Chapitre 1

Le langage OCaml

1 Les types du langage OCaml **2**
1.1 Les types simples en OCaml 2
1.2 Les types complexes en OCaml 5

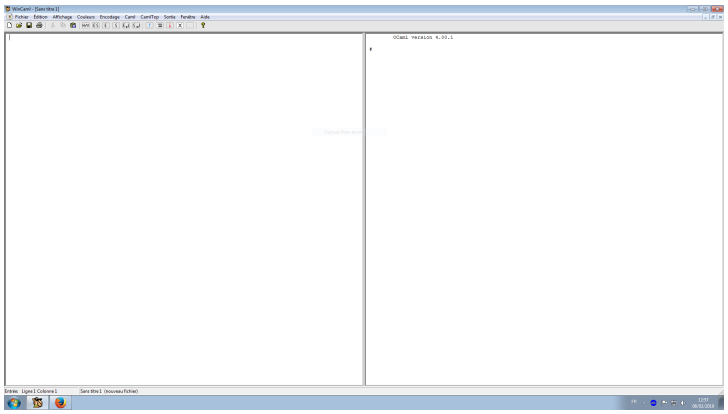
2 Les définitions ou déclarations en langage OCaml **8**
2.1 Définitions globales 8
2.2 Définitions locales 9

3 La programmation fonctionnelle en langage OCaml **12**
3.1 Différentes syntaxes comparées 12
3.2 Le filtrage en langage OCaml 13
3.3 Fonctions récursives 15

4 OCaml, langage impératif **16**
4.1 Les références 16
4.2 Séquence d'instructions 17
4.3 La conditionnelle 17
4.4 La répétitive 18

5 Exercices **19**

Notre nouvel environnement :



OCaml est un langage :

1. **fonctionnel**, c'est-à-dire basé sur des définitions et compositions de fonctions de plusieurs variables,
2. fortement **typé**, c'est-à-dire les objets manipulés sont systématiquement et automatiquement typés avant toute utilisation,
3. orienté objet, mais nous ne développerons pas ce point.

L'utilisation de l'aide permet de répondre à de nombreuses interrogations et de découvrir d'autres fonctions.
Quelques généralités sur l'éditeur :

- les phrases OCaml se terminent par ;;
- les phrases OCaml sont validées par Ctrl + ↵ ,
- les commentaires en OCaml sont délimités par `(*` et `*)` ,

généralités que nous complèterons par la pratique.

1 Les types du langage OCaml

L'objet de ce paragraphe est double :

- définir les principaux types utilisés par OCaml,
- comprendre le typage des objets manipulés.

Exemple. Décryptons le typage de la fonction `sin` :

```
sin ;;
- : float -> float = <fun>
```

`sin` est une fonction `<fun>` qui à un objet de type `float` associe un objet de type `float`.

```
sin 3.0 ;;
- : float = 0.14112000806
```

Lorsqu'on applique `sin` à un objet de type `float`, on obtient un résultat de type `float` suivi de sa valeur.

Une fonction est naturellement en **mode préfixe**, c'est-à-dire que les arguments suivent le nom de la fonction (comme la fonction `sin`), sauf quelques-unes en **mode infixe** où les arguments apparaissent de part et d'autre du nom de la fonction (par souci de commodité d'écriture... `2+3` se lit plus facilement que `+ 2 3`). Pour connaître le typage de ces dernières, il est nécessaire de les convertir en mode préfixe en entourant leur nom par des parenthèses `(...)`. Par exemple :

```
(+) ;;
- : int -> int -> int = <fun>
```

1.1 Les types simples en OCaml

Le type booléen : `bool`

- 2 valeurs : `true` ou `false`.
- Fonctions de résultat booléen : `= , <= , >= , < , > , <>` .

  Par exemple :

  ```
  2 + 2 = 4 ;;
  - : bool = true
  (<) ;;
  - : 'a -> 'a -> bool = <fun>
  ```

  La fonction `<` attend deux arguments de même type et retourne un résultat de type booléen. `'a` représente une variable de type, car la même fonction `<` peut être utilisée avec plusieurs types différents. On peut s'en servir pour comparer des entiers, des flottants... à condition bien sûr de comparer deux entiers entre eux, ou deux flottants entre eux. La fonction `<` est dite **polymorphe**.

- Opérations sur les booléens : `not , && , ||` .

  Par exemple :

  ```
  (2+2 = 4) && (1 < 0) ;;
  - : bool = false
  (||) ;;
  - : bool -> bool -> bool = <fun>
  ```

  La fonction `||` attend deux arguments de type booléen et retourne un résultat de type booléen.

  OCaml évalue les expressions booléennes de gauche à droite donc n'évalue pas les arguments si cela n'est pas logiquement nécessaire. Par exemple, lors de l'évaluation du booléen `a && b`, dans le cas où le booléen `a` est évalué à `false`, le booléen `a && b` est dans tous les cas évalué à `false` et l'évaluation de `b` devenant inutile, elle n'est pas faite. Les opérateurs `&&` et `||` sont dits **paresseux**.

  Par exemple :

  ```
  (1 > 0) || (1/0 > 0) ;;
  - : bool = true
  (1 = 0) && (1/0 <> 1) ;;
  - : bool = false
  ```

Il existe des allias pour la conjonction et la disjonction logique qui sont respectivement `&` et `or` . Pour des raisons de cohérence, ces allias sont à éviter, notamment pour ne pas utiliser les faux-ami `and` (qui sert à effectuer une affectation simultanée) ou `|` (qui sert à effectuer un filtrage).

Le type entier : `int`

- Opérations sur les entiers : `+` , `-` , `*` , `/` , `mod` , `abs` , `min` , `max`.

  Par exemple :

```
4 / 3 ;;
- : int = 1
(+) (-10) 25 ;;
- : int = 15
```

  Notons au passage l'importance des parenthèses pour éviter que OCaml ne dissocie le signe `-` du nombre 10. Les parenthèses en OCaml sont des délimiteurs. Nous prendrons l'habitude de les utiliser à bon escient, en évitant de trop charger l'écriture avec des parenthèses inutiles.

  Notons également l'absence de fonction puissance. En cas de besoin d'une telle fonction, nous devrons l'implémenter.

- Valeurs limitées entre $-4\,611\,686\,018\,427\,387\,904 = -2^{62}$ et $4\,611\,686\,018\,427\,387\,903 = 2^{62} - 1$, ces bornes étant stockées dans les constantes `min_int` et `max_int` :

```
min_int ;;
- : int = -4611686018427387904
max_int ;;
- : int = 4611686018427387903
```

  OCaml effectue les calculs sur les entiers modulo 2^{63} en représentant chaque entier par le reste de sa division par 2^{63} dans $[\![-2^{62}, 2^{62} - 1]\!]$. Cette limite peut entrainer des résultats surprenants comme :

```
9*9*9*9*9*9*9*9*9*9*9*9*9*9*9*9*9*9*9*9*9;;
- : int = -1261475310744950487
2147483648*2147483648;;
- : int = -4611686018427387904
2147483648*2147483648-1;;
- : int = 4611686018427387904
```

Le type flottant : `float`

- OCaml distingue complètement les types entier et flottant car la représentation interne en machine est tout à fait différente. Par exemple, l'entier 2 n'est pas du tout représenté de la même façon que le nombre décimal 2.0. Le type entier n'est donc pas un sous-type du type flottant (différence avec les maths où l'ensemble des entiers relatifs est inclus dans celui des réels). De ce fait, les algorithmes correspondants aux opérations usuelles sur les nombres (addition, multiplication...) sont différents suivant qu'il s'agit de nombres de type entier ou de type flottant. OCaml n'utilise donc pas les mêmes identificateurs pour désigner par exemple l'addition entre nombres de type entier et celle entre nombres de type flottant.

- Opérations sur les flottants : `+.` , `-.` , `*.` , `/.` , `**` , `sin` , `cos` , `tan` , `exp` , `log` (logarithme népérien) , `sqrt` , `sinh` , `cosh` , `tanh` , `asin` , `acos` , `atan` , `abs_float` , `min` , `max`.

  Par exemple :

```
3.14 +. 2.5 ;;
- : float = 5.64
(*.) ;;
- : float -> float -> float = <fun>
2 * 3.2 ;;
Toplevel input:
>2 * 3.2 ;;
>     ^^^
This expression has type float, but is used with type int.
```

```
2 *. 3.2 ;;
Toplevel input:
>2 *. 3.2 ;;
>^
This expression has type int, but is used with type float.
```

- Valeurs limitées à une partie finie de l'ensemble des nombres décimaux, les bornes étant stockées dans les constantes `min_float` et `max_float`.

Le type caractère : `char`

Par exemple :

```
'a' ;;
- : char = 'a'
```

Le type chaîne : `string`

- Suite ordonnée de caractères délimitée par des doubles guillemets, chaque caractère étant numéroté à partir de 0.

  Par exemple :

  ```
  "Bonjour, ça va ?" ;;
  - : string = "Bonjour, ça va ?"
  ```

- Opérations sur les chaînes de caractères.

  – Longueur d'une chaîne :

  ```
  String.length ;;
  - : string -> int = <fun>
  String.length "informatique" ;;
  - : int = 12
  ```

  – Extraction d'un caractère d'une chaîne :

  ```
  String.get ;;
  - : string -> int -> char = <fun>
  String.get "informatique" 3 ;;
  - : char = `o`
  "informatique".[3] ;;
  - : char = `o`
  ```

  – Concaténation de deux chaînes :

  ```
  (^) ;;
  - : string -> string -> string = <fun>
  "mon" ^ "sieur" ;;
  - : string = "monsieur"
  ```

  – Extraction d'une sous-chaîne d'une chaîne de caractères :

  ```
  String.sub ;;
  - : string -> int -> int -> string = <fun>
  String.sub "cavalerie" 2 3 ;;
  - : string = "val"
  ```

Le type `unit`

OCaml type systématiquement tous les objets manipulés. Mais que fait-il des actions telles que l'affectation, les entrées-sorties... Ces actions sont typés par OCaml en type `unit` . Ce type ne comporte qu'une valeur `()` :

```
() ;;
- : unit = ()
```

Par exemple, les fonctions `print_int` et `print_float`, qui permettent d'afficher des valeurs entières ou flottantes, renvoient un type `unit` :

```
print_int 3 ;;
3- : unit = ()
print_float 2.3 ;;
2.3- : unit = ()
```

Fonctions de conversion

Des fonctions de conversion de type sont disponibles, mais nous éviterons de les utiliser. Pour information, le sens de lecture correspond au sens de composition des fonctions mathématiques :

```
float_of_int ;;
- : int -> float = <fun>
int_of_float ;;
- : float -> int = <fun>
(* Correspond à la partie entière informatique. *)
int_of_char ;;
- : char -> int = <fun>
(* Retourne le code ASCII du caractère. *)
char_of_int ;;
- : int -> char = <fun>
```

1.2 Les types complexes en OCaml

Un objet de type complexe est un ensemble fini et ordonné d'objets typés :

- soit par une constante de type comme `int, float...` ou un type complexe,
- soit par une variable de type (`'a, 'b, ...`).

Le type tableau : `array`

- Type complexe ayant les caractéristiques suivantes :
  - les objets le composant sont tous de même type,
  - chaque objet est numéroté à partir de 0 et est directement accessible, on parle d'**accès en temps constant**,
  - l'ensemble est de taille fixe (non modifiable en cours de route), nous qualifierons le type de **statique**,
  - la valeur de chaque objet peut être modifiée, nous qualifierons le type de **mutable**.
- Les objets d'un tableau sont séparés par des `;` et délimités par `[|` et `|]` .

  Par exemple :

  ```
  [|3;5;8|] ;;
  - : int array = [|3; 5; 8|]
  ```

- Opérations sur les tableaux :
  - Accès direct à un élément :

    ```
    [|3;5;8|].(1) ;;
    - : int = 5
    ```

  - Longueur :

    ```
    Array.length ;;
    - : 'a array -> int = <fun>
    Array.length [|2;2;2|] ;;
    - : int = 3
    ```

  - Création d'un tableau :

    ```
    Array.make ;;
    - : int -> 'a -> 'a array= <fun>
    Array.make 5 0 ;;
    - : int array = [|0; 0; 0; 0; 0|]
    ```

- Extraction d'un sous-tableau :
```
Array.sub ;;
- : 'a array -> int -> int -> 'a array = <fun>
Array.sub [|2;5;4;8;7;1;0|] 2 3 ;;
- : int array = [|4; 8; 7|]
```
- Affectation dans un tableau :
```
[|1;2;3|].(1) <- 4 ;;
- : unit = ()
```
  On a ici un type `unit` car on effectue une action sans résultat (l'affectation). Pour voir qu'on a bien modifier la deuxième valeur du tableau, il faut faire une définition locale (on y reviendra plus tard) :
```
let v = [|1;2;3|] in
  v.(1)<-5 ;
  v
;;
- : int array = [|1; 5; 3|]
```
- Application d'une fonction à un tableau :
```
Array.map ;;
- : ('a -> 'b) -> 'a array -> 'b array = <fun>
Array.map (fun x->x*x)  [|1;2;3|] ;;
- : int array = [|1; 4; 9|]
```

Le type liste : `list`

- Type complexe ayant les caractéristiques suivantes :
  - les objets le composant sont tous de même type,
  - chaque objet contient l'adresse du suivant et ainsi le temps d'accès à chaque objet n'est plus identique, il est proportionnel à l'éloignement, on parle d'**accès en temps linéaire**,
  - l'ensemble est de taille modifiable (nous pouvons supprimer ou ajouter des éléments), nous qualifierons le type de **dynamique**,
  - la valeur de chaque objet ne peut être modifiée, nous qualifierons le type de **non mutable**.
- Les objets d'une liste sont séparés par des `;` et délimités par `[` et `]` .

  Par exemple :
```
[3;5;8]  ;;
- : int list = [3; 5; 8]
```
- Opérations sur les listes :
  - Création d'une liste vide :
```
[ ]  ;;
- : 'a list = [ ]
```
    Ce **polymorphisme** est nécessaire car selon le type du premier élément ajouté, on changera le type de la liste complète.
  - Ajout en-tête d'une liste :
```
2 :: [1;5;3] ;;
- : int list = [2; 1; 5; 3]
```
  - Tête d'une liste :
```
List.hd ;;
- : 'a list -> 'a = <fun>
List.hd [1;5;2;4] ;;
- : int = 1
```
    Prendre garde à la tête d'une liste vide : erreur !
```
List.hd [ ] ;;
Uncaught exception: Failure "hd"
```

- Queue d'une liste (ou suppression de la tête) :

```
List.tl ;;
- : 'a list -> 'a list = <fun>
List.tl [1;3;5;8] ;;
- : int list = [3; 5; 8]
```

  Prendre garde à la queue d'une liste vide : erreur !

```
List.tl [ ] ;;
Uncaught exception: Failure "tl"
```

- Longueur d'une liste :

```
List.length ;;
- : 'a list -> int = <fun>
List.length [1;2;3;4] ;;
- : int = 4
```

- Concaténation de deux listes :

```
(@)  ;;
- : 'a list -> 'a list -> 'a list = <fun>
[1;2;3;4] @ [4;2] ;;
- : int list = [1; 2; 3; 4; 4; 2]
```

- Inversion d'une liste :

```
List.rev ;;
- : 'a list -> 'a list = <fun>
List.rev [1;2;3;4] ;;
- : int list = [4; 3; 2; 1]
```

- Appartenance à une liste :

```
List.mem ;;
- : 'a -> 'a list -> bool = <fun>
List.mem 5 [3;5;4;5] ;;
- : bool = true
```

- Application d'une fonction à une liste :

```
List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list=<fun>
List.map sin [1.0;2.0;3.0] ;;
- : float list = [0.841470984808; 0.909297426826; 0.14112000806]
```

Le type produit cartésien : *

- Type complexe ayant les caractéristiques suivantes :
  - les objets le composant ne sont pas tous de même type,
  - type non mutable (même si on peut le rendre mutable).
- Les objets d'un produit cartésien sont séparés par des `,` et délimités par `(` et `)` .

  Par exemple :

```
(2 + 3 , 4.0) ;;
- : int * float = (5, 4.0)
```

- Opérations sur les produits cartésiens et plus particulièrement aux couples (le cas des n-uplets sera vu plus tard) :
  - Accès au premier élément du couple :

```
fst ;;
- : 'a * 'b -> 'a = <fun>
fst ((1,2) , 3.0) ;;
- : int * int = 1, 2
```

– Accès au second élément du couple :

```
snd ;;
- : 'a * 'b -> 'b = <fun>
( [1;2] , [|3.2;4.5|] ) ;;
- : int list * float array = [1; 2], [|3.2; 4.5|]
snd ([1;2] , [|3.2;4.5|]) ;;
- : float array = [|3.2; 4.5|]
```

2 Les définitions ou déclarations en langage OCaml

Au lancement de OCaml, nous avons un **environnement** (ou une **configuration**) initial(e) noté(e) dans la suite $< E_0 >$. Il nous est possible d'enrichir cet environnement par des **définitions** ou **déclarations**. Jusqu'alors, nous avons uniquement manipulé des expressions OCaml composées d'éléments prédéfinis. Nous pouvons définir en fonction de nos besoins, d'autres éléments et les nommer, nom que nous appellerons **identificateur**.

2.1 Définitions globales

On utilise la syntaxe : `let <identificateur>=<expression en OCaml>;;` . L'environnement est alors enrichi du couple `(<identificateur>,<expression>)`.

Par exemple :

```
let x = 2 ;;
val x : int = 2
```

En supposant que nous étions dans la configuration initiale, notre nouvel environnement est : $\{(x, 2), < E_0 >\}$. Nous constatons l'enrichissement global de l'environnement par le `x` qui se substitue au tiret `-` usuel.

```
let x = 6 ;;
val x : int = 6
x ;;
- : int = 6
```

L'état précis de l'environnement peut se présenter ainsi $\{(x, 6), (x, 2), < E_0 >\}$. Nous constatons le retour du tiret `-` . Le nom de la variable est uniquement précisé lors de la définition.

Remarque. L'évaluation automatique et systématique de toute expression et les définitions globales peuvent amener des confusions.

Par exemple :

$\{< E_0 >\}$
`let x = 5 ;;`
`val x : int = 5`
$\{(x, 5), < E_0 >\}$
`let y = x + 1;;`
`val y : int = 6`
$\{(y, 6), (x, 5), < E_0 >\}$
`let x = 2 ;;`
`val x : int = 2`
$\{(x, 2), (y, 6), (x, 5), < E_0 >\}$
`y ;;`
`- : int = 6`
$\{(x, 2), (y, 6), (x, 5), < E_0 >\}$

Le lien $(x, 5)$ est désormais inaccessible dans l'environnement. Il est devenu obsolète et encombre inutilement l'environnement.

De plus, il n'existe pas en OCaml de commande spécifique permettant de "récupérer" l'environnement initial au cours d'une session. La seule possibilité consiste à interrompre la session et en ouvrir une nouvelle.

Aussi, en OCaml, nous ne manipulons des définitions globales que lorsqu'elles auront un intérêt au cours de la session tout entière. En pratique, cela revient à dire que nous privilégions presque systématiquement les définitions locales et que les définitions globales sont presque exclusivement réservées à la définition de fonctions.

2.2 Définitions locales

Principe

On utilise la syntaxe :

```
let <identificateur> = <expression1> in
    <expression2>
;;
```

L'environnement est enrichi du couple `(<identificateur>,<expression1>)` localement (c'est-à-dire entre le `in` et le `;;`). Par exemple :

$\{< E_0 >\}$
```
let x = 2 in
```
$\{(x, 2), < E_0 >\}$
```
  x + 3
;;
```
$\{< E_0 >\}$
```
- : int = 5
```

On peut vérifier que la variable `x` n'existe pas en dehors du bloc où elle est définie :

```
x ;;
 Characters 0-1:
   x ;;
   ^
 Error: Unbound value x
```

Définitions locales imbriquées

Exemple 1.

$\{< E_0 >\}$
```
let x = 2 in
```
$\{(x, 2), < E_0 >\}$
```
  let y = 3 in
```
$\{(y, 3), (x, 2), < E_0 >\}$
```
    x + y
;;
```
$\{< E_0 >\}$
```
- : int = 5
```

Exemple 2.

$\{< E_0 >\}$
```
let x = 4 in
```
$\{(x, 4), < E_0 >\}$
```
  let y = x + 1 in
```
$\{(y, 5), (x, 4), < E_0 >\}$
```
    x + y
;;
```
$\{< E_0 >\}$
```
- : int = 9
```

Définitions locales juxtaposées

Exemple 1.

$\{< E_0 >\}$
```
let x = 2 and y = 3 in
```
$\{(y, 3), (x, 2), < E_0 >\}$
```
  x + y
;;
```
$\{< E_0 >\}$
```
- : int = 5
```

Notons l'utilisation du `and` et non du connecteur logique `&&` .

Exemple 2.

```
let x = 4 and y = x + 1 in
  x + y
;;
Toplevel input:
let x = 3 and y = x + 1 in x + y;;
>               ^
The value identifier x is unbound.
```

Remarques.

1. Lors de définitions juxtaposées, les différentes définitions faites doivent être indépendantes les unes des autres. OCaml refuse de compiler l'exemple précédent car l'identificateur `y` est défini en fonction de l'identificateur `x` .

2. Dans une définition locale juxtaposée, l'ordre dans lequel les définitions sont faites ne doit avoir aucune importance. Il est clair que le code de l'exemple 1 est équivalent au suivant :

   ```
   let y = 3 and x = 2 in
     x + y
   ;;
   ```

   En revanche, on se rend compte immédiatement que l'exemple 2 n'est pas correct si l'on intervertit les définitions locales concernant `x` et `y` :

   ```
   let y = x + 1 and x = 4 in
     x + y
   ;;
   ```

   Le message d'erreur de OCaml est alors parfaitement clair : lors de la définition de `y` , l'identificateur `x` n'est pas défini.

Portée d'une définition locale

Nous avons déjà précisé que la portée par défaut d'une définition locale est délimitée par le `in` et le `;;` . Cependant, une définition locale peut commencer et prendre fin à tout moment. Pour cela, il suffit de définir sa portée par les délimiteurs `begin` et `end` (ou par `(` et `)` mais c'est à éviter) qui permettent l'**encapsulage** de la définition locale.

Exemple.

Première version :

$\{< E_0 >\}$
```
let x = 2.0 in
```
$\{(x, 2.0), < E_0 >\}$
```
  print_float x ;
  print_newline () ;
  let x = 4.0 in
```
$\{(x, 4.0), (x, 2.0), < E_0 >\}$
```
    print_float x ;
    print_newline () ;
    x
;;
```
$\{< E_0 >\}$
```
2.0
4.0
- : float = 4.0
```

Seconde version :

$\{< E_0 >\}$
```
let x = 2.0 in
```
$\{(x, 2.0), < E_0 >\}$
```
  print_float x ;
  print_newline () ;
  begin
    let x = 4.0 in
```
$\{(x, 4.0), (x, 2.0), < E_0 >\}$
```
      print_float x ;
      print_newline ()
  end ;
```
$\{(x, 2.0), < E_0 >\}$
```
  x
;;
```
$\{< E_0 >\}$
```
2.0
4.0
- : float = 2.0
```

3 La programmation fonctionnelle en langage OCaml

Un **paradigme** est une manière de penser et de représenter le monde. Les **paradigmes de programmation** sont les différentes philosophies de méthodes et de rédaction de code qui sont utilisables par le programmeur pour résoudre un problème informatique. Sans être complètement incompatibles, ils sont plus ou moins développés selon les langages de programmation.

La **programmation fonctionnelle** est la programmation naturelle en langage OCaml. En programmation fonctionnelle, on considère le calcul comme l'évaluation successive de fonctions sans modifier l'état de la mémoire. Une fois créé, un objet ne peut pas être modifié.

Nous allons voir dans ce paragraphe :

- comment déclarer globalement ou localement une fonction,
- comment utiliser une fonction.

3.1 Différentes syntaxes comparées

Avec la commande `fun`

Ici, la fonction s'applique à un ou plusieurs arguments.
On utilise la syntaxe :

```
fun <argument 1> <arg 2> ... <arg n> -> <expression en OCaml> ;;
```

L'`expression en OCaml` utilise les arguments `arg 1, arg 2, ... , arg n`.

Exemple.

```
fun x -> x + 1 ;;
- : int -> int = <fun>
```

Ici, l'opération `+` indique que l'argument est un entier (sinon, il y aurait une erreur de syntaxe) donc de type `int`. OCaml en déduit le typage de cette fonction : `int -> int`. Et pour appliquer cette fonction à un argument :

```
(fun x -> x + 1) 2 ;;
- : int = 3
```

Dans le but d'une utilisation, nous déclarerons nos fonctions (globalement la plupart du temps) :

- dans le cas d'une déclaration globale :

  ```
  let <ident> = fun <arg 1> <arg 2> ... <arg n> -> <expr OCaml> ;;
  ```

- dans le cas d'une déclaration locale :

  ```
  let <ident> = fun <arg 1><arg 2> ... <arg n> -> <expr OCaml 1> in <expr OCaml 2> ;;
  ```

Exemple.

```
let carre = fun x -> x * x ;;
val carre : int -> int = <fun>
```

L'environnement est alors enrichi globalement :

```
carre 4 ;;
- : int = 16
```

Notre fonction s'utilise comme une fonction classique. Attention, la priorité est d'abord donnée à l'évaluation de la fonction :

```
carre 2 * 3 ;;
- : int = 12
carre (2 * 3) ;;
- : int = 36
```

Donnons une version allégée de la syntaxe, utilisée très majoritairement en pratique lors d'une déclaration de fonction :

```
let <ident> <arg 1><arg 2> ... <arg n> = <expression en OCaml> ;;
```

Exemple.

```
let somme1 x y = x + y ;;
val somme1 : int -> int -> int = <fun>
somme1 4 3 ;;
- : int = 7
somme1 4 ;;
- : int -> int = <fun>
```

Avec la commande `function`

Ici, la fonction s'applique à un seul argument. Cependant, plusieurs arguments peuvent être regroupés sous la forme d'un n-uplet (de type produit cartésien `*`).
Voici la syntaxe :

```
function <argument> -> <expression en OCaml> ;;
```

Une version allégée de la syntaxe :

```
let <ident> (<arg 1>,<arg 2>, ... , <arg n>) = <expression en OCaml> ;;
```

Cette fonction n'a en toute rigueur, qu'un seul argument qui est `(<arg 1>,<arg 2>, ... ,<arg n>)` auquel on impose d'être un produit cartésien.

Exemple.

```
let somme2 (x,y) = x + y;;
val somme2 : int * int -> int = <fun>
somme2 (4,3);;
- : int = 7
```

Préférence entre les deux syntaxes allégées.

Nous disposons donc de deux commandes `fun` et `function` pour définir une fonction. Mais elles ne correspondent pas à la même fonction mathématique. Par exemple :

- Pour `somme1`, on a le schéma suivant :

$$f : \left\{ \begin{array}{rcl} \texttt{int} & \to & (\texttt{int} \to \texttt{int}) \\ x & \mapsto & f_x : \left\{ \begin{array}{rcl} \texttt{int} & \to & \texttt{int} \\ y & \mapsto & f_x(y) = x + y \end{array} \right. \end{array} \right.$$

- Pour `somme2`, on a le schéma suivant :

$$g : \left\{ \begin{array}{rcl} (\texttt{int} \times \texttt{int}) & \to & \texttt{int} \\ (x, y) & \mapsto & g(x, y) = x + y \end{array} \right.$$

La fonction f est appelée version **curryfiée** de g (qui est donc dite **non curryfiée**). La première version est préférable à la seconde car elle fait apparaitre une fonction à n arguments comme une composée de n fonctions à un seul argument. Par exemple, la fonction `somme1 4` est directement accessible à partir de `somme1` alors que ce n'est pas le cas à partir de `somme2`.

3.2 Le filtrage en langage OCaml

L'idée du filtrage consiste à définir des fonctions "par morceaux" en séparant les différents cas.

```
let f = function
  true -> 1 | false -> 0
;;
```

Il est essentiel d'utiliser la syntaxe de `function` (et donc de n'utiliser qu'un seul argument) et que le résultat soit du même type dans tous les cas.

Afin de se rapprocher de la syntaxe allégée, nous utiliserons l'instruction `match ... with ...` :

```
match <identificateur> with
  | <motif 1> -> <expr 1>
  | <motif 2> -> <expr 2>

  | <motif n> -> <expr n>
```

Exemple.

```
let f b = match b with
  | true -> 1
  | false -> 0
;;
val f : bool -> int = <fun>
```

Remarques.

1. Un seul identificateur est possible, son type peut être complexe... ce qui permet de contourner cette limitation en utilisant un produit cartésien.

2. Les motifs `<motif 1> , ... , <motif n>` sont tous de même type, et du type de l'identificateur. Les expressions `<expr 1> , ... , <expr n>` sont toutes de même type, mais pas nécessairement du même type de l'identificateur. Par exemple :

```
let f b = match b with
  | true -> 1
  | false -> 0.
;;
Characters 57-59:
  | false -> 0.
             ^^
Error: This expression has type float but an expression was expected of type int
```

3. L'ordre est important : le filtrage s'arrête au premier test positif.

4. Le filtrage peut s'effectuer sur le résultat de l'application d'une fonction. Par exemple :

```
let pair n = match n mod 2 with
  | 0 -> true
  | 1 -> false
;;
val pair : int -> bool = <fun>
```

5. Le filtrage se doit d'être exhaustif, c'est-à-dire d'inclure la totalité des cas. Il se peut qu'il le soit mais que le compilateur OCaml ne le détecte pas. Dans ce cas, OCaml nous préviendra avec un message d'avertissement mais cela n'empêchera pas de compiler :

```
Warning 8: this pattern-matching is not exhaustive.
```

   C'est par exemple le cas pour la fonction `pair` précédente. Pour éviter ce problème, la dernière ligne du filtrage sera :

```
  | _ -> <expr n>
```

   c'est-à-dire "dans tous les autres cas". Le filtrage sera ainsi exhaustif. Par exemple :

```
let pair n = match n mod 2 with
  | 0 -> true
  | _ -> false
;;
val pair : int -> bool = <fun>
```

6. On parle plus généralement de filtrage par motifs. On compare une variable avec un motif et non avec une valeur. Intuitivement, l'opération de filtrage correspond aux étapes suivantes :

   - on compare notre variable à tel motif,
   - on effectue une liaison locale des différentes parties du motif.

   Pour contourner en partie la rigidité du filtrage par motifs, on peut avoir recours à l'instruction `when`. Par exemple :

```
let pair n = match n with
  | n when n mod 2 = 0 -> true
  | _ -> false
;;
val pair : int -> bool = <fun>
```

7. L'encapsulage d'une déclaration locale est intéressant par exemple lorsqu'on fait une déclaration locale dans un filtrage et qu'on veut la limiter à ce cas là.

3.3 Fonctions récursives

Récursif vient du latin "recurrere" qui signifie "courir en arrière". Une **fonction récursive** est une fonction qui s'appelle elle-même sur des données plus petites. On associera la récursivité à la programmation fonctionnelle.

La récursivité simple

Voici la syntaxe lors d'une déclaration globale de fonction récursive :

```
let rec <ident> = fun <arg 1><arg 2> ... <arg n> -> <expression en OCaml> ;;
```

ou, en version abrégée :

```
let rec <ident> <arg 1><arg 2> ... <arg n> = <expression en OCaml> ;;
```

Lors de la définition d'une fonction récursive, nous prendrons soin de séparer à l'aide d'un filtrage :

- l'étape de base,
- l'étape d'induction.

Exemple. Voici une fonction récursive qui calcule la factorielle d'un entier naturel :

```
let rec facto n = match n with
  | 0 -> 1
  | _ -> n * facto (n-1)
;;
```

Remarque. Nous avons un mode qui nous permet de visualiser les appels récursifs d'une fonction, le mode `trace` (c'est-à-dire exécution pas à pas). En reprenant l'exemple de la fonction récursive factorielle :

```
#trace facto ;;
facto is now traced.

facto 5;;
facto <-- 5
facto <-- 4
facto <-- 3
facto <-- 2
facto <-- 1
facto <-- 0
facto --> 1
facto --> 1
facto --> 2
facto --> 6
facto --> 24
facto --> 120
- : int = 120

#untrace facto;;
facto is no longer traced.
```

Les récursivités croisées

Cela se produit lorsqu'on est en présence de deux (ou plusieurs) fonctions récursives qui s'appellent mutuellement. Aucune de ces fonctions ne peut être définie sans avoir défini les autres, ce qui pose un véritable problème.

Par exemple :

$$\begin{cases} u_0 = 1 \\ v_0 = 2 \end{cases}, \qquad \forall n \in \mathbb{N}, \begin{cases} u_{n+1} = \dfrac{1}{2}(u_n + v_n) \\ v_{n+1} = \sqrt{u_n\ v_n} \end{cases}$$

En OCaml, il faut alors procéder comme pour une déclaration juxtaposée (bien que, dans ce cas, les différentes déclarations ne soient justement pas du tout indépendantes !).

```
let rec u n = match n with
  | 0 -> 1.
  | _ -> (u (n-1) +. v (n-1)) /. 2.
and v n = match n with
  | 0 -> 2.
  | _ -> sqrt (u (n-1) *. v (n-1))
;;
```

On peut évidement calculer le résultat de l'une des fonctions sans préciser l'autre :

```
u 7;;
- : float = 1.45679103104690677
```

4 OCaml, langage impératif

La **programmation impérative** consiste à modifier la mémoire lors de l'exécution des calculs. Les objets informatiques y sont alors modifiables. Ce paradigme est en opposition à celui de la programmation fonctionnelle. Bien qu'étant principalement fonctionnel, OCaml dispose également d'éléments de programmation impérative.

Nous allons voir dans ce paragraphe comment définir sur OCaml des fonctions itératives. Itératif vient du latin "iterare" qui signifie "recommencer, répéter". Une **fonction itérative** est une fonction qui répète des instructions qui modifient l'état de la mémoire un certain nombre de fois (connu à l'avance ou non). On associera l'itérativité à la programmation impérative.

4.1 Les références

La base de notre problème est que la plupart des types natifs ne sont pas mutables (c'est-à-dire que nous ne pouvons pas modifier leur contenu). Ainsi :

```
let i = 0 in
  i <- i+1
;;
Characters 13-23:
let i = 0 in i <- i + 1 ;;
             ^^^^^^^^^^
Error: The value i is not an instance variable
```

Remarquons que les "i" de l'instruction :

$$i \leftarrow i + 1$$

ont deux rôles distincts :

- le i de droite correspond à la valeur de la variable, le contenu,
- le i de gauche correspond à l'emplacement de stockage de la valeur $i + 1$, le contenant appelé **référence**.

OCaml fait la différence entre le contenu et le contenant, ce qui pose problème.

Pour solutionner ceci, nous passerons par l'intermédiaire d'une **référence**. Voici la syntaxe :

```
let i = ref 0 ;;
val i : int ref = {contents = 0}
```

et le contenu du contenant `i` est `!i` :

```
!i ;;
- : int = 0
```

Notons le typage de OCaml pour une référence : `'a ref` , et de son contenu : `'a` .
Remarque importante, le `!` est toujours précédé d'un espace ou entouré par des parenthèses.
Pour l'affectation dans une référence, on utilise `:=` avec à gauche de ce symbole, le contenu ou la référence de la variable, et à droite de ce symbole, le contenu de la variable. L'affectation est de type `unit` .

```
(:=) ;;
- : 'a ref -> 'a -> unit = <fun>
```

Par exemple, en algorithmique :

$$i \leftarrow 0$$
$$i \leftarrow i + 1$$

et en OCaml :

```
let i = ref 0 in
  i := !i+1
;;
- : unit=()
```

Ceci nous permettra de faire des déclarations locales dans une fonction, avec obligation de donner une valeur initiale à la référence même si cette valeur initiale n'est pas utilisée par la suite.

4.2 Séquence d'instructions

Nous appellerons **instruction** toute expression en OCaml. Par exemple : `i := !i+1` , ...
La programmation impérative procéde par séquence (suite...) d'instructions. Dans une séquence d'instructions,

- les instructions sont séparées par un point virgule `;` ,
- la dernière instruction donne le type de toute la séquence,
- les instructions précédant la dernière doivent toutes être de type `unit` .

Exemple.

```
2+3;
2=3;
[1;4] ;;
Characters 0-3:
  2+3;
  ^^^
Warning 10: this expression should have type unit.
Characters 6-9:
   2=3;
   ^^^
Warning 10: this expression should have type unit.
- : int list = [1; 4]
```

Exemple.

```
let i = ref 0 in
  i := !i+1;
  !i
;;
- : int = 1
```

Une séquence contenant plusieurs instructions aura parfois besoin d'être délimitée : c'est l'encapsulage, les délimiteurs utilisés étant `begin` et `end` :

```
begin
    <séquence d'instructions>
end
```

4.3 La conditionnelle

Voici la syntaxe :

```
if <condition>
    then
       begin
          <séquence d'instructions S1>
       end
    else
       begin
          <séquence d'instructions S2>
       end
```

où :

- `<condition>` est nécessairement de type `bool`,
- `<séquence d'instructions S1>` et `<séquence d'instructions S2>` doivent être de même type, et ce type commun sera le type de l'instruction `if`.

- l'encapsulage n'est pas nécessaire si les séquences d'instructions S1 et S2 ne comportent qu'une seule instruction.

Si la condition est vérifiée, la séquence d'instructions S1 est réalisée, sinon la séquence d'instructions S2 est réalisée.

Exemple.

```
let absolue x =
    if (x>0.)
        then x
        else (-.x)
;;
absolue : float -> float = <fun>
```

Remarque. La clause sinon n'est pas totalement facultative !!! OCaml ajoute systématiquement ... `else ()` ... de type `unit`, et par conséquent la séquence du `then` se doit d'être de type `unit`. Par exemple :

```
let monlog x =
    if (x>0.)
        then log x
;;
Characters 30-35:
  then log x
       ^^^^^
Error: This expression has type float but an expression was expected of type unit
```

Une solution consiste à placer un message d'erreur de type polymorphe dans le sinon tel que :

```
raise( Failure "..." )
```

ou :

```
failwith "..." .
```

Exemple.

```
let monlog x =
    if (x>0.)
      then log x
      else failwith "monlog : argument négatif"
;;
monlog : float -> float = <fun>

monlog (-1.) ;;
Uncaught exception: Failure "monlog : argument négatif"
```

4.4 La répétitive

Une répétitive est une instruction OCaml de type `unit` .

Boucle indexée (pour)

Voici la syntaxe :

```
for <identificateur> = <val_init> to (downto) <val_fin> do
     <séquence d'instructions>
done
```

où :

- `<ident>` , `<val_init>` , `<val_fin>` sont de type `int`,
- les instructions de la séquence sont toutes de type `unit`,
- l'encapsulage est automatique par `do .... done`.

La séquence d'instructions est réalisée une fois pour chaque valeur de l'identificateur, celui-ci variant de 1 en 1 à partir de la valeur initiale et jusqu'à la valeur finale incluses.

Boucle conditionnelle (tant que)

Voici la syntaxe :

```
while <condition> do
      <séquence d'instructions>
done
```

où :

- `<condition>` est nécessairement de type `bool`,
- les instructions de la séquence sont toutes de type `unit`,
- l'encapsulage est automatique par `do .... done`.

Dans le cas où la condition est fausse, nous passons à l'instruction suivante sans réaliser la séquence d'instructions, et dans le cas où la condition est vraie, nous réalisons la séquence d'instructions puis revenons tester la condition.... Nous devrons nous assurer de deux choses :

- que la condition a été initialisée,
- que la condition ne sera pas toujours vérifiée, autrement dit que la répétitive s'arrête.

5 Exercices

Exercice 1

Quelle est la réponse de OCaml à :

```
2 ** 3 ;;
1.  + 1.  ;;
(>) 2 3 ;;
2 + 5 = 7 ;;
```

Exercice 2

1. Calculer une valeur approchée de π :
   (a) en utilisant la fraction $\dfrac{355}{113}$,
   (b) en utilisant la valeur $4 \times \arctan(1)$.
2. Calculer une valeur approchée de $\cos(\arctan(2))$.

Exercice 3

Corriger la faute d'orthographe dans la chaîne `"parassol"` .

Exercice 4

Quelle est la réponse de OCaml à :

1. `[|[|1;2|];[|2;3;5;4|];[|1;1;1|]|] ;;`
2. `[|[|1;2|];[|2;3;5;4|];[|1;1;1|]|].(1) ;;`

Exercice 5

1. Quelle est la réponse de OCaml à :

```
7::3::2::8::[ ] ;;
[ ]::[ ] ;;
List.tl [3] ;;
```

2. (a) Déterminer le quatrième élément de la liste `[1;2;3;4;5]` .
   (b) Supprimer le quatrième élément de la liste `[1;2;3;4;5]` .

3. Expliquer la façon dont OCaml type les expressions suivantes (typage de `l` et du résultat) :

```
List.length ([2;4]::l) ;;
l @ [[| |]] ;;
l::[[ ]] ;;
```

Exercice 6

Quelle est la réponse de OCaml à :

```
(1,2,3,4) ;;
((1,2),(3,4)) ;;
(1,2,(3,4)) ;;
fst ((1,2.0),3) ;;
fst (1,2,3.0) ;;
fst ((1,2),3) ;;
[1,2.2,3] ;;
```

Exercice 7

En utilisant une affectation locale, calculer :

$$\frac{1+\sqrt{2}+\sqrt{2}^3}{e^{\sqrt{2}}-1}, \qquad \frac{\ln(17)+\sin(\ln(17))}{\sqrt{3}+\ln(\sqrt{3})}, \qquad \tanh(2).$$

Exercice 8

Déterminer le résultat de l'évaluation des instructions suivantes ou si elles comportent des erreurs :

1. `let x = 1 and y = 3 in let x = 2 in x + y ;;`
2. `let x, y, z = 1, 2, 3 in x + y + z ;;`
3. `let x, y, z = 1, (2, 3) in x + y + z ;;`
4. `let x, y, z = true, 2, 3 in x || y = z;;`
5. `let x, y, z = true, 2, 3 in x and y < z ;;`

Exercice 9

Déterminer le type des fonctions suivantes :

1. `let somme_fct f g x = (f x) + (g x) ;;`
2. `let composition f g x = f (g x) ;;`

Exercice 10

Écrire les fonctions :

1. `funct1` qui associe à $f \in \mathcal{F}(\mathbb{R},\mathbb{R})$ la valeur $\frac{f(0)+f(1)}{2}$.
2. `funct2` qui associe à $f \in \mathcal{F}(\mathbb{R},\mathbb{R})$ et $x \in \mathbb{R}$ la valeur $f(x)^2$.
3. `funct3` qui associe à $f \in \mathcal{F}(\mathbb{R},\mathbb{R})$ la fonction f^2 (le carré de f).
4. `funct4` qui associe à $f \in \mathcal{F}(\mathbb{R},\mathbb{R})$ la fonction $x \mapsto f(x+1)$.

Exercice 11

Considérons la fonction `max3` suivante :

```
let max3 x y z = max max x y z ;;
```

Justifier que cette fonction ne renvoie pas le résultat attendu, corriger puis donner son typage.

Exercice 12

1. Écrire une fonction `module : float * float -> float` qui prend un nombre complexe sous forme d'un couple (partie réelle, partie imaginaire) et qui calcule son module.

2. Écrire une fonction `test_egalite : int * int -> int * int -> bool` qui prend deux rationnels sous forme de couples (numérateur, dénominateur) et qui teste l'égalité de ses arguments. Par exemple,

```
test_egalite (1, 2) (3, 6) ;;
- : bool = true
```

Exercice 13

Écrire une fonction qui prend une fonction avec un argument de type produit cartésien (version non curryfiée), et qui renvoie la fonction sémantiquement équivalente sans le produit cartésien (version curryfiée), et vice-versa (fonction de curryfication et de décurryfication).

Exercice 14

Déterminer le type des fonctions suivantes :

```
let f x y = match y with
  | 0 -> 0.
  | _ -> 1.
;;
```

```
let g x y z = match (z,x) with
  | 0, 0 -> 0
  | a, b -> 1
;;
```

Exercice 15

1. Écrire une fonction `xor` correspondant au ou exclusif.

2. Écrire une fonction `signe` sur l'ensemble des entiers relatifs, c'est-à-dire :

$$\texttt{signe} : \mathbb{Z} \to \mathbb{Z}\,,\ n \mapsto \begin{cases} 1 & \text{si } n > 0, \\ 0 & \text{si } n = 0, \\ -1 & \text{si } n < 0. \end{cases}$$

Exercice 16

1. Écrire une fonction récursive qui calcule la puissance entière d'un entier relatif.

2. Écrire une fonction récursive qui calcule le pgcd de deux entiers naturels.

Exercice 17

1. Écrire une fonction récursive qui calcule le terme général de la suite définie par :

$$\begin{cases} u_0 \in \mathbb{R} \\ \forall n \in \mathbb{N}\,,\ u_{n+1} = \sin(u_n) \end{cases}$$

2. Écrire une fonction récursive qui calcule le terme général de la suite (de type Fibbonacci) définie par :

$$\begin{cases} F_0 \in \mathbb{R}\,,\ F_1 \in \mathbb{R} \\ \forall n \in \mathbb{N}\,,\ F_{n+2} = F_{n+1} + F_n \end{cases}$$

3. Écrire une fonction récursive qui calcule le terme général des suites u et v définies par :

$$\begin{cases} u_0 = 2 \\ v_0 = 1 \end{cases}, \qquad \forall n \in \mathbb{N}, \begin{cases} u_{n+1} = \dfrac{u_n + v_n}{2} \\ v_{n+1} = \dfrac{2u_n v_n}{u_n + v_n} \end{cases}.$$

Exercice 18
Définir récursivement sur OCaml les fonctions f et g suivantes :

$$f : (m,n) \mapsto \begin{cases} n & \text{si } m = 0 \\ f(m-1, n+1) & \text{sinon,} \end{cases} \qquad \text{et} \qquad g : (m,n) \mapsto \begin{cases} n & \text{si } m = 0 \\ g(m-1,n)+1 & \text{sinon.} \end{cases}$$

Exercice 19

1. Écrire une fonction récursive `somme` qui, à une fonction $f : \mathbb{N} \to \mathbb{N}$ et à un entier n, associe $\displaystyle\sum_{k=0}^{n} f(k)$.
2. En déduire une fonction récursive `catalan` utilisant `somme` et renvoyant le n-ième nombre de la suite de Catalan (c_n) définie par $c_0 = 1$, $c_1 = 1$ et

$$\forall n \geq 2, \quad c_n = \sum_{p+q=n-1} c_p c_q.$$

Exercice 20
On considère la suite définie par $u_0 = 1$ et pour tout $n \in \mathbb{N}$, $u_{n+1} = \sin(u_n)$.

1. Écrire une suite d'instructions calculant u_{50}.
2. Écrire une suite d'instructions permettant de déterminer le plus petit entier n tel que $u_n \leq 10^{-3}$.

Exercice 21

1. Écrire une fonction itérative de calcul de factorielle.
2. Écrire une fonction itérative de calcul de la puissance d'un nombre entier.
3. Écrire une fonction itérative de calcul de $\displaystyle\sum_{k=m}^{n} \frac{1}{k}$, m et n étant deux entiers naturels non nuls.
4. Écrire une fonction itérative qui calcule le pgcd de deux entiers naturels.
5. Écrire une fonction itérative du calcul de la valuation adique d'une entier naturel non nul.

Exercice 22

1. Écrire une fonction itérative `maximum : 'a array -> 'a` qui détermine la valeur du plus grand élément d'un tableau.
2. Écrire une fonction itérative `miroir : 'a array -> 'a array` qui renvoie un tableau qui est le retourné du tableau donné en argument.
3. Écrire une fonction itérative `appartenance : 'a array -> 'a -> bool` qui détermine si un élément appartient à un tableau.
4. Écrire une fonction itérative `permutation : 'int array -> bool` qui détermine si un tableau représente une permutation.