TP 2 : Premiers pas en OCaml

Comme vu en cours, on peut exécuter du code OCaml de deux façons : en utilisant un *interpréteur* (ou REPL) avec la commande `ocaml` ou `utop`, ou en utilisant un *compilateur* et en exécutant le fichier obtenu avec la commande `ocamlc` ou `ocamlopt`. Pour le début de l'année, on utilisera le premier.
Trois options sont configurées pour programmer en OCaml : les notebook Jupyter que vous avez déjà rencontrés au TP1, VSCodium et Emacs. VSCodium et Emacs permettent d'éditer un fichier source OCaml (dont l'extension est `.ml`), et peuvent intégrer l'interpréteur `ocaml` ou `utop`. Dans ce TP, on présente l'utilisation de VSCodium, mais Emacs dispose d'un mode adapté à OCaml très complet, qui est déjà configuré quand vous ouvrez des fichiers `.ml` / `.mli` et qui est très apprécié : c'est une option parfaitement viable pour programmer efficacement en OCaml.

Exercice 1 – Comment programmer en OCaml

▶ **Question 1** Ouvrir VSCodium/`betterocaml.ml`/votre éditeur de texte, et ouvrir un fichier `nom_fichier.ml`.*Sur* `betterocaml.ml`*, c'est la colonne de gauche.*
Si vous commencez à écrire dans la syntaxe OCaml, VSCodium devrait colorer différemment les mots-clés `let in match ...`, les noms de variable, afficher les parenthèses ouvrantes et fermantes, etc. VSCodium dispose de plusieurs outils pour lancer des commandes / exécuter des actions sur des fichiers, qui sont accessibles par des raccourcis clavier ou par la *palette de commande*, accessible avec Ctrl+Maj+P. Par exemple : la commande `OCaml : Open REPL` ouvre un interpréteur OCaml (par défaut, `utop`, qui est plus agréable à utiliser). La coloration syntaxique est aussi disponible dans Emacs, mais aussi dans les blocs Jupyter ou encore les éditeurs `nano`, `vim`, etc.
Dans le REPL OCaml, pour évaluer une expression, on l'écrit en la terminant par `;;`.

▶ **Question 2** Ouvrir l'interpréteur OCaml : sur `betterocaml.ml`, c'est la colonne de droite, sinon il faut lancer `utop` dans un terminal. Ensuite, écrire des expressions qui effectue les calculs suivants :
- multiplier `25` par `-18`,
- diviser `2.25` par `3.14`,
- mettre `238.4` à la puissance `5.0`

▶ **Question 3** Écrire dans le fichier une fonction `norme : float -> float -> float` telle que `norme a b` est évalué à $\sqrt{a^2+b^2}$. Ensuite, sélectionner la fonction et lancez le raccourci clavier Maj+Entrée. *La fonction racine carrée est définie dans la bibliothèque standard avec* `sqrt : float -> float`.

La fonction est envoyée au REPL (qui est lancé s'il ce n'est pas déjà fait), qui l'évalue. Il est aussi possible de l'utiliser directement :

▶ **Question 4** Évaluer la norme du vecteur $(0,4) \in \mathbb{R}^2$ dans le REPL.
Observez que VSCodium devine le type de la fonction pendant que vous l'écrivez et l'affiche en petit au-dessus de la fonction. Il fait la même chose avec les variables globales. En passant la souris sur une valeur / variable, VSCodium affiche son type, même si l'expression n'a pas encore été évaluée dans le REPL.

▶ **Question 5** Écrire une fonction `div_euclidienne : int -> int -> int * int` où `div_euclidienne a b` renvoie le couple du quotient et du reste de la division euclidienne de a par b.

Remarque

Dans les modules de base d'OCaml, il est habituel de rencontrer deux versions d'une même fonction, l'une étant `<fon> : 'a -> ... -> 'b` qui soulève une exception quand la fonction n'est pas défini (par exemple, le maximum d'une liste vide), et une fonction `<fon>_opt : 'a -> ... -> 'b option` qui renvoie `Some res` quand le résultat est bien défini, et `None` sinon. On peut aussi mêler les deux solutions : par exemple, la fonction `List.nth_opt : 'a list -> int -> 'a` lève une exception quand l'entier fourni est négatif, `Some x` si `x` est le $n$ième élément de la liste et `None` si $n \geq |l|$.

▶ **Question 6** Écrire une fonction puissance entière `puissance : int -> int -> int`, tel que `puissance x n` évalue x^n pour $x \in \mathbb{Z}, n \in \mathbb{N}$.

▶ **Question 7** Écrire la fonction `fib : int -> int` tel que `fib n` est évalué en le $n$ième nombre de la suite de Fibonacci, défini par récurrence pour $n \in \mathbb{N}$ comme :
- $u_0 = u_1 = 1$,
- $u_{n+2} = u_n + u_{n+1}$

▶ **Question 8** Testez la fonction sur des petites valeurs, puis progressivement montez jusqu'à $n = 40$. Que remarque-t-on ? Pourquoi ?

▶ **Question 9** Écrire une fonction `fib_efficace : int -> int` qui calcule efficacement le $n$ième élément de la suite de Fibonacci. *Indication : on pourra utiliser une fonction auxiliaire qui prend comme argument les deux derniers termes de la suite et le nombre restant de termes à calculer.*

▶ **Question 10** Évaluer le 90ième terme de la suite de Fibonacci : que remarque-t-on ? Pourquoi ?

▶ **Question 11** Écrire une fonction `suite_arith_geometrique : float -> float -> int -> float` tel que pour $n \in \mathbb{N}$, `suite_arith_geometrique a b n` renvoie le $n$ième terme de la suite définie par récurrence :

$$u_0 = 0 \text{ et } \forall n \in \mathbb{N}, u_{n+1} = au_n + b$$

▶ **Question 12** Que se passe-t-il quand vous appelez cette fonction avec un entier `n` strictement négatif ?

Remarque 1

Pour définir une fonction partielle (ici : non définie sur les entiers strictement négatifs), on peut utiliser `failwith : string -> 'a` qui lève une exception quand la fonction n'est pas définie (ici, quand $n < 0$). Dans ce cas précis, on utilisera plutôt la fonction `assert : bool -> unit` qui évalue son argument et renvoie `()` s'il est vrai, et lève une exception s'il est faux.

Exercice 2 – Fonctions sur les listes

▶ **Question 1** Écrire une fonction `double_premier : int list -> int list` qui renvoie la liste dont le premier élément est doublé (et la liste vide si l'entrée est vide). Même chose pour `double_dernier : int list -> int list` avec le dernier élément.

▶ **Question 2** Écrire une fonction `somme_liste : int list -> int` qui calcule la somme des éléments d'une liste d'entiers.

▶ **Question 3** Écrire une fonction `longueur : 'a list -> 'a` qui calcule la longueur d'une liste quelconque. *En OCaml, une fonction est prédéfinie pour cela :* `List.length`.

▶ **Question 4** Écrire une fonction `moyenne_liste : float list -> float` qui calcule la moyenne d'une liste de flottant. En écrire une ne nécessitant qu'un seul parcours de la liste.

▶ **Question 5** En OCaml, l'opérateur `( @ ) : 'a list -> 'a list -> 'a list` est l'opérateur de concaténation entre listes. En proposer une implémentation dans une fonction `concatener : 'a list -> 'a list -> 'a list` *évidemment sans utiliser* `@`.

▶ **Question 6** Quelle est le nombre d'appels récursifs nécessaires pour évaluer `concatener l1 l2` ?

▶ **Question 7** En déduire une fonction `miroir : 'a list -> 'a list` telle que `miroir l` renvoie la liste `l` "retournée". Par exemple, `miroir [1;2;3;5]` est évaluée à `[5;3;2;1]`.

▶ **Question 8** Pour une liste de taille n, combien de fois est appliquée la fonction `concatener` ? *Indication : on peut commencer par compter le nombre de fois qu'on appelle la fonction sans compter les appels récursifs.*

▶ **Question 9** Sans utiliser de concaténation (ni `concatener` ni `@`), écrire une fonction `miroir_efficace : 'a list -> 'a list` qui calcule le miroir d'une liste. *Consigne : ne parcourir la liste qu'une seule fois.*