

TP n°0

Rappels Ocaml

Pour lancer l'interpréteur OCaml sous **emacs** :

- ouvrez un nouveau fichier **tp0.ml** (l'extension **.ml** est nécessaire),
- dans le menu **Tuareg**,
dans le sous-menu **Interactive Mode**,
choisissez l'entrée **Run Caml Toplevel**
- confirmez le lancement de **ocaml** par un retour-chariot.

Chaque expression entrée dans la fenêtre de **tp1.ml** peut être évaluée en se plaçant sur un caractère quelconque de l'expression (avant “;”) , puis : ou bien par **Evaluate phrase** dans le sous-menu **Interactive Mode** du menu **Tuareg** d'emacs; ou bien par **ctrl-x**, **ctrl-e**.

1 Liaison, Fonctions d'ordre supérieur, Filtrage

Exercice 1 [Rappels sur le mécanisme de liaison]

Prévoir le résultat fourni par l'interpréteur OCaml après chacune des commandes suivantes :

```
#let x = 2;;  
#let x = 3  
  in let y = x + 1  
     in x + y;;  
#let x = 3 and y = x + 1  
  in x + y;;
```

Pourquoi la deuxième et la troisième commande ne fournissent-elles pas le même résultat ?
Expliquer à présent le comportement suivant :

```
#let x = 3;;  
x : int = 3  
#let f y = y + x;;  
f : int -> int = <fun>  
#f 2;;  
- : int = 5  
#let x = 0;;  
x : int = 0  
#f 2;;  
- : int = 5
```

Exercice 2 [Placement des parenthèses] Ajouter les parenthèses nécessaires pour que le code ci-dessous compile :

```
let somme x y = x + y;;
somme
  somme somme 2 3 4
  somme 2 somme 3 4
;;
```

Exercice 3 [Fonctions sur les listes] Retrouver le code des fonctions suivantes :

- **append** : concatène deux listes.
Exemple : `append [1;2] [3;4] = [1;2;3;4]`.
- **flatten** : aplanir d'un niveau une liste de listes.
Exemple : `flatten [[2];[];[3;4;5]] = [2;3;4;5]`.
- **rev** : inverse l'ordre des éléments d'une liste (une version naïve suffira).
Exemple : `rev [1;2;3] = [3;2;1]`.

Exercice 4 [Utilisation de la bibliothèque List]

- Écrivez une fonction `somme_liste : int list -> int`
- Ré-implémentez la fonction `fold_left`, puis `somme_liste` en utilisant `fold_left`.
- Écrivez une fonction `scal l l'` calculant le produit scalaire des vecteurs `l` et `l'` représentés sous forme de liste. Servez vous des fonctions `fold_left` et `map2`.
- Même question en utilisant uniquement `fold_left2`.

Exercice 5 Écrivez un encodage possible, avec seulement des conditionnelles, de la fonction suivante :

```
let f x y z = match x, y, z with
| _ , false , true -> 1
| false , true , _ -> 2
| _ , _ , false -> 3
| _ , _ , true -> 4 ;;
```

Exercice 6 Que renvoie la fonction `est_ce_moi` ?

```
type contact =
| Tel of int
| Email of string;;

let mon_tel = 0123456789;;
let mon_email = "gabriel.scherer@gmail.com";;

let est_ce_moi = function
| Tel mon_tel -> true
| Email mon_email -> true
| _ -> false
```

2 Manipulation des outils standards

Exercice 7 [Trace et ses limitations]

- Quelle est la complexité de la version naïve de `rev` ?
- Tracez les appels de `rev` grâce à l’outil `trace` sur `rev` avec `#trace rev`. Essayez avec `rev [1;2;3;4]`.

Malheureusement cet outil ne permet pas de donner les valeurs lorsque celles-ci sont des instanciations de valeurs polymorphes (d’où l’indication `<poly>`).

- Restreignez la fonction `rev` au type `int list -> int list` et tracer son appel sur `rev [1;2;3;4]`. Remarquez que l’on descend dans la structure puis que l’on remonte.

Exercice 8 [Compilation séparée]

- Ecrivez trois fichiers `plus.ml`, `fois.ml`, `exp.ml` contenant respectivement la définition de la fonction `plus`, `fois`, et exponentielle. La fonction `fois` devra utiliser la fonction `plus`, et la fonction `exp` devra utiliser la fonction `fois`.
- Ecrivez un fichier `main.ml` qui affiche (`print_int`) le résultat de 2^4 .
- On veut maintenant compiler le programme en code natif. Pour cela on utilisera deux commandes :
 - `ocamlc -c toto.ml` qui produit le fichier object `toto.cmo` à partir du fichier source. Si le fichier d’interface `toto.mli` n’est pas présent, il produit aussi le fichier d’interface compilée `toto.cmi`.
 - `ocamlc toto.cmo tata.cmo titi.cmo -o monprog` qui produit l’exécutable `monprog` en liant ensemble des fichiers objects. On peut ensuite vérifier que `./monprog` exécute bien le programme.
- La plupart du temps on pourra simplement utiliser la commande `ocamlbuild` qui permet de compiler n’importe quel fichier en code octet ou natif en s’occupant des dépendances et de l’ordre d’exécution.

Utilisez le pour générer le code octet du module `main`. Il suffit de lui demander de produire `main.byte` (pour obtenir un programme bytecode) ou `main.native` (programme natif).

3 Questions difficiles

Exercice 9 [Inventer les paires] En utilisant seulement les constructions `let` et `fun` du langage (en particulier, pas le droit aux paires `(a,b)...`), définir trois fonctions `pair`, `first` et `second` telles que pour toutes valeurs `a` et `b`, `first (pair a b)` soit égal à `a` et `second (pair a b)` à `b`.

Exercice 10 [Récursion terminale] Écrire une implémentation tail-réursive de la fonction `leaves` suivante :

```
type 'a tree =  
| Node of 'a tree * 'a tree  
| Leaf of 'a  
  
let rec leaves = function  
| Leaf v -> [v]  
| Node (a, b) -> leaves a @ leaves b
```