

## Expression Logique et Fonctionnelle ... Évidemment

### Programmation fonctionnelle, filtrage

#### Objectifs du TP

- Déclarations de types par le programmeur ;
- types sommes, enregistrements ;
- définition de fonctions par filtrage ;
- programmation de Core ML en CAML : la syntaxe.

## 1 Déclaration de types

Comme beaucoup d'autres langages de programmation, OCAML offre au programmeur la possibilité de définir ses propres types de données à partir des types de base. Pour cela, il faut utiliser le mot clé **type** suivi du nom du type à définir et de la définition de ce type.

```
# type identificateur = string ;;
type identificateur = string
```

Il est même possible de déclarer simultanément plusieurs types, leurs définitions faisant référence à d'autres types en cours de déclaration.

```
# type identificateur = string
  and valeur = int
  and variable = identificateur * valeur
  and environnement = variable list ;;
type identificateur = string
and valeur = int
and variable = identificateur * valeur
and environnement = variable list
```

Supposons que nous voulions définir une fonction qui à partir d'un identificateur, d'une valeur et d'un environnement construit un nouvel environnement dans lequel une variable a été ajoutée, c'est-à-dire une fonction dont le type est

identificateur -> valeur -> environnement -> environnement.

Nous pourrions écrire :

```
let ajoute x v e = (x,v)::e ;;
```

Malheureusement, cette définition donne une fonction dont le type est plus général que celui désiré :

```
# let ajoute x v e = (x,v)::e ;;
val ajoute : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

Cela provient du mécanisme de type qui ne peut pas déduire de la définition de la fonction que **x** doit être de type **identificateur**, etc... même si c'est pourtant la volonté du programmeur.

Une solution consiste à laisser le soin au programmeur de typer la fonction qu'il définit :

```
# let ajoute (x : identificateur)
  (v : valeur)
  (e : environnement) : environnement =
  (x,v)::e ;;
val ajoute : identificateur -> valeur -> environnement -> environnement = <fun>
```

ou plus simplement encore

```
# let ajoute x v e : environnement = (x,v)::e ;;
val ajoute : identificateur -> valeur -> environnement -> environnement = <fun>
```

Il existe d'autres solutions pour permettre une reconnaissance syntaxique du type des expressions lorsque ce type est défini par le programmeur, sans avoir recours au typage explicite : les types sommes et les enregistrements.

## 2 Types sommes

Un *type somme* est un type déclaré par énumération des différentes *formes* que peuvent prendre les valeurs de ce type. Les formes sont déclarées au moyen de *constructeurs* dont les identificateurs en CAML doivent nécessairement débuter par une lettre majuscule.

Ainsi, le programmeur peut (s'il en éprouve le besoin) déclarer un type **booléen** n'ayant que deux valeurs **Vrai** et **Faux**.

```
# type booléen = Vrai | Faux ;;
type booléen = Vrai | Faux
# Vrai ;;
- : booléen = Vrai
# Faux ;;
- : booléen = Faux
```

Une telle déclaration de type correspond à ce qu'on appelle dans d'autres langages de programmation à un type énuméré. Le nombre de valeurs de types ainsi déclarés est fini.

En CAML, les types sommes ne se limitent pas à des types finis. Les constructeurs ne sont pas nécessairement constants, mais peuvent être paramétrés par d'autres types. Par exemple, le type **int\_etendu** étend le type **int** avec un constructeur non paramétré **Non\_defini**, et un constructeur, **Int**, paramétré par les **int**.

```
# type int_etendu = Non_defini | Int of int ;;
type int_etendu = Non_defini | Int of int
# Non_defini ;;
- : int_etendu = Non_defini
# Int 3 ;;
- : int_etendu = Int 3
# Int (3+2) ;;
- : int_etendu = Int 5
```

### Exercice 1

Programmez la fonction **quotient** de type **int -> int -> int\_etendu** qui donne le quotient entier des deux entiers passés en paramètres lorsque le second n'est pas nul, et la valeur **Non\_defini** dans le cas contraire. On doit avoir

```
# quotient 17 3 ;;
- : int_etendu = Int 5
# quotient 17 0 ;;
- : int_etendu = Non_defini
```

Les types sommes permettent même de définir des structures de données récursives. Par exemple, pour définir un type pour les listes d'entiers, si on se souvient qu'une liste d'entiers est soit la liste vide, soit un couple dont le premier élément est un entier et le second une liste d'entiers, on peut écrire la déclaration :

```
# type liste_entiers = Vide | Cons of (int*liste_entiers) ;;
type liste_entiers = Vide | Cons of (int * liste_entiers)
# Vide ;;
- : liste_entiers = Vide
# Cons (1,Vide) ;;
- : liste_entiers = Cons (1, Vide)
```

### Exercice 2

Construisez en CAML une valeur de type **liste\_entiers** représentant la liste (1,2,3,1).

## 3 Filtrage

Comment peut-on définir des fonctions dont les paramètres sont d'un type somme? Par exemple, comment définir les opérateurs logiques usuels sur notre type **booléen**? Comment définir l'arithmétique usuelle sur notre type **int\_etendu**? Comment calculer la longueur d'une liste?

Lorsque le type somme ne contient que des constructeurs d'arité 0 (ou encore non paramétrés), une simple énumération des cas peut suffire. Par exemple, on peut programmer l'opérateur **et** en écrivant

```
# let et a b =
  if a = Vrai then
    b
  else
    Faux ;;
val et : boolean -> boolean -> boolean = <fun>
# et Vrai Faux ;;
- : boolean = Faux
# et Vrai Vrai ;;
- : boolean = Vrai
# et Faux Vrai ;;
- : boolean = Faux
# et Faux Faux ;;
- : boolean = Faux
```

Mais ce style de programmation ne convient plus dès qu'un des constructeurs du type a une arité au moins égale à 1. En effet comment extraire la valeur du paramètre d'un tel constructeur ?

La réponse réside dans le *filtrage de motifs* (*pattern matching* en anglais). En CAML, c'est l'expression `match ... with ...` qui permet d'effectuer du filtrage de motifs. Sa syntaxe est

```
match expression with
| motif1 -> expr1
| motif2 -> expr2
...
| motifn -> exprn
```

Voici comme premier exemple, une définition de notre opérateur logique **et** à l'aide du filtrage des quatre motifs possibles pour les deux booléens.

```
let et a b =
  match a,b with
  | Vrai, Vrai -> Vrai
  | Vrai, Faux -> Faux
  | Faux, Vrai -> Faux
  | Faux, Faux -> Faux ;;
```

En utilisant le motif universel `_`, satisfait par toutes les instances, on peut écrire de manière plus concise :

```
let et a b =
  match a,b with
  | Vrai, Vrai -> Vrai
  | _ -> Faux ;;
```

ce qui signifie : lorsque les deux paramètres sont **Vrai** alors on renvoie **Vrai** et dans tous les autres cas on renvoie **Faux**.

**Remarque 1 :** La première version de la fonction **et** énumère quatre motifs deux à deux incompatibles, tandis que la deuxième version ne donne que deux motifs non incompatibles : le second (motif universel) contient le premier. L'incompatibilité ou non des motifs dans une énumération d'un filtrage a une conséquence sur l'ordre d'écriture des motifs. Dans le premier cas (incompatibilité des motifs), l'ordre importe peu. Dans le second cas (non incompatibilité des motifs), l'ordre est important. Par exemple, si on écrit

```
# let et a b =
  match a,b with
  | _ -> Faux
  | Vrai, Vrai -> Vrai ;;
Warning U: this match case is unused.
# et Vrai Vrai =
- : boolean = Faux
```

nous pouvons remarquer un message d'avertissement de l'interprète qui prévient qu'un motif est inutile. De plus la fonction ainsi définie est fautive.

**Remarque 2 :** Si l'énumération des différents cas d'un filtrage n'est pas exhaustif, l'interprète avertit le programmeur en donnant un exemple de motif non couvert par l'énumération.

```
# let et a b =  
  match a,b with  
  | Vrai, Vrai -> Vrai  
  | Vrai, Faux -> Faux  
  | Faux, Vrai -> Faux ;;  
Warning P: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
(Faux, Faux)  
val et : boolean -> boolean -> boolean = <fun>
```

**Remarque 3 :** L'expression `match` peut être utilisée dans tous contextes. Dans les exemples qui précèdent, elle est utilisée dans le contexte d'une définition de fonctions à deux paramètres, le filtrage portant sur les deux paramètres. Dans le contexte d'une définition d'une fonction à un seul paramètre, une autre écriture du filtrage de motif est possible qui n'utilise pas l'expression `match` :

```
let une_fonction = function  
  | motif1 -> expr1  
  | motif2 -> expr2  
  ...  
  | motifn -> exprn
```

En fait toutes les fonctions que l'on définit en OCAML le sont par filtrage de motifs.

### Exercice 3

Programmez la négation avec du filtrage de motifs sans utiliser l'expression `match`.

Voyons maintenant comment par le filtrage de motifs, on peut extraire des parties d'une valeur d'un type somme. Nous allons l'illustrer sur la fonction calculant la somme de deux `int_etendu`. Si les deux valeurs sont de la forme `Int n` et `Int p` alors la somme est de la forme `Int (n+p)`. Dans le cas contraire, la somme est `Non_definie`.

```
let plus n1 n2 =  
  match n1,n2 with  
  | Int n, Int p -> Int (n+p)  
  | _ -> Non_defini
```

### Exercice 4

Reprenez la fonction `quotient` pour l'étendre aux `int_etendu`. Le type de la fonction doit maintenant être

`int_etendu -> int_etendu -> int_etendu.`

### Exercice 5

Définissez une fonction calculant la longueur d'une liste d'entiers. Son type est

`liste_entiers -> int.`

```
# longueur Vide ;;  
- : int = 0  
# longueur (Cons (1,Vide)) ;;  
- : int = 1  
# longueur (Cons (2, Cons (1,Vide))) ;;  
- : int = 2
```

**Remarque 4 :** Dans un motif, on ne peut pas utiliser plusieurs occurrences de la même variable de motif. Par exemple, pour définir l'égalité de deux `int_etendu` on pourrait être tenté d'écrire

```
let egal n1 n2 =  
  match n1,n2 with  
  | Int n, Int n -> Vrai  
  | Non_defini, Non_defini -> Vrai  
  | _ -> Faux ;;
```

mais ce code produit le message d'erreur `Variable n is bound several times in this matching.`

Si on tient malgré tout à suivre ce schéma de motif, on peut écrire

```
let egal n1 n2 =  
  match n1,n2 with  
  | Int n, Int p -> if n=p then Vrai else Faux  
  | Non_defini, Non_defini -> Vrai  
  | _ -> Faux ;;
```

ou bien utiliser la notion de *garde*

```
let egal n1 n2 =  
  match n1,n2 with  
  | Int n, Int p when n=p -> Vrai  
  | Non_defini, Non_defini -> Vrai  
  | _ -> Faux ;;
```

### Exercice 6

Définissez une fonction de type `int -> liste_entiers -> int_etendu` qui donne le rang de l'entier passé en premier paramètre dans la liste passée en second paramètre si cet entier y figure, qui donne la valeur `Non_defini` dans le cas contraire.

```
# rang 1 Vide ;;  
- : int_etendu = Non_defini  
# rang 1 (Cons (1,Vide)) ;;  
- : int_etendu = Int 0  
# rang 1 (Cons (2, Cons (1,Vide))) ;;  
- : int_etendu = Int 1
```

## 4 Enregistrements

Les enregistrements sont une façon de représenter des types produits qui diffèrent des  $n$ -uplets sur plusieurs points :

1. les différentes composantes d'un enregistrement sont nommées ;
2. l'ordre dans lequel les composantes sont précisées n'est pas important ;
3. dans un filtrage de motifs, il n'est pas toujours nécessaire de faire figurer toutes les composantes.

Par exemple, si on veut manipuler des dates représentées sous forme d'un triplet d'entiers, on peut les représenter par des triplets (au sens CAML)

```
type date1 = int*int*int
```

ou bien par un enregistrement

```
type date2 = {jour : int ; mois : int ; annee : int}
```

Si on veut représenter la date du 14 juillet 1789, il n'est pas très clair d'après la définition du type `date1` si dans le triplet on doit mettre le jour en premier suivi du mois lui-même suivi de l'année, ou bien s'il faut adopter l'ordre inverse tel que l'utilisent les anglophones. Il faut lire la documentation du type `date1` (et s'en souvenir) pour le savoir. D'ailleurs, quelque soit l'ordre adopté, l'interprète ne parvient pas à reconnaître un objet de type `date1` mais attribue un type plus général (`int * int * int`).

```
# 14,7,1789 ;;  
- : int * int * int = (14, 7, 1789)  
# 1789,7,14 ;;  
- : int * int * int = (1789, 7, 14)
```

En revanche, avec le type `date2` l'ambiguïté n'existe plus et le type est reconnu.

```
# let prise_bastille1 = {jour=14 ; mois = 7 ; annee = 1789} ;;  
val prise_bastille1 : date2 = {jour = 14; mois = 7; annee = 1789}  
# let prise_bastille2 = {mois = 7 ; annee = 1789 ; jour = 14} ;;
```

```
val prise_bastille2 : date2 = {jour = 14; mois = 7; annee = 1789}
# prise_bastille1 = prise_bastille2 ;;
- : bool = true
```

Bien entendu, une expression d'un type enregistrement doit définir chacune de ces composantes. Dans le cas contraire, un message d'avertissement est produit :

```
# {jour = 13 } ;;
Some record field labels are undefined: mois annee
```

Pour accéder à une composante d'un enregistrement, on utilise la notation pointée.

```
# prise_bastille1.jour ;;
- : int = 14
# prise_bastille1.mois ;;
- : int = 7
# prise_bastille1.annee ;;
- : int = 1789
```

Dans les filtrages de motifs portant sur les enregistrements, il n'est pas nécessaire de mentionner tous les champs de l'enregistrement dans les motifs. Voici par exemple, la fonction `fin_mois` de type `date2 -> bool` qui renvoie la valeur `true` lorsque la date passée en paramètre correspond à une fin de mois, et la valeur `false` dans le cas contraire.

```
let fin_mois = function
| {jour=29 ; mois=2 } -> true
| {jour=28 ; mois=2 ; annee=a} when not(bissextile(a)) -> true
| {jour=30 ; mois=m} when List.mem m [4;6;9;11] -> true
| {jour=31} -> true
| _ -> false ;;
```

Le filtrage utilise deux clauses avec garde :

1. la première garde fait appel à une fonction `bissextile` de type `int -> bool` qui reconnaît les années bissextiles<sup>1</sup>, fonction qui est bien évidemment à définir préalablement ;
2. la seconde utilise la fonction booléenne `List.mem` de type `'a -> 'a list -> bool` qui permet de tester la présence d'un élément dans une liste.

## Exercice 7

Réalisez une fonction qui calcule la date du lendemain d'une date passée en paramètre.

## 5 Syntaxe de Core ML

Le but de cette partie est de débiter l'implantation d'un noyau ML en CAML. On ne s'occupe ici que de la partie syntaxique de ce noyau, l'évaluation et le typage étant reporté aux TP ultérieurs.

On rappelle la grammaire définissant les termes (ou programmes) de Core ML.

$t$	<code>::=</code>	$x$	<i>variable</i>
		<b>f</b>	<i>fonction primitive</i>
		<b>c</b>	<i>constructeur de donnée</i>
		$\lambda x.t$	<i>abstraction (terme fonctionnel <b>function</b> <math>x \rightarrow t</math>)</i>
		$tu$	<i>application (d'une fonction <math>t</math> à un argument <math>u</math>)</i>
		<b>let</b> $x = t$ <b>in</b> $u$	<i>liaison (d'un <math>\lambda</math>-terme <math>t</math> à une variable <math>x</math>)</i>

Pour représenter les termes de Core ML on peut déclarer les types suivants :

```
type nom = Name of string | Int of int
and constant = {nom : nom ; arity : int}
and prim_func = Fun of constant
and prim_constr = Cons of constant
and expression =
```

<sup>1</sup>On rappelle que dans le calendrier grégorien, établi en 1582 par le pape Grégoire, une année est déclarée bissextile si son millésime est divisible par 4 mais pas par 100 sauf si elle est divisible par 400.

Var of string	(* variables *)
Prim of prim_func	(* fonctions primitives *)
Constr of prim_constr	(* constructeurs de donnees *)
Abstr of (string * expression)	(* abstraction *)
App of (expression * expression)	(* application *)
Let of (string * expression * expression)	(* liaison *)

### Exercice 8

Définissez les termes Core ML suivant en CAML

1.  $terme_1 = x$ ;
2.  $terme_2 = 1$ ;
3.  $terme_3 = +$ ;
4.  $terme_4 = \lambda x. (+ \ x \ 1)$ ;
5.  $terme_5 = \lambda x. (+ \ x \ 1) \ 1$ ;
6.  $terme_6 = \lambda x. (+ \ x \ 1) \ y$ ;
7.  $terme_7 = (\lambda x. (* \ x \ x)) ((\lambda x. (+ \ x \ 1)) \ 2)$ ;
8.  $terme_8 = \lambda x. (x \ x)$ ;
9.  $terme_9 = \omega = (\lambda x. (x \ x)) (\lambda x. (x \ x))$ .

### Exercice 9

Réalisez un imprimeur (ou pretty-printer) pour les termes. Cet imprimeur permet de passer de la représentation interne (en CAML) des termes à une représentation externe plus lisible. Les variables, les fonctions primitives et les constructeurs seront imprimés par leur propre nom. Les applications seront parenthésées. Les abstractions seront notées avec le mot `\lambda` suivi du nom de la variable d'abstraction. Et les liaisons seront notées en utilisant la forme `Let ... = ... in ....`