Université de Marne-la-Vallée
Licence d'informatique 2004-2005

Cours d'Objective Caml

Marie-Pierre Béal

Transparents du cours, également disponibles sur :
`http://www-igm.univ-mlv.fr/~beal`

Objective Caml

1. Bibliographie
2. Principales caractéristiques
3. Première utilisation
4. Le noyau
5. Les bibliothèques
6. L'aspect impératif
7. Les flux
8. Modules et foncteurs
9. L'aspect objet

Bibliographie

- Thomas Ehrhard, Note de cours, Deug, Polycopié, Université de Marne-la-Vallée, 1993.
- Émmanuel Chailloux, Pascal Manoury et Bruno Pagano, *Développement d'applications avec Objective Caml*, Éditions O'Reilly, 2000.

Styles de programmation

Style applicatif

- Fondé sur l'évaluation d'expressions, où le résultat ne dépend que de la valeurs des arguments (et non de l'état de la mémoire).
- Donne des programmes courts, faciles à comprendre.
- Usage intensif de la récursivité.
- Langages typiques : Lisp, ML, Caml (Camlight, Ocaml).

Style impératif

- Fondé sur l'exécution d'instructions modifiant l'état de la mémoire.
- Utilise une structure de contrôle et des structures de données.
- Usage intensif de l'itération.
- Langages typiques : Fortran, C, Pascal.

Style objet

- Un programme est vu comme une communauté de composants autonomes (objets) disposant de ses ressources et de ses moyens d'interaction.
- Utilise des classes pour décrire les structures et leur comportement.
- Usage intensif de l'échange de message (métaphore).
- Langages typiques : Simula, Smalltalk, C++, Java.

Principales caractéristiques d'Objective Caml

Langage fonctionnel

- Une seule notion fondamentale : la notion de valeur. Les fonctions elles-mêmes sont considérées comme des valeurs.
- Un programme est essentiellement constitué d'expressions et le calcul consiste en l'évaluation de ces expressions.

Le système de typage

- Le langage est typé statiquement.
- Il possède un système d'inférence de types : le programmeur n'a besoin de donner aucune information de type : les types sont synthétisés automatiquement par le compilateur.

Le polymorphisme paramétrique

- Un type peut être non entièrement déterminé. Il est alors polymorphe.
- Ceci permet de développer un code générique utilisable pour différentes structures de données tant que la représentation exacte de cette structure n'a pas besoin d'être connue dans le code en question.

Mécanismes particuliers

- Objective Caml possède un mécanisme de programmation par cas : "pattern matching" très puissant.
- Objective Caml possède un mécanisme d'exceptions pour interrompre l'évaluation d'une expression. Ce mécanisme peut aussi être adopté comme style de programmation.

- Il y a des types récursifs mais pas de pointeurs.
- La gestion de la mémoire est différente de celle du C. Objective Caml intègre un mécanisme de récupération automatique de la mémoire.

Autres aspects de Objective Caml

- Objective Caml possède des aspects impératifs. La programmation impérative est possible mais non souhaitée.
- Objective Caml interagit avec le C.
- Objective Caml possède des aspects objets. Les fonctions peuvent manipuler des objets.
- Les programmes peuvent être structurés en modules paramétrés (séparation du code et de l'interface).
- Objective Caml exécute des processus léger (threads).
- Objective Caml communique avec le réseau internet.

Environnement de programmation

- Objective Caml dispose de trois modes d'exécution.
  - le mode interactif : c'est une boucle d'interaction. La boucle d'interaction d'Objective Caml utilise du code-octet pour exécuter les phrases qui lui sont envoyées.
  - compilation vers du code-octet (byte-code) interprété par une machine virtuelle (comme Java).
  - compilation vers du code machine exécuté directement par un micro-processeur.
- Objective Caml dispose de bibliothèques.

Modes de compilation

Noms des commandes

(détails dans le chapitre 7 du livre de référence)

- `ocaml` : lance la boucle d'interaction
- `ocamlrun` : interprète de code-octet
- `ocamlc` : compilateur en ligne de code-octet
- `ocamlopt` : compilateur en ligne de code natif
- `ocamlmktop` : constructeur de nouvelles boucles d'interaction

Unités de compilation

C'est la plus petite décomposition d'un programme pouvant être compilée.

- Pour la boucle d'interaction, c'est une phrase du langage. On va voir plus loin ce qu'est une phrase.
- Pour les compilateurs en ligne, c'est un couple de fichiers (le source `.ml` et l'interface facultative `.mli`).

Les fichiers objets en code-octet ont pour extension `.cmo`, `.cma` (bibliothèque objet), `.cmi` (interface compilée). Les fichiers objets en code natif ont les extensions `.cmx` et `.cmxa`.

L'option `-c` indique au compilateur de n'engendrer que le code objet. L'option `-custom` de `ocamlrun` permet de créer un code autonome il contient le code-octet du programme plus l'interprète du code-octet).

```
[Ocaml]$ cat essai.ml
let x = 3;;
print_int x;;
print_newline();;
[Ocaml]$ ocamlc essai.ml
[Ocaml]$ a.out
3
[Ocaml]$ ocamlc -o essai.exe essai.ml
[Ocaml]$ head -1 essai.exe
#!/usr/local/bin/ocamlrun
[Ocaml]$ ocamlc -help
---> donne la liste des options.
[Ocaml]$ ls
a.out  essai.cmi  essai.cmo  essai.exe  essai.ml
```

Boucle d'interaction

- L'option `-I catalogue` permet d'indiquer un chemin dans lequel se fera la recherche de fichiers sources ou compilés.
- La boucle possède plusieurs directives de compilation qui permettent de modifier interactivement son comportement. Ces directives commencent par `#` et se terminent par `;;`.
  - `#quit;;` sort de la boucle d'interaction
  - `#directory catalogue;;` ajoute un chemin de recherche
  - `#cd catalogue;;` change de catalogue courant
  - `#load "fichier_objet";;` charge un fichier `.cmo`
  - `#use "fichier_source";;` compile et charge un fichier `.ml` ou `.mli`
  - `#print_depth profondeur;;` modifie la profondeur d'affichage
  - `#trace fonction;;` trace les arguments d'une fonction
  - `#untrace fonction;;` enlève cette trace
  - `#untrace_all;;` enlève toute les traces

On peut lancer la boucle d'interaction sous emacs par `M-x run-caml`.

Première utilisation

```
[Ocaml]$ ocaml
        Objective Caml version 3.02
# 3;;
- : int = 3
# #quit;;                   --> sortie de la boucle
[Ocaml]$

[Ocaml]$ ocaml
        Objective Caml version 3.02
# let addition x = x+1;;
val addition : int -> int = <fun>
# addition 12;;
- : int = 13
# #quit;;
[Ocaml]$

[Ocaml]$ ocaml
        Objective Caml version 3.02
# #use "essai.ml";;
val x : int = 3
3- : unit = ()

- : unit = ()
# #load "essai.cmo";;
3
# #quit;;
```

```
[Ocaml]$ ocaml
        Objective Caml version 3.02
# let rec fib n =
  if n < 2 then 1
  else (fib (n-1) + fib (n-2));;
val fib : int -> int = <fun>
# #trace fib;;
fib is now traced.
# # fib 3;;
fib <-- 3
fib <-- 1
fib --> 1
fib <-- 2
fib <-- 0
fib --> 1
fib <-- 1
fib --> 1
fib --> 2
fib --> 3
- : int = 3
# #quit;;
```

```
[Ocaml]$ ocaml
        Objective Caml version 3.02
# let sum x y = x+y;;
val sum : int -> int -> int = <fun>
# #trace sum;;
sum is now traced.
# sum 2 3;;
sum <-- 2
sum --> <fun>
sum* <-- 3
sum* --> 5
- : int = 5
# #quit;;
```

Les phrases

Une phrase est une suite de caractères qui se termine par `;;`.
Il y a essentiellement quatre genres de phrases :

- Les expressions.
- Les déclarations globales de valeurs.
- Les déclarations de types.
- Les déclarations d'exceptions.

Les expressions

Expressions de type int

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

```
# -(12+2*5)/(14-7);;
- : int = -3
```

Expressions booléennes

Les opérateurs booléens sont

- `not` : négation
- `&&` ou `&` : et séquentiel
- `||` ou `or` : ou séquentiel

```
# not (true & false);;
- : bool = true
```

```
# if (true or false) then 2+4 else -1;;
- : int = 6
```

La construction `if` sert ici à écrire des *expressions*.

```
# true = false;;
- : bool = false
# 2+3 < 7-4;;
- : bool = false
# 2*3 >= 5;;
- : bool = true
```

Expressions chaînes de caractères

```
# "exemple";;
- : string = "exemple"
# "exe"^"mple";;
- : string = "exemple"
#
```

Il existe une bibliothèque permettant de manipuler les chaînes de caractères (le module `String`).

```
# String.uppercase "exemple";;
- : string = "EXEMPLE"
# open String;;   --> ouverture de la bibliothèque
# uppercase "exemple";;
- : string = "EXEMPLE"
```

Produits cartésiens, n-uplets

```
# (2,true,"toto");;
- : int * bool * string = 2, true, "toto"
```

Le type `unit`

On dispose également d'un type qui s'appelle `unit` et qui est un peu étrange : il existe une unique valeur du type `unit` qui est `()`. Cette valeur est utilisée dans les programmes impératifs.

```
# ();;
- : unit = ()
```

Les déclarations globales

Une déclaration associe un nom à une valeur. Dans une déclaration globale, un nom déclaré est connu de toutes les expressions qui suivent la déclaration. Dans une déclaration locale, les noms déclarés ne sont connus que d'une expression.

```
# let x = 3;;
val x : int = 3
# let x = 3 and y = 4;; -> déclarations simultanées
val x : int = 3
val y : int = 4
```

Attention En Objective Caml, une variable est un nom pour une valeur, alors qu'en C une variable est un nom pour une case mémoire. En Objective Caml, il est donc hors de question de pouvoir changer la valeur d'une variable. La déclaration globale (ou locale, voir plus loin) d'Objective Caml, n'a *rien à voir* avec l'affectation du C. Les variables Objective Caml et C sont des objets de natures complètement différentes.

```
# let x = 3;;
val x : int = 3
# let x = 6;;
val x : int = 6
# 3 * z;;
Unbound value z
# let z = 5 - 2;;
val z : int = 3
# 3 * z;;
- : int = 9
# x;;
- : int = 6
```

Expressions fonctionnelles

On peut écrire en Objective Caml une expression dont la “valeur” est cette fonction.

```
# function x -> 2*x+7;;
- : int -> int = <fun>
```

La variable `x` est un paramètre formel tandis que `function` est un mot clé d'Objective Caml.
Le type donné par Objective Caml pour cette expression est `int -> int`. (C'est-à-dire : fonction de `int` dans `int`). Il a été inféré automatiquement par le compilateur qui a “vu” que la variable `x` figurait comme argument d'une multiplication, et donc devait être du type `int` et que le résultat était une addition et devait être aussi du type `int`.

```
# (function x -> 2*x+7) 3;;
- : int = 13

# let f = function x -> 2*x+7;;
val f : int -> int = <fun>
# f 5;;
- : int = 17
# 3 + f (2+4);;
- : int = 22

# let g = function x -> (f x) + (f(2*x)) - x;;
val g : int -> int = <fun>
# g 2;;
- : int = 24
```

Voici un exemple qui illustre ce qui a été dit dans la section précédente.

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

Avec Objective Caml, les noms de variables et de fonctions doivent commencer par une minuscule. Les noms commençant par une majuscule sont réservés pour les constructeurs.

```
# let Fou = function x -> x+1;;

Unbound constructor Fou
# let foU  = function x -> x+1;;
val foU : int -> int = <fun>
```

Remarque sur l'égalité

- Le symbole `=` est le symbole d'égalité structurelle. Elle teste l'égalité de deux valeurs en explorant leur structure.
- L'égalité physique, représentée par le symbole `==`, teste si les deux valeurs occupent les mêmes zones mémoires.
- Les deux tests retournent les mêmes valeurs pour les valeurs simples (entiers, caractères, booléens, et constructeurs constants).

```
# "toto" = "toto";;
- : bool = true
# "toto" == "toto";;
- : bool = false
# let f = function x -> x+1;;
val f : int -> int = <fun>
# let g = function x -> x+1;;
val g : int -> int = <fun>
# f = g;;
Uncaught exception:
Invalid_argument "equal: functional value".
# f == g;;
- : bool = false
```

Fonctions plus compliquées

```
# let h = function x ->(x<=7) or ((x>15)&(x<>54));;
val h : int -> bool = <fun>

# let f = function (b,x) -> if not b then x else x*x;;
val f : bool * int -> int = <fun>

# let g = function (x,y,z) ->
((if x then y else 1), if y>10 then "bip" else z^"a");;
val g : bool * int * string -> int * string = <fun>
```

Fonction qui rend une fonction en résultat

```
# let f = function x -> (function y -> x+y);;
val f : int -> int -> int = <fun>
# let g = f 2;;
val g : int -> int = <fun>
# g 5;;
- : int = 7
# (f 3) 4;;
- : int = 7
# f 3 4;;
- : int = 7
```

Attention La fonction `f` est celle qui à un entier x associe la fonction qui à un entier y associe $x+y$. Elle est du type

`int -> (int -> int)`.

Par convention, on décide que le symbole `->` associe à droite, et on écrit ce type

`int -> int -> int`.

Attention le symbole `->` n'est pas associatif (pensez aux formules logiques du calcul propositionnel).
Par opposition, si `e`, `f`, `g` sont des expressions, l'expression

`(e f) g`

est notée par convention

`e f g`.

C'est seulement dans ce cas que les parenthèses peuvent être omises. L'application n'est pas une opération associative. Ainsi l'application d'une fonction à des arguments associe à gauche (`f 3 4` équivaut à `(f 3) 4`). Ces deux associations marchent de pair.

Fonctionnelles

Une fonction qui prend une fonction en argument est appelée une fonctionnelle.

```
# let g = function f -> (f 3) + 5;;
val g : (int -> int) -> int = <fun>
# g (function x -> x*(x-1));;
- : int = 11

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

```
# let g = function f ->
          (function x -> f(function y -> x+y));;
val g : ((int -> int) -> 'a) -> int -> 'a = <fun>
```

Une variante syntaxique

Comme le mot **function**, et aussi le symbole **->**, sont longs à écrire, Objective Caml propose une variante syntaxique pour les définitions globales (ou locales, voir plus loin), de la forme suivante. A la place de

$$\texttt{let}\ f = \texttt{function}\, p_1 \rightarrow \texttt{function}\, p_2 \rightarrow \ldots \texttt{function}\, p_n \rightarrow e,$$

on peut écrire

$$\texttt{let}\ f\ p_1\, p_2\ \ldots\ p_n = e.$$

Par exemple, au lieu de

```
# let f = function x-> 2*x + 7;;
val f : int -> int = <fun>
```

on peut écrire

```
# let f x = 2*x + 7;;
val f : int -> int = <fun>
```

Au lieu de

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

on peut écrire

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

Au lieu de

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

on peut écrire

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

Les fonctions de plusieurs arguments

La notion de fonction à plusieurs arguments n'est pas une notion primitive d'Objective Caml. Ce qui existe fondamentalement en Objective Caml ce sont des fonctions à *un seul argument* qui rendent *un seul résultat*. Les fonctions à plusieurs arguments sont représentées par des fonctions d'une forme particulière.

```
# let f1 = function (x,y,z) -> (x+y)*z;;
val f1 : int * int * int -> int = <fun>

# let f2 =
  function x -> function y -> function z -> (x+y)*z;;
val f2 : int -> int -> int -> int = <fun>

# f1 (2,1,3);;
- : int = 9
# f2 2 1 3;;
- : int = 9

# let f2 x y z = (x+y)*z;;
val f2 : int -> int -> int -> int = <fun>
```

On préfère `f2`, fonction sous une forme dite *curryfiée*, à `f1`.

Fonctions récursives

Les fonctions récursives doivent être définies à l'aide du mot clé `rec`.

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

```
Unbound value fact
```

```
# let rec fact n=if n=0 then 1 else n*fact(n-1);;
val fact : int -> int = <fun>
```

```
# let rec add(x,y) = if x=0 then y else add(x-1,y)+1;;
val add : int * int -> int = <fun>
```

Les récursions croisées ou simultanées sont possibles.

```
# let rec f n = if n=0 then 1 else g (n-1)
      and g n = if n=0 then 1 else f (n-1) + g (n-1);;
val f : int -> int = <fun>
val g : int -> int = <fun>
```

Les déclarations locales de valeurs

Les déclarations locales sont une nouvelle façon de construire des expressions. La syntaxe générale est la suivante.

$$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\ \texttt{;;}$$

où x est un identificateur et e et e' sont des expressions. L'expression construite est e'.

Insistons bien : à la différence des déclarations globales, les déclarations locales sont des expressions.

```
# let x = 2+3 in (let y = x*x in y-x+1);;
- : int = 21

# let f x = (x*x)+1 in (f 5)/(f 2);;
- : int = 5

# let rec f n = if n=0 then 1
                else n*f(n-1) in (f 3)+(f 4);;
- : int = 30

# let rec exp' n = if n=0 then 1
                   else (let x = exp'(n-1) in x+x);;
val exp' : int -> int = <fun>
```

Typage et polymorphisme

La synthèse des types se fait à la compilation. Le compilateur calcule le type “le plus général” qu’on puisse donner aux identificateurs et à l’expression elle-même qui soit compatible avec les différentes constructions syntaxiques utilisées. L’algorithme utilisé est un algorithme d’unification de termes.

```
# fun x y -> if x then y+1 else y*y;;
- : bool -> int -> int = <fun>
```

Dans l’exemple suivant, on voit que l’accès au premier champ d’un couple n’a pas à être différencié selon le type de la valeur de ce champ.

```
# let first = function (x,y) -> x;;
val first : 'a * 'b -> 'a = <fun>
# first (2,3);;
- : int = 2
# first ("toto",true);;
- : string = "toto"
```

Les fonctions dont l’un des paramètres ou la valeur de retour est d’un type qu’il n’est pas nécessaire de préciser sont dites polymorphes.

La fonction `first` est prédéfinie an Objective Caml sous le nom de `fst`.

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

# let g = function f -> f 0;;
```

```
val g : (int -> 'a) -> 'a = <fun>
# g (function x->x+1);;
- : int = 1
# g (function x->if x=0 then "toto" else "tata");;
- : string = "toto"
```

La fonction `compose` permet de composer deux fonctions.

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

Les limites du polymorphisme en Objective Caml

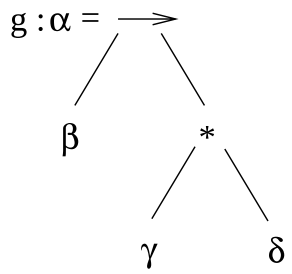
Le polymorphisme a cependant des limites. Considérons l'exemple suivant.

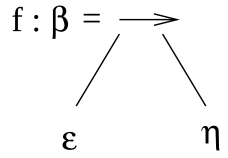
```
# let id x = x;;
val id : 'a -> 'a = <fun>

# ((id true),(id 2));;
- : bool * int = true, 2
```

Ici, la première occurrence de `id` a le type `bool -> bool` et la seconde a le type `int -> int`.
Cependant

```
# let g f = (f true),(f 2);;

This expression has type int
but is here used with type bool
```

Le fonctionnement du synthétiseur de types sur cet exemple est en effet le suivant.





$$\varepsilon = \text{bool}$$

$$\varepsilon = \text{int}$$

Les listes

Des valeurs d'un même type peuvent être regroupées en des listes. Le type `list` est en fait un schéma de type prédéfini. La plupart des fonctions de manipulation des listes sont définies dans la bibliothèque `List`.

La liste vide est une valeur non fonctionnelle qui est de type polymorphe. Objective Caml permet de générer des listes d'objets de même type sans que ce type soit précisé. Ce polymorphisme est appelé *paramétrique*.

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

```
# 1::[];;
- : int list = [1]
# [1;2] @ [3;4;5];;
- : int list = [1; 2; 3; 4; 5]
```

On note `[e1;...;en]` la liste `e1::(e2::...  (en::[])...)`.

```
# [[1;2];[];[6];[7;8;5;2]];;
- : int list list = [[1; 2]; []; [6]; [7; 8; 5; 2]]
```

La "déstructuration" des listes (extraction des éléments) se fait par "pattern-matching".

```
# let f = function
          [] -> 0
        | x::l -> x+1;;
val f : int list -> int = <fun>
# f [];;
```

```
- : int = 0
# f [3;2;7];;
- : int = 4
```

```
# let f = function
           [] -> 0
        |  []::l -> 1
        |  (x::l)::m -> x+1;;
val f : int list list -> int = <fun>
# f [ []; [1;2] ];;
- : int = 1
# f [ [3;2]; []; [1;4] ];;
- : int = 4
```

Fonctions récursives sur les listes

Dans l'exemple suivant on calcule la somme des éléments d'une liste d'entiers.

```
# let rec somme = function
        [] -> 0
  |  x::l -> x+somme l;;
val somme : int list -> int = <fun>
# somme [3;2;7];;
- : int = 12
```

Dans l'exemple suivant on calcule la conjonction des éléments d'une liste de booléens.

```
# let rec conj = function
        [] -> true
  |  x::l -> x & conj l;;
val conj : bool list -> bool = <fun>
# conj [true; false; true];;
- : bool = false
```

```
# let rec longueur = function
      [] -> 0
  | x::l -> 1 + longueur l;;
val longueur : 'a list -> int = <fun>
# longueur [true; false];;
- : int = 2
# longueur [[1];
```

On peut également utiliser la fonction `tl` (tail) de la librairie `List`

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

On peut enfin utiliser la fonction `length` de la bibliothèque `List`. Attention elle prend un temps proportionnel à la longueur de la liste.

```
# List.length [1;3;5];;
- : int = 3
# List.hd [1;3;5];;
- : int = 1
```

La fonctionnelle `map`

Considérons l'exemple suivant.

```
# let rec succl=function
      [] -> []
 | x::l -> (x+1)::succl l;;
val succl : int list -> int list = <fun>
# succl [2;1;5];;
- : int list = [3; 2; 6]
```

Voici maintenant une fonction qui à une liste d'entiers $[n_1; \ldots, n_k]$ associe la liste de booléens $[b_1; \ldots; b_k]$ telle que b_i soit `true` si et seulement si n_i est pair.

```
let rec pairl = function
      [] -> []
 | x::l -> ((x mod 2)=0)::pairl l;;
val pairl : int list -> bool list = <fun>
# pairl [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

La fonctionnelle `map` est définie dans la librairie `List`. Son type est

```
# List.map;;
- : f:('a -> 'b) -> 'a list -> 'b list = <fun>
```

et son effet est le suivant.

$$\texttt{List.map f}\,[e_1; \ldots; e_n] = [\texttt{f}\,e_1; \ldots; \texttt{f}\,e_n]$$

Cette fonction peut aussi s'écrire de la façon suivante.

```
# let rec map f = function
       [] -> []
  | x::l -> (f x)::map f l
```

```
;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

On peut alors réécrire les expressions vues plus haut en

```
# List.map (function x->x+1) [3; 2; 6];;
- : int list = [4; 3; 7]
# List.map (function x->(x mod 2)=0) [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

Exercice Synthétiser le type de `map`.

Fonctionnelles d'itérations sur les listes

Il y a en Objective Caml deux fonctionnelles d'itération `fold_left` et `fold_right` définies dans la librairie `List`. Ces deux fonctionnelles permettent l'écriture compacte d'une fonction de calcul sur les éléments d'une liste comme la somme des éléments.

Leurs effets sont les suivants.

$$\begin{aligned} \texttt{fold\_right f } [e_1;e_2;\ldots;e_n]\ a &= (\texttt{f}\, e_1\, (\texttt{f}\, e_2 \ldots (\texttt{f}\, e_n\, a)) \\ \texttt{fold\_left f } a\ [e_1;e_2;\ldots;e_n] &= (\texttt{f} \ldots (\texttt{f}\, (\texttt{f}\, a\, e_1)\, e_2) \ldots e_n) \end{aligned}$$

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

La fonction somme peut s'écrire

```
# let add x y = x+y;;
val add : int -> int -> int = <fun>
# let somme l = List.fold_right add l 0;;
val somme : int list -> int = <fun>
# somme [3;1;9];;
- : int = 13
```

```
# List.fold_right;;
- : f:('a ->'b ->'b) ->'a list->init:'b ->'b = <fun>
# List.fold_left;;
- : f:('a ->'b ->'a) ->init:'a ->'b list ->'a = <fun>
```

L'écriture directe de `fold_right` peut se faire de la façon suivante.

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

On peut également redéfinir la fonction `map`.

```
# let map f l =
 List.fold_right (fun x m -> (f x)::m) l [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

De même, la fonction qui concatène deux listes peut s'écrire :

```
# let append  l m  =
  List.fold_right (fun x n -> x::n) l m;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6;7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

Elle est prédéfinie dans la bibliothèque `List`.

```
# List.append [1;2;3] [4;5;6;7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

Quelques autres fonctions du module `List`

```
# List.rev;;
- : 'a list -> 'a list = <fun>
# List.mem;;
- : 'a -> 'a list -> bool = <fun>
# List.flatten;;
- : 'a list list -> 'a list = <fun>
# List.find;;
- : f:('a -> bool) -> 'a list -> 'a = <fun>
# List.split;;
- : ('a * 'b) list -> 'a list * 'b list = <fun>
# List.combine;;
- : 'a list -> 'b list -> ('a * 'b) list = <fun>
# List.assoc;;
- : 'a -> ('a * 'b) list -> 'b = <fun>
```

Exercice Avec leurs noms et types, essayez de deviner ce que font ces fonctions et écrivez leur code.

Compléments sur le “pattern matching”

Un exemple déjà vu.

```
# let rec fold_right f l a =
   match l with
     [] -> a
   | x::reste -> f x (fold_right f reste a);;
val fold_right:('a ->'b ->'b)->'a list->'b->'b =<fun>
# let somme l = fold_right (+) l 0;;
val somme : int list -> int = <fun>
# (+);;   --> plus préfixe
- : int -> int -> int = <fun>
```

Les différentes formes de motifs vues jusqu'ici sont les suivantes.

- Toute constante est un motif, étant entendu qu'une constante est soit un élément d'un type de base prédéfini (`bool`, `int`, `string`, ...), soit un constructeur d'arité 0, comme par exemple le constructeur `[]`).
- Toute variable est un motif.
- Si `C` est un constructeur (le constructeur binaire des listes `::`entre bien sûr dans cette catégorie même si, dans ce cas, la notation est infixe) d'arité $n \geq 1$ et si $p_1, \ldots, p_n$ sont des motifs, alors $p =$ `C` $(p_1, \ldots, p_n)$ est un motif
- Si $l_1, \ldots, l_k$ sont certains des noms de champs d'un type enregistrement et si $p_1, \ldots, p_k$ sont des motifs, alors $p = \{l_1 = p_1; \ldots; l_k = p_k\}$ est un motif.

La construction **match** a la signification suivante. Si e, e_1,...,e_n sont des expressions et si p_1,...,p_n sont des motifs, alors

$$\texttt{match}\, e\, \texttt{with}\, p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n$$

est une expression, qui est équivalente à

$$(\texttt{function}\, p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n)\, e$$

Inversement,

$$\texttt{function}\, p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n$$

peut s'écrire

$$\texttt{function x} \rightarrow (\texttt{match x with}\, p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n)$$

Pattern matching non exhaustif

```
# let f l = match l with
            [] -> 0;;

Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::_
val f : 'a list -> int = <fun>
# f [];;
- : int = 0
# f [1];;
Uncaught exception: Match_failure ("", 10, 35).

# let (x::l)=[1;2;3];;

Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x : int = 1
val l : int list = [2; 3]
```

Il existe un motif qui filtre n'importe quelle valeur : _ . La valeur filtrée est perdue après le filtrage.

```
# let rec meme_lg l1 l2 =
    match (l1,l2) with
    ([],[]) -> true
  | (_::reste1, _::reste2) ->
              meme_lg reste1 reste2
  | (_, _) -> false;;
val meme_lg : 'a list -> 'b list -> bool = <fun>
# meme_lg [1] ["toto"];;
- : bool = true
```

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

Autre écriture

```
# let nulle l = (l = []);;
val nulle : 'a list -> bool = <fun>
```

Un dernier point important sur les motifs : ils doivent être *linéaires*, c'est-à-dire qu'une même variable ne peut apparaître qu'au plus une fois dans un motif. Par exemple

```
# let eg = function (x,x) -> true | _ ->
false;;

This variable is bound several times in this matching
```

La construction `as` permet de nommer un sous-motif d'un motif.

```
# let f = function
    [] -> []
  | x::[] -> []
  | x::((y::m) as l) -> m@l;;
val f : 'a list -> 'a list = <fun>
```

Définitions de type

Les types énumérés

La définition d'un type se fait à l'aide du mot clé `type`. Les noms des constructeurs commencent par une majuscule. Un constructeur permet de construire les valeurs d'un type et d'accéder aux composantes de ces valeurs grâce au filtrage des motifs.

```
# type couleur = Bleu | Rouge | Vert;;
type couleur = Bleu | Rouge | Vert
# Vert;;
- : couleur = Vert
# let valeur = function
    Bleu  -> 1
  | Vert  -> 2
  | Rouge -> 3;;
val valeur : couleur -> int = <fun>
# valeur Vert;;
- : int = 2
```

La somme de deux types

La somme de deux types correspond à l'union disjointe.

```
# type int_ou_bool = I of int | B of bool;;
type int_ou_bool = I of int | B of bool

# I;;

The constructor I expects 1 argument(s),
but is here applied to 0 argument(s)
```

```
# I 4;;
- : int_ou_bool = I 4
# B true;;
- : int_ou_bool = B true
```

```
# B 5;;
```

```
This expression has type int
but is here used with type bool
```

L'utilisation des expression de ce type se fait par “pattern matching”.

```
# let f = function
   B x -> if x then 1 else 2
|  I x -> x+4;;
val f : int_ou_bool -> int = <fun>
# let f' = function
   B true -> 1
|  B false -> 2
|  I x -> x+4;;
val f' : int_ou_bool -> int = <fun>
```

```
# type string_ou_intlist =
        S of string | IL of int list;;
type string_ou_intlist = S of string | IL of int list
```

```
# type ('a,'b) somme = G of 'a | D of 'b;;
type ('a, 'b) somme = G of 'a | D of 'b
```

```
# G 1;;
- : (int, 'a) somme = G 1
# G true;;
- : (bool, 'a) somme = G true
# D [1; 2];;
- : ('a, int list) somme = D [1; 2]
```

```
# let gd = function G x -> true | D x -> false;;
val gd : ('a, 'b) somme -> bool = <fun>
# gd (G true);;
- : bool = true
# gd (D "hello");;
- : bool = false
```

```
#let f = function
      G x -> D x
    | D y -> G (y + 1);;
val f : ('a, int) somme -> (int, 'a) somme = <fun>
# f (G "hello");;
- : (int, string) somme = D "hello"
# f (D 3);;
- : (int, '_a) somme = G 4   --> problème au typage
```

Les types récursifs

On utilise les constructeurs pour définir des types récursifs.

```
# type entier = Zero | Succ of entier;;
type entier = Zero | Succ of entier

# let succ x = Succ x;;
val succ : entier -> entier = <fun>
# let pred = function
     Zero -> Zero
 | Succ x -> x;;
val pred : entier -> entier = <fun>

# let rec add x y =
    match x with
       Zero -> y
   | Succ z -> Succ (add z y);;
val add : entier -> entier -> entier = <fun>

# let rec entier_fold_right f e a =
   match e with
       Zero -> a
   | Succ x -> f (entier_fold_right f x a);;
val entier_fold_right:('a ->'a ->entier->'a->'a=<fun>
```

On peut redéfinir le type liste d'entiers avec

```
# type liste_entiers =
   Nil | Cons of entier * liste_entiers;;
type liste_entiers =
     Nil | Cons of entier * liste_entiers
```

Dans cet exemple, `Nil` est un constructeur d'arité 0 et `Cons` un constructeur d'arité 2.

```
# let rec somme = function
   Nil -> Zero
| Cons(x,l) -> add x (somme l);;
val somme : liste_entiers -> entier = <fun>
```

On peut également recréer le type polymorphe des listes en définissant le type paramétré suivant.

```
# type 'a liste =
   Nil | Cons of 'a * ('a liste);;
type 'a liste = Nil | Cons of 'a * 'a liste

# let rec liste_fold_right f l b =
     match l with
  Nil -> b
| Cons(x,reste) -> f x (liste_fold_right f reste b);;
# let rec liste_fold_right f l b =
     match l with
  Nil -> b
| Cons(x,reste) -> f x (liste_fold_right f reste b);;
val liste_fold_right :
    ('a -> 'b -> 'b) -> 'a liste -> 'b -> 'b = <fun>
```

On peut utiliser les types paramétrés pour créer par exemple des listes pouvant contenir des valeurs de types différents.

```
# type ('a,'b) liste =
   Nil
 | Acons of 'a * ('a,'b) liste
 | Bcons of 'b * ('a,'b) liste;;
type ('a, 'b) liste =
    Nil
  | Acons of 'a * ('a, 'b) liste
  | Bcons of 'b * ('a, 'b) liste
```

Définir des arbres binaires

La définition d'arbres binaires représentant des noeuds dont les étiquettes sont d'un type quelconque (mais toutes du même type) peut se faire par.

```
# type 'a arbre =
    Vide | Noeud of 'a * ('a arbre) * ('a arbre);;
type
'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
# Noeud (1, Noeud(2,Vide,Vide), Vide);;
- : int arbre = Noeud(1,Noeud(2,Vide,Vide),Vide)
```

Le constructeur `Vide` est un constructeur d'arité 0 et le constructeur `Noeud` est un constructeur d'arité 3.

```
# let rec somme = function
   Vide -> 0
 | Noeud(x,t1,t2) ->
     x + (somme t1) + (somme t2);;
val somme : int arbre -> int = <fun>

# let rec nbe = function
   Vide -> 0
 | Noeud(x,t1,t2) ->
       1 + (nbe t1) + (nbe t2);;
val nbe : 'a arbre -> int = <fun>
```

La fonction `flat` suivante donne la projection verticale d'un arbre ou l'ordre interne ou encore infixe.

```
# let rec flat = function
  Vide -> []
| Noeud(x,t1,t2) -> (flat t1)@(x::flat t2);;
val flat : 'a arbre -> 'a list = <fun>
```

On peut écrire un `flat` de meilleure complexité à l'aide d'une variable appelée accumulateur.

```
# let rec flat_accu t accu = match t with
   Vide -> accu
 | Noeud(x,t1,t2) ->
       flat_accu t1  (x::flat_accu t2 accu);;
val flat_accu : 'a arbre -> 'a list -> 'a list = <fun>
# let flat t = flat_accu t [];;
val flat : 'a arbre -> 'a list = <fun>
```

Le `flat` se fait alors en temps linéaire.

Fonctions d'itération sur les arbres

Elles sont semblables aux fonctionnelles d'itérations sur les listes mais ne sont pas prédéfinies. La fonction suivante est l'équivalent d'un `fold_right`.

```
# let rec arbre_it f t b =
 match t with
   Vide -> b
 | Noeud(x,t1,t2) ->
     f x (arbre_it f t1 b) (arbre_it f t2 b);;
val arbre_it :
 ('a ->'b ->'b ->'b) ->'a arbre ->'b ->'b=<fun>
```

Exemples d'utilisation de la fonctionnelle d'itération.

```
# let nbe t = arbre_it (fun x n1 n2 -> 1+n1+n2) t 0;;
val nbe : 'a arbre -> int = <fun>
# let ht t =
   arbre_it (fun x h1 h2 -> 1+(max h1 h2)) t 0;;
val ht : 'a arbre -> int = <fun>
# let flat t =
   arbre_it (fun x l1 l2 -> l1@(x::l2)) t [];;
val flat : 'a arbre -> 'a list = <fun>
```

Syntaxes abstraites

On définir des expressions arithmétiques, des formules logiques, ..., de la façon suivante.

```
# type expr = Const of int
            | Plus of expr * expr
            | Mult of expr * expr
            | Moins of expr * expr
            | Div of expr * expr
            | Opp of expr;;
type expr =
    Const of int
  | Plus of expr * expr
  | Mult of expr * expr
  | Moins of expr * expr
  | Div of expr * expr
  | Opp of expr
```

```
# let rec eval = function
    Const n -> n
 |  Plus (e1,e2) -> (eval e1)+(eval e2)
 |  Mult (e1,e2) -> (eval e1)*(eval e2)
 |  Moins(e1,e2) -> (eval e1)-(eval e2)
 |  Div (e1,e2)  -> (eval e1)/(eval e2)
 |  Opp e -> -(eval e);;
val eval : expr -> int = <fun>
```

Les enregistrements

Ce sont des n-uplets dont les champs sont nommés comme les **struct** du C. Les noms des champs doivent commencer par une lettre minuscule.

```
# type int_et_bool = { i : int; b : bool };;
type int_et_bool = { i : int; b : bool; }
#{i=3; b=true};;
- : int_et_bool = {i=3; b=true}
#{b=false; i=7};;
- : int_et_bool = {i=7; b=false}

# let f = function
   {b=true; i=x} -> x*x
 | {b=false; i=x} -> x+x;;
val f : int_et_bool -> int = <fun>

# let f c = let x=c.i in
      if c.b then x*x else x+x;;
val f : int_et_bool -> int = <fun>

# let g = function {i=x} -> x;;
val g : int_et_bool -> int = <fun>

# type ('a,'b) couple = { fst :'a; snd :'b };;
type ('a, 'b) couple = { fst : 'a; snd : 'b; }

# type 'a arbre = Vide | Noeud of 'a noeud
  and  'a noeud =
      { valeur :'a; d:'a arbre; g:'a arbre };;
type 'a arbre = Vide | Noeud of 'a noeud
```

```
type 'a noeud =
     { valeur : 'a; d : 'a arbre; g : 'a arbre; }
```

```
# let rec somme = function
   Vide -> 0
 | Noeud {d=t2; valeur=x; g=t1} ->
         x + (somme t1) + (somme t2);;
val somme : int arbre -> int = <fun>
```

```
# let rec somme = function
   Vide -> 0
 | Noeud n ->
     n.valeur + (somme n.g) + (somme n.g);;
val somme : int arbre -> int = <fun>
```

Les exceptions

Nous allons décrire le mécanisme d'exception d'Objective Caml.

```
# let tete = function
      [] -> failwith "tete"
    | x::_ -> x;;
val tete : 'a list -> 'a = <fun>
# failwith;;
- : string -> 'a = <fun>
```

```
# tete [];;
Uncaught exception: Failure "tete".
```

En Objective Caml, les exceptions appartiennent à un type prédéfini `exn`. Ce type a une particularité. C'est un type somme qui est extensible en ce sens que l'on peut étendre l'ensemble de ses valeurs en déclarant de nouveaux constructeurs. On définit comme cela de nouvelles exceptions. La syntaxe pour définir de nouvelles exceptions est

```
# exception Erreur;;
exception Erreur
# Erreur;;
- : exn = Erreur
```

Le constructeur `Failure` est un constructeur d'exception.

```
# Failure "hello !";;
- : exn = Failure "hello !"
```

La levée d'une exception se fait à l'aide de la fonction `raise`. Remarquez son type.

```
# let tete = function
      [] -> raise (Failure "tete")
  | x::_ -> x;;
val tete : 'a list -> 'a = <fun>
# raise;;
- : exn -> 'a = <fun>
# let tete = function
      [] -> raise Erreur
  | x::_ -> x;;
val tete : 'a list -> 'a = <fun>
```

```
# exception Trouve of int;;
exception Trouve of int
# Trouve 5;;
- : exn = Trouve 5
# raise (Trouve 7);;
Uncaught exception: Trouve 7.
```

```
# exception Erreur_fatale of string;;
exception Erreur_fatale of string
# raise (Erreur_fatale "cas non prevu");;
Uncaught exception: Erreur_fatale "cas non prevu".
```

La construction try ... with

En cas de levée d'une exception, la capture de l'exception permet de continuer un calcul. On peut utiliser les captures d'exception comme un véritable style de programmation. La construction `try` ... `with` permet de le réaliser les captures. Les différentes étapes à l'exécution sont les suivantes.

- Essayer de calculer l'expression.
- Si aucune erreur n'est déclenchée, on retourne l'expression.
- Si cette évaluation déclenche une erreur qui tombe dans un cas de filtrage, alors on retourne la valeur correspondante de la clause sélectionnée par le filtrage.
- En cas d'échec sur le filtrage, la valeur exceptionnelle est propagée.

Dans ce qui suit, `p` est un prédicat de type `'a -> bool`.

```
# let rec ch p n = function
   [] -> raise Erreur
|x::l -> if (p x) then
         raise (Trouve n) else ch p (n+1) l;;
val ch : ('a -> bool) -> int -> 'a list -> 'b = <fun>

# let cherche p l = try (ch p 1 l) with
  Erreur -> raise (Failure "cherche: rien trouve")
| Trouve n -> n;;
val cherche : ('a -> bool) -> 'a list -> int = <fun>
# cherche pair [1;5;3;4;6;7];;
- : int = 4
# cherche pair [1;5;3];;
Uncaught exception: Failure "cherche: rien trouve".
```

Les aspects impératifs d'Objective Caml

(à utiliser avec modération)

Les enregistrements à champs modifiables

Les champs d'un enregistrement peuvent être déclarés modifiables à l'aide du mot clé `mutable`.

```
# type ex={ a:int; mutable b:(bool*int);
            mutable c:int->int };;
type ex = { a : int; mutable b : bool * int;
            mutable c : int -> int; }

# let r = {b=(false, 7); a=1; c=function x->2*x+1};;
val r : ex = {a=1; b=false, 7; c=<fun>}
# r.b;;
- : bool * int = false, 7
# r.c 3;;
- : int = 7

# let (x,y) = r.b in r.b<-(true, y+7);;
- : unit = ()
# r;;
- : ex = {a=1; b=true, 14; c=<fun>}
# r.c <- (function x->x*x);;
- : unit = ()
# r.c 3;;
- : int = 9
```

Les expressions peuvent être séquencées. Si $e_1, e_2, \ldots, e_n$ sont des expressions, alors $e = e_1; e_2; \ldots; e_n$ est une expression de type, le type de e_n, et de valeur, la valeur de e_n.

```
# type compteur = { mutable cpt:int };;
type compteur = { mutable cpt : int; }
# let incr = let c = { cpt=0 } in
    function () -> c.cpt <- c.cpt+1; c.cpt;;
val incr : unit -> int = <fun>
# incr ();;
- : int = 1
# incr ();;
- : int = 2
```

Les références

Objective Caml fournit un type polymorphe **ref** qui peut être vu comme le type des pointeurs sur une valeur quelconque. On parle de référence sur une valeur. Le type **ref** est défini comme un enregistrement à un champ modifiable.

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

Ce type est muni de raccourcis syntaxiques prédéfinis.

```
# let compteur = ref 0;;
val compteur : int ref = {contents=0}
# !compteur;;
- : int = 0
# compteur := 2;;
- : unit = ()
# compteur := !compteur +1;;
- : unit = ()
# !compteur;;
- : int = 3

# let incr c = c := !c + 1;;
val incr : int ref -> unit = <fun>
# incr compteur; !compteur;;
```

```
- : int = 4
```

Les boucles

Les différents types de boucles sont

- `while` expression `do` expression `done`
- `for` ident=expression `to|downto` expression
- `do` expression `done`

```
# let imprime_chiffres () =
   for i=0 to 9 do
       print_int i ;
       print_string " "
   done;
   print_newline ();;
val imprime_chiffres : unit -> unit = <fun>
# imprime_chiffres ();;
0 1 2 3 4 5 6 7 8 9
- : unit = ()
```

Les vecteurs

Les vecteurs sont des tableaux à une dimension. Beaucoup de fonctions de manipulations sont définies dans la librairie `Array`. Les indices commencent à 0.

```
# [|1;2;3|];;
- : int array = [|1; 2; 3|]
# let v = Array.create 4 3.14;;
val v : float array = [|3.14; 3.14; 3.14; 3.14|]
# v.(0);;
- : float = 3.14
# v.(4);;
Uncaught exception: Invalid_argument "Array.get".
#  v.(4) <- 5.;;
Uncaught exception: Invalid_argument "Array.set".
#  v.(3) <- 8.;;
- : unit = ()
# v;;
- : float array = [|3.14; 3.14; 3.14; 8|]
# Array.length v;;
- : int = 4
#  let w = Array.copy v;;
val w : float array = [|3.14; 3.14; 3.14; 8|]
# v = w;;
- : bool = true
# v == w;;
- : bool = false
# Array.of_list;;
- : 'a list -> 'a array = <fun>
```

```
# Array.of_list [3;2;8];;
- : int array = [|3; 2; 8|]

# let invt t=
   let n = Array.length t in
    (let rec f i =
       if i=n/2  then ()
       else let x=t.(i) in
              (t.(i)<-t.(n-i-1);
               t.(n-i-1)<-x;
               f(i+1)
               )
     in f 0
    );;
val invt : 'a array -> unit = <fun>

# let t=[|8; 10; 2; 1|];;
val t : int array = [|8; 10; 2; 1|]
# invt t;;
- : unit = ()
# t;;
- : int array = [|1; 2; 10; 8|]
```

Des tableaux à deux dimensions peuvent être créés de la façon suivante.

```
# Array.create_matrix;;
- : dimx:int -> dimy:int ->'a ->'a array array=<fun>
# Array.create_matrix 2 3 "a";;
# let m = Array.create_matrix 2 3 "a";;
val m : string array array =
        [|[|"a"; "a"; "a"|]; [|"a"; "a"; "a"|]|]
#  m.(1).(2);;
- : string = "a"
#  m.(1).(2) <- "toutou";;
- : unit = ()
# m;;
- : string array array =
    [|[|"a"; "a"; "a"|]; [|"a"; "a"; "toutou"|]|]
# Array.length m;;
- : int = 2
# Array.length m.(0);;
- : int = 3
```

Pour d'autres fonctions de la librairie `Array`, se reporter au chapitre 8 du livre de référence.

Les flux

Les flux ou flots permettent de faire de l'analyse lexicale. Les principales fonctions de manipulation des flots sont définies dans la bibliothèque `Stream`. Des outils d'analyse lexicale sont donnés dans le module `Genlex`. Il permet également une analyse syntaxique simple. Des outils plus puissants sont fournis avec `ocamllex` et `ocamlyacc`.

On peut définir des flots de valeurs quelconques. Le type des flots est défini dans la bibliothèque `Stream` par

```
type 'a t
```

Il permet de créér des flots de type `'a`.

Jusqu'à la version 3.03, les flux sont directement accessibles dans la boucle d'interaction. Ensuite, on charge la bibliothèque Camlp4 (voir documentation sur le site Caml de l'INRIA) par

```
# #load "camlp4o.cma";;
      Camlp4 Parsing version 3.07+2

# [<'1>];;
- : int Stream.t = <abstr>
# Stream.of_string "1.0 + x";;
- : char Stream.t = <abstr>
# let s = Stream.of_string "1.0 + x";;
val s : char Stream.t = <abstr>
# Stream.next s;;
- : char = '1'
# Stream.next s;;
- : char = '.'
```

Plus curieux :

```
# let s = [<'1;'2;'3>];;
val s : int Stream.t = <abstr>
# let t = [<'4;'5>];;
val t : int Stream.t = <abstr>
# let u = [<s;t>];;
val u : int Stream.t = <abstr>
# Stream.next u;;
- : int = 1
# u;;
- : int Stream.t = <abstr>
# [<>];;
- : 'a Stream.t = <abstr>
```

Le module `Genlex`

Ce module permet de manipuler plusieurs catégories d'unités lexicales prédéfinies données dans le type

```
type token =
          Kwd of string
        | Ident of string
        | Int of int
        | Float of float
        | String of string
        | Char of char ;;
```

Le constructeur `Ident` désigne la catégorie des identificateurs (ne commençant pas par un chiffre). On considère aussi comme identificateur toute suite de symboles d'opérations ou de relations. Le constructeur `Kwd` définit la catégorie des mots clés qui contient des identificateurs distingués ou des caractères spéciaux. La seule catégorie paramétrable est celle des mots clés.

La fonction `make_lexer` construit un analyseur lexical qui prend en paramètre un ensemble de mots clés. Son type est

```
Genlex.make_lexer ;;
- :
string list -> char Stream.t -> Genlex.token Stream.t
```

Elle s'utilise de la façon suivante.

```
# let mots_cles = [ "IF"; "LET"; "+"; "*"; ];;
val mots_cles : string list = ["IF"; "LET"; "+"; "*"]
# let mon_lexer l =
  Genlex.make_lexer mots_cles (Stream.of_string l);;
val mon_lexer : string -> Genlex.token Stream.t = <fun>
# let s = mon_lexer "LET x = y * 3";;
val s : Genlex.token Stream.t = <abstr>
# Stream.next s;;
- : Genlex.token = Kwd "LET"
# Stream.next s;;
- : Genlex.token = Ident "x"
# Stream.next s;;
- : Genlex.token = Ident "="
# Stream.next s;;
- : Genlex.token = Ident "y"
# Stream.next s;;
- : Genlex.token = Kwd "*"
#  Stream.next s;;
- : Genlex.token = Int 3
```

On a ainsi construit un flux de “token” qui peuvent ensuite être traités syntaxiquement.

Exemple de traitement lexical : élimination des espaces, tabulations et sauts de ligne.

```
# let rec elimine_espaces s =
   match s with parser
        [<'' ' ; reste >] -> elimine_espaces reste
     |  [<''\t' ; reste >] -> elimine_espaces reste
     |  [<''\n' ; reste >] -> elimine_espaces reste
     |  [<>]  ->()
;;
val elimine_espaces : char Stream.t -> unit = <fun>
```

Exemple d'analyse syntaxique

```
# open Genlex;;
# let mon_lexer = make_lexer ["+";"-"];;
val mon_lexer :
char Stream.t -> Genlex.token Stream.t = <fun>
# let s = mon_lexer (Stream.of_string "+ 1.0  x");;
val s : Genlex.token Stream.t = <abstr>
# Stream.next s;;
- : Genlex.token = Kwd "+"
# Stream.next s;;
- : Genlex.token = Float 1
# Stream.next s;;
- : Genlex.token = Ident "x"
```

On définit maintenant un type pour des expressions arithmétiques.

```
# type expr = Const of float
            | Var of string
            | Plus of expr * expr
            | Moins of expr * expr
;;
```

La construction de l'expression à partir de la chaîne postfixée se fait simplement par

```
let rec mon_parser s =
   match s with parser
 [<'Kwd "+";e1=mon_parser;e2=mon_parser>]-> Plus(e1,e2)
|[<'Kwd "-";e1=mon_parser;e2=mon_parser>]-> Moins(e1,e2)
|[<'Ident x>] -> Var x
|[<'Int i>] -> Const(float i)
|[<'Float f>] -> Const f
;;
val mon_parser : Genlex.token Stream.t -> expr = <fun>
# let s = mon_lexer (Stream.of_string "+ 1.0  x");;
val s : Genlex.token Stream.t = <abstr>
# mon_parser s;;
- : expr = Plus (Const 1., Var "x")
# let s =  mon_lexer (Stream.of_string "+ 1.0  - x y");;
val s : Genlex.token Stream.t = <abstr>
# mon_parser s;;
- : expr = Plus (Const 1., Moins (Var "x", Var "y"))
```

La syntaxe est particulière : l'appel interne `e1=mon_parser` est un appel récursif qui traite le flux de l'intérieur est où `e_1` est le nom de la valeur renvoyée.

Autre exemple : mise sous forme clausale d'une formule

On calcule un ensemble de clauses sémantiquement équivalent à une formule du calcul propositionnel. Les clauses sont représentées par des listes de formules et les ensembles de clauses par des listes de listes de formules.

Le type des formules est :

```
(* Mise sous forme clausale d'une formule *)
(* Version sans accumulateurs *)
type binconn = Et  | Ou | Impl | Equiv;;

type formule =
| Const of bool                          (* valeur constante *)
| Var of string                                  (* variable *)
| Bin of binconn * formule * formule (* con. binaire *)
| Non of formule                               (* con. unaire *)
;;

let f1 = Bin(Impl,Var "a",Var "a");;

(* pour manipuler les formules sans Impl, Equiv et  *)
(* et Const                                         *)
exception BadConnectorException;;
```

On renvoie à la partie logique du cours pour la description de l'algorithme. La descente des négations se fait par :

```
(* fait descendre les negations *)
let rec descente f = match f with
   Const _ -> f
|  Var _ -> f
|  Bin(co,f1,f2) -> Bin(co,descente f1, descente f2)
```

```
|  Non (Non f1) -> descente f1
|  Non (Var _) -> f
|  Non (Bin(Et,f1,f2)) ->
      Bin(Ou,descente (Non f1), descente (Non f2))
|  Non (Bin(Ou,f1,f2)) ->
      Bin(Et,descente (Non f1), descente (Non f2))
|  _  -> raise BadConnectorException
;;
```

La mise sous forme de clause est ensuite :

```
(* ajoute une liste en tête de chaque liste          *)
(* d'une liste de listes                             *)
let addHeadListe l ll =
     List.map (function l1 -> l@l1) ll;;
(* calcule le ``ou'' de deux listes de clauses       *)
let rec computeOr lc1 lc2 = match (lc1,lc2) with
|  ([],_)  -> [] (* [] <-> liste vide, tautologie *)
|  (_,[])  -> []
(* simplification ici,                            *)
(* [[],...] <-> présence de la clause vide        *)
|  ([]::_,_) -> lc2
|  (_,[]::_) -> lc1
|  (c1::reste1,_) ->
        (addHeadListe c1 lc2)@(computeOr reste1 lc2)
;;
(* on peut l'améliorer en testant les répétitions  *)

(* exception si la formule comporte des négations *)
(* non descendues                                 *)
exception FormuleException;;
```

```
(* mise sous forme clausale d'une formule f    *)
(*   [] pas de clauses, implique f tautologie   *)
(*   [[]] clause vide, implique f antilogie     *)
let rec clauses f = match f with
   Const b -> if (b) then [] else [[]]
|  Var _  -> [[f]]
|  Bin(Et,f1,f2) -> (clauses f1)@(clauses f2)
|  Bin(Ou,f1,f2) -> computeOr (clauses f1) (clauses f2)
|  Non (Var _)  -> [[f]]
|  Non _ -> raise FormuleException
            (* f a des negations non descendues *)
|  _ ->  raise BadConnectorException
;;

let make_clauses f = clauses (descente f);;
(* test sur un exemple *)
let f = Non ( Bin(Et, Bin(Ou, Var "p", Var "q"),
        Var "r"));;
make_clauses f;;
(* [[Non (Var "p"); Non (Var "r")];           *)
(* [Non (Var "q"); Non (Var "r")]]            *)
```

Les entrées-sorties sont :

```
(* entree-sortie a l'aide de Genlex           *)
open Genlex;;
let mon_lexer s =
make_lexer ["non";"et";"ou";"(";")"](Stream.of_string s);;

(* entrée des formules sous formes de chaînes
de caracteres, avec une forme préfixe *)
let rec mon_parser st = match st with parser
```

```
    [<'Kwd "("; e1 = mon_parser; 'Kwd ")" >]  -> e1
 |  [<'Kwd "non"; e1 = mon_parser >]  -> Non e1
 |  [<'Kwd "ou"; e1 = mon_parser; e2 = mon_parser >]
             -> Bin(Ou,e1,e2)
 |  [<'Kwd "et"; e1 = mon_parser; e2 = mon_parser >]
             -> Bin(Et,e1,e2)
 |  [<'Ident x>]  -> Var x
;;
let traite s = mon_parser (mon_lexer s);;
traite "non(et (ou p q) r)";;
```

Par exemple,

```
(* un exemple *)
# make_clauses (traite "non(et (ou p q) r)");;
- : formule list list =
[[Non (Var "p"); Non (Var "r")];
 [Non (Var "q"); Non (Var "r")]]
```

qui représente deux clauses : $(\neg p \vee \neg r)$ et $(\neg q \vee \neg r)$. Les clauses ne sont pas simplifiées.

```
# #use "clause.ml";;
type binconn = Et | Ou | Impl | Equiv
type formule =
    Const of bool
  | Var of string
  | Bin of binconn * formule * formule
  | Non of formule
val f1 : formule = Bin (Impl, Var "a", Var "a")
exception BadConnectorException
val descente : formule -> formule = <fun>
val addHeadListe :
'a list -> 'a list list -> 'a list list = <fun>
```

```
val computeOr :
'a list list -> 'a list list -> 'a list list = <fun>
exception FormuleException
val clauses : formule -> formule list list = <fun>
val make_clauses : formule -> formule list list = <fun>
val f : formule =
Non (Bin (Et, Bin (Ou, Var "p", Var "q"), Var "r"))
- : formule list list =
[[Non (Var "p"); Non (Var "r")];
[Non (Var "q"); Non (Var "r")]]
val mon_lexer : string -> Genlex.token Stream.t = <fun>
val mon_parser : Genlex.token Stream.t -> formule = <fun>
val traite : string -> formule = <fun>
- : formule =
Non (Bin (Et, Bin (Ou, Var "p", Var "q"), Var "r"))
        Camlp4 Parsing version 3.08.3
```

Modules et foncteurs

Les modules permettent l'encapsulation. Les modules peuvent implémenter des interfaces (sources suffixées par `.mli`) qui définissent ainsi un moule pour les modules.

Exemple : une interface et un module pile

Pour une interface, l'usage est de mettre le nom du type en majuscule. Une interface déclare un type, des exceptions et donne le type des fonctions.

```
module type PILE =
sig
  type 'a t
  exception Empty
  val create: unit -> 'a t
  val push: 'a -> 'a t -> unit
  val pop: 'a t -> 'a
  val top: 'a t -> 'a
  val clear : 'a t -> unit
  val length: 'a t -> int
  val iter: ('a -> unit) -> 'a t -> unit
end;;
```

Le module suivant définit au moins tous les types, exceptions et fonctions de l'interface `PILE`. Le nom d'un module commence par une lettre majuscule obligatoirement. Les noms des fichiers sont libres.

```
module Pile_liste =
struct
  type 'a t = { mutable c : 'a list }
```

```
    exception Empty
    let create () = { c = [] }
    let clear p = p.c <- []
    let push x p = p.c <- x::p.c
    let pop p = match p.c with
       [] -> raise Empty
       |x::reste -> p.c <- reste; x

    let top p = match p.c with
       [] -> raise Empty
       |x::_ -> x
    let length p = List.length p.c
    let iter f p = List.iter f p.c
    let inutile () = true
  end;;
```

Pour forcer le typage du module `Pile_liste` à `PILE` on ajoute

```
module Pile_liste = (Pile_liste : PILE);;
```

L'effet pour l'utilisateur du module est de ne pouvoir utiliser que les fonctions déclarées dans l'interface. C'est un peu comme si la fonction `let inutile () = true` était alors une méthode privée (statique) de Java.

```
# #use "pile.mli";;
module type PILE =
  sig
    type 'a t
    exception Empty
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val top : 'a t -> 'a
```

```
    val clear : 'a t -> unit
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end
# #use "pile.ml";;
module Pile_liste :
  sig
    type 'a t = { mutable c : 'a list; }
    exception Empty
    val create : unit -> 'a t
    val clear : 'a t -> unit
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val top : 'a t -> 'a
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
    val inutile : unit -> bool
  end
module Pile_liste : PILE
# let p = Pile_liste.create();;
val p : '_a Pile_liste.t = <abstr>
# Pile_liste.push 2 p;;
- : unit = ()
#  Pile_liste.push 3 p;;
- : unit = ()
#  Pile_liste.length p;;
- : int = 2
# Pile_liste.iter (fun x -> Printf.printf "%d " x) p;;
3 2 - : unit = ()
# Pile_liste.inutile ();;
Unbound value Pile_liste.inutile
```

Les foncteurs sont des modules qui permettent de construire un module à partir d'un autre module. Ils sont donc aux modules ce que les fonctions sont aux valeurs.

Les bibliothèques Map, Set, Hashtbl contiennent par exemple des foncteurs. Le foncteur `Map.Make` permet de construire une table (l'équivalent d'une map Java) où les clés sont définies dans un module contenant une fonction compare (l'équivalent d'une méthode compare de Java). La table contient des entrées (clés, valeurs). Le synthétiseur de type détermine le type des valeurs qui peut être polymorphe.

Exemples de modules définissant des clés

Des entiers :

```
module ComparableInt  =
    struct
     type t =  int
     (* analogue du compareTo de java *)
     let compare x y  = x - y
    end
;;
```

Des paires d'entiers avec l'ordre lexicographique

```
module ComparableIntPair  =
    struct
     type t =  int * int
     (* analogue du compareTo de java *)
     let compare (x1,y1) (x2,y2) =
          if x1 < x2 then -1
          else if x1 > x2 then 1
          else if y1 < y2 then -1
          else if y1 > y2 then 1
          else 0
```

```
      end
  ;;
```

Des paires d'entiers avec l'ordre inverse.

```
module OppositeComparableIntPair  =
    struct
     type t =  int * int
     (* analogue du compare de java *)
     let compare (x1,y1) (x2,y2) =
          - ComparableIntPair.compare (x1,y1) (x2,y2)
     end
;;
```

Exemple de construction de table ou map

L'appel du foncteur se fait alors de la façon suivante.

```
module MapIntPair = Map.Make (ComparableIntPair);;
let m = MapIntPair.empty;;
let m = MapIntPair.add (1,2) "toto" m;;
let m = MapIntPair.add (2,4) "tutu" m;;
MapIntPair.find (2,4) m;;
```

les réponses ocaml sont :

```
# module MapIntPair :
 sig
  type key = ComparableIntPair.t
  and 'a t = 'a Map.Make(ComparableIntPair).t
  val empty : 'a t
  val add : key -> 'a -> 'a t -> 'a t
  val find : key -> 'a t -> 'a
  val remove : key -> 'a t -> 'a t
  val mem : key -> 'a t -> bool
  val iter : (key -> 'a -> unit) -> 'a t -> unit
```

```
    val map : ('a -> 'b) -> 'a t -> 'b t
    val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
   end
# val m : 'a MapIntPair.t = <abstr>
# val m : string MapIntPair.t = <abstr>
# val m : string MapIntPair.t = <abstr>
# - : string = "tutu"
```

Exemple de construction d'ensemble ordonné

La fabriquation d'un module définissant un ensemble ordonné se fait avec le foncteur `Set.Make` de la bibilothèque `Set`. C'est l' équivalent de `OrderedSet` de Java. Il prend en argument un module définissant des valeurs comparables comme `Map.Make`. Ainsi,

```
module SetIntPair = Set.Make (ComparableIntPair);;
let s = SetIntPair.empty;;
let s = SetIntPair.add (1,2) s;;
let s = SetIntPair.add (2,4) s;;
SetIntPair.cardinal s;;
SetIntPair.min_elt s;;
SetIntPair.max_elt s;;
SetIntPair.mem (2,4) s;;
SetIntPair.elements s;;
```

donne

```
# module SetIntPair :
  sig
    type elt = ComparableIntPair.t
    and t = Set.Make(ComparableIntPair).t
    val empty : t
    val is_empty : t -> bool
```

```
    val mem : elt -> t -> bool
    val add : elt -> t -> t
    val singleton : elt -> t
    val remove : elt -> t -> t
    val union : t -> t -> t
    val inter : t -> t -> t
    val diff : t -> t -> t
    val compare : t -> t -> int
    val equal : t -> t -> bool
    val subset : t -> t -> bool
    val iter : (elt -> unit) -> t -> unit
    val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
    val for_all : (elt -> bool) -> t -> bool
    val exists : (elt -> bool) -> t -> bool
    val filter : (elt -> bool) -> t -> t
    val partition : (elt -> bool) -> t -> t * t
    val cardinal : t -> int
    val elements : t -> elt list
    val min_elt : t -> elt
    val max_elt : t -> elt
    val choose : t -> elt
  end
# val s : SetIntPair.t = <abstr>
# val s : SetIntPair.t = <abstr>
# val s : SetIntPair.t = <abstr>
# - : int = 2
# - : SetIntPair.elt = (1, 2)
# - : SetIntPair.elt = (2, 4)
# - : bool = true
# - : SetIntPair.elt list = [(1, 2); (2, 4)]
```

Exemple de construction de table de hachage

Une table de hachage peut être obtenue à l'aide du foncteur `Hashtbl.Make` de la bibliothèque `Hashtbl`. Les clés de la table doivent être définies dans un module compatible avec l'interface prédéfinie :

```
module type HashedType =
  sig
    type t
      (** The type of the hashtable keys. *)
    val equal : t -> t -> bool
      (** The equality predicate used to compare keys. *)
    val hash : t -> int
      (** A hashing function on keys. It must be such
       that if two keys are equal according to [equal],
       then they have identical hash values as computed
        by [hash].
       Examples: suitable ([equal], [hash]) pairs for
       arbitrary key types include
       ([(=)], {!Hashtbl.hash}) for comparing objects by
       structure, and
       ([(==)], {!Hashtbl.hash}) for comparing objects by
       addresses (e.g. for mutable or cyclic keys). *)
   end
```

Par exemple, des clés couples d'entiers peuvent être définies par

```
module HashedIntPair  =
   struct
     type t =  int * int
     let equal = (=)
     let hash  = Hashtbl.hash
   end
```

```
;;
```

La table est ensuite créée par :

```
module Hach = Hashtbl.Make (HashedIntPair);;
let tbl = Hach.create 100;;
Hach.add tbl (1,2) "toto" ;;
Hach.add tbl (2,4) "tutu" ;;
Hach.find tbl (2,4);;
```

qui donne

```
# module HashedIntPair :
  sig
    type t = int * int
    val equal : 'a -> 'a -> bool
    val hash : 'a -> int
  end
# module Hach :
  sig
    type key = HashedIntPair.t
    and 'a t = 'a Hashtbl.Make(HashedIntPair).t
  val create : int -> 'a t
  val clear : 'a t -> unit
  val copy : 'a t -> 'a t
  val add : 'a t -> key -> 'a -> unit
  val remove : 'a t -> key -> unit
  val find : 'a t -> key -> 'a
  val find_all : 'a t -> key -> 'a list
  val replace : 'a t -> key -> 'a -> unit
  val mem : 'a t -> key -> bool
  val iter : (key -> 'a -> unit) -> 'a t -> unit
  val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  end
```

```
# val tbl : '_a Hach.t = <abstr>
# - : unit = ()
# - : unit = ()
# - : string = "tutu"
```

L'aspect objet

L'extension objet s'intègre au noyau fonctionnel et au noyau impératif. Son intérêt est sa compatibilité avec le système de typage. On obtient ainsi un langage objet typé statiquement avec inférence de types.

Dans un premier exemple, on définit une classe `point` qui permet de manipuler des points du plan. Une classe définit tout à la fois un type et un moule pour les objets. Elle contient des champs de données appelées variables d'instance et des méthodes. Les instances d'une classe sont appelées objets.

```
class point x_init y_init =
 object
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method move dx dy = x <- x + dx; y <- y +dy
 end;;
class point :
  int ->
  int ->
  object
    val mutable x : int
    val mutable y : int
    method get_x : int
    method get_y : int
    method move : int -> int -> unit
  end
```

La création d'une instance de la classe se fait par

```
# new point;;
- : int -> int -> point = <fun>
# let p = new point 2 3;;
val p : point = <obj>
```

Le type d'un objet est la liste des noms des méthodes avec leur type. Dans l'exemple, `point` est une abréviation pour

```
<get_x : int ; get_y : int; move : int -> int -> unit>
```

Comme les variables d'instance n'apparaissent pas dans le type, on ne peut y accéder *que par l'intermédiaire des méthodes.*
L'appel des méthodes se fait à l'aide du symbole `#`.

```
# p#get_x;;
- : int = 2
# p#get_y;;
- : int = 3
# p#move 1 1;;
- : unit = ()
# p#get_x;;
- : int = 3
# p#get_y;;
- : int = 4
```

On peut donc ajouter les méthodes

```
# class point x_init y_init =
 object
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method set_x z = x <- z
  method set_y z = y <- z
  method move dx dy = x <- x + dx; y <- y + dy
  method to_string () =
   "("^(string_of_int x)^","^( string_of_int y)^")"
 end;;

class point :
  int ->
  int ->
  object
    val mutable x : int
    val mutable y : int
    method get_x : int
    method get_y : int
    method move : int -> int -> unit
    method set_x : int -> unit
    method set_y : int -> unit
    method to_string : unit -> string
  end
# let p = new point 2 3;;
val p : point = <obj>
# p#to_string();;
```

```
- : string = "(2,3)"
# print_string ( p#to_string() );;
(2,3)- : unit = ()
```

La notion d'héritage

On définit une classe de points colorés en ajoutant un champ ou attribut couleur à la classe `point`.

```
class point_colore  x_init  y_init c_init =
 object (this)
 inherit point x_init  y_init as super
 val mutable c = c_init
 method get_color = c
 method set_color nc = c <- nc
 method to_string()=super#to_string()^this#get_color
 end;;
class point_colore :
  int ->
  int ->
  string ->
  object
    val mutable c : string
    val mutable x : int
    val mutable y : int
    method get_color : string
    method get_x : int
    method get_y : int
    method move : int -> int -> unit
    method set_color : string -> unit
    method set_x : int -> unit
    method set_y : int -> unit
    method to_string : unit -> string
```

```
  end

# let p = new point_colore 2 3 "bleu";;
val p : point_colore = <obj>
# p#get_color;;
- : string = "bleu"
# p#to_string();;
- : string = "(2,3)bleu"
```

Ce serait incorrect d'écrire

```
class point_colore  x_init  y_init c_init =
 object (this)
  inherit point x_init  y_init as super
  val mutable c = c_init
  method get_color = c
  method set_color nc = c <- nc
 end;;
Some type variables are unbound in this type:
  class point_colore :
    int ->
    int ->
    'a ->
    object
      val mutable c : 'a
      val mutable x : int
      val mutable y : int
      method get_color : 'a
      method get_x : int
      method get_y : int
      method move : int -> int -> unit
      method set_color : 'a -> unit
      method set_x : int -> unit
```

```
      method set_y : int -> unit
      method to_string : unit -> string
    end
The method get_color has type 'a where 'a is unbound
```

On peut forcer le type des couleurs à `string`

```
class point_colore  x_init  y_init (c_init:string) =
 object (this)
  inherit point x_init  y_init as super
  val mutable c = c_init
  method get_color = c
  method set_color nc = c <- nc
 end;;
```

La méthode à utiliser est déterminée à l'exécution. On appelle cela la liaison retardée.

Il n'y a pas de transtypage implicite du type dérivé vers le type de base.

```
# class point x_init y_init =
 object
  val mutable x = x_init
  val mutable y = y_init
  method get_x = x
  method get_y = y
  method set_x z = x <- z
  method set_y z = y <- z
  method move dx dy = x <- x + dx; y <- y + dy
  method to_string () =
   "("^(string_of_int x)^ ","^( string_of_int y)^")"
  method f (p:point) = 1
 end;;
# class point_colore  x_init  y_init c_init =
```

```
    object (this)
      inherit point x_init  y_init as super
    val mutable c = c_init
    method get_color = c
    method set_color nc = c <- nc
    method to_string () = super#to_string()^this#get_color
  end;;
# let p = new point_colore 2 3 "bleu";;
val p : point_colore = <obj>
# let q = new point 1 1;;
val q : point = <obj>
# q#f(p);;
Characters 3-6:
  q#f(p);;
     ^^^
This expression has type point_colore
but is here used with type point
Only the first object type has a method get_color
```

On peut remédier à ceci en castant le point coloré en point. Il s'agit alors de sous-typage : possibilité pour un objet d'un certain type d'être utilisé comme un objet d'un autre type. Attention, bien que connu comme un point, le point coloré n'en reste pas moins un point coloré et les méthodes des points colorés seront déclenchées lors de l'application d'une méthode sur l'instance (comme `to_string` par exemple).

```
# let p2 = (p1 :> point);;
val p2 : point = <obj>
# p1#to_string();;
- : string = "(4,5)bleu"
# p2#to_string();;
```

```
- : string = "(4,5)bleu"
# q#f(p2);;
- : int = 1
```

Classes paramétrées

Les classes paramétrées permettent d'utiliser le polymorphisme paramétrique d'Objective Caml dans les classes. Ce procédé s'intègre dans le système de typage.

Sur l'exemple suivant, on définit une pile contenant des éléments d'un type quelconque, mais tous du même type. Cette pile est implémentée par une liste d'éléments d'un type quelconque.

```
class ['a] pile  =
object (this)
   val mutable p: 'a list = []
   method est_vide = (p = [])
   method get_top = List.hd p
   method push x =  p <- x::p
   method pop = if (this#est_vide)
                then raise (Failure "pile vide")
                else p <- List.tl p;
   method to_string () = p
 end;;
```

Exemple d'utilisation de cette pile

```
# let p1 = new pile ;;
val p1 : '_a pile = <obj>
# p1#push 3;;
- : unit = ()
# p1#push 8;;
- : unit = ()
# p1#push 2;;
```

```
- : unit = ()
# p1#to_string();;
- : int list = [2; 8; 3]
# p1;;
- : int pile = <obj>
# p1#get_top;;
- : int = 2
# p1#pop;;
- : unit = ()
# p1#pop;;
- : unit = ()
# p1#pop;;
- : unit = ()
# p1#pop;;
Uncaught exception: Failure "pile vide".
```

Un autre exemple

On désire construire une paire d'éléments d'un type quelconque. La définition de classe suivante est erronée.

```
# class pair x0 y0 =
    object
       val x = x0
       val y = y0
       method fst = x
       method snd = y
    end ;;
```

Il faut écrire

```
# class ['a,'b] pair (x0:'a) (y0:'b) =
    object
       val x = x0
```

```
        val y = y0
        method fst = x
        method snd = y
     end ;;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b
   method fst : 'a method snd : 'b end
```

```
# let p = new pair 2 'x';;
val p : (int, char) pair = <obj>
# let q = new pair true [1;2];;
val q : (bool, int list) pair = <obj>
# p#fst;;
- : int = 2
# q#snd;;
- : int list = [1; 2]
```

Dérivation dans les classes paramétrées

On crée ci-dessous une classe qui dérive de la classe `pair` et qui ajoute une méthode permettant d'appliquer des fonctions sur chaque élément de la paire.

```
# class ['a,'b] map_pair (x0 : 'a) (y0 : 'b) =
        object
          inherit ['a,'b] pair x0 y0
          method  map f g  = new map_pair (f x0) (g y0)
        end;;
class ['a, 'b] map_pair :
  'a ->
  'b ->
  object
    val x : 'a
```

```
    val y : 'b
    method fst : 'a
    method map :
      ('a -> 'a) -> ('b -> 'b) -> ('a, 'b) map_pair
    method snd : 'b
  end
```

```
# let q = new map_pair true [1;2];;
val q : (bool, int list) map_pair = <obj>
# let r = q#map (function x -> not x)
                (function l -> 0::l);;
val r : (bool, int list) map_pair = <obj>
# r#fst;;
- : bool = false
# r#snd;;
- : int list = [0; 1; 2]
```

Il est aussi possible de créer une classe dérivée pour préciser les type de la classe paramétrée. On veut par exemple créer des paires de points.

```
# class pair_point (p1,p2) =
        object
          inherit [point,point] pair p1 p2
        end;;
     class pair_point :
       point * point ->
       object
         val x : point
         val y : point
         method fst : point
         method snd : point
       end
```

Formes, rectangles et ellipses

Les classes abstraites Java correspondent aux classes virtuelles ocaml. On définit ci-dessous une classe virtuelle `forme_rectangulaire` et deux classes dérivées `rectangle` et `ellipse`. Les deux classes implémentent les méthodes virtuelles de la classe virtuelle. De plus on ajoute dans la classe `ellipse` une méthode non déclarée dans `forme_rectangulaire`, `special_ellipse`.

```
class virtual forme_rectangulaire (x_init : float)
                                  (y_init : float) =
 object (this)
   val mutable x = x_init
   val mutable y = y_init
   method get_x = x
   method get_y = y
   method virtual get_aire : float
   method to_string () = string_of_float this#get_aire
   method equals (f:forme_rectangulaire) =
         (this#get_x = f#get_x)&&(this#get_y = f#get_y)
 end;;

class rectangle x_init y_init =
 object (this)
  inherit forme_rectangulaire x_init y_init as super
  method get_aire = this#get_x *. this#get_y
 end;;

class ellipse x_init y_init =
 object (this)
  inherit forme_rectangulaire x_init y_init as super
  method get_aire = this#get_x *. this#get_y
```

```
                                      *. atan 1.0
    method special_ellipse = 0
   end;;
```

Les types synthétisés sont

```
#         class virtual forme_rectangulaire :
  float ->
  float ->
  object
    val mutable x : float
    val mutable y : float
    method equals : forme_rectangulaire -> bool
    method virtual get_aire : float
    method get_x : float
    method get_y : float
    method to_string : unit -> string
  end
#         class rectangle :
  float ->
  float ->
  object
    val mutable x : float
    val mutable y : float
    method equals : forme_rectangulaire -> bool
    method get_aire : float
    method get_x : float
    method get_y : float
    method to_string : unit -> string
  end
#           class ellipse :
  float ->
  float ->
```

```
object
  val mutable x : float
  val mutable y : float
  method equals : forme_rectangulaire -> bool
  method get_aire : float
  method get_x : float
  method get_y : float
  method special_ellipse : int
  method to_string : unit -> string
end
```

On crée des objets rectangles et ellipses,

```
let r = new rectangle 2.0 3.0;;
let e1 = new ellipse 4.0 5.0;;
let e2 = new ellipse 6.0 10.0;;
let e3 = new ellipse 6.0 10.0;;
# val r : rectangle = <obj>
# val e1 : ellipse = <obj>
# val e2 : ellipse = <obj>
# val e3 : ellipse = <obj>
```

et l'on construit une liste `[r; e1 ; e2]` d'objets de type `forme_rectangulaire`.

```
let cons (f,l) = (f :> forme_rectangulaire)::l;;
let l = cons(r, cons(e1, cons(e2,[])));;
# val cons :
  #forme_rectangulaire * forme_rectangulaire list
       -> forme_rectangulaire list = <fun>
# val l : forme_rectangulaire list = [<obj>;<obj>;<obj>]
```

On peut calculer la liste des aires des objets de la liste `l` par

```
let aire_map l = List.map (function o -> o#get_aire) l;;
```

```
aire_map l;;
# val aire_map :
< get_aire : 'a; .. > list -> 'a list = <fun>
# - : float list =
 [6.; 15.7079632679489656; 47.1238898038468932]
```

Cet exemple illustre la liaison retardée (comme en Java). Par contre, le transtypage entraîne une perte d'information, due ici à la présence de la fonction spéciale sur ellipse. En effet, le transtypage inverse, même explicite, sur un objet ellipse typé `forme_rectangulaire` est impossible.

```
# e3;;
- : ellipse = <obj>
# let e = (e3 :> forme_rectangulaire);;
val e : forme_rectangulaire = <obj>
# (e :> ellipse);;
Characters 1-2:
  (e :> ellipse);;
   ^
This expression cannot be coerced to type
  ellipse =
    < equals : forme_rectangulaire -> bool;
      get_aire : float; get_x :
      float; get_y : float; special_ellipse : int;
      to_string : unit -> string >;
it has type
  forme_rectangulaire =
    < equals : forme_rectangulaire -> bool;
      get_aire : float; get_x :
      float; get_y : float; to_string : unit -> string >
but is here used with type
  #ellipse as 'a =
```

```
     < equals : forme_rectangulaire -> bool;
       get_aire : float; get_x :
       float; get_y : float; special_ellipse : int;
       to_string : unit -> string; .. >
Only the second object type has a method special_ellipse.
This simple coercion was not fully general.
Consider using a double coercion.
```

Par contre, il est possible de le faire sur un objet rectangle.

```
# r;;
- : rectangle = <obj>
# let f = (r :> forme_rectangulaire);;
val f : forme_rectangulaire = <obj>
# (f :> rectangle);;
- : rectangle = <obj>
```

Cet exemple illustre les limites de la programmation par objets en Objective Caml, surtout lorsque l'on connaît Java.