# L2/LD – TP 1 – Initiation à Objective Caml

Le but de ce TP est de vous initier à la partie fonctionnelle du langage Caml et de vous fournir les outils dont vous aurez besoin pour réaliser les TPs suivants.

Dans toute la suite de ce document, les lignes commençant par " À taper : " sont les lignes que vous devez taper et les lignes commençant par "Réponse : " sont les réponses de l'interpréteur.

## 1 L'interpréteur Caml

Il y a deux façons d'utiliser Caml : l'interpréteur et le compilateur. L'interpréteur lit des expressions et des déclarations de fonctions, les évalue immédiatement et affiche les résultats.

On utilise l'interpréteur dircetement dans un terminal en tapant la commende ledit ocaml. L'utilisateur ledit permet d'utiliser les flèches pour se déplacer dans le ligne de commande ou de rappemer une ligne de l'historique.

Une fois OCAml lancé, un prompt (ou invite de commande) # et le curseur s'affichent. On entre le programme et on valide avec **entrée**. On quitte avec **CTRL-D**.

## 2 Premier contact avec Caml

Dans cette première partie, l'interpréteur est une calculatrice : vous donnez des expressions et vous les évaluez. **Attention**, pour déclencher l'évaluation d'une expression en Caml, il faut la conclure par `;;` .

Par exemple,

À taper : `34*56;;`
Réponse : *`- : int = 1904`*

La réponse de l'interpréteur est de la forme

`nom : type = valeur`

qui signifie : la variable *nom* (ici `-` signifie *pas de nom*) est de type `type` (ici `int`) et a pour valeur `valeur` (ici 1904).

On peut donner un nom à cette valeur en utilisant le mot clé `let` :

À taper : `let x=34*56;;`
Réponse : *`val x : int = 1904`*

`x` désigne alors un entier qui a pour valeur 1904.

`x` est désormais connu et on peut l'utiliser par la suite :

À taper : `x*2;;`
Réponse : `- : int = 3808`

## 3 Types définis

- entiers (*int*), opérateurs : `+ - * /`
- réels (*float*), opérateurs : `+. -. *. /.`
- caractères (*char*) : `'a', 'b', '1'...`
- chaînes (*string*) : `"Bonjour", "ab"` ; opérateur de concaténation : `^`
- booléens (*bool*) : `true, false` ; opérateurs : `or, &&, not`

À taper : `1. +. 2.5 *. 3.78;;`
Réponse : `- : float = 10.45`

Notez que les opérateurs `+, -, *, /` ne s'appliquent qu'aux entiers, et les opérateurs `+., -., *., /.` ne s'appliquent qu'aux flottants. Caml étant un langage *fortement typé*, il n'y a aucune conversion automatique entre entiers et flottants.

À taper : `1 + 2.5 ;;`
Réponse : `This expression has type float but is here used with type int`

À taper : `1<2;;`
Réponse : `- : bool = true`
À taper : `"ga" = "bu";;`
Réponse : `- : bool = false`
À taper : `"aab" < "aac";;`
Réponse : `- : bool = true`

L'interpréteur Caml refuse de faire n'importe quoi, comme l'illustrent les deux exemples suivants :

À taper : `x*"une chaine de caracteres";;`
Réponse :
```
Characters 2-28:
  x*"une chaine de caracteres";;
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
This expression has type string but is used with type int
```

car l'expression soulignée devrait être un entier.

À taper : `x*y;;`
Réponse :
```
Characters 2-3:
  x*y;;
    ^
Unbound value y
```

car la variable `y` n'a pas été définie (unbound).
Toutes les variables doivent être définies (par le mot clé `let`) **avant** d'être utilisées.

## 4 Un programme Caml

Un *programme* Caml est une suite de définitions suivie d'une expression à évaluer. Les commentaires doivent être placés entre `(*` et `*)`.

```
À taper :   let x=2   (* definition de x *);;
À taper :   let y=3   (* definition de y *);;
À taper :   let z=x*y (* definition de z utilisant les valeurs de x et de y *);;

À taper :   x*y*z (* expression a evaluer *);;
Réponse :   - : int = 36
```

**Exercice :** Proposez un programme qui résoud le problème suivant : combien vaut $(3157 * h) * w$ sachant que $h = 543 * w$ et $w = 19$ ?

# 5 Définition locale

La construction `let...in...` permet de créer un nom local. On peut créer simultanément plusieurs noms locaux avec la construction `let...and...and...in....`

```
À taper :   let a = 22 + 47 in a / 5 ;;
Réponse :   - : int = 13
À taper :   let b = 3 and c = 8 in b + c;;
Réponse :   - : int = 11
À taper :   c+2;;
Réponse :   Error: Unbound value c
```

Une fois le calcul `b+c` fini, `c` n'est plus lié à 8.

# 6 Inclusion de fichiers

Pour conserver des déclarations de types, de variables, de fonctions ..., on peut les écrire dans des fichiers. On peut alors inclure ces fichiers dans d'autres fichiers, comme dans l'exemple de la figure 1.

```
#use "chemin/du/dossier/contenant/le/fichier/tp1.ml";;
```

FIGURE 1 – Inclusion du fichier `tp1.ml` dans un autre fichier Caml

Cette fonctionnalité permet de réutiliser l'ensemble des types, fonctions et variables définis dans les fichiers inclus. A noter que cette inclusion se propage. Si vous incluez dans un fichier C un fichier A incluant un fichier B, alors le fichier B sera également inclus dans le fichier C.

# 7 Définition de fonctions

On peut définir une fonction $f$ qui prend $n$ arguments de la manière suivante :

$$\texttt{let}\ (f : t_1 * \cdots * t_n \to t) = \texttt{function}\ (v_1, \ldots, v_n) \to \textit{expression};;$$

FIGURE 2 – Définition de fonction simple

La fonction $f$ prend $n$ arguments $v_1 : t_1$, ..., $v_n : t_n$, et retourne une valeur de type $t$.
Par exemple,

À taper : `let (inc : int -> int) = function x -> x + 1;;`
Réponse : *`val inc : int -> int = <fun>`*

L'interpréteur Caml répond que `inc` est maintenant défini comme une fonction qui prend un entier en argument et retourne un entier ; comme l'interpréteur ne peut pas afficher de valeur pour une fonction, il affiche `<fun>`.

À taper : `let (plus : int * int -> int) = function (x,y) -> x + y;;`
Réponse : *`val plus : int * int -> int = <fun>`*

**Exercice :**
- Que vaut `inc 0` ?
- Que vaut `inc (x*3)` ?
- Que vaut `(inc x) * 3` ?
- Que vaut `inc x * 3` ?
- Que vaut `inc` ?
- Que vaut `inc * 3` ?

**Exercice :** Écrire la fonction `double` qui prend en entrée un entier $x$ et retourne $2 * x$.

# 8 Redéfinition n'est pas modification

**Exercice :** Tapez ce qui suit et assurez-vous de comprendre ce qui se passe.

À taper : `let (x : int) = 3`
Réponse : *`val x : int = 3`*

`x` désigne maintenant l'entier 3

À taper : `let (ajouterX : int -> int) = function y -> x + y`
Réponse : *`val ajouterX : int -> int = <fun>`*

En mémoire, `ajouterX` vaut `y -> 3 + y *)`

À taper : `ajouterX 2 ;;`
Réponse : *`- : int = 5`*

À taper : `let x = 4`
Réponse : *`val x : int = 4`*

`x` désigne maintenant l'entier 4

À taper : `ajouterX 3 ;;`
Réponse : *`- : int = 6`*

À taper : `let (ajouterX : int -> int) = function y -> x + y`
Réponse : *`val ajouterX : int -> int = <fun>`*

En mémoire, cela donne : `ajouterX = y -> 4 + y *)`

À taper : `ajouterX 3 ;;`
Réponse : *`- : int = 7`*

# 9 Filtrage

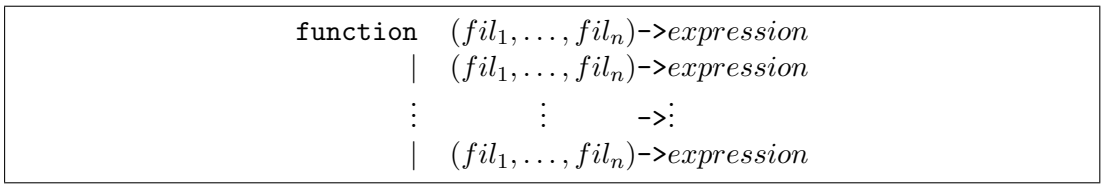


FIGURE 3 – Une fonction peut être définie par filtrage

À taper :
```
let (estIlDAccord : string -> string) =
       function  "o"  -> "Il est d'accord"
              |  "n"  -> "Il n'est pas d'accord"
              |  "O"  -> "Il est d'accord"
              |  "N"  -> "Il n'est pas d'accord"
              |  _    -> "Je n'ai pas compris"
;;
```
Réponse :
```
val estIlDAccord : string -> string = <fun>
```

À taper :
```
estIlDAccord "n";;
```
Réponse :
```
- : string = "Il n'est pas d'accord"
```
À taper :
```
estIlDAccord "o";;
```
Réponse :
```
- : string = "Il est d'accord"
```
À taper :
```
estIlDAccord "quoi?";;
```
Réponse :
```
- : string = "Je n'ai pas compris"
```

**Exercice :** Écrire la fonction `FeuTricolore` qui prend en entrée la couleur du feu sous forme d'une chaîne de caractères et indique à l'automobiliste s'il peut avancer.

# 10 Type indéterminé

Quand on définit, la fonction `identite`

À taper :
```
let identite x = x ;;
```
Réponse :
```
val identite : 'a -> 'a = <fun>
```

rien ne permet de connaître le type de l'argument, ni le type de retour de cette fonction. Par contre, ces deux types sont identiques. OCaml utilise alors le type indéfini `'a` . La fonction `identite` peut prendre des arguments de tout type.

À taper :
```
identite 3;;
```
Réponse :
```
- : int = 3
```

À taper :
```
identite 4.2 +. 2.4 ;;
```
Réponse :
```
- : float = 6.6
```

À taper :
```
identite "bonjour";;
```
Réponse :
```
- : string = "bonjour"
```

**Exercice :** Proposez une fonction qui répond `oui` sur n'importe quelle entrée.

## 11 Fonctions récursives simples

Une fonction $f$ peut être récursive, c'est-à-dire que la définition de $f$ peut utiliser $f$. Il faut prévenir l'interpréteur que l'on veut définir une fonction récursive, en utilisant le mot clé `rec`.

Par exemple,

```
À taper :   let rec (factoriel : int -> int) =
                     function x -> if x <= 1 then 1
                                             else x * factoriel (x-1)
            ;;
Réponse :   val factoriel : int -> int = <fun>
À taper :   factoriel 1;;
Réponse :   - : int = 1
À taper :   factoriel 4;;
Réponse :   - : int = 24
À taper :   factoriel 20;;
Réponse :   - : int = 45350912
À taper :   factoriel 30;;
Réponse :   - : int = -738197504
À taper :   factoriel 67;;
Réponse :   - : int = 0
```

Attention, la récursion risque de définir des programmes qui ne terminent pas.

```
À taper :   let rec (mauvais : int -> int) =
                      function x -> if x <= 1 then 1
                                              else x * mauvais (x+1)
            ;;
Réponse :   val mauvais : int -> int = <fun>
À taper :   mauvais 2;;
Réponse :   Stack overflow during evaluation (looping recursion?).
```

**Exercice :** Définissez récursivement la fonction `sum : int -> int` telle que

$$\texttt{sum}\ n = \sum_{i=0}^{n} i,$$

en utilisant $\texttt{sum}\ 0 = 0$ et $\texttt{sum}\ n = \texttt{sum}(n-1) + n$ si $n > 0$.

**Exercice :** Redéfinir la fonction `sum` vue précédemment en utilisant le filtrage.

## 12 Fonctions mutuellement récursives

Deux fonctions sont dites *mutuellement récursives* si chacune d'elles fait appel à l'autre. Dans ce cas, utiliser les mots clé `rec` et `and` comme dans l'exemple suivant :

```
À taper :   let rec (pair : int -> bool) =
              function  0 -> true
                    |   n -> impair(n-1)
            and (impair : int -> bool) =
```

```
  function  0 -> false
          |   n -> pair(n-1)
;;
```

Réponse :

```
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>
```