Option informatique, cours 1

Syntaxe du langage Ocaml

C. MULLAERT

Lycée Saint-Louis

Année 2024-2025

Plan du cours

Introduction au langage OCaml

Programmation fonctionnelle

OCaml est un langage de programmation adapté à la **programmation fonctionnelle**, par opposition à la **programmation impérative**. Cela signifie que :

- le langage est conçu pour limiter les **effets de bord**
- les "variables" sont, par défaut, **immuable** (c'est-à-dire non modifiable en place)
- les **fonctions** ont le même statut que les autres types (peuvent être passés en argument d'une autre fonction, ou être renvoyées comme valeur de retour)

Illustrons tous ces concepts importants par des exemples.

Effet de bord

Un **effet de bord** (side-effect en anglais) indique qu'une fonction ou une instruction fait "autre chose" que de réaliser le calcul de la valeur à renvoyer (ex : modifier une variable globale, modifier un argument passé en référence, afficher un résultat, ...).

En Python, la fonction suivante n'a pas d'effet de bord :

```
def add(a,b):
    return(a+b);
```

mais cette fonction en comporte plusieurs :

```
def f(L):
    L[1] = 6
    print L[0]
    return(L)
```

Effet de bord

Écrire un programme qui comporte des effets de bord comporte des avantages et des inconvénients :

- ⊕ facilité de la programmation (pas de restriction)
- ⊖ difficulté à prouver la correction/terminaison des algorithmes

En Caml, la possibilité d'utiliser des effets de bord est limitée. Il en résulte un recours fréquent à la **récursivité**.

Variables immuables

Caml permet de définir des "variables", mais il n'est (en principe) pas possible de les modifier.

En Python, on peut écrire :

```
a = 3 # Déclaration et initialisation
a = 4 # Affectation
```

En Ocaml, une telle écriture (bien que possible), n'est pas le comportement par défaut. On parle de liaison entre un nom et une valeur immuable.

Statut des fonctions

En Ocaml, les fonctions sont des valeurs de **première classe**, ce qui signifie qu'elles se comportent comme n'importe quelle valeur (int, float, ...). Elles peuvent :

- être renvoyée par d'autres fonctions
- être passées en paramètre d'une fonction

Une fonction qui manipule une autre fonction est appelée fonction **d'ordre supérieur**.

Installation de la distribution et de l'interface

Pour pouvoir s'entrainer, il est possible d'installer sur son ordinateur le logiciel WinCaml pour une utilisation interactive. Il suffit de consulter l'adresse suivante :

`https://www.wincaml7.fr/index.html#WinCamlbinaries`, puis de suivre les instructions correspondant à votre architecture et votre système d'exploitation.

Le cas le plus courant (Windows 11, x64) nécessite de télécharger les deux archives suivantes :

- `https://www.wincaml7.fr/archives/WinCaml7_2_x64.zip`
- `https://www.wincaml7.fr/archives/ocaml64.zip`

Il faudra alors extraire la première, puis placer les fichiers de la deuxième à la racine du répertoire d'extraction. Après l'exécution de WinCaml.exe, vous disposez d'un éditeur sur la gauche et d'une console Ocaml sur la droite.

Expressions et types de base

Expression

Une **expression** est un groupe syntaxique qui peut être évalué.

Elles sont toujours associées à un type, déterminé à la compilation (typage **statique**, contrairement au langage Python qui est typé **dynamiquement**).

Exemples :

- `2`
- `2.`
- `2+1`
- `2.+.1.`

Les nombres suivi du symbole . sont considérés comme des flottants. L'opérateur +. est l'opérateur d'addition pour les flottants.

Le type **int**

Le type `int` contient les entiers signés de l'intervalle $[\![-2^{62}; 2^{62}-1]\!]$. Ils sont représentés en mémoire sur 63 bits avec la convention du complément à 2 pour les entiers négatifs.

Les opérateurs élémentaires qui agissent sur ce type sont :

```
+ (* addition *)
- (* soustraction *)
* (* multiplication *)
/ (* division *)
```

Attention, les dépassements ne génèrent pas d'erreur :

```
4611686018427387903+1
```

est une expression de type `int` dont la valeur est $-2^{62} = -4611686018427387904$.

Le type **float**

Le type `float` contient les nombres à virgule flottante. Ils sont stockés en double précision (sur 64 bits selon le standard IEEE 754).

Les opérateurs élémentaires qui agissent sur les flottants sont :

```
+. (* addition *)
-. (* soustraction *)
*. (* multiplication *)
/. (* division *)
** (* exponentiation *)
```

Les opérations sur les flottants ne génèrent pas d'erreur. Par exemple,

```
1./.0.
```

est une expression de type float qui est évaluée comme `infinity` (une des valeurs possible du type `float`).

Le type **bool**

Le type `bool` peut correspondre à seulement deux valeurs : `true` et `false`.

Les opérations sur les booléens sont :

```
&& (* conjonction (et) *)
|| (* disjonction (ou) *)
not (* négation *)
```

L'évaluation d'une conjonction ou d'une disjonction est **paresseuse** (comme en Python) : les expressions sont évaluées de la gauche vers la droite, sans évaluer les dernières expressions lorsqu'elles sont inutiles.

Les opérateurs = (égalité structurelle) et <> (différence structurelle) permettent de construire une expression booléenne à partir de deux expressions de type `int` ou `float`. On dispose également des opérateurs <, >, <= et >=.

Autres types courants

Le type `unit` ne contient qu'une seule valeur, notée `()`.

Le type `char` contient des caractères (non accentués car stockés uniquement sur 8 bits). On les écrit entre deux symboles `'` : `'a'`, `'b'`.

Le type `string` contient des chaines de caractères. On les écrit entre deux symboles `"` : `"mpsi"`, `"c"`, `"éééè"`.

Le type `'a` est un type non spécifié, utilisé lorsque caml ne peut déterminer le type d'une expression.

Définition globale

Une **définition ou liaison globale** est une instruction qui permet de donner un nom à une expression. La syntaxe est la suivante :

```
let nom = expression;;
```

Le nom doit commencer par une lettre minuscule (ou bien le caractère underscore _).

L'évaluation de l'expression n'est faite qu'une seule fois lors de la définition. Le nom peut ensuite être utilisé dans le programme.

Tout comme l'expression, la valeur a un type qui est déterminé par celui de l'expression et qui ne pourra être modifiée.
Lors d'une liaison, l'interpréteur affiche une ligne du type
`val nom : type = valeur` .

Définition locale

Une **définition ou liaison locale** permet de donner un nom à une expression, mais uniquement dans une autre expression (et pas dans le reste du programme), la syntaxe est la suivante :

```
let nom = expression1 in expression2
```

Exemple :

```
let a = 3 in a*a
```

est une expression de type int et de valeur 9.

```
let a = 3.2 and b=2. in a*b
```

est une expression de type float et de valeur 6.4.

Quizz 1

Quels sont les types et les valeurs des expressions suivantes ?

```
10.
3*3
3=2
"c"='c'
let a=2. in a*.a;;
let a=5 in let b=a+2;;
let b=let a=5 in a+2;;
let a=5 in let b=a+2 in a+b;;
let c= let a=5 in let b=a+2 in a+b;;
```

Définition et manipulation de fonctions

Fonctions d'un argument

En Caml, on définit une fonction de la façon suivante :

```
let nom_fonction = function nom_argument -> expression;;
```

Cette écriture fait apparaitre que les fonctions sont un type possible pour une expression. Des variantes syntaxiques possibles sont :

```
let nom_fonction = fun nom_argument -> expression;;
```

```
let nom_fonction nom_argument = expression;;
```

On préfère généralement cette dernière écriture car elle est plus concise.

Fonctions d'un argument

Comme pour les autres constantes, Caml attribue un type à une fonction lors de sa déclaration. Ce type sera de la forme :

`type_argument -> type_retour`

Exemples :

```
fun x->x*x (* est de type int -> int *)
fun x->x*.x (* est de type float -> float *)
fun x->1 (* est de type 'a -> int *)
fun ()->1 (* est de type unit -> int *)
```

Appel de fonction

L'application d'une fonction sur un argument s'écrit :

`nom_fonction expression`

Exemple : après exécution de l'instruction suivante

```
let f = fun x->x*x;;
```

l'expression `f 3` est de type `int` et de valeur 9.

Notons que les parenthèses ne sont pas requises, et servent uniquement à gérer l'ordre des opérations dans des appels plus complexes comme .

```
f 3+3;; (* a pour valeur 3² + 3 = 12 *)
f (3+3);; (* a pour valeur 36 *)
f (-3);; (* a pour valeur 9; f -3 renvoie un message d'erre
```

Attention, lorsque l'on passe la valeur () de type `unit` à une fonction, les parenthèses sont obligatoires.

Fonctions prédéfinies à connaitre

Quelques fonctions sont directement utilisables sans déclaration et sont à connaitre :

- Les fonctions mathématiques : `exp`, `log`, `sqrt`, `cos`, `sin`, `tan`, `acos`, `abs_float`. Ces fonctions sont du type `float->float`.
- Les fonction de conversion de type : `float_of_int` (`int->float`) et `int_of_float` (`float->int`).
- Les fonctions d'affichage : `print_char` (`char->unit`), `print_string` (`string->unit`), `print_int` (`int->unit`), `print_float` (`float->unit`), `print_newline` (`unit->unit`).

Fonctions avec plusieurs arguments

Pour définir une fonction de plusieurs arguments, il y a deux stratégies possibles. Prenons l'exemple de la fonction distance entre deux flottants $(x, y) \mapsto |x - y|$. On peut :

1. Utiliser un type produit (la construction de tels type sera vue en détails plus tard dans le cours) :

   ```
   let dist = fun (x,y)->abs_float(x-.y);;
   ```

   La fonction dist est alors de type `float * float -> float` et peut être appelée, par exemple par `dist (3.,4.5)`.

2. Utiliser des fonctions imbriquées :

   ```
   let dist = fun x-> fun y->abs_float(x-.y);;
   ```

   La fonction dist est alors de type `float -> float -> float` et peut être appelée, par exemple par `dist 3. 4.5`.

C'est cette deuxième stratégie qui est la plus utilisée.

Fonctions à plusieurs arguments

Reprenons la définition de notre fonction dist :

```
let dist = fun x-> fun y->abs_float(x-.y);;
```

Deux écritures équivalentes (et plus simples) :

```
let dist = fun x y -> abs_float(x-.y);;
let dist x y = abs_float(x-.y);; (* à préférer *)
```

Dans l'exemple précédent, `dist 3.` est une expression de type `float -> float`, ce qui justifie l'écriture `dist 3. 4.5` qui est interprétée comme `(dist 3.) 4.5` et qui est donc de type `float`.

Ce principe de voir des fonctions à plusieurs arguments comme des fonctions à un argument renvoyant d'autres fonctions à un argument est appelé la **curryfication**.

Types fonctionnels

Une fonction a un type fonctionnel, formé d'un type d'entrée et d'un type de retour, reliés par une flèche.

On omet d'écrire certaines parenthèses : une fonction de type `int->int->int` est en réalité de type `int -> (int->int)`.

Attention cependant à ne pas confondre avec une fonction de type `(int->int) -> int`, qui prend une fonction en argument et renvoie un entier. On dit que l'opérateur `->` est associatif à droite

La fonction f ci dessous évalue une fonction en 0 et ajoute 1.

```
let eval f = f 0 + 1;;
```

Elle est du type `(int->int) -> int` et l'appel `eval 0` déclenche une erreur.

Filtrage

Conditions et filtrage

La structure conditionnelle if/then/else peut être utilisée pour construire une expression, par exemple dans la définition d'une fonction en caml.

```
let syracuse n = if (n mod 2 = 0) then n/2 else 3*n+1;;
```

Il est possible d'écrire une version équivalente de cette fonction à l'aide des fonctionnalités de filtrage (*pattern matching* en anglais) d'Ocaml :

```
let syracuse n = match (n mod 2) with
 | 0 -> n/2
 | 1 -> 3*n+1;;
```

Conditions et filtrage

```
let syracuse n = match (n mod 2) with
 | 0 -> n/2
 | 1 -> 3*n+1;;
```

Cette écriture est habituelle en Ocaml et présente plusieurs avantages :

- elle est commode à lire et permet d'éviter les imbrications, lorsqu'il y a plus de deux possibilités à considérer;
- c'est un outil très puissant car le *motif* de filtrage peut être complexe et utiliser des opérateurs (voir la suite du cours)

Conditions et filtrage

Quelques remarques de syntaxe sur les filtrages :

- Le premier filtrage n'a pas besoin de commencer par le caractère `|`. Il est cependant d'usage de le mettre pour des raisons esthétiques
- Les motifs n'ont pas besoin d'être exclusifs : le premier motif compatible sera appliqué.
- Les motifs n'ont pas besoin d'être exhaustifs. Un *warning* est levé dans ce cas. Si, lors d'un appel, aucun motif ne correspond, une *exception* est levée.
- Le motif `_` est universel. Il est donc à utiliser en dernier pour capturer tous les cas non traités avant.

Conditions et filtrage

Une variante syntaxique plus concise est d'utiliser le mot clé function pour éviter de nommer un paramètre qui ne sert qu'au filtrage:

```
let estZero  = function
 | 0 -> true
 | _ -> false;;
```

plutôt que :

```
let estZero n = match n with
 | 0 -> true
 | _ -> false;;
```

Contraintes de filtrage

Le mot-clé `when` permet d'ajouter une contrainte booléene à un motif de filtrage :

```
let syracuse = function
  | n when (n mod 2 = 0) -> n/2
  | n -> 3*n+1;;
```

Types personnalisés

Déclaration de type

Avec Ocaml, il est possible de définir des types de données personnalisés, en plus des types de base déjà rencontrés (`int`, `float`,...). On utilise pour cela le mot clé `type`. Les types que l'on rencontrera le plus souvent sont :

- le type "Somme" (ou énumération)
- le type "Produit" (ou enregistrement)

On va présenter les bases de ces deux types, en laissant pour d'autres cours les fonctionnalités avancées suivantes :

- les types récursifs (dont la définition fait référence au type lui même)
- les types polymorphes (dont la définition fait intervenit un type variable)
- les types mutables (qui peuvent être modifiés)

Le type Somme

La syntaxe pour la déclaration d'un type Somme est la suivante :

```
type couleurDesYeux = Bleu | Vert | Marron;;
```

Ces types doivent commencer par une lettre minuscule. Les valeurs, elles, doivent commencer par une majuscule.

```
# let a = Bleu;;
val a : couleurDesYeux = Bleu
```

Le type Somme

Il est possible que les modalités d'un type Somme soient d'un type déjà défini. Par exemple, si l'on souhaite définir l'union des nombres entiers et des produits d'entiers, on peut écrire

```
type nouveauType = Entier of int | Paire of int*int ;;
```

On peut alors définir une liaison avec la commande

```
# let a = Entier 4;;
val a : nouveauType = Entier 4
```

Le type Somme et le filtrage

Dans la définition précédente, les constructeurs Entier et Paire peuvent être utilisés comme motif de filtrage dans la définition d'une fonction.

```
# let composante1 = function
        | Entier k -> k
        | Paire (a,b) -> a;;
val composante1 : nouveauType -> int = <fun>
# composante1 (Entier 4);;
- : int = 4
# composante1 (Paire (3,4));;
- : int = 3
```

Les types produits

On peut définir un type pour représenter les nombres complexes. Il s'agit d'un type produit :

```
type complexe = float * float
```

Dans cet exemple, on parle d'alias car le type `float*float` existe déjà. On peut le personnaliser en nommant les composantes : on parle de type enregistrement.

```
type complexe = {re: float; im: float};;
```

Le type et les composantes doivent commencer par une lettre minuscule.

Le types enregistrement

Avec les types enregistrement, on peut alors créer des liaisons, et des fonctions qui manipulent ce nouveau type :

```
type complexe = {re: float; im: float};;
let z = {re=1. ; im=1.};;
let conjugue z = {re = z.re; im = -. z.im};;
```

Dans cet exemple, la fonction `conjugue` est de type `complexe -> complexe`.
On accède aux champs d'un enregistrement à l'aide de l'opérateur .

Exceptions

Exceptions : définition

Lorsque Ocaml rencontre un problème dans l'évaluation d'une expression, un message d'erreur s'affiche. Voici deux exemples :

```
# 1/0;;
Exception: Division_by_zero
# let f = function
  | 0 -> 0;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
1
val f : int -> int = <fun>
# f 1;;
Exception: Match_failure ("//toplevel//", 1, 8).
```

Le type exn

Les exceptions sont d'un types particuliers : le type `exn`

Il comporte deux types de valeurs : les exceptions simples (comme `Division_by_zero`) et paramétrées (comme `Failure`, qui doit être associé à une chaine de caractères) :

```
# Division_by_zero;;
- : exn = Division_by_zero
# Failure "oulala!";;
- : exn = Failure "oulala!"
```

Lever une exception

Pour lever une exception, on utilise la fonction `raise`. Failwith est aussi très utilisé, et remplace l'appel de la fonction raise avec une exception construite avec `Failure` . Il permet de disposer d'exceptions personnalisées avec un message d'erreur explicite.

```
# raise Division_by_zero;;
Exception: Division_by_zero.
# raise (Failure "trop grand!!");;
Exception: Failure "trop grand!!".
# failwith "trop grand!";;
Exception: Failure "trop grand!".
```

Rattraper une exception

Dans certains cas, on souhaite que le déclenchement d'une exception n'affiche pas le message d'erreur, mais évalue une expression. On utilise pour cela la structure `try ... with ...`

```
# let safe_div n p = try n/p with Division_by_zero -> 1;;
val safe_div : int -> int -> int = <fun>
# safe_div 6 3;;
- : int = 2
# safe_div 6 0;;
- : int = 1
```

Quelques exercices

1. Écrire une fonction `est_diviseur` en utilisant la fonction `mod`.
2. Écrire une fonction `maximum`.
3. Écrire une fonction `composée`.
4. Définir un type complexe (plusieurs versions)
5. Écrire une fonction `inverse` qui prend en paramètre un complexe de partie réelle a et de partie imaginaire b et affiche $a + ib$.
6. Écrire une fonction `inverse` qui prend en paramètre un complexe et renvoie son inverse lorsque c'est possible et une exception sinon.
7. Écrire une fonction `argument` qui prend en paramètre un complexe et renvoie son argument principal (c'est-à-dire dans $]-\pi, \pi]$) lorsque c'est possible et une exception sinon. On utilisera les fonction `acos` et `asin`

Quelques exercices

On considère les fonctions :

```
let vrai a b = a;;
let faux a b =b ;;
```

Retrouver les opérateurs logiques définis par ces fonctions

```
let m1 p =  p faux vrai;;
let m2 p q = p q faux;;
let m3 p q = p q vrai;;
let m4 p q = p vrai q;;
let m5 p q = p faux q;;
```