unit -> t           (* 'ensemble vide *)
val add : int -> t -> unit       (* l'ajout d'un élément *)
val remove : int -> t -> unit    (* la suppression d'un élément *)
...
```

Ici la fonction d'ajout `add` ne retourne rien, car elle a ajouté l'élément en place dans la structure de données, et il en sera de même pour les autres opérations. D'autre part la fonction `create` prend un argument de type `unit` car chaque appel à `create` doit construire une nouvelle instance de la structure de données, afin que les modifications en place sur l'une n'affecte pas l'autre.

Malheureusement, rien n'empêche de donner à une structure impérative une interface « persistante » (en retournant la valeur passée en argument) et inversement de donner à une structure persistante une interface « impérative » (en jetant les valeurs que l'on vient de construire). Dans les deux cas, c'est stupide et dangereux.

Cela ne signifie pas pour autant qu'une structure de données persistante est nécessairement codée sans aucun effet de bord. La bonne définition de *persistant* est

$$\textit{persistant} = \textit{observationnellement immuable}$$

et non *purement fonctionnel* (au sens de l'absence d'effet de bord). On a seulement l'implication dans un sens :

$$\textit{purement fonctionnel} \Rightarrow \textit{persistant}$$

La réciproque est fausse, à savoir qu'il existe des structures de données persistantes faisant usage d'effets de bord. Un exemple classique est celui d'arbres binaires de recherche évoluant au fur et à mesure des recherches pour optimiser l'accès aux éléments les plus recherchés (*Splay trees*). Ceci peut être réalisé en codant la structure de données comme une référence sur un arbre, lui-même codé par une structure persistante. L'arbre optimisé peut alors être substitué à l'arbre courant par modification de cette référence, mais la structure complète reste persistante (pas de modification observable du contenu, mais seulement des performances).

Un autre exemple plus simple est celui des files (*first-in first-out* ou *queue* en anglais). Si l'on cherche à en réaliser une version persistante à l'aide de listes, les performances ne sont pas bonnes, car ajouter un élément en tête de liste est immédiat mais supprimer un élément en queue de liste ne l'est pas (le coût est proportionnel au nombre d'éléments en temps et en espace). Une idée astucieuse consiste à représenter la file par *deux* listes, l'une recevant les éléments en entrée et l'autre contenant les éléments prêts à sortir (donc rangés dans l'autre sens). Les opérations d'ajout et de retrait ont donc toujours un coût $O(1)$ *sauf* dans le cas particulier où la liste de sortie se trouve être vide ; on retourne alors la liste d'entrée pour l'utiliser comme nouvelle liste de sortie, la liste d'entrée devenant vide. Le coût de ce retournement est proportionnel à la longueur de la liste retournée, mais ceci ne sera fait qu'une fois pour l'ensemble des éléments de cette liste. La complexité *amortie* (i.e. ramenée à l'ensemble des opérations d'ajout et de retrait) reste donc $O(1)$.

Avec l'interface minimale suivante pour des files persistantes :

```
type 'a t
val create : unit -> 'a t
val push : 'a -> 'a t -> 'a t
exception Empty
val pop : 'a t -> 'a * 'a t
```

on peut proposer le code suivant basé sur l'idée ci-dessus :

```
type 'a t = 'a list * 'a list
let create () = [], []
```

```
let push x (e,s) = (x :: e, s)
exception Empty
let pop = function
  | e, x :: s -> x, (e,s)
  | e, [] -> match List.rev e with
    | x :: s -> x, ([], s)
    | [] -> raise Empty
```

Mais il se peut tout de même que l'on soit amené à retourner plusieurs fois la liste d'entrée `e`, si l'opération `pop` est effectuée plusieurs fois sur une même file de la forme `(e,[])`. Après tout, cela est susceptible d'arriver puisque l'on s'évertue à construire des files persistantes.

Pour remédier à cela, on va enregistrer le retournement de la liste d'entrée `e` par un effet de bord dans la structure de données. Cela n'en affectera pas le contenu — la file `(e,[])` contient exactement les mêmes éléments que la file `([],List.rev e)` — et évitera d'avoir à retourner la même liste la fois suivante. On remplace donc la paire de liste par un enregistrement composé de deux champs `entrée` et `sortie` tous deux `mutable` :

```
type 'a t = { mutable entrée : 'a list; mutable sortie : 'a list }
let create () = { entrée = []; sortie = [] }
let push x q = { entrée = x :: q.entrée; sortie = q.sortie }
exception Empty
```

et le code de la fonction `pop` est à peine plus complexe :

```
let pop q = match q.sortie with
    | x :: s ->
      x, { entrée = q.entrée; sortie = s }
    | [] -> match List.rev q.entrée with
      | [] ->
          raise Empty
      | x :: s as r ->
          q.entrée <- []; q.sortie <- r;
          x, { entrée = []; sortie = s }
```

La seule différence tient dans les deux modifications en place des champs de la file `q`. (Notez l'utilisation du mot clé `as` dans le filtrage qui permet de nommer l'ensemble de la valeur filtrée afin d'éviter de la reconstruire.)