# Cours d'informatique

## MP2I

Lou Chalmain

2022-20223

# Table des matières

# Chapitre 1

# Initiation à Ocaml

Ocaml est un langage de programmation relevant notamment du paradigme de la programmation fonctionnelle, c'est à dire qu'il repose sur l'évaluation de fonctions, mais permettant également d'adopter un style de programmation impérative.

Un programme Ocaml est une suite d'expressions et de déclarations séparées par un double point-virgule `;;` dans lesquelles l'indentation et les retours à la ligne n'a pas d'importance mais facilitent grandement la lisibilité et la compréhension du code.

Une expression peut elle-même être formée de plusieurs expressions séparées d'un point-virgule `;`. Dans ce cas c'est l'évaluation de la dernière expression qui est renvoyé.

## 1.1 Déclaration de variables

### 1.1.1 Variables et types

La déclaration d'une variable s'effectue en Ocaml grâce au mot-clé `let` selon la syntaxe suivante :

```
let ma_variable = ma_valeur;;
```

**Exemple 1.1.**

```
# let a = 8;;
val a : int = 8
```

La sortie indique qu'on vient de créer une ***variable*** dont :

- le ***nom*** est `a`
- le ***type*** est `int`
- la ***valeur*** est `8`

Pour faciliter la compréhension du code on choisit des noms de variables explicites lorsque cela fait sens (comme `prix` ou `distance` plutôt que `a` ou `x`).
Dans tous les cas on choisira des mots ***sans espaces*** et ***non accentués*** (on pourra utiliser le symbole `_`).

**Exemple 1.2.**

```
# let duree_en_h = 3.5;;
val duree_en_h : float = 3.5
```

Une variable définie comme ci-dessus est ***immuable*** (sa valeur n'est pas modifiable) et possède obligatoirement un ***type***, on parle de ***typage statique***.
Le type est ***inféré*** par le compilateur ou l'interpréteur : il est déduit des différentes fonctions et opérateurs impliqués dans l'expression évaluée lors de la déclaration.

Le tableau suivant récapitule les types de base et les opérateurs associés :

| | Type | Opérateur | |
|---|---|---|---|
| Entiers relatifs | `int` | + | addition |
| | | - | soustraction |
| | | * | multiplication |
| | | / | division euclidienne |
| | | mod | reste de la diviion euclidienne (si positifs) |
| Nombres à virgule flottante | `float` | +. | addition |
| | | -. | soustraction |
| | | *. | multiplication |
| | | /. | division exacte |
| | | ** | puissance |
| Booléen (`true` ou `false`) | `bool` | `not` | négation |
| | | `&&` | ET logique |
| | | `\|\|` | OU logique |
| Caractère | `char` | | ex : `'c'` |
| Chaîne de caractères | `string` | | ex : "une chaine" |
| Rien | `unit` | | `()` est de type `unit` |

**Remarque 1.1.** Les nombres à virgule flottante (ou ***flottants***) s'écrivent avec un point et non une virgule ! Sinon, il s'agira d'un couple d'entiers (type `int * int`).

**Exemple 1.3.**

```
# 2.718;;
- : float = 2.718
# 2,718;;
- : int * int = (2, 718)
```

### 1.1.2 Overflow

Les entier et les flottants sont sujets au dépassement de capacité. En effet, les variables sont stockées dans la mémoire RAM de l'ordinateur, or celle-ci n'est pas infinie.

• Le plus grand entier représentable est stocké dans la variable `max_int`, si on augmente ce nombre de 1, on tombe sur le plus petit entier représentable.
On parle de dépassement d'entier ou encore d'***integer overflow***.

• De même, `max_float` est le plus grand flottant représentable.

**Exemple 1.4.**

```
# max_int;;
- : int = 4611686018427387903
# max_int + 1;;
- : int = -4611686018427387904
# max_float;;
- : float = 1.79769313486231571e+308
```

**Remarque 1.2.** Le dépassement d'entier est notamment responsable du crash de la fusée Ariane 5...

### 1.1.3 Comparaisons et type bool

Pour effectuer une comparaison entre deux objets de même type on utilise les opérateurs `=`, `<`, `>`, `<=`, `>=` ou `<>` (pour tester la différence).
Le résultat est de type `bool` : `true` (vrai) ou `false` (faux).

**Exemple 1.5.**

```
# 4.2 < 2.4;;
- : bool = false
# 7 >= (-2);;
- : bool = true
# 0.1 +. 0.2 <> 0.3;;
- : bool = true          (* ce resultat est plus etonnant, mais deja vu plus haut *)
# 64 mod 5 = 0;;
- : bool = false
```

Les opérateurs `&&` et `||` effectuent une ***évaluation paresseuse*** des expressions situées de part et d'autre, cela signifie que :

• pour évaluer l'expression `a && b` (ET logique), c'est l'expression `a` qui est évaluée la première.

⟶ si le résultat est `true`, l'évaluation de `b` est retournée
⟶ sinon c'est directement `false` qui est retourné

• pour évaluer l'expression `a || b` (OU logique), c'est l'expression `a` qui est évaluée la première.

⟶ si le résultat est `false`, l'évaluation de `b` est retournée
⟶ sinon c'est directement `true` qui est retourné

**Exemple 1.6.**

```
# 4 < 6 && 1024 <= 1024;;
- : bool = true
# 5.2 > 8. || 13 mod 2 = 0;;
- : bool = false
# not (4 = 7);;
- : bool = true
```

### 1.1.4 Variables locales

La construction vue ci-dessus permet de définir une ***variable globale***.
Une variable est dite ***locale*** lorsqu'elle n'est définie que dans le cadre d'une expression.
On utilise pour cela la construction `let ... in`

**Exemple 1.7.** On souhaite convertir 47 heures en minutes :

```
# let min_par_h = 60 in
      47 * min_par_h;;
- : int = 2820
# min_par_h;;
Error: Unbound value min_par_h     (* min_par_h n'est plus definie *)
```

### 1.1.5 Références

On a vu que les variables définies comme précédemment étaient ***immuables*** (leur valeur n'est pas modifiable).
Lorsqu'on souhaite utiliser des variables ***mutables*** on utilise des ***références***, que l'on déclare à l'aide du mot-clé `ref` en adoptant la syntaxe suivante :

```
let ma_ref = ref ma_valeur;;
```

• L'accès à la valeur de la réfécence se fait en ajoutant un ! devant son nom : `!ma_ref`

• L'assignation d'une nouvelle valeur s'effectue selon la syntaxe `ma_ref := autre_valeur;;`

**Exemple 1.8.**

```
# let x = ref 3;;
val x : int ref = {contents = 3}
# x := 4 * !x;;
- : unit = ()
# x;;
- : int ref = {contents = 12}
# !x;;
- : int = 12
```

**Remarque 1.3.** Le type d'une référence dont la valeur est de type `int` est de type `int ref`.

**Remarque 1.4.** On n'utilisera des références que lorsque c'est vraiment nécessaire.

## 1.2 Fonctions

### 1.2.1 Déclaration et signature

On rappelle que Ocaml est avant tout un langage de ***programmation fonctionnelle***. Les fonctions y occupent donc naturellement une place importante.

Une fonction Ocaml ne devrait pas générer d'***effet de bord***. On parle d'effet de bord lorsque l'évaluation d'une expression affecte l'environnement extérieur à cette expression, par exemple en modifiant la valeur d'une variable globale, en faisant de l'affichage sur écran ou bien en modifiant le contenu d'un fichier.

Pour déclarer une fonction on utilise la syntaxe suivante : `let fonction argument = expression;;`

**Exemple 1.9.**

```
# let carre x = x * x;;
val carre : int -> int = <fun>
# carre 12;;      (* inutile de mettre des parentheses *)
- : int = 144
# carre (-9);;      (* sauf cas particuliers *)
- : int = 81
```

Une fonction possède une ***signature***, inférée par le compilateur ou l'interpréteur.
En notant `type_entree` le type de `argument` et `type_sortie` le type de `expression`,
la signature de la fonction est : `type_entree -> type_sortie`

**Exemple 1.10.** La fonction `carre` définie plus haut à pour signature `int -> int`.
Pour déclarer une fonction à plusieurs variables on utilise la syntaxe :

```
let fonction arg1 arg2 arg3 ... = expression;;
```

La signature d'une telle fonction est alors de la forme : `type1 -> type2 -> type -> ... -> type_sortie`

En réalité il s'agit d'une suite de fonctions à un seul argument imbriquées entre elles. C'est ce qu'on appelle la ***curryfication***. Un avantage notable est qu'on peut appliquer partiellement une fonction.

**Exemple 1.11.**

```
# let prod x y = x * y;;
val prod : int -> int -> int = <fun>
# prod 2;;
- : int -> int = <fun>
# let double = prod 2;;
val double : int -> int = <fun>
# double 3;;
- : int = 6
```

On trouve fréquemment des variables locales dans des fonctions.

**Exemple 1.12.**

```
# let conversion_h_min heure = let min_par_h = 60 in
            heure * min_par_h;;
val conversion_h_min : int -> int = <fun>
# conversion_h_min 3;;
- : int = 180
```

### 1.2.2 Polymorphisme

Il peut arriver que les types ne puissent par être inférés car il n'y a pas de contrainte particulière induite par l'expression évaluée (contrairement à l'utilisation de l'opérateur + par exemple).
Dans ce cas on note les différents types possibles `'a`, `'b`, etc. On conserve la même notation lorsque deux type doivent être identiques dans une signature.

**Exemple 1.13.**

```
# let first x y = x;;
val first : 'a -> 'b -> 'a = <fun>
# let second x y = y;;
val second : 'a -> 'b -> 'b = <fun>
# first 4 8;;
- : int = 4
# second true false;;
- : bool = false
```

### 1.2.3 Type unit

Certaines expressions ne renvoient aucune valeur après évaluation, ce qui se note `()`. Ces expressions sont de type `unit`. Celles-ci peuvent en revanche générer des effets de bord. C'est le cas par exemple des fonctions `print_int`, `print_float`, `print_string` et `print_newline` qui permettent de faire de l'affichage sur écran.

**Exemple 1.14.**

```
# print_string "la racine de ";
  print_int 16;
  print_string " vaut ";
  print_int 4;
  print_newline;;
la racine de 16 vaut 4- : unit -> unit = <fun>
```

## 1.3 Structure if ... then ... else ...

L'expression `if condition then expr1 else expr2` évalue d'abord l'expression `condition`, qui doit être un booléen. Si le résultat est `true`, c'est `expr1` qui est évaluée puis retournée. Sinon, c'est `expr2`.

**Exemple 1.15.**

```
# let abs x = if x > 0.
                then x
            else -. x
;;
val abs : float -> float = <fun>
```

## 1.4 Structures itératives

**Exemple 1.16.**

~ *Fin* ~