CHAPITRE 10 - FONCTIONNELLES

1. Introduction, rappels

Syntaxe allégée pour l'écriture d'une fonction Au passage, il existe une syntaxe allégée pour définir les fonctions, que nous pourrons utiliser à partir de maintenant :

```
let f x = x+2 ; ;
(* f  : int -> int *)

#f 3 ; ;
(*-  : int = 5 *)
```

Le polymorphisme

Exemple 1

La fonction

```
let rec longueur = function
| x  : : q -> 1 + longueur(q)
| _ -> 0  ; ;
```

a pour type : `'a list -> int`. Le type `'a list` est un type **polymorphe**, et cette fonction est, elle-aussi, qualifiée de polymorphe. Cela signifie que l'on peut l'utiliser sur plusieurs types : `int list, bool list, string list...`
`'a, 'b, 'c...` sont appelés des variables polymorphes.

Lorsque l'on souhaite éviter le polymorphisme d'une fonction, il est possible de contraindre le type d'une expression à l'aide de la syntaxe :

```
(expr  : type)
```

Exemple 2

La fonction `egal`

```
let egal = function (x,y)->x=y ; ;
(* egal  : 'a * 'a -> bool *)
```

ainsi écrite, est polymorphe. Pour l'éviter, nous pouvons ainsi annoter la fonction :

```
let egal = function ((x  : int),(y  : int)) -> x=y ; ;
(*egal  : int * int -> bool *)
```

Plutôt que de *deviner* le type de votre fonction (nous appelons cela **l'inférence de type**), Caml va *vérifier* que le corps de votre fonction est cohérent avec les annotations de types (nous appelons cela **la vérification de type**).

Remarque : L'annotation de type peut également s'avérer très utile, lorsqu'on écrit des fonctions complexes à plusieurs paramètres, pour se rappeler qui est qui. Cf semestre 6...

L'objet de ce chapitre Une des caractéristiques des langages fonctionnels est que les fonctions y sont des expressions à part entière (et non une super-structure comme dans les langages impératifs), cela signifie qu'elles peuvent apparaître dans des listes, des n-uplets, et bien sûr comme paramètre ou résultat d'une fonction !

Pour bien comprendre ce chapitre, il est important de **bien regarder les types** !!

2. Fonctionnelles

2.1. Définition et premiers exemples

Définition 1

Nous appelons **fonctionnelle (ou fonction d'ordre supérieur), une fonction dont les arguments et/ou le résultat sont eux-mêmes des fonctions.**

Remarque : Ces fonctions peuvent être polymorphes.

Exemple 3

La fonction `fois_n` fabrique la fonction qui multiplie par n :

```
let fois_n = function n -> (function m -> n*m)  ; ;
(* fois_n  : int -> int -> int *)
```

Comprenez vous le type retourné par Caml ?
`fois_n` est bien une fonction : appliquée à un argument, le résultat est une fonction.

```
let double = fois_n (2) ; ;
(* double  : int -> int *)
```

Cette fonction doit à son tour être appliquée à un argument pour obtenir une valeur entière :

```
double (3) ; ;
-  : int = 6

fois_n(2)(3) ; ;
-  : int = 6
```

Attention ! Source d'erreur :

- Le constructeur de types `->` est prioritaire à droite : `int -> int -> int` équivaut à `int -> (int -> int)`.
- En revanche, l'opérateur d'application des fonctions est prioritaire à gauche : `fois_n 2 3` équivaut à `(fois_n 2) 3`.

Exemple 4

Ecrivons une fonction qui étant donnée une fonction f passée en paramètre, construit la fonction qui à x non nul, renvoie $\frac{f(x)}{x}$:

```
#let divx= function f-> (function 0.-> failwith "x nul"
|x->f(x)/.x) ; ;
(* divx  : (float -> float) -> float -> float *)
```

Comprenez-vous le type de `divx` ??
Voici des exemples d'utilisation :

```
#divx(sin) ; ;
-  : float -> float = <fun>

#divx(sin)(0.1) ; ;
-  : float = 0.998334166468

#let divsin=divx(sin) ; ;
divsin  : float -> float = <fun>

#divsin(0.1) ; ;
-  : float = 0.998334166468
```

Exemple 5

Autre exemple avec des chaînes de caractères :

```
#let star = function f-> function ch->"**"^f(ch)^"**" ; ;
star  : ('a -> string) -> 'a -> string = <fun>

#let duplique = function ch->ch^ch ; ;
#let dupliqueEtStar = star (duplique) ; ;


#dupliqueEtStar("bonjour") ; ;
-  : string = "**bonjourbonjour**"


#let convertirEtStar = star(string_of_int) ; ;
#convertirEtStar (1234) ; ;
-  : string = "**1234**"
```

2.2. Définition d'une fonction à plusieurs paramètres

Une fonctionnelle étant une fonction comme une autre, il est toujours possible d'utiliser la syntaxe classique :

```
function a_1 -> function a_2 -> ... function a_n -> Corps
```

où `Corps` est le corps de la fonction de type `t` non fonctionnel.
Il est possible d'utiliser également la syntaxe allégée :

```
function a_1 a_ 2 ... a_n -> Corps
```

Exemple 6

Observer la différence des types :

```
let foisN1 = function (n,a) -> n*a  ; ;
(* foisN1  : int*int -> int *)
(* Il s'agit d'une fonction à un seul paramètre, qui est un couple *)

let foisN2 = function n -> function a -> n*a  ; ;
(* foisN2  : int -> int -> int *)
(* Il s'agit cette fois, d'une fonctionnelle *)
```

Nous pouvons écrire également :

```
let foisN2 = function n a -> n * a  ; ;
```

ou encore :

```
let foisN2 n a = n*a  ; ;
```

Attention toutefois, ces deux dernières syntaxes ne permettent pas le filtrage.

2.3. Application totale ou partielle

Définition 2

L'application totale d'une fonction à n paramètres nécessite n arguments. Le résultat n'est plus une fonction mais une valeur de type `t`.

Exemple 7

```
let divise = function n -> function
0 -> failwith "division par 0"
|p -> n / p  ; ;
(* divise  : int -> int-> int *)
```

L'appel de fonction étant prioritaire, le parenthésage est inutile :

```
divise 6 3  ; ;
-  : int = 2
```

Définition 3

Tout l'intérêt d'une fonctionnelle réside dans la possibilité d'utiliser des applications partielles. On ne fournit alors pas tous les arguments à la fonction et le résultat est une fonction.

Exemple 8

```
#let div10 = divise 10 ; ;
val div10  : int -> int = <fun>

#div10 3 ; ;
-  : int = 3
```

Important : Il est donc fondamental de choisir avec soin l'ordre des paramètres pour pouvoir définir les applications partielles intéressantes.

2.4. Curry de Caml

Une fonction à plusieurs arguments peut donc prendre deux formes :

```
let f = function (x,y) -> 2*x + 3*y  ; ;
(* f  : int * int -> int *)
```

Cette forme est dite **non curryfiée**.

```
let f = function x y -> 2*x + 3*y  ; ;
(* f  : int -> int -> int *)
```

S'il est intéressante d'écrire des applications partielles, alors cette deuxième forme est préférable. Elle est dite **curryfiée**, en hommage au mathématicien et logicien américain Haskell Curry (1900 - 1982).
Remarque : Il est toujours possible de passer d'une forme à l'autre :

```
let curry = function f x y -> f(x,y)  ; ;
let ecurry = function f (x,y) -> f x y  ; ;
```

Exercice . Exercice TP 1

1. Déterminer le type de la fonction suivante et tester :

   ```
   let fzz = function f a -> f(a+1) = 0  ; ;
   ```

2. Pour chaque requête suivante, donner la réplique Caml et commenter

   ```
   fzz(3)  ; ;
   let g = fzz(function n -> 3)  ; ;
   g 4  ; ;
   ```

3. Construire un exemple de fonction de chaque type suivant :
   (a) `int -> int -> bool`
   (b) `(int -> int) -> bool`

(c) `int -> bool -> float`

(d) `(int -> bool) -> float`

3. Fonctionnelles et récursivité

Exercice . Exercice TP 2

Nous appelons **prédicat** une fonction p qui renvoie un booléen.

1. Ecrire la fonctionnelle `ilexiste : ('a -> bool) -> 'a list -> bool` qui, étant donnés un prédicat p et une liste ℓ, retourne Vrai si au moins l'un des éléments de ℓ satisfait la propriété p et Faux sinon.
2. Ecrire la fonctionnelle `qqsoit : ('a -> bool) -> 'a list -> bool` qui, étant donnés un prédicat p et une liste ℓ, renvoie Vrai si tous les éléments de ℓ satisfont la propriété p, Faux sinon.
3. En déduire une fonction `estMembre :'a -> 'a list -> bool` qui teste l'appartenance d'un élément à une liste ; et `estInclus : 'a list -> 'a list -> bool` qui teste l'inclusion d'un ensemble dans un autre lorsque les deux ensembles sont stockés sous forme de liste.
4. Ecrire la fonctionnelle `filtrer : ('a -> bool) -> 'a list -> 'a list` qui retourne la liste des éléments de ℓ qui satisfont la propriété p.
5. En déduire une fonction `diffEns : 'a list -> 'a list -> 'a list` qui, étant donnés deux ensembles stockés sous forme de listes ℓ_1 et ℓ_2, retourne la liste des éléments de ℓ_1 n'apparaissant pas dans ℓ_2.

Un petit évaluateur Caml

Exercice . TP 3

Nous avons étudié les modes d'évaluation des différentes expressions Caml. Le but de cet exercice est d'écrire un petit interpréteur Caml reconnaissant quelques expressions arithmétiques en nombre entier et les définitions globales et locales. Nous proposons d'écrire un évaluateur pour cette version très allégée de Caml que nous appellerons *Caml O'Chocolat*.

Les expressions Caml O'Chocolat En Caml O'Chocolat, une expression peut être une constante entière, un identificateur (réduit à un seul caractère), la somme de deux expressions, le produit de deux expressions ou une expression à une certaine puissance entière. Nous définissons donc le type expression comme suit :

```
type Expression = Const of int | Var of char
| Add of Expression*Expression
| Mult of Expression*Expression
| Puiss of Expression*int ; ;
```

1. Définir deux expressions e_1 et e_2 représentant respectivement $1+2x^3$ et $1+a^2$.

   Pour évaluer nos expressions, nous devons disposer d'un environnement courant. On définit une liaison entre un identificateur et la valeur d'une expression par le type produit suivant :

   ```
   type liaison = {id   : char ; valeur   : int} ; ;
   ```

   On peut alors définir un environnement comme une liste de liaisons. On pourra supposer qu'initialement, notre environnement courant, que nous noterons `envC` est :

   ```
   let envC = [{id='a' ;valeur=3} ;{id='b' ;valeur=4}] ; ;
   ```

2. Écrire une fonction `evalVar : char * liaison list -> int` qui, à un caractère c et à un environnement *env* associe la valeur entière liée à c dans l'environnement *env* si cette liaison existe et un message d'erreur sinon.

   ```
   evalVar ('a',envC) ; ;
   -  : int = 3
   evalVar ('b',envC) ; ;
   -  : int = 4
   evalVar ('x',envC) ; ;
   Exception non rattrapée : Failure " identificateur inconnu"
   ```

3. Écrire une fonction `puissance : int * int -> int` qui calcule (naïvement) la puissance entière puissance entière positive d'un nombre entier. On affichera une erreur si la puissance est strictement négative.
4. Écrire une fonctionnelle curryfiée `evalExp :liaison list -> Expression -> int` qui, à un environnement et une expression donnés associe la valeur de cette expression dans l'environnement ou un message d'erreur si l'expression contient un identificateur

non lié.
On définira une fonction récursive auxiliaire locale ne portant que sur l'expression à évaluer et on pourra bien sûr utiliser `evalVar` et puissance.

Définitions globales et locales : Une définition crée la liaison entre un identificateur et la valeur d'une expression. Nous représentons une définition par le type produit suivant :

```
type Definition={ident :char  ; exp : Expression } ; ;
```

1. Écrire une fonction `ajoute : liaison list -> Definition -> liaison list` qui, à un environnement *env* et à une définition *d* associe le nouvel environnement obtenu en ajoutant à *env* la liaison entre l'identificateur et la valeur de l'expression dans l'environnement *env*.

```
ajoute envC {ident='x' ;exp = Add(Var 'a',Const 3)} ; ;
-  : liaison list =
[{id = 'x' ; valeur = 6} ; {id = 'a' ; valeur = 3} ; {id = 'b' ; valeur = 4}]
```

Programmes Caml O'Chocolat Un programme Caml O'Chocolat peut être soit une expression, soit une définition globale, soit une définition locale permettant la réalisation d'une définition le temps de l'évaluation d'un programme. On définit donc le type récursif suivant :

```
type Programme =
Elementaire of Expression
| DefGlob of Definition
| DefLocale of Definition*Programme ; ;
```

Le petit programme Caml O'Chocolat suivant :

```
Soit x=7 dans
Soit y = x + 3 dans x + 3*y  ; ;
```

sera alors représenté par l'objet de type `Programme` suivant :

```
let p  = DefLocale ({ident = `x` ; exp = Const 7},
DefLocale ({ident = `y` ; exp = Add (Var `x`, Const 3)},
Elementaire (Add (Var `x`, Mult (Const 3, Var `y`)))) ; ;
```

1. Écrire une fonction `evalProg : Programme * liaison list -> int * liaison list` qui, à un programme et un environnement *env* donnés, associe la valeur de ce programme dans l'environnement *env* (la valeur affichée par la réplique Caml) et le nouvel environnement.

```
Pour le programme  : 1+a2  ; ;
#evalProg( Elementaire e2,envC) ; ;
```

```
-  : int * liaison list = 10, [{id = 'a' ; valeur = 3} ; {id = 'b' ; valeur = 4}]


Pour le programme  : soit x = 1+a2  ; ;
#evalProg( DefGlob {ident='x'  ; exp=e2 },envC) ; ;
-  : int * liaison list =
10, [{id = 'x' ; valeur = 10} ; {id = 'a' ; valeur = 3} ; {id = 'b' ; valeur = 4}]


Pour le programme  : soit x = 7 dans soit y = x+3 dans x+3*y  ; ;
#evalProg ( p ,envC) ; ;
-  : int * liaison list = 37, [{id = 'a' ; valeur = 3} ; {id = 'b' ; valeur = 4}]
```

Exercice . Exercice TP 4

1. Écrire une fonction `sigma : (int -> int) -> int -> int` récursive permettant de calculer :

$$\sum_{i=0}^{n} f(i)$$

pour une fonction f et un entier n quelconques.

```
# sigma (function i-> i*i) 10 ; ;
-  : int = 385
```

Cette version possède l'inconvénient que la récursivité porte sur n et sur f, alors que f ne change pas. L'application partielle est donc recalculée à chaque appel récursif.

2. Redéfinir la fonction `sigma`, non récursive, mais possédant une fonction récursive définie localement et ne portant que sur le paramètre n.

4. Bonus : Les exceptions

Il arrive souvent que l'on veuille écrire des fonctions qui ne sont pas définies partout. Dans de tels cas, comme toute expression Caml doit avoir une valeur, il nous faut un moyen pour signaler qu'une fonction ne peut pas être calculée : on appelle cela des exceptions.

4.1. Exceptions prédéfinies de Caml

```
1/0  ; ;
Exception  : Division_by_zero
```

Nous disons alors qu'une exception a été levée. L'évaluation normale de l'expression en cours est stoppée, et l'exécution s'arrête en affichant l'exception.
La fonction prédéfinie (que nous connaissons déjà !)

`failwith`

permet de lever l'exception `Failure`, accompagnée de la chaîne de caractères passée en paramètre.

```
let f = function
0 -> failwith "pas d'argument nul"
| x -> 1./.x  ; ;

f(0.)  ; ;
Exception non rattrapée  : Failure "pas d'argument nul"
```

4.2. Exceptions déclarées par le programmeur

Il est également possible de déclarer de nouvelles exceptions, grâce à la syntaxe :

```
exception NomException  ; ;
```

Attention En OCaml, les noms d'exceptions devront commencer par une majuscule !
Ensuite, une fonction peut lever une telle exception grâce à la syntaxe :

```
raise NomException  ; ;
```

Exemple 9

```
#exception NombreNegatif ; ;

#let rec fact = function
0->1
| n-> if n<0 then raise NombreNegatif
else n*fact(n-1) ; ;

(* fact  : int -> int = <fun> *)

#fact(-3) ; ;
Exception  : NombreNegatif
```

Exercice . TP 4

Écrire une fonction `associe : 'a -> ('a * 'b) list -> 'b` qui, à un élément a et une liste de couples $liste$ associe b si le couple (a,b) appartient à $liste$ et lève une exception "NonTrouvé" sinon.

4.3. Capturer une exception

Nous avons vu que la levée d'une exception interrompait l'évaluation normale. Ce comportement est parfois gênant : on voudrait pouvoir parfois appeler des fonctions qui risquent de lever des exceptions, et dans le cas où une exception est levée, faire un traitement particulier. Ceci peut se faire à l'aide de la construction de capture d'une ou plusieurs exceptions, qui a la forme suivante (analogue au filtrage) :

```
try expression with
exception_1 -> expression_1
|exception_2 -> expression_2
...
|exception_n -> expression_n
```

et qui s'évalue en `expression_i` si l'évaluation de *expression* lève l'exception `exception_i`. Toutes ces expressions doivent avoir le même type.

Exemple 10

```
fact(-3) ;;
Exception : NombreNegatif

try fact(-3) with
NombreNegatif -> 0 ;;
- : int = 0
```

Exercice

Nous définissons :

```
let dico= [("a","un") ; ("called","appelé") ;
("cat","chat") ; ("hand","main") ;
("is","est") ; ("language","langage") ;
("my","mon") ; ("us","nous") ;
("wonderful","magnifique")] ;;
```

1. Définir une exception `PasTrouve`.
2. Ecrire une fonction `trouve : 'a -> ('a * 'b) list -> 'b` qui cherche la traduction d'un mot à l'aide du dictionnaire, l'exception `PasTrouve` sera levée en cas d'échec.

   ```
   #trouve "cat" dico ;;
   - : string = "chat"

   #trouve "dog" dico ;;
   Exception : PasTrouve
   ```

3. Ecrire une fonction `traduire : string -> string` permettant de traduire un mot si on le trouve dans le dictionnaire, et rattrapant l'exception `PasTrouvé` en laissant le mot en anglais.