Chapitre 1. Introduction
Bases de la programmation en Caml

I Introduction

I.1 Qu'est ce que Caml ?

Caml est un langage de programmation développé par l'INRIA depuis 1985 ; il se range dans la catégorie des langages **fonctionnels**, mais se prête aussi à la programmation **impérative** (et orientée objet pour OCaml). Ces différents types de programmation sont présentés rapidement ci-dessous (cf. figure 1), mais nous en reparlerons plus tard.

Il existe deux implémentations du langage : Caml Light est une version légère destinée jusqu'à présent à l'enseignement et OCaml , qui est la version la plus récente et la plus complète ; elle est destinée à un usage professionnel. Depuis l'année scolaire 2017-2018 **c'est la version préconisée pour les concours**, et celle que nous utiliserons donc en cours, en TD et en TP.

La **programmation impérative** repose sur la notion de machine abstraite possédant une mémoire et des instructions modifiant les états de celle-ci grâce à des affectations successives. Cela nécessite pour le programmeur de connaître en permanence l'état de la mémoire que le programme modifie, ce qui peut se révéler une tâche complexe.

La **programmation fonctionnelle** va mettre en avant la définition et l'évaluation de fonctions pour résoudre un problème, en interdisant toute opération d'affectation. Dans ce mode de programmation, la mémoire est donc gérée automatiquement et de manière transparente pour le programmeur.

La **programmation orientée objet** s'articule autour des données d'un programme. Ceux-ci sont des objets constitués d'attributs les caractérisant et de méthodes les manipulant (ces dernières pouvant être fonctionnelles ou impératives), l'ensemble des attributs et méthodes constituant une classe.

Figure 1 – Les trois paradigmes principaux de la programmation .

Vous trouverez sur la page officielle du langage le manuel complet de OCaml :

`http://ocaml.org/`

I.2 Comment programmer en Caml ?

Se procurer Caml :

- En ligne, sur internet, vous pouvez vous rendre sur

  `https://try.ocamlpro.com/`

  qui présente un éditeur et une boucle interactive (Toplevel) en ligne .
- sous Mac OSX ou Windows
  Vous pouvez essayer de télécharger MacCaml ou WinCaml 7.2 à l'adresse suivante :

  `https://jean-mouric.pagesperso-orange.fr/`

  Mais cette page est maintenant relativement en fouillis et confuse. Pas si facile de s'y retrouver.
  Vous obtiendrez un environnement de développement intégré regroupant un éditeur de texte et un interprète pour l'utilisation interactive de OCaml .
- sous Linux ou Mac OSX
  Le mode Tuareg est un excellent éditeur de code OCaml pour Emacs. Vous pouvez-donc installer emacs, puis le mode Tuareg.

Utilisation interactive : CAML est un **langage compilé** : on écrit les programmes à l'aide d'un éditeur de textes, et le tout est traité par un **compilateur** pour créer un **fichier exécutable**.
Il existe cependant une boucle interactive (un peu comme l'interpréteur PYTHON) qui permet d'interpréter le langage : l'utilisateur tape des morceaux de programme qui sont traitées instantanément par le système, lequel les compile, les exécute et écrit les résultats à la volée. C'est un mode privilégié pour l'apprentissage du langage.
Lors d'une session interactive, le caractère # qui apparait à l'écran est le symbole d'invite du système (le prompt). Le texte écrit par l'utilisateur commence après ce caractère et se termine par deux points-virgules consécutifs, et constitue une **phrase**. Une fois lancé, l'interprète va afficher les types et valeurs des expressions évaluées, avant de réafficher le prompt.

Commençons donc par observer une première session CAML :

Code Caml 1

```
1  >          Objective Caml version 3.12.1
2  # let rec fact = function (* définition d'une fonction *)
3      | 0 -> 1
4      | n -> n * fact (n - 1) ;;
5  val fact : int -> int = <fun>
6  # fact 5;; (* application d'une fonction *)
7  - : int = 120
8  # fact 2.3;; (* une erreur de typage *)
9  Characters 5-8::
10 > fact 2.3 ;;
11 >      ^^^
12 Error: This expression has type float, but an expression was expected of type
       int
13 #
```

Dans ce premier exemple, nous avons commencé par définir une fonction `fact` à l'aide de l'instruction d'affectation `let ... = ...`. Cette définition est **récursive**, car la fonction s'appelle elle-même, comme l'indique le mot-clé `rec`, et procède par **filtrage** (nous reviendrons à la fin de ce chapitre sur ces deux notions), ce qui nous permet d'obtenir une définition très proche de la définition mathématique de la fonction factorielle :

$$n! = \begin{cases} 1 \text{ si } n = 0 \\ n \times (n-1)! \text{ sinon} \end{cases} \tag{1}$$

L'interprète nous répond (ligne 5) que nous avons défini un nouveau nom (fact), qui est de type int -> int (-> se prononce « flèche »), et que ce nom a pour valeur une fonction (<fun>).

Nous pouvons observer que CAML est pourvu d'une reconnaissante automatique de type : cette fonction ne peut s'appliquer qu'à des entiers, et retournera toujours un entier. Nous en avons l'illustration dans la suite de l'exemple : à la commande suivante, l'interprète nous répond (ligne 7) que nous avons simplement calculé une valeur sans la nommer (signalé par le caractère - , que cette valeur est de type int, et qu'elle vaut 120. En revanche, appliquer cette fonction à un nombre non entier (ici de type float) conduit à un message d'erreur. CAML est en effet un langage **fortement typé** : les erreurs de typage sont détectées et **les conversions implicites de type sont formellement interdites**.

I.3 Définitions globales et locales

Attribuer un nom à une valeur à l'aide de l'instruction `let` est une *définition*. Les définitions sont des **liaisons** de noms à des valeurs. Ces définitions ne sont pas modifiables, elles diffèrent donc fondamentalement du mécanisme d'affectation propre à la programmation de style impératif que nous étudierons plus tard (et que nous avons déjà rencontré dans PYTHON).

Une fois défini, un nouveau nom est utilisable dans d'autres calculs :

Code Caml 2

```
# let n = 10 + 2;;
val n : int = 12
# n * n;;
-: int = 144
```

La définition précédente est *globale* : la liaison qui a été établie entre le nom et sa valeur est permanente.

La syntaxe `let ... = ... in` permet de définir temporairement un nom pour la seule durée du calcul en question. C'est une définition **locale**. On dit que la portée du nom est limité au contenu du `in` :

Code Caml 3

```
# let n = 5 in n * n;;
- : int = 25
# n * n;;
-: int = 144
```

On peut observer sur l'exemple précédent que même si elles portent le même nom, variables locale et globale restent bien différenciées.
Notons enfin que le mot `and` permet des définitions multiples (en parallèle et simultanément) :

Code Caml 4

```
# let a = 3 and b = 2 in a + b ;;
- : int = 5
```

Attention, les valeurs ne deviennent visibles qu'après toutes les déclarations simultanées :

Code Caml 5

```
# let x = 1 and y = x + 1 ;;
Characters 18-19::
let x = 1 and y = x + 1 ;;
                  ^
Error: Unbound value
```

Une remarque très importante : `let ... = ... in...` est une **expression** et a donc un résultat, ce qui n'est pas le cas de la liaison globale `let ... = ...;;`. Remarquer la différence de comportement entre les deux phrases suivantes :

Code Caml 6

```
# let x = let a = 2 in a * a ;;
x : int = 4
# let a = 2 in let x = a * a ;;
Characters 26-28:
let a = 2 in let x = a * a;;
                         ^^
Error: Syntax error
```

I.4 Fonctions

I.4.1 Fonction à un seul argument

Définir une fonction en Caml est simple et proche de la notation mathématique usuelle : on utilise principalement la syntaxe `let f arg = expr`, où `arg` désigne l'argument de la fonction et `expr` le résultat souhaité. Par exemple, pour définir la fonction `f : x -> x + 1` on écrira :

Code Caml 7

```
# let f x = x + 1 ;;
val f : int -> int = <fun>
# f 2;;
- : int = 3
```

On peut noter que l'usage des parenthèses n'est ici pas nécessaire :

`f x` est équivalent à `f(x)`.

En revanche, elles peuvent se révéler indispensables pour des expressions plus complexes :

Code Caml 8

```
#f 2 * 3 ;;
- : int = 9
# (f 2) * 3 ;;
- : int = 9
# f (2 * 3) ;;
- : int = 7
```

Cet exemple permet de mettre en évidence le fait que les expressions sont associées à **gauche** :

`f x y` est équivalent à `(f x) y`.

I.4.2 Fonctions à plusieurs arguments

Les fonctions possédant plusieurs arguments se définissent à l'aide de la syntaxe `let f args = expr`, où cette fois-ci args désigne les différents arguments de la fonction, séparés par un espace. Par exemple, pour définir l'application $g : (x, y) -> x + y$ on écrira :

Code Caml 9

```
# let g x y = x + y ;;
val g : int -> int -> int = <fun>
# g 2 3 ;;
- : int = 5
```

Une telle définition de fonction est dite *curryfiée*[1], car elle permet aisément de définir des applications partielles.
On peut en effet remarquer que $f(x) = g(1, x)$ et que l'on aurait donc pu définir la fonction f en écrivant : `let f x = g 1 x`, ou encore `let f x = (g 1) x` suivant la règle de l'association à gauche. Le langage Caml autorisant la simplification à droite, il nous est alors loisible de définir f de la façon suivante :

Code Caml 10

```
# let f = g 1;;
val f : int -> int = <fun>
```

1. du nom du mathématicien Haskell Curry

I.4.3 Fonctions anonymes

Une fonction est donc un objet typé, qui en tant que tel peut être calculé, passé en argument, retourné en résultat. Pour ce faire, on peut construire une valeur fonctionnelle anonyme en suivant la syntaxe `function x -> expr`. Par exemple, les fonctions f et g peuvent aussi être définies de la façon suivante :

Code Caml 11

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

Cette dernière écriture a le mérite de mettre en évidence la raison pour laquelle `f = g 1` (grâce à la commutativité de l'addition) : `g 1` est en effet équivalent à : `function y -> 1 + y`. On peut en outre observer que le typage est associatif à droite :

int -> int -> int est équivalent à int -> (int -> int) .

Attention cependant : il n'est pas permis d'utiliser `function` avec deux variables ou plus (d'où l'écriture un peu lourde de la dernière définition de la fonction g). On peut néanmoins utiliser l'instruction `fun` pour alléger l'écriture d'une fonction anonyme à plusieurs variables, sachant que `fun a b` est un racourci pour `function a ->function b`. On pouvait donc aussi écrire : `let g = fun x y -> x + y`.

II Les types en Caml

Nous l'avons déjà dit, toute valeur manipulée par Caml possède un *type* reconnu automatiquement par le compilateur sans qu'il soit nécessaire de le déclarer : connaissant les types des valeurs de base et des opérations primitives, le contrôleur de types produit un type pour une phrase en suivant des règles de typage pour les constructions du langage (nous en avons eu un premier aperçu avec la définition des fonctions).
Nous allons maintenant passer en revue les principaux types élémentaires ainsi que quelques opérations primitives, avant de décrire les premières règles de construction des types composés.

II.1 Types élémentaires

II.1.1 Le vide

Le type unit est particulièrement simple, puisqu'il ne comporte qu'une seule valeur, notée `()`, et qu'on appelle *void*.
Pour comprendre sa raison d'être, imaginons que l'on veuille afficher une chaîne de caractères à l'écran : nous n'avons rien à calculer, seulement modifier l'environnement (c'est-à-dire réaliser ce qu'on appelle **un effet de bord**). Or un langage fonctionnel ne peut que définir et appliquer des fonctions. Voilà pourquoi la fonction `print_string` est de type string -> unit : elle renvoie le résultat void , tout en réalisant au passage un effet de bord.

Code Caml 12

```
# print_string "Hello World" ;;
Hello World - : unit = ()
```

Autre exemple, la fonction `print_newline` est de type unit -> unit et a pour effet de passer à la ligne. Pour l'utiliser, il ne faut donc pas oublier son argument (qui ne peut qu'être le void) :

Code Caml 13

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

II.1.2 Les entiers

Sur les ordinateurs équipés d'un processeur 64 bits, les éléments de type int sont les entiers de l'intervalle $[-2^{62}, 2^{62}-1]$, les calculs s'effectuant modulo 2^{63} suivant la technique du complément à deux (voir cours d'informatique de tronc commun). De même pour une machine 32 bits on peut manipuler les entiers de l'intervalle $[-2^{30}, 2^{30}-1]$ [2], modulo 2^{31}. Il faut donc veiller à ne pas dépasser ces limites sous peine d'obtenir des résultats surprenants (l'exemple suivant correspond à une machine 64 bits, pour une machine 32 bits, utiliser a = 32768) :

Code Caml 14

```
# let a = 2147483648 in a * a ;;
- : int = -4611686018427387904
# let a = 2147483648 in a * a - 1;;
- : int = 4611686018427387903
```

Vous avez très certainement deviné que $2^{31} = 2147483648$.
Notez que `max_int` et `min_int` vous donnent les valeurs maximales et minimales de type entier :

Code Caml 15

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

Les opérations usuelles se notent `+, -, *` (multiplication), `/` (quotient de la division euclidienne) et `mod` (reste de la division euclidienne). Multiplication, quotient et reste ont priorité face à l'addition et la soustraction, dans les autres cas la règle d'association à gauche s'applique, et on dispose des parenthèses pour hiérarchiser les calculs si besoin est :

Code Caml 16

```
# 2 * 5 / 3 ;;
- : int = 3
# 5 / 3 * 2 ;;
- : int = 2
# 1 + 3 mod 2 ;;
- : int = 2
# (1 + 3) mod 2 ;;
- : int = 0
```

II.1.3 Les réels

Les éléments de type float sont des nombres décimaux (par exemple `3.141592653`) éventuellement accompagnés d'un exposant représentant une puissance de 10 (par exemple `1.789e3`). Les calculs effectués sur les flottants sont bien évidemment approchés.
Les opérateurs usuels se notent : `+., -., *., /.` (noter le **point** qui permet de les différencier des opérateurs sur le type int), `**` (élévation à une puissance). On dispose en outre d'un certain nombre de fonctions mathématiques telles que : `sqrt` (racine carrée), `log` (qui désigne le **logarithme népérien** !), `exp`, `sin`, `cos`, `tan`, `asin` (arcsinus), `acos`, `atan` ...

Code Caml 17

```
# let a = 3.141592654 in tan(a /. 4.) ;;
- : float = 1.00000000021
```

2. On peut se demander pourquoi on n'utilise que 31 des 32 bits. Ceci est dû à une optimisation de la gestion des entiers qui réserve un bit à des fins personnelles...

Attention à ne pas oublier que Caml est un langage fortement typé : il n'y a pas de conversion implicite de type, et vous ne pouvez donc pas appliquer une fonction du type float à une valeur de type int :

Code Caml 18

```
# let a = 3.141592654 in tan (a /. 4) ;;
Characters 32-33::
let a = 3.141592653 in tan ( a /. 4) ;;
                                    ^
Error: This expression has type int but an expression was expected of type
    float
```

II.1.4 Caractères et chaînes de caractères

Les caractères sont les éléments du type char ; on les note en entourant le caractère par le symbole ' (quote sous la touche 4). Les éléments de type string sont des chaînes de caractères entourées du symbole " (double quote sous la touche 3). Nous reviendrons sur la notion de chaîne de caractères lorsque nous aborderons les vecteurs ; pour l'instant, contentons nous de noter quelques fonctions qui pourront nous être utiles :

- La concaténation des chaînes de caractères est notée par le symbole ^ :

Code Caml 19

```
# "liberté, " ^ "égalité, " ^ "fraternité" ;;
- : string = " liberté , égalité , fraternité"
```

- La fonction prédéfinie `String.length` du module **String** renvoie la longueur d'une chaîne de caractères ; c'est donc une fonction de type string -> int.
- La fonction `String.get` renvoie le n-ième caractère d'une chaîne de caractères (attention, dans une chaîne les indices commencent à 0). C'est une fonction de type string -> int -> char.
- La fonction `String.sub` retourne une sous-chaîne partant d'un indice donné et d'une longueur donnée ; c'est une fonction de type string -> int -> int -> string.

Code Caml 20

```
# String.get "Caml" 1 ;;
- : char = 'a'
# String.sub "informatique" 2 6 ;;
- : string = "format"
```

II.1.5 Les booléens

Le type bool comporte deux valeurs : le vrai (`true`) et le faux (`false`), et dispose des opérations logiques non (`not`) et (`&&`) et ou (`||`). Les deux derniers opérateurs fonctionnent suivant le principe de **l'évaluation paresseuse** : `expr1 && expr2` ne va évaluer la deuxième expression que si la première est vraie, et `expr1 || expr2` ne va évaluer la deuxième expression que si la première est fausse.

Code Caml 21

```
# not (1 = 2) || (1 / 0 = 1) ;;
- : bool = true
# not (1 = 2) && (1 / 0 = 1) ;;
Exception: Division_by_zero
```

II.1.6 Les paires

Il est possible de définir le produit cartésien de deux types : si `x` est un élément de type 'a et `y` un élément de 'b, alors `(x, y)` est un élément de type 'a * 'b.
Par exemple, la version non currifiée de la fonction $g : (x, y) \to x + y$ se définit de la façon suivante :

Code Caml 22

```
# let g (x, y) = x + y ;;
val g : int * int -> int = < fun >
```

Il est bien entendu possible de faire le produit cartésien d'un nombre plus important de types, mais seuls les paires disposent des fonctions `fst` et `snd` pour récupérer le première et la seconde composante du couple :

Code Caml 23

```
# fst ("caml", 0.75) ;;
- : string = "caml"
# snd ("caml", 0.75) ;;
- : float = 0.75
```

II.2 Polymorphisme

On a pu constater que Caml est pourvu d'une reconnaissance automatique de type : par exemple, c'est la présence de l'entier `1` et de l'opérateur entier `+` qui permet d'associer à la fonction `function n -> n + 1` le type int -> int .
Mais il peut arriver qu'une fonction puisse s'appliquer indifféremment à tous les types ; on dit dans ce cas que cette fonction est *polymorphe*. On utilise alors les symboles 'a, 'b, 'c ,... pour désigner des types quelconques.
Revenons par exemple sur les fonctions fst et snd qui agissent sur les paires. Leur définition est évidente, et n'imposent aucune contrainte de typage :

Code Caml 24

```
# let fst (x, y) = x ;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (x, y) = y ;;
val snd : 'a * 'b -> 'b = <fun>
```

Nous avons vu que les opérations algébriques, l'addition par exemple, ne sont pas polymorphes : l'opérateur infixe `+` est de type int -> int alors que `+.` est de type float -> float . En revanche, les opérations de comparaison telle l'égalité `=` le sont. On peut utiliser le même opérateur pour comparer des entiers ou des nombres complexes (ou tout autre objet pour lesquels la comparaison a un sens) :

Code Caml 25

```
# 1 = 1 ;;
- : bool = true
# 1.0 = 2.0 ;;
- : bool = false
# (=) ;; (* La version préfixe de l'égalité *)
- : 'a -> 'a -> bool = <fun>
# (=) 1 1 ;;
- : bool = true
```

Les autres opérateurs de comparaison polymorphes sont `<>` (la négation de l'égalité), `<`, `>`, `<=` (inférieur ou égal), `>=` (supérieur ou égal).

Code Caml 26

```
# "anaconda" < "zebre" ;;
- : bool = true
```

(La comparaison des chaînes de caractères utilise l'ordre lexicographique.)

III Définition d'une fonction

Nous l'avons déjà dit, une fonction se définit à l'aide de la syntaxe `let f = function ... -> ...` et son type est alors de la forme type1 -> type2 , le typage étant implicite, et polymorphe le cas échéant. Cette syntaxe est en outre équivalente à `let f ... = ...`
Rappelons aussi que `let f = fun x y -> ...` est la forme raccourcie de :

```
let f = function x -> function y -> ...
```

et qu'on peut aussi écrire `let f x y = ...` Son type est alors type_de_x -> type_de_y -> type_du_résultat .

III.1 Analyse par cas : le filtrage

Le **filtrage** est un trait extrêmement puissant de Caml , et à ce titre très fréquemment employé. La syntaxe d'une fonction travaillant par filtrage peut prendre une des formes suivantes :

Code Caml 27

```
let f = function
  | motif1 -> expr1
  | motif2 -> expr2
  | ..............
  | motifn -> exprn ;;
```

Code Caml 28

```
let f x = match x with
| motif1 -> expr1
| motif2 -> expr2
| ..............
| motifn -> exprn ;;
```

Lors du calcul d'une telle fonction, l'interprète de commande va essayer de faire concorder l'argument avec les motifs successifs qui apparaissent dans la définition. C'est l'expression correspondant au premier motif reconnu qui sera calculée. Bien entendu, il est nécessaire que les différentes expressions du résultat soient du même type.
Lorsque le motif est un nom, il s'accorde avec n'importe quelle valeur, et la reçoit dans l'expression exécutée. Par exemple, la définition de la fonction sinus cardinal sera :

Code Caml 29

```
# let sinc = function
  | 0. -> 1.
  | x -> sin(x) /. x ;;
sinc : float -> float = <fun>
```

Lorsqu'il n'est pas nécessaire de nommer un motif, on utilise le caractère _ qui s'accorde avec n'importe quelle valeur (tout comme un nom), à la différence près que cette valeur est perdue après filtrage :

Code Caml 30

```
# let est_nul x =
    match x with
    | 0 -> true
    | _ -> false ;;
est_nul : int -> bool = <fun>
```

III.2 Motif gardé

Pour filtrer sur des cas que l'on ne peut exprimer structurellement, il est possible d'utiliser un motif gardé dont la syntaxe est :

Code Caml 31

```
| motif when condition ->
```

Dans un tel cas de filtrage, le motif n'est reconnu que si la condition (une expression de type bool) est vérifiée. Nous reparlerons et utiliserons cette possibilité plus tard.

III.3 Récursivité

Considérons de nouveau la définition de la fonction factorielle que nous avons donnée en introduction :

Code Caml 32

```
>          Objective Caml version 3.12.1
# let rec fact = function (* définition d'une fonction *)
    | 0 -> 1
    | n -> n * fact (n - 1) ;;
val fact : int -> int = < fun >
```

ou encore en utilisant la structure `match ... with` , en notant l'utilisation de `_`

Code Caml 33

```
# let rec fact n =
    match n with
    | 0 -> 1
    | _ -> n * fact (n - 1) ;;
val fact : int -> int = < fun >
```

Le mot clé `rec` indique que nous avons défini une fonction récursive, c'est-à-dire une fonction dont le nom intervient dans sa définition. Nous approfondirons plus tard cette notion, contentons-nous pour l'instant d'observer sa proximité avec nombre de définitions mathématiques, à commencer par les suites récurrentes.
La suite de Fibonacci, par exemple, peut être définie ainsi :

Code Caml 34

```
#let rec fib n
   match n with
   | 0 -> 1
   | 1 -> 1
   | n -> fib (n - 1) + fib (n - 2) ;;
```

(Nous apprendrons plus tard que cette définition n'est pas très efficace, mais ce n'est pas la question pour l'instant.)
Notons enfin qu'il est possible de définir deux fonctions mutuellement récursives à l'aide de la syntaxe `let rec f = ... and g = ...`. Les deux fonctions suivantes, par exemple, déterminent la parité d'un entier naturel :

Code Caml 35

```
# let rec est_pair = function
    | 0 -> true
    | n -> est_impair (n -1)
  and est_impair = function
  | 0 -> false
  | n -> est_pair (n - 1) ;;
est_pair : int -> bool = <fun>
est_impair : int -> bool = <fun>
```