TP1: Premiers pas en OCaml

Option Informatique MPSI
Lycée Pierre de Fermat

Machine virtuelle

Rappel : pour lancer la machine virtuelle, lancez VMWare, puis lancez la machine "NonOS 2024". Le mot de passe est **concours**.

Sauvegarder son travail

ATTENTION, vous n'avez pas de session personnelle sur les machines virtuelles au lycée. Si vous y laissez un fichier, aucune garantie que vous le retrouverez à la session suivante. Donc, à la fin de chaque cours/TP, **mettez votre travail sur une clé USB** ou envoyez-le vous par mail. Lorsque vous branchez votre clé USB, une fenêtre s'affiche et vous demande si vous voulez connecter votre clé à la machine virtuelle où à Windows. Connectez-la à la machine virtuelle, et vous la verrez apparaître sur celle-ci après quelques secondes. Si ce n'est pas le cas, essayez un autre port USB. Alternativement, il est aussi possible de copier-coller des fichiers de la machine virtuelle vers Windows et inversement, avec CTRL+C / CTRL+V.

Mise en place

Nous avons vu en cours que l'on peut écrire des lignes de code OCaml dans l'interpréteur utop. En TP, nous allons plutôt écrire dans des fichiers, et importer les fichiers dans utop pour les tester. Là où les fichiers Python ont l'extension `.py`, les fichiers OCaml ont l'extension `.ml`. Il est conseillé d'organiser votre écran de la manière suivante :

1. A gauche, le terminal où vous allez utiliser utop.
2. A droite, l'éditeur de texte que vous allez utiliser pour programmer.

De cette manière, vous pouvez facilement passer de l'édition de code au test. Sur la machine virtuelle, l'éditeur de texte à privilégier est Visual Studio Code. Lorsque vous êtes dans un terminal, vous pouvez :

— Ouvrir un fichier `bla.ml` dans VSCode avec la commande `code bla.ml`, ce qui créera le fichier s'il n'existe pas.

— Lancer utop avec `utop`.

Notons que dans les fichiers sources OCaml, il n'y a pas besoin de terminer let `let` par des `;;`. Il faut néanmoins utiliser les `;;` comme délimiteur lorsque l'on veut écrire des expressions seules :

```
1 let ajouter x y = x + y (* pas de ;; ici car la ligne suivante est un let *)
2 let echanger (x, y) = (y, x);; (* ;; nécessaire car la ligne suivante n'est pas un let *)
3 print_int (ajouter 2 3);; (* ;; nécessaire car la ligne actuelle n'est pas un let *)
4 let a = echanger (3, 5)
```

On peut aussi importer des fichiers `.ml` dans l'interpréteur utop avec la directive `#use`. Par exemple, l'on a écrit les 4 lignes précédentes dans un fichier `bla.ml`, alors dans utop on pourra taper :

```
1 #use "bla.ml" ;;
2 let b = echanger a;;
```

La première ligne va exécuter tout le code de `bla.ml`, et donc rajouter les deux fonctions `ajouter` et `echanger` au contexte global, ainsi que `a`, ce qui fait que la deuxième ligne fonctionne bien.

Pendant le TP vous devez donc implémenter les fonctions dans les fichiers `.ml` que vous créez, puis importer le code dans utop avec `#use` afin de le tester.

Exercice 1. Créez un fichier OCaml, y écrire `let x = 1`, puis importer le fichier dans utop.

Exercice 2.

Écrivez les réponses pour cet exercice dans un fichier "exercice2.ml".

Q1. Écrivez les fonctions suivantes (veillez à bien respecter les types demandés) :

a) Une fonction `double: int -> int` renvoyant le double de son entrée

b) Deux fonctions `first` et `second` prenant en entrée un tuple de type `'a * 'b` et renvoyant respectivement la première et la deuxième composante.

c) Une fonction `somme3: float * float * float -> float` qui ajoute les trois composantes du tuple donné en entrée.

d) Une fonction `est_pair; int -> bool` déterminant si un entier est pair. En OCaml, l'opérateur de modulo se note `mod` (par exemple : `10 mod 3`).

e) Une fonction `divise: int -> int -> bool` qui détermine si sa première entrée est un diviseur de sa deuxième entrée.

Q2. Déterminez ce que les fonctions `fst` et `snd`, qui existent par défaut, sont/font.

Q3. Que représente l'expression `divise 2` ?

Q4. Écrire une fonction `ajouteur: int -> (int -> int)` telle que `ajouteur k` est une fonction ajoutant k à son entrée.

Q5. Écrire une fonction `composee: ('a -> 'b)-> ('c -> 'a)-> ('c -> 'b)` prenant en entrée deux fonctions f et g et renvoyant leur composée $f \circ g$.

Exercice 3.

En OCaml, on peut faire des if-then-else avec la syntaxe suivante :

```
1 if b then e1 else e2
```

où :

— `b` est une expression de type `bool`

— `e1` et `e2` sont des expressions de même type

Un tel if-then-else est une **expression**, dont la valeur est soit la valeur de e_1, soit celle de e_2, dépendant de si b est le booléen true ou false. Par exemple :

```
1 let a = if 3 = 5 then "lapin" else "hibou"
2 (* [résultat] a: string = "hibou" *)
3
4 let valeur_absolue x =
5   if x < 0 then -x else x
```

Q1. Écrivez une fonction `n_roots: (float * float * float)-> int` qui prend en entrée un triplet (a, b, c) et calcule le nombre de racines réelles distinctes du polynôme aX^2+bX+c.

Q2. Écrivez une fonction `nom_chiffre: int -> string` qui prend en entrée un entier n et :

— si n est un chiffre entre 2 et 5 inclus, renvoie son nom en toutes lettres (“trois” pour $n = 3$ par exemple)

— sinon, renvoie la chaîne vide “”.

On veut écrire cette fonction de manière plus concise. En OCaml, il existe une généralisation du if-else appelée le ***match with***, ou ***pattern matching***. Pour la fonction précédente, on peut écrire en OCaml :

```
1 let nom_chiffre n = match n with
2   | 2 -> "deux"
3   | 3 -> "trois"
4   | 4 -> "quatre"
5   | 5 -> "cinq"
6   | _ -> "" (* _ veut dire "Tous les cas" *)
```

Pour évaluer un ***match with***, on évalue l'expression à matcher (ici, n lors de l'appel de la fonction), et on compare avec chaque motif possible : 2, 3, 4, 5, _. Dès que l'on en trouve un qui correspond, on évalue l'expression associée. Par exemple, si l'on appelle `nom_chiffre` avec $n = 4$, on va comparer 4 avec 2, puis avec 3, puis avec 4. On renverra donc “quatre”. Le motif `_` est un attrape-tout : toutes les valeurs lui correspondront.

Cette syntaxe est particulièrement puissante. Voyons un exemple plus poussé :

Q3. Lisez le code suivant et tentez de deviner ce qu'il affiche. La fonction `print_int: int -> unit` sert à afficher un entier, et `print_newline: unit -> unit` affiche un retour à la ligne. Le point virgule simple sert ici à exécuter plusieurs print d'affilées.

```
let f x y = match (x-1, y) with
  | (0, 0) -> 0
  | (0, _) -> y + 1
  | (z, 0) -> z + 100
  | _ -> x * y
;;

print_int (f 3 5); print_newline ();;
print_int (f 1 0); print_newline ();;
print_int (f 1 3); print_newline ();;
print_int (f 6 0); print_newline ();;
```

Q4. Recopiez le code précédent pour vérifier vos suppositions

On veut écrire une fonction qui prend en entrée deux entiers x et y et qui :

— Si x vaut $\pm y$, renvoie 0

— Si $x \in \{y+1, y-1\}$, renvoie $(x+y)^2 + 1$

— Si $x + y \in \{1, -1\}$, renvoie $(x-y)^2 - 1$

— Sinon, renvoie $x * y$

Q1. Compléter le code suivant pour implémenter la fonction décrite :

```
let g x y = match (x-y, x+y) with
  | 0, _ -> 0
  | _, 0 -> (* A COMPLETER *)
  | 1, z -> (* A COMPLETER *)
  | (* A COMPLETER *)
  | (* A COMPLETER *)
  | (* A COMPLETER *)
  | _    -> (* A COMPLETER *)
```

Testez sur quelques exemples pour vérifier.

Lors de l'évaluation d'un `match with`, les différents motifs sont testés dans l'ordre de haut en bas. Essayez d'exploiter ce comportement pour répondre à la question suivante.

Q2. Écrire une fonction prenant en entrée un entier n et renvoyant :

— Si n est multiple de 3, "ga"

— Si n est multiple de 5 mais pas de 3, "bu"

— Si n est multiple d'aucun des deux, n sous forme de string (via la fonction `string_of_int`).

Exercice 4.

Le type `unit` n'a qu'une seule valeur : `()` (aussi prononcée "unit"). Ce type est utilisé pour les fonctions qui "ne renvoient rien mais font quelque chose". Par exemple, les fonctions suivantes prédéfinies en OCaml servent à ***afficher*** des valeurs de différents types :

```
print_int;;
print_float;;
print_string;;
print_newline;;
```

Q1. Vérifiez le type de ces fonctions, et utilisez les pour afficher un entier, un flottant, une chaîne de caractère, et un retour à ligne.

Le type unit possède une syntaxe particulière : le point-virgule ";" permet d'enchaîner plusieurs expressions de type unit, ce qui a pour effet de toutes les "exécuter" dans l'ordre :

```
print_int 5 ; print_string " bonjour " ; print_float 9.8 ; print_newline ()
```

Q2. Tapez l'expression précédente. Quel est son type ?

";" n'est ni une fonction ni un opérateur, mais on peut informellement le voir comme un opérateur binaire sur les `unit`.
On peut donc voir une expression de type unit comme une instruction impérative, et le point-virgule sert à combiner deux instructions en séqunce. [1]

Q3. Créez une fonction `print_triplet: (int*int*int)-> unit` qui prend en entrée trois flottants et les affiche, chacun sur une ligne.

Remarque 1. En Ocaml, lorsque l'on écrit `if a then b else ()`, autrement dit si l'on veut effectuer une commande `b` de type `unit` si une condition `a` booléenne est vérifiée, et ne rien faire sinon, on peut écrire simplement `if a then b`. En revanche ça ne marche pas pour les autres types, ce qui est logique : il serait impossible de donner un type à `1 + (if b then 5);;`

Attention, si l'on veut exécuter plusieurs instructions dans un if-else, il faut les mettre entre parenthèses. Par exemple : dans le code ci-contre, la première fonction va toujours afficher le retour à la ligne, car seul le `print_string "Plus grand"` est considéré comme étant à l'intérieur du if. La deuxième version est correcte

```
let f x =
  if x > 10 then
    print_string "Plus grand";
    print_newline ()

let g x =
  if x > 10 then
    (print_string "Plus grand";
     print_newline ())
```

En pratique, lorsque l'on utilise des parenthèses dans ce cas spécifique, on peut les remplacer par les mots-clés **begin** et **end** : [2]

```
let g x =
  if x > 10 then begin
    print_string "Plus grand";
    print_newline ()
  end
```

1. Il existe même des façons de faire des boucles for et while en OCaml, mais pour l'instant c'est interdit !
2. En réalité, on peut remplacer n'importes quelles parenthèses par begin et end, mais on les utilise généralement pour signaler qu'on fait une suite d'instructions.

Q4. Créez une fonction `print_pair: int -> unit` qui prend en entrée un entier, l'affiche s'il est pair, et ne fait rien sinon.

Q5. Créez une fonction `print_pairs: (int*int*int)-> unit` qui prend en entrée un triplet d'entiers, et affiche uniquement ceux qui sont pairs.

Q6. Créez une fonction `print_ordre: int -> int -> unit` qui prend en entrée deux entiers et les affiche dans l'ordre croissant.

Remarque 2. La dernière fonction existe en OCaml : elle s'appelle `print_endline: string -> unit` !

On peut aussi utiliser le point virgule pour effectuer une commande avant de calculer une valeur :

```
let x = 5 ;;
let y =
  print_int x;
  print_newline ();
  x + 3;;
(* y vaut 8, et l'exécution a affiché 5 suivi d'un retour ligne *)
```

Récursivité

En OCaml, l'outil principal de programmation est la ***récursivité***, c'est à dire le fait qu'une fonction peut s'appeler elle-même.

Prenons la fonction factorielle. On sait que pour $n \in \mathbb{N}$, $n! = \prod_{i=1}^{n} i$, ce qui donne assez facilement une manière de calculer la factorielle à l'aide d'une boucle :

```
def factorielle(n):
  res = 1
  for i in range(1, n+1):
    res = res * i
  return res
```

En OCaml, pour écrire la fonction factorielle, il faudra trouver une relation de récursivité permettant de ***définir*** la factorielle. On remarque :

$$\begin{aligned} \textbf{factorielle}(0) &= 1 \\ \textbf{factorielle}(n) &= n \times \textbf{factorielle}(n-1) \quad \text{pour } n > 0 \end{aligned}$$

Ces formules permettent de ***définir récursivement*** ce qu'est la factorielle d'un entier. En OCaml, on doit utiliser le mot clé `let rec` pour préciser qu'une fonction est récursive. On écrira donc :

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

Remarquons que si $n < 0$, cette fonction va s'appeler à l'infini. En réalité, la fonction factorielle au dessus n'est définie que pour $n \geq 0$. On dit que "$n \geq 0$ est une **précondition** de la fonction. Lorsqu'une fonction possède des préconditions, on les vérifie dans le code lorsque c'est possible. La fonction `failwith` en OCaml permet de renvoyer un message d'erreur et d'arrêter le programme. En l'utilisant, on peut donc forcer l'arrêt lorsque les préconditions ne sont pas vérifiées :

```
(* Renvoie n! = n * (n-1) * ... * 1. n doit être positif ou nul *)
let rec factorielle n =
  if n < 0 then failwith "Factorielle d'un entier négatif"
  else if n = 0 then 1
  else n * factorielle (n-1)
```

Si l'on évalue `factorielle (-3)`, ocaml affichera le message d'erreur :

```
Exception: Failure "Factorielle d'un entier négatif".
```

La programmation en OCaml va donc souvent consister à trouver des définitions récursives pour les objets et les fonctions que l'on manipule.

Exemple 1. On remarque que pour $x \in \mathbb{N}$, on a :

$$\begin{aligned} x \times 0 &= 0 \\ x \times y &= x \times (y-1) + x \quad \text{pour } y > 0 \end{aligned}$$

On peut en déduit la fonction suivante (très lente) qui calcule le produit de deux entiers :

```
let rec produit x y =
  match y with
  | 0 -> 0
  | _ -> produit x (y-1) + x
```

Exercice 5.

Trouver une définition récursive des fonctions suivantes, puis les implémenter en OCaml. Commencez par réfléchir aux cas de base, c'est à dire aux entrées pour lesquelles la fonction peut répondre immédiatement.

Q1. `puiss: float -> int -> float` qui calcule de manière naïve la puissance d'un flottant par un entier.

Q2. `reste: int -> int -> int` qui calcule le reste de la division euclidienne d'un entier a par un autre b, sans utiliser `mod`. On supposera $a \geq 0$ et $b > 0$.

Q3. `pgcd: int -> int -> int` qui calcule le PGCD de deux entiers avec l'algorithme d'Euclide

Q4. `div_eucl: int -> int -> (int * int)` qui calcule le couple (quotient, reste) de la division euclidienne d'un entier a par un autre entier b, sans utiliser les opérateurs `/` et `mod`. On supposera $a \geq 0$ et $b > 0$.

Q5. `decomp: int -> unit` telle que `decomp x` affiche les chiffres de x un par un sur une ligne chacun. Par exemple :

```
>>> decomp 1879
1
8
7
9
```

Q6. `fibo: int -> int -> int -> int` telle que `fibo a b n` renvoie le n-ème terme de la suite de Fibonacci de premiers termes $u_0 = a$ et $u_1 = b$. Essayez d'écrire cette fonction avec **un seul** appel récursif !

Q7. (*Difficile*) `a_racine: (int -> int)-> bool` telle que `a_racine f` renvoie `true` si f admet une racine dans $\mathbb{Z}$, et boucle à l'infini sinon. On pourra passer par une fonction auxiliaire récursive prenant plus de paramètres en entrée.