# TP de programmation n° 1

Julien Reichert *

Mardi 1er octobre 2013

**AVERTISSEMENT** : Le langage de programmation de référence pour ces séances est OCaml. Si vous souhaitez utiliser un autre langage lors de l'épreuve de modélisation, faites-m'en part dès aujourd'hui. [1] [2]

### Le premier pas du débutant

Ouvrez un terminal et tapez `ocaml` (voire, si possible, `ledit ocaml` [3]). Ouvrez aussi un éditeur de texte (ou **e**nvironne**m**ent de bure**a**u **c**rade **s**ans éditeur de texte) pour sauvegarder vos bouts de code.

## 1 La base

### 1.1 Les expressions

Une expression en OCaml se termine par `;;`, du moins lorsqu'elle est saisie dans un interpréteur. Par exemple, entrez

```
2+2;;
print_string "Hello World!";;
```

Notez que l'expression interprétée est assortie d'un type.

Une expression peut consister en plusieurs instructions, qui sont alors séparées par des `;`.

*TPs inspirés de ceux de César Rodríguez (2012/2013)

1. ... afin que je vous fasse changer d'avis !
2. Non, Jill-Jênn, on ne fera pas de Python !
3. Je vous laisse faire le jeu de mots qui s'impose

```
print_string "Hello "; print_string "World!";;
```

Pour définir une variable, une fonction ou quelque nouvel élément que ce soit, on utilise le mot-clé `let`.

```
let answer = 42;;
let my_list = [19;12;1988];;
```

Encore une fois, le type de la variable ainsi créée est ajouté.

Les définitions de variables peuvent être locales, grâce à `in`.

```
let x = 6 in x * 9;;
let unbound_error = x * 9;;
```

## 1.2 Les types classiques

Il est possible de créer des types en OCaml, et nous y reviendrons. En attendant, les principaux types existants sont `int`, `float`, `bool`, `unit`, `string`, `char`, `'a list`[4], `'a array` entre autres.

Certaines fonctions permettent de passer d'un type à un autre, car OCaml n'accepte pas d'utiliser un type à la place d'un autre, ce qui conduit à l'existence de plusieurs versions d'une même fonction. Les fonctions de changement de type figurent dans le manuel en ligne, mais mieux vaut avoir une intuition du nom que l'on recherche, il s'agit par exemple de `int_of_float` (valeur entière) et de `Array.of_list`[5].

Au vu du paragraphe précédent, l'arithmétique n'est pas si triviale qu'il ne paraît : l'addition se fait par l'opérateur infixe `+` quand il s'agit de deux entiers mais `+.` quand il s'agit de deux flottants (ce dernier type regroupe tous les nombres pourvus d'un point). De même pour `-`, `*` et `/`, ce dernier ne tenant pas compte du reste dans la division entière. L'opérateur `mod` n'existe que pour les entiers, et les fonctions de type `log` et `sqrt` que pour les flottants.

Les comparaisons se font à l'aide de `<`, `>`, `<=`, `>=`, `=` et `!=`. Les opérateurs booléens sont `&&`, `||` et `not`.

Notez que si une expression consiste en plusieurs instructions, toutes sauf la dernière (éventuellement) doivent être de type `unit`. Le type global de l'expression sera alors celui de la dernière instruction.

4. toute ressemblance avec le nom d'un BDE est fortuite
5. Il ne m'a fallu que 9 essais dans un terminal pour la retrouver !

Quelques mots sur les chaînes de caractères. Elles se concatènent via l'opérateur infixe `^`, on accède au `(i+1)`$^{\texttt{e}}$ caractère de `str` par `str.[i]` et elles s'impriment de plusieurs façons différentes :

```
print_string str;;
print_endline str;;
Printf.printf "%s" str;;
```

### 1.3 Les fonctions

Pour définir une fonction, on utilise `let` ainsi que le mot-clé `fun` (ou `function` à condition qu'il n'y ait qu'un argument), avec la syntaxe suivante :

```
let increment = function x -> x + 1;;
let add = fun a b -> a + b;;
```

On peut aller plus vite en omettant le mot-clé :

```
let increment x = x + 1;;
let add a b = a + b;;
```

On peut aussi se servir d'une fonction sans la nommer :

```
(fun x -> x * x) 12;;
```

Et si la fonction n'a pas d'argument, on écrit `()` (de type `unit`).

```
let fun_42 () = print_string "42";;
```

Le type d'une fonction est `<type du premier argument> (-> <type du deuxième argument> ...) -> <type de la sortie>`.

Notez aussi qu'il n'y a pas besoin de parenthèses autour du ou des arguments d'une fonction, tant qu'il n'y a pas d'ambiguïté [6] possible.

## 2 Une variable, c'est fait pour varier !

Par défaut, une variable qui n'est pas un tableau...ne varie pas. Les variables modifiables sont les références (d'entiers, de listes, etc.) créées en ajoutant le mot-clé `ref`. On accède au contenu d'une référence en précédant son nom de `!`

6. c'est toi l'ennéacontaphobe

(lui-même précédé d'une espace, sans quoi OCaml déclenchera une erreur), et on la modifie avec `:=`.

```
let answer = ref 42;;
!answer;;
answer := 6 * 9;;
```

Pour in/décrémenter une référence entière, on peut utiliser directement `incr` ou `decr`, qui sont donc de type `int ref -> unit`.

# 3 Structures de contrôle principales

La disjonction de cas a pour syntaxe `if <condition> then <expression1> [else <expression2>]`. La condition doit être de type `bool` et les expressions 1 et 2 de même type. En particulier, si on omet la partie `else`, l'expression 1 doit être de type `unit`.

La boucle conditionnelle a pour syntaxe `for <variable1>=<variable2> [down]to <variable3> do <bloc> done;;`, le pas étant toujours de 1 ou -1. Ici aussi, le type des variables 1 à 3 est fixé : ce sont des entiers.

La boucle inconditionnelle a pour syntaxe `while <condition> do <bloc> done;;`. La condition est un booléen.

```
let x = ref 2 in while incr x; !x < 5 do print_int !x done;;
```

# 4 Filtrage

Pour ne pas avoir à enchaîner les `if`, il est possible de faire un filtrage similaire au `switch ... case` présent dans d'autres langages de programmation. Il consiste à précéder chaque cas (sauf éventuellement le premier) d'une barre verticale, et éventuellement de contraindre les variables grâce au mot-clé `when` suivi d'un booléen, dans des expressions de type :

```
let abs = function
0 -> 0
| x when x > 0 -> x
| x -> -x;;
```

Notez que si vous ne couvrez pas tous les cas, OCaml déclenchera un aver-

tissement (et vous fera confiance pour que les cas restants n'apparaissent jamais). Il est également possible d'utiliser la variable muette _. En-dehors d'une déclaration de fonction, la syntaxe est (ici dans un contexte où `x`, de n'importe quel type, est utilisé) :

```
match x with
| 0 -> 1
| 1 -> 0
| _ -> 2;;
```

# 5 Récursion

Une fonction peut être rendue récursive en ajoutant au moment de la définition le mot-clé `rec`. Cela marche même pour d'autres types d'expressions, testez par exemple

```
let rec l = 1::2::3::l;;
```

Ainsi, une fonction comme celle de McCarthy s'écrira

```
let rec mccarthy = function
x when x > 100 -> x - 10
| x -> mccarthy (mccarthy (x + 11));;
```

# 6 Exercices

**Exercice 1** : Implémentez la factorielle de trois façons, sous forme de fonction récursive, avec `while` et avec `for`.

**Exercice 2** : Implémentez le tri fusion en OCaml [7].

**Exercice 3** : Implémentez un algorithme qui convertit un chiffre arabe entre 1 et 3999 en chiffres romains.

**Exercice 4** : Implémentez un algorithme de décomposition en facteurs premiers.

7. et si vous êtes pressés le jour de l'oral, utilisez `List.sort` ou `Array.sort`