Initiation à OCaml

I. Découvert de l'environnement

1. Terminal et ligne de commande

Nous allons nous familiariser un peu avec les terminaux qui sont des programmes permettant d'interagir avec l'ordinateur de façon purement textuelle. Pour être précis, le terminal fait tourner un programme, un *shell* et c'est avec ce dernier que nous allons interagir.

```
olibrunet@MACHINE-218:~$ _
```

1.1. Navigation

La commande « `ls` » (pour *list*) permet d'afficher la liste des fichiers.

```
$ ls
Bureau      gravity.py             nsmail        secret
Documents   Informatique Commune   public_html   Téléchargements
```

On peut se poser la question suivante : Quels sont les fichiers affichés ? Où sont-ils ? En fait, dans le shell, on se trouve toujours dans un répertoire particulier, le répertoire de travail ou *working directory*. Pour l'afficher, on peut utiliser la commande « `pwd` » (pour *print working directory*).

```
$ pwd
/home/olibrunet
```

Dans le résultat affiché (on parle de *chemin* pour un enchaînement de répertoires imbriqués), on trouve le répertoire à la racine noté « `/` », dans lequel se trouve le répertoire « `home` » dans lequel se trouve le répertoire « `olibrunet` ».

Pour changer de répertoire, on peut utiliser la commande « `cd` » (pour *change directory*) suivie d'un chemin.

```
$ cd /
$ ls
bin    etc    lib32        media   opt    run       srv   usr
boot   home   lib64        mnt     proc   sbin      sys   var
dev    lib    lost+found   mnt2    root   selinux   tmp
```

1.2. Quelques mots sur les chemins

À la base, un chemin est une succession de noms de répertoires séparés par des

« / » (avec éventuellement un nom de fichier à la fin, mais alors pas pour la commande `cd`).

On peut distinguer plusieurs points de départ. Ainsi, le chemin part

- ⋆ de la racine s'il commence par « / » ;
- ⋆ du répertoire personnel de `machin` s'il commence par « `~machin/` » ;
- ⋆ de **votre** répertoire personnel s'il commence par « `~/` » ;
- ⋆ du répertoire de travail dans les autres cas.

Dans les trois premiers cas, on parle de chemin absolu, alors que pour le dernier, on a un chemin relatif.

Terminons par deux autres répertoires spéciaux :

- ⋆ « `.` » désigne le répertoire de travail ;
- ⋆ « `..` » désigne le répertoire parent du répertoire actuel.

Exemple On suppose que l'on a deux utilisateurs nommés `alice` et `bob` dont les répertoires personnels se trouvent dans `/home`. On commence chez Alice :

```
$ pwd
/home/alice
$ cd ..
$ pwd
/home
$ cd ..
$ pwd
/
```

```
$ cd ~bob
$ pwd
/home/bob
$ cd ~/Bureau
$ pwd
/home/alice/Bureau
$ cd ../Documents
/home/alice/Documents
```

Notons que la commande `ls` peut elle aussi être appelée avec un chemin vers un répertoire. C'est alors le contenu du répertoire en question qui est affiché.

Exercice 1

- ⋆ Allez dans le répertoire personnel du professeur, et revenez chez vous.
- ⋆ En supposant que tous les chemins sont corrects, connaissant à chaque fois le répertoire de départ (obtenu à l'aide de `pwd`) et le changement de répertoire effectué, indiquer le répertoire d'arrivée.
  1. « `cd Photos` » depuis « `~Alice` » ;
  2. « `cd ~Bob/Photos/Noel2020` » depuis « / » ;
  3. « `cd Musiques/..` » depuis « `~Bob` ».

1.3. Manipulation de fichiers

Voici quelques fonctions utiles pour manipuler des fichiers.

- « `cp` » (pour *copy*) créée une nouvelle copie d'un fichier. Ainsi,

```
cp aaa Dossier/bbb
```

  va créer une copie du fichier `aaa` situé dans le répertoire courant, intitulée `bbb` et située dans le répertoire `Dossier`.
  Pour dupliquer un répertoire (avec son contenu), on précisera l'option `-r` (pour *récursif*). Ainsi, pour créér une copie du répertoire `Dossier` nommée `Dossier2`, on exécutera la commande

```
cp -r Dossier Dossier2
```

- « `mv` » (pour *move*) modifie le nom ou l'emplacement d'un fichier (ou les deux, bien sûr). Ainsi « `mv aaa bbb` » renomme le fichier `aaa` en `bbb` alors que

```
mv aa Dossier/
```

  le déplace dans le répertoire `Dossier`. On peut combiner les deux en tapant « `mv aaa Dossier/bbb` ».
- « `rm` » (pour *remove*) supprime un fichier.

Concernant spécifiquement les répertoires, on a :

- « `mkdir` » pour créer un répertoire.
- « `rmdir` » pour supprimer un répertoire vide. Notons que l'on peut aussi supprimer un répertoire non vide en utilisant la commande `rm` avec l'option `-r` mais cela est à utiliser avec précautions.

Exercice 2

- Copier chez vous, avec son contenu, le répertoire nommé `option_sup` présent dans mon répertoire personnel (mon nom d'utilisateur étant `olibrunet`).

2. L'éditeur Emacs

L'éditeur de texte Emacs, avec le mode `Tuareg`, permet de programmer en OCaml

Voici tout d'abord quelques commandes de base, valables pour tout type de fichiers sous Emacs.

- `Ctrl` + `X` suivi de `Ctrl` + `F` : ouvrir un fichier
- `Ctrl` + `X` suivi de `Ctrl` + `S` : sauvegarder
- `Ctrl` + `X` suivi de `Ctrl` + `C` : quitter

Concernant l'utilisation spécifique d'Emacs pour des fichiers OCaml (finissant en `.ml`), on ajoutera :

⋆ Ctrl + C suivi de Ctrl + E : évaluer l'expression OCaml sous le curseur.

Exercice 3 Ouvrez avec Emacs le fichier `tp1.ml` présent dans le répertoire `option_sup` que vous avez copié précédemment, en entrant la commande `emacs` suivie du chemin vers ce fichier.

II. Prise en main d'OCaml

Dans la suite, vous allez avoir une longue liste de commande simple à entrer dans l'interpréteur OCaml. Faites en sorte de comprendre ce que fait chaque commande et, en cas d'erreurs (il y en aura), de les comprendre elles aussi.

Chaque commande est terminée par `;;`, et rien n'est exécuté tant que ces points-virgules n'ont pas été entrés.

Les parties du type `(* Bonjour, comment allez-vous ? *)` sont des commentaires.

1. Entiers et flottants

```
2 + 2 ;;
11 / 2 ;;  (* division entière *)
1 + 0.5 ;;          (* interdit *)
.5 ;;               (* interdit *)
2. ;;               (* autorisé *)
2. + 0.5 ;;         (* problème *)
2. +. 0.5 ;;
```

```
float_of_int ;;
float_of_int 42 ;;
int_of_float 3.1415 ;;
cos 1 ;;            (* problème *)
cos ;;             (* la preuve *)
cos 3.1415926535 ;;
cos 5. ** 2. +. sin 5. ** 2. ;;
```

2. Listes et tableaux

Les listes se construisent par ajout à gauche d'un élément à une liste, noté « `::` ». On peut aussi définir une liste en notant ses éléments entre crochets, séparés par un point-virgule. La liste vide se note « `[]` ».

```
[1; 2; 3] ;;
[1; 2.0; 3] ;;    (* interdit *)
1 :: 2 :: [3] ;;
1 :: [] ;;
1 :: [2; 3] ;;
1 :: 2 ;;         (* interdit *)
[2] :: 1 ;;       (* interdit *)
[1] :: [2] ;;     (* interdit *)
[1] :: [[2]] ;;   (* regardez bien le type *)
```

Les tableaux se notent entre « `[|` » et « `|]` », avec les différents éléments séparés eux aussi par des points-virgule. On rappelle qu'ils sont de taille fixe.

```
[| 1; 2; 3 |] ;;
[| 1; 2; 3 |].(2) ;;
[| 1; 2.0; 3 |] ;;           (* interdit *)
[| 1; 2 |] + [| 3; 4 |] ;;   (* interdit *)
Array.make 10 true ;;
Array.length [| 1; 2; 3 |] ;;
```

3. Caractères et chaînes de caractères

Les caractères sont entourés de `'`, les chaînes de caractères de `"`.

```
'a' ;;
"Bonjour" ;;
'Bonjour' ;; (* interdit *)
"Bonjour".[3] ;;
'\n' ;;      (* autorisé *)
"Bon" ^ "jour" ;;
```

Pour la dernière expression, un certain nombre de caractères peut s'obtenir en utilise le *caractère d'échappement* \. Ainsi, pour obtenir le caractère `'`, on peut écrire `'\''`, pour obtenir un \, on écrira `'\\'`. Le caractère `'\n'` correspond à un passage à la ligne. On peut les utiliser aussi dans des chaînes de caractères.

4. Booléens

```
not true ;;
true || false ;;
true and false ;; (* interdit *)
true && false ;;
1 = 2 ;;
1 <> 2 ;;
```

5. Jouons avec quelques variables

```
let x = 3 ;;
x ;;
let y = (let x = "pou" in x ^ "let") ;;
y ;; (* c'est ce que l'on vient de définir *)
x ;; (* "pou" n'était qu'une définition locale *)
```

6. Paires et tuples

```
(1, 2) ;;
("a", true) ;;
fst ("a", true) ;;
```

```
let p = (1, 'a') in (snd p, fst p) ;;
snd (1, 2, 3) ;;  (* probleme *)
[ 1, 2 ] ;;       (* attention aux , et aux ; *)
[ 1, 2; 3 ] ;;    (* probleme *)
[ 1, 2; 3, 4 ] ;;
[ 1; 2; 3; 4 ] ;;
```

7. Et pour finir, quelques fonctions

```
let additionne x y = x + y ;; (* comprenez bien les types *)
let additionne2 (x, y) = x + y ;;
let abs x = if x < 0 then (- x) else x ;;
abs -4 ;;                     (* erreur *)
abs (-4) ;;                   (* c'est mieux *)
```

Question 4 Écrire une fonction `delta : bool -> int` telle que `delta true` renvoie `1` et `delta false` renvoie `0`.

On peut passer des fonctions en argument, on fait de la programmation fonctionnelle.

```
let double f x = f (f x) ;;
let double2 f = fun x -> f (f x) ;; (* c'est la même fonction *)
```

Question 5 On veut écrire une fonction `comp` telle que `comp f g` renvoie la fonction composée $f \circ g$. Autrement dit, sous réserve de définition, on veut que `(comp f g) x` renvoie la même chose que `f (g x)`.

1. Quel doit être le type de `comp` ?
2. Écrire cette fonction.

Question 6 On veut écrire une fonction `cond` telle que `cond b f g x` renvoie `f x` si le booléen `b` vaut `true` et renvoie `g x` sinon.

1. Quel est le type de cette fonction ?
2. Écrire cette fonction.

Une fonction récursive, maintenant, instructive à défaut d'être originale.

```
let rec factorielle n =
    if n = 0 then 1 else n * factorielle (n - 1) ;;
```

Question 7 On veut écrire une fonction `somme_images` telle que `somme_images f n` renvoie la somme

$$f(0) + f(1) + \cdots + f(n)$$

1. Quel est le type de cette fonction ?
2. Écrire cette fonction.

Question 8 On veut écrire une fonction `itere_fonction` telle que l'expression « `itere_fonction f n x` » renvoie

$$\underbrace{f \circ f \circ \cdots \circ f}_{n \text{ fois}}(x)$$

1. Quel est le type de cette fonction ?
2. Écrire cette fonction.

Question 9 On veut écrire une fonction `suite_recurrente` telle que l'expression « `suite_recurrente f n x` » renvoie la liste

$$[\,u_0\,,\,u_1\,,\,\cdots\,,\,u_{n-1}\,]$$

avec la suite (u_n) définie par récurrence par

$$u_0 = x \qquad \forall\, n \in \mathbf{N},\, u_{n+1} = f(u_n)$$

1. Quel est le type de cette fonction ?
2. Écrire cette fonction.