Cours de programmation fonctionnelle et logique
Cours 1 : présentation de `Ocaml`

Table des matières

1 Le langage

Le langage qui servira de base à la partie fonctionnelle de ce cours est `Ocaml` (Objective Caml).
Documentation et distribution : `http://caml.inria.fr/`.

2 Exécuter des programmes `Ocaml`

Deux méthodes :
- compilation : classique.
- système interactif : l'interpréteur
  - pour lancer l'interpréteur, taper `Ocaml` ; le signe d'invite « # » apparaît ;
  - entrer l'expression à évaluer ;
  - clore avec la marque de fin de phrase « ;; ».

– exemple :

```
# 2+2;;
  - : int = 4
```

Lecture : le résultat de l'évaluation de l'expression est une valeur ; cette valeur est de type entier ; elle est égale à « 4 ».

L'expression `Ocaml` est juste `2+2;;`. Je n'indique le « # » que parce que je veux aussi présenter la réponse de l'interpréteur.

À noter : il y a déduction (inférence) automatique du type.

3 Les définitions

3.1 Définitions globales

```
# let s = 1+2+3;;
 val s : int = 6
```

Lecture : le résultat de l'évaluation de l'expression est la définition d'un nouvel objet, appelé « s » ; cet objet est de type entier ; sa valeur est égale à « 6 ».

La définition est **permanente** :

– Version compilée : l'objet « s » est défini pour tout le reste du programme.

– Version interprétée : l'objet « s » est défini tant que l'interpréteur n'est pas tué.

Par conséquent, une fois l'objet défini de manière globale, on peut l'évaluer n'importe quand, on peut demander sa valeur à l'interpréteur.

```
# s;;
 - : int = 6
```

```
# let s2 = s * s;;
 val s2 : int = 36
```

On ne spécifie pas forcément de type, mais on peut le faire :

```
# let (t : int) = 1+2+3;;
 val t : int = 6
```

Les **types de base** sont :

– int ;

– float ;

– bool (« true » ou « false ») ;

– char ;

– string.

3.2 Définitions locales

Il s'agit de définitions **temporaires** et non plus **permanentes**.

```
# let u = 20 in u*4;;
 - : int = 80
```

Ici, l'objet « u » n'est défini qu'au sein de l'expression « let u = 20 in u*4;; », et n'est pas défini en dehors. Par conséquent si on interroge l'interpréteur, il répondra qu'il ne connaît pas l'objet en question.

```
# u;;
 Characters 0-1:
Unbound value u
```

Dans une définition locale, on peut réutiliser un identificateur d'un objet défini globalement.

```
#let s = 20 in s*4;;
 - : int = 80
```

La valeur de l'objet global n'est bien évidemment pas modifiée :

```
#s;;
 - : int = 6
```

La définition locale du nom est indépendante du type actuel du nom (si celui-ci est déjà défini globalement).

```
#let s = "Bon" and s2 = "jour" in s^s2;;
 - : string = "Bonjour"
```

Le mot clé `and` permet des définitions multiples et simultanées.

4 Les fonctions

4.1 Définition globale de fonctions

On veut définir une fonction `successeur` qui à l'entier `x` associe l'entier `x+1`.

```
#let successeur (x) = x + 1;;
 val successeur : int -> int = <fun>
```

```
#let successeur x = x + 1;;
 val successeur : int -> int = <fun>
```

Lecture :

- Le résultat de l'évaluation de l'expression est la définition d'un nouvel objet, appelé `successeur`.
- Cet objet est de type `int -> int`, c'est-à-dire une fonction qui prend un entier et renvoie un entier. Cette écriture est à rapprocher de l'écriture mathématique : `successeur` : $\mathbb{N} \to \mathbb{N}$
- C'est une fonction et l'interpréteur, incapable d'afficher sa valeur, indique juste `<fun>`.

Le nom `successeur` a une valeur :

```
#successeur;;
 - : int -> int = <fun>
```

La définition d'une fonction n'est pas fondamentalement différente de la définition d'un entier. De même, la valeur d'une fonction n'est pas fondamentalement différente de celle d'un entier... Les fonctions ont le même statut que les entiers, ce sont des « citoyens de première classe ».

4.2 Application de fonctions

```
#successeur(2);;
 - : int = 3
```

```
#successeur 2;;
 - : int = 3
```

```
#(successeur 2);;
 - : int = 3
```

Les parenthèses sont optionnelles. En général ont ne les mets que si elles sont nécessaires.

4.3 Définition locale de fonctions

```
#let prédécesseur x = x-1 in
    (prédécesseur 3)*(prédécesseur 4);;
   - : int = 6
```

```
#prédécesseur 5;;
```

```
 Characters 0-12:
Unbound value prédécesseur
```

```
#let prédécesseur_carré x =
    let pred_de_x x = x-1
    in (pred_de_x x)*(pred_de_x x);;
     val prédécesseur_carré : int -> int = <fun>
```

```
#prédécesseur_carré 8;;
 - : int = 49
```

4.4 Fonctions à plusieurs arguments

```
#let moyenne a b = (a + b)/2;;
 val moyenne : int -> int -> int = <fun>
```

```
#moyenne 5 3;;
 - : int = 4
```

```
#(moyenne 5 3);;
 - : int = 4
```

```
#moyenne (5 3);;
 Characters 9-10:
This expression is not a function, it cannot be applied
```

```
#(moyenne (5))(3);;
 - : int = 4
```

Fonction sans argument :

```
#let fonction_deux () = 2;;
 val fonction_deux : unit -> int = <fun>
```

```
#let valeur_deux      = 2;;
 val valeur_deux : int = 2
```

4.5 Fonctions anonymes

Ce sont des valeurs fonctionnelles qui sont définies mais pas nommées.

```
#function x -> 2*x+1;;
 - : int -> int = <fun>
```

Application :

```
#(function x -> 2*x+1) 2;;
 - : int = 5
```

Définition mathématique des fonctions

$$\texttt{successeur} : \begin{array}{l} \mathbb{Z} \to \mathbb{Z} \\ x \mapsto x+1 \end{array}$$

Expression `Ocaml` totalement équivalente :

```
#let (successeur : int -> int) = function x -> x+1;;
 val successeur : int -> int = <fun>
```

Écriture effective :

```
#let successeur = function x -> x+1;;
 val successeur : int -> int = <fun>
```

Fonctions à plusieurs arguments

Une fonction anonyme prend en entrée **un seul** argument.

```
#let moyenne = function a -> function b-> (a+b)/2;;
 val moyenne : int -> int -> int = <fun>
```

4.6 Conventions syntaxiques

`let f = function x -> ...`	$\Leftrightarrow$	`let f x = ...`
`f (x)`	$\Leftrightarrow$	`f x`
`(f x) y`	$\Leftrightarrow$	`f x y`
`f (g y)`	$\Leftrightarrow$	`f g y`
`function x -> f x`	$\Leftrightarrow$	`f`

4.7 Les tests et l'alternative

« `if ... then ... else ...` »

```
#let valeur_absolue x = if x >= 0 then x else -x;;
 val valeur_absolue : int -> int = <fun>
```

Les tests calculent un résultat qui est de type booléen :

```
#2 < 1;;
 - : bool = false
```

```
#(valeur_absolue 3) = (valeur_absolue (-3));;
 - : bool = true
```

4.8 Les programmes

Ce sont des fonctions !

Un programme est une fonction qui calcule le résultat souhaité. Une fonction manipule des valeurs typées, une fonction est une valeur typée, les programmes `Ocaml` sont donc des valeurs typées !

5 Programmation impérative

`Ocaml` est loin d'être un langage fonctionnel pur. Il contient en effet les principales structures impératives. Cette partie de `Ocaml` est décrite à des fins d'exhaustivité **mais ne doit en aucun cas être utilisée**.

5.1 Séquencement

```
#print_string "Bon"; print_string "jour";;
 Bonjour- : unit = ()
```

- Opérateur de séquencement : « ; ».
- `Bonjour` : impression.
- Le résultat est de type `unit`.
- L'évaluation de l'expression renvoie la valeur `()`, c'est-à-dire « rien ». C'est le seul objet de type `unit`.

Toutes les fonctions `Ocaml` rendent un résultat.

```
# begin
    1+2; 0;
  end;;
      Characters 8-11:
Warning: this expression should have type unit.
- : int = 0

# 1+2; 0;;
 Characters 0-3:
Warning: this expression should have type unit.
- : int = 0
```

Le résultat du premier calcul a été jeté. Le résultat de l'évaluation d'une séquence est le résultat de l'évaluation du dernier terme de cette séquence. Le couple `begin-end` sert à délimiter la séquence. Ici il est superflu.

5.2 Les références

Il s'agit de cases mémoires **modifiables** : on peut les lire et les réécrire.
Création :

```
# let compteur = ref(0);;
 val compteur : int ref = \contents=0\
```

Lecture :

```
# !compteur;;
 - : int = 0
```

Écriture :

```
# compteur := 2;;
 - : unit = ()

# !compteur;;
 - : int = 2
```

Les références se manipulent comme les autres objets :

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

# incrémente compteur; !compteur;;
 - : int = 3
```

5.3 Les boucles

Boucle *tant que* :

```
while expression-condition
do
  expression-actions
done;;
```

Boucle *pour* :

```
for identificateur = expression-début to/downto expression-fin
do
  expression-actions
done;;
```

```
# let imprime_chiffres () =
    for i=0 to 9 do
       print_int i
    done;
    print_newline ();;
         val imprime_chiffres : unit -> unit = <fun>
```

```
# imprime_chiffres();;
 0123456789
- : unit = ()
```

5.4 Les tableaux

Création (définition) :

```
# let p = [|3;2;1|];;
 val p : int array = [|3; 2; 1|]
```

```
# let q = Array.make 4 2;;
 val q : int array = [|2; 2; 2; 2|]
```

Taille :

```
# Array.length q;;
 - : int = 4
```

Lecture (le premier élément d'un tableau est toujours l'élément d'indice « 0 ».) :

```
# p.(0);;
 - : int = 3
```

Écriture :

```
# q.(0) <- 1;;
 - : unit = ()
```

```
# q;;
 - : int array = [|1; 2; 2; 2|]
```

5.5 Caractères et chaînes de caractères

```
# let mot = "Ocaml";;
 val mot : string = "Ocaml"
```

```
# mot.[0];;
 - : char = 'O'
```

```
# mot.[0] <- 'o';;
 - : unit = ()
```

```
# mot;;
 - : string = "ocaml"
```

```
# String.length mot;;
 - : int = 5
```