

Introduction à Caml

Syntaxe et types

Table des matières

1	Installer OCaml	2
1.1	Solution facile	2
1.2	Solution experte	2
2	Généralités	3
2.1	Commentaires	3
2.2	Séparateurs, expressions et liaisons	3
2.3	Parenthèses et blocs d'instructions	3
3	Les types	4
3.1	Types de base	4
3.1.1	Entiers	4
3.1.2	Flottants	5
3.1.3	Caractères et chaînes de caractères	5
3.1.4	Booléens	5
3.1.5	Unit	6
3.2	Types paramétrés	6
3.2.1	Couples et n -uplets	6
3.2.2	Listes	6
3.2.3	Vecteurs (ou tableaux)	7
3.2.4	Références	7
3.2.5	Fonctions	7
3.3	Types déclarés (construits)	8
3.3.1	Abréviations de type	8
3.3.2	Types somme	8
3.3.3	Types enregistrement	8
3.3.4	Types récursifs, types paramétrés	9
4	Quelques commandes utiles	9
4.1	Conversions de types	9
4.2	Affichage	9
4.3	Entrée au clavier	10
5	Structures de contrôle	10
5.1	Conditionnelles	10
5.2	Boucles définies	10
5.3	Boucles indéfinies	10
5.4	Liaisons locales	11
5.5	Filtrage	11
5.6	Gestion des exceptions	12

2 Généralités

Contrairement à Python, les caractères blancs (espaces, retours à la ligne...) ne jouent aucun rôle syntaxique, et le programmeur peut organiser l'indentation et les retours à la ligne comme il le souhaite. Bien entendu, il reste fortement recommandé d'indenter proprement son code pour qu'il soit lisible.

2.1 Commentaires

Un commentaire est ouvert avec les deux caractères (*, et fermé avec *). Les commentaires imbriqués sont gérés correctement.

```

1 (* ceci est un commentaire *)
2
3 (* ceci est un autre commentaire (* avec un commentaire dans le commentaire *) *)

```

2.2 Séparateurs, expressions et liaisons

Chaque phrase en caml est terminée par un double point-virgule ;;.

Une phrase peut consister en :

- Une expression (ou une suite d'expressions séparées par des point-virgule ;).
- Une liaison (*let-binding*) de la forme let identifiant = valeur qui définit une nouvelle variable.
- On peut définir plusieurs variables en même temps :

```
let identifiant1 = valeur1 and identifiant2 = valeur2
```

– En utilisant let rec au lieu de let, les variables peuvent être définies récursivement.

```

1 print_string "bla bla";
2 print_newline ();
3 print_string "bla bla bla";;
4
5 let somme = 3 + 5;;
6
7 let x = 3 and y = 5;; (* Deux liaisons simultanées *)
8
9 let f = function n -> n + 1;; (* Pas de différence entre la définition d'une fonction
10                               et celle d'un entier *)
11
12 5 + f 4;;
13
14 let rec factorielle = (* Une fonction factorielle définie récursivement *)
15   function n ->
16     if n=0 then 1 else n * factorielle(n-1);;

```

2.3 Parenthèses et blocs d'instructions

Comme on a pu le voir dans l'exemple ci-dessus, les parenthèses ne sont pas indispensables pour appeler une fonction : on peut écrire f x au lieu de f(x). Les parenthèses restent indispensables pour indiquer la priorité des opérations : (f 3)+4 ne calcule pas la même chose que f(3+4).

Les blocs d'instructions sont délimités par begin ... end, comme dans l'exemple suivant :

```

1 if 5-(2+1) = 3 then
2   begin
3     print_string "Ouh la...";
4     print_string "Il y a un problème quelque part";
5   end;;

```

Remarque : Le couple (...) parenthèse ouvrante - parenthèse fermante est exactement synonyme du couple begin ... end. L'exemple précédent peut aussi s'écrire :

```

1 if 5 = begin 2+1 end = 3 then
2   (
3     print_string "Ouh la... ";
4     print_string "Il y a un problème quelque part";
5   );

```

Mais on préfèrera (...) dans les calculs, et begin ... end pour les blocs d'instructions.

3 Les types

Le système de typage est l'un des points forts de Caml. Donnons-en quelques caractéristiques :

- Caml est **fortement typé**. Toute valeur, toute fonction, toute expression du langage a un type. Les types doivent être respectés strictement, il n'y a aucune conversion implicite de type. Par exemple, il est impossible d'ajouter directement un entier et un flottant, il faut d'abord convertir à la main l'entier en flottant. On n'utilise pas la même commande pour ajouter deux entiers (+) et deux flottants (+.).
- Caml pratique l'**inférence de type** : Le programmeur n'a pas besoin d'indiquer les types des valeurs manipulés, Caml est capable de les inférer en analysant le code source.
- Caml est **statiquement typé**, c'est-à-dire que la détermination des types et la vérification de leur correction se fait lors de la compilation, et non lors de l'exécution. La rigueur du système de typage permet de détecter la plupart des bugs lors de la compilation.
- Caml supporte le **polymorphisme** : il est possible par exemple d'écrire une fonction générique prenant un argument de type indéterminé 'a, et renvoyant un résultat du même type 'a; au lieu d'écrire plusieurs fonction int -> int, float -> float, etc...

3.1 Types de base

3.1.1 Entiers

Type : int.

Description : Les entiers signés.

Syntaxe : Une simple suite de chiffres, sans séparateur décimal. Exemple : 142857

Quelques fonctions et valeurs fournies par le langage :

+	Addition	
-	Soustraction	
*	Multiplication	
/	Division entière	7 / 2 renvoie 3
mod	Reste	7 mod 2 renvoie 1
succ	Fonction successeur	Synonyme de function n -> n + 1
pred	Fonction prédécesseur	Synonyme de function n -> n - 1
abs	Valeur absolue	
max_int	Le plus grand entier (positif)	$2^{30} - 1$ ou $2^{62} - 1$
min_int	Le plus petit entier (négatif)	-2^{30} ou -2^{62}

3.1.2 Flottants

Type : float.

Description : Les flottants.

Syntaxe : Une suite de chiffres, avec séparateur décimal « . ». Exemple : 12.3

Quelques fonctions et valeurs fournies par le langage :

+. . .	Addition	
-. . .	Soustraction	
*. . .	Multiplication	
/. . .	Division	
**. . .	Puissance (opérateur infixé)	2. ** 3. renvoie 8.
abs_float	Valeur absolue	
sqrt	Racine carrée	
log	Logarithme	Il s'agit du logarithme népérien.
exp	Exponentielle	
sin cos tan	Fonctions trigonométriques	
asin acos atan	Fonctions trigonométriques réciproques	
sinh cosh tanh	Fonctions trigonométriques hyperboliques	

3.1.3 Caractères et chaînes de caractères

Type : char.

Description : Caractère individuel.

Syntaxe : Un caractère entre '...'. Exemple : 'A'

Caractères spéciaux :

- '\\' pour \
- '\' pour '
- '\n' pour un retour à la ligne
- '\t' pour un caractère tabulation
- '\123' pour indiquer un code Ascii (en décimal)

Type : string.

Description : chaîne de caractères : une suite finie de n caractères, indexés de 0 à $n - 1$.

Syntaxe : Une suite de caractères entre "....". Exemple : "bla bla"

Pour utiliser le caractère ", on tape \".

Quelques fonctions et valeurs fournies par le langage :

-	Concaténation	
String.length	Longueur d'une chaîne	Type string -> int
String.get	Renvoie le n -ième caractère	Type string -> int -> char
s.[i]	Synonyme de String.get s i	
String.sub	Découpe une sous chaîne	Type string -> int -> int -> string
String.make	Crée une chaîne constante	Type int -> char -> string

3.1.4 Booléens

Type : bool.

Description : Booléen : vrai ou faux.

Syntaxe : true ou false.

Quelques fonctions et valeurs fournies par le langage :

<code>true</code>	Vrai	
<code>false</code>	Faux	
<code>&&</code>	Et	L'évaluation est paresseuse.
<code> </code>	Ou	L'évaluation est paresseuse.
<code>not</code>	Négation	
<code>< <= > >=</code>	Opérateurs de comparaison	Type ' <code>a -> 'a -> bool</code>
<code>= <></code>	Teste l'(in)égalité de deux valeurs	Égalité structurelle
<code>== !=</code>	Teste l'(in)égalité de deux valeurs	Égalité physique
<code>min max</code>	Minimum ou maximum de deux valeurs	Type ' <code>a -> 'a -> 'a</code>
<code>compare</code>	Renvoie 1, 0 ou -1	Type ' <code>a -> 'a -> int</code>

3.1.5 Unit

Type : `unit`.

Description : Le type « Aucune valeur ». Il est notamment consommé par les fonctions sans argument, et renvoyé par les fonctions sans résultat.

Syntaxe : `()` est la seule valeur de type `unit`.

Exemples :

`print_string "blabla"` affiche `blabla` et renvoie `()`.
`print_newline()` écrit un saut de ligne, et renvoie `()`.

3.2 Types paramétrés

3.2.1 Couples et n-uplets

Type : `'a * 'b`.

Description : Un couple de valeurs

Syntaxe : `(valeur1 , valeur2)`. Exemple : `(1,"A")` de type `int * string`.

Quelques fonctions et valeurs fournies par le langage :

<code>fst</code>	Premier élément d'un couple	<code>fst (1,"A")</code> renvoie 1
<code>snd</code>	Deuxième élément d'un couple	<code>snd (1,"A")</code> renvoie "A"

Type : `'a * 'b * 'c ...`

Description : Un n-uplets de valeurs

Syntaxe : `(valeur1 , valeur2 , ...)`.

Exemple : `(1, "A", function n->n+1, ())` est de type `int * string * (int -> int) * unit`.

3.2.2 Listes

Type : `'a list`.

Description : Listes de valeurs (qui ont toutes le même type).

Syntaxe : `[valeur1 ; valeur2 ; valeur3 ; ...]`.

Quelques fonctions et valeurs fournies par le langage :

<code>[]</code>	Liste vide	
<code>a :: q</code>	Construit une liste formée de <code>a</code> , suivi de <code>q</code>	« cons ».
<code>List.hd</code>	Premier élément (head)	Type <code>'a list -> 'a</code>
<code>List.tl</code>	Tout sauf le premier élément (tail)	Type <code>'a list -> 'a list</code>
<code>@</code>	Concaténation	
<code>List.length</code>	Nombre d'éléments	
<code>List.rev</code>	Renverse une liste	<code>rev [1;2;3]</code> renvoie <code>[3;2;1]</code>

Des précisions sur le type `list` seront données dans la suite du cours.

3.2.3 Vecteurs (ou tableaux)

Type : '*a* array.

Description : Tableaux de *n* valeurs (qui ont toutes le même type), indexées de 0 à *n* - 1.

Syntaxe : [| valeur₁ ; valeur₂ ; valeur₃ ; ... |].

Quelques fonctions et valeurs fournies par le langage :

[]	Tableau vide	
Array.make	Construit un tableau constant	Type int → ' <i>a</i> → ' <i>a</i> array
Array.get	Renvoie le <i>i</i> -ième élément d'un tableau <i>v</i>	Type ' <i>a</i> array → int → ' <i>a</i>
v.(i)	Synonyme de Array.get <i>v</i> <i>i</i>	
Array.set	Modifie le <i>i</i> -ième élément d'un tableau <i>v</i>	Type ' <i>a</i> array → int → ' <i>a</i> → unit
v.(i) <- <i>c</i>	Synonyme de Array.set <i>v</i> <i>i</i> <i>c</i>	

Des précisions sur le type array seront données dans la suite du cours.

3.2.4 Références

Type : '*a* ref.

Description : Pointeur sur une valeur mutable de type '*a*

Syntaxe : ref valeur.

Quelques fonctions et valeurs fournies par le langage :

ref	Constructeur de référence	let <i>a</i> = ref 1;;
:=	Opérateur d'affectation	<i>a</i> := 3;;
!	Opérateur de déréférencement	! <i>a</i> renvoie 3

Des précisions sur le type ref seront données dans la suite du cours.

3.2.5 Fonctions

Type : '*a* → '*b*.

Description : Fonction d'un type '*a* vers un type '*b*.

Syntaxe : function motif → expression ou fun motif → expression.

- On peut raccourcir let *f* = function *x* → *x*+1 en let *f* *x* = *x*+1
- Une fonction à deux arguments de types '*a* et '*b*, qui renvoie un argument de type '*c*, n'est rien d'autre qu'une fonction qui prend un argument de type '*a*, et qui renvoie une fonction de type '*b* → '*c*. Une fonction à deux arguments est donc de type '*a* → '*b* → '*c*.

Elle peut s'écrire function *x* → function *y* → *x*+*y* ou, plus simplement, fun *x* *y* → *x*+*y*.

Dans une liaison (*let-binding*), on peut aussi utiliser let *f* *x* *y* = *x*+*y*;;.

Quelques exemples :

```

1 let identite x = x;;
2
3 let rec f n =
4   if n = 1 then 0 else 1+f(n/2);;
5
6 let echange (x,y) = (y,x);;
7
8 let compose f g = function x -> f (g x);;
9
10 let f_couple f g = function (x,y) -> (f x, g y);;
11
12 let transvase = f_couple succ pred;;

```

3.3 Types déclarés (construits)

En plus des types prédéfinis, Caml permet de définir de nouveaux types avec le mot-clef **type**.

3.3.1 Abréviations de type

Il s'agit juste de donner un alias pour un type déjà existant. Par exemple :

```
1 type complexe = float * float;;
```

On peut forcer une expression à adopter ce type avec la syntaxe `(expression:type)`, et définir ainsi :

```
1 let somme_complexe ((x,y):complexe) ((z,t):complexe) = ((x+.z, y+.t):complexe);;
2
3 let produit_complexe ((x,y):complexe) ((z,t):complexe) =
4   ( (x*.z -. y*.t, x*.t +. y*.z) :complexe);;
```

3.3.2 Types somme

Un type somme (ou *variant*) consiste en l'union disjointe de plusieurs types. Voilà la syntaxe :

```
1 type limite =
2   | Reel of float
3   | Infini of bool (* true -> +infini, false -> -infini *)
4   | Indetermine;;
```

Les noms *Reel*, *Infini* et *Indetermine* sont appelé des *constructeurs de type*.

Reel se comporte comme une fonction `float -> limite`.

Infini se comporte comme une fonction `bool -> limite`.

Indetermine se comporte comme une constante de type *limite*.

On peut ensuite manipuler le type somme notamment grâce au filtrage (voir plus loin).

```
1 let somme_limites = fun
2   | (Reel x) (Reel y) -> Reel (x+.y)
3   | (Reel _) (Infini signe) -> Infini signe
4   | (Infini signe) (Reel _) -> Infini signe
5   | (Infini signe1) (Infini signe2) ->
6     if signe1 = signe2
7     then Infini signe1
8     else Indetermine
9   | _ _ -> Indetermine;;
```

3.3.3 Types enregistrement

Un type enregistrement (*record*) est un *n*-uplet dont les différentes composantes portent des labels.

Exemple :

```
1 type complexe =
2   { re : float;
3   im : float };;
4
5 let i = { re=0.; im=1. };;
6
7 let somme_complexe z1 z2 = { re = z1.re +. z2.re; im = z1.im +. z2.im };;
```

Les champs d'une valeur d'un type enregistrement sont accessibles par `identifiant.champ`.

3.3.4 Types récursifs, types paramétrés

Les types construits par l'utilisateur peuvent être définis récursivement :

```

1 type arbre =
2   | Feuille
3   | Noeud of arbre * arbre;;

```

On peut même définir des types mutuellement récursifs :

```

1 type gouache = Rouge | Jaune | Bleu
2 and couleur =
3   | Pur of gouache
4   | Melange of couleur * couleur;;
5
6 let vert = Melange (Pur Jaune,Pur Bleu);;
7
8 let rec contient_rouge = function
9   | Pur Rouge -> true
10  | Pur _ -> false
11  | Melange(x,y) -> (contient_rouge x) || (contient_rouge y);;
12
13 contient_rouge vert;;

```

Les types définis par l'utilisateur peuvent aussi être paramétrés. Voilà un nouveau type arbre dont les feuilles sont décorées :

```

1 type 'a arbre =
2   | Feuille of 'a
3   | Noeud of ('a arbre) * ('a arbre);;
4
5 let mon_arbre = Noeud(Feuille 1,Feuille 2);;

```

Dans cette exemple, `mon_arbre` sera de type `int arbre`.

4 Quelques commandes utiles

4.1 Conversions de types

Quelques commandes de conversion :

int_of_float	float_of_int	string_of_int
int_of_string	string_of_float	float_of_string
string_of_bool	int_of_char	char_of_int
Array.of_list	Array.to_list	

4.2 Affichage

On dispose entre autres des fonctions suivantes pour afficher à l'écran :

<code>print_char</code>	Afficher un caractère	Type <code>char -> unit</code>
<code>print_string</code>	Afficher une chaîne de caractères	Type <code>string -> unit</code>
<code>print_int</code>	Afficher un entier	Type <code>int -> unit</code>
<code>print_float</code>	Afficher un flottant	Type <code>float -> unit</code>
<code>print_newline</code>	Passer à la ligne	Type <code>unit -> unit</code>

4.3 Entrée au clavier

Les fonctions suivantes sont là pour lire des données entrées au clavier :

<code>read_line</code>	Lit une chaîne de caractères terminée par un saut de ligne	Type <code>unit -> string</code>
<code>read_int</code>	Composée de <code>read_line</code> et <code>int_of_string</code>	Type <code>unit -> int</code>
<code>read_float</code>	Composée de <code>read_line</code> et <code>float_of_string</code>	Type <code>unit -> float</code>

5 Structures de contrôle

5.1 Conditionnelles

Première version (sans else) :

```
1 if expression_booléenne then expression
```

Dans ce cas, l'expression après le `then` doit être de type `unit`.

La structure complète est elle-même de type `unit`.

Deuxième version (avec else) :

```
1 if expression_booléenne
2   then expression1
3   else expression2
```

Les deux expressions après `then` et `else` doivent être de même type, qui sera le type de la structure complète.

5.2 Boucles définies

Version ascendante :

```
1 for variable = expr1 to expr2 do expression done
```

Les bornes `expr1` et `expr2` sont des nombres entiers. La structure complète est de type `unit`.

Version descendante :

```
1 for variable = expr1 downto expr2 do expression done
```

Les bornes `expr1` et `expr2` sont des nombres entiers. La structure complète est de type `unit`.

5.3 Boucles indéfinies

```
1 while expression_booléenne do expression done
```

La structure complète est de type `unit`.

5.4 Liaisons locales

```
1 let identifiant = valeur in expression
```

Ce *let-binding* local opère des liaisons qui ne seront valables que le temps de l'évaluation de l'expression.

La syntaxe avant le `in` est la même que celle du *let-binding* global. On peut en particulier utiliser les mots clefs `rec` et `and`.

```
1 let rec f n = if n=0 then 1 else n*f(n-1) in f 7;;
```

5.5 Filtrage

Le filtrage (*pattern-matching*) est une structure fondamentale de Caml. Il s'agit de tester si une valeur se conforme à un certain *motif*, et exécuter des instructions correspondantes.

La syntaxe est la suivante :

```

1 match expression with
2   | motif1 -> expr1
3   | motif2 -> expr2
4   ...
5   | motifn -> exprn

```

Caml tente de filtrer *expression* avec *motif1*. En cas de succès, *expr1* est exécutée. En cas d'échec, les motifs suivants sont essayés dans l'ordre. Si aucun motif ne filtre *expression*, une exception est levée.

Voilà trois exemples simples :

```

1 match 3*3-2*2*2 with
2   | 0 -> print_string "Le résultat est zéro"
3   | 1 -> print_string "Le résultat est un"
4   | n -> print_string "Le résultat n'est ni zéro, ni un, mais "; print_int n;;
5
6
7 match vert with
8   | Pur _ -> "couleur pure"
9   | Mélange (_,_) -> "mélange";;
10
11 let contient_zero couple = match couple with
12   | (0,_) -> true
13   | (_,0) -> true
14   | _ -> false;;

```

Les motifs peuvent être de la forme suivante :

- identifiant	filtre tout
constante (0, 1, true...)	filtre tout, en effectuant une liaison locale (comme let .. in)
[]	Une constante ne filtre qu'elle-même
[motif1;motif2;...;motifn]	filtre la liste vide
motif1 :: motif2	filtre une liste à <i>n</i> éléments
(motif1,motif2,...,motifn)	filtre une liste de tête <i>motif1</i> , de queue <i>motif2</i>
Constructeur argument	filtre un <i>n</i> -uplet
{ c1=motif1; ... cn=motifn }	pour filtrer un type somme
motif1 motif2	pour filtrer un type enregistrement
motif as identifiant	pour filtrer <i>motif1</i> ou <i>motif2</i>
motif when condition	pour filtrer <i>motif</i> tout en le liant
	pour rajouter une condition booléenne

Quelques nouveaux exemples :

```

1 let string_of_bool b = match b with
2   | true -> "true"
3   | false -> "false;;"
4
5 let égal_ou_nul couple = match couple with
6   | (x,y) when x=y -> true
7   | (0,_) | (_,0) -> true
8   | _ -> false;;

```

Un peu de sucre syntaxique : `function x -> match x with | motifs -> résultats` peut être réécrit plus simplement `function | motifs -> résultats`.

```
1 let rec fibonacci = function
2   | 0 | 1 -> 1
3   | n -> fibonacci(n-1) + fibonacci(n-2);;
```

Il est aussi possible de filtrer dans les *let-bindings* :

```
1 let (x,_,[y,_]) = (3,5,[2;3]);;
```

et dans les arguments de fonctions :

```
1 let mange_milieu (a,_,b) = (a,b);;
```

5.6 Gestion des exceptions

Le mot-clé `exception` permet de définir une nouvelle exception :

```
1 exception Erreur_perso of int;;
```

La commande `raise` : `exn -> 'a` permet de lever une exception. On dispose aussi de la commande `failwith` : `string -> 'a` qui lève l'exception `Failure chaîne_de_caractères`.

```
1 raise (Erreur_perso 3);;
2
3 failwith "truc";;
```

La structure `try expression_test with motif -> expression` permet de rattraper des exceptions :

```
1 try .... with
2   | Erreur_perso n -> ...
3   | Failure st -> ...
4   | x -> raise x (* exception non rattrapée *)
```

Le mécanisme des exceptions sera expliqué en détail plus loin dans le cours.