Université Paris Diderot
M1 Informatique

Programmation Fonctionnelle Avancée
Année 2015-2016

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`, choisir `Run OCaml Toplevel`
- confirmez le lancement de `ocaml` par un retour-chariot.

Chaque expression entrée dans la fenêtre de `tp0.ml` peut être évaluée en se plaçant sur un caractère quelconque de l'expression (avant "`;;`") , puis en utilisant le raccourci `ctrl-x, ctrl-e`, qui correspond à `Evaluate phrase` dans le sous-menu `Interactive Mode` du menu `Tuareg` d'emacs.

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 ?
Considérons maintenant les phrases suivantes :

```
let x = 3;;

let f y = y + x;;

f 2;;

let x = 0;;

f 2;;
```

Quels sont les résultats des appels de fonctions successifs `f 2` ?

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 listes. 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 ;;
```

En générant tous les triplets de booléens, vérifiez que votre encodage se comporte bien comme la fonction d'origine.

Exercice 6 Que renvoie la fonction `est_ce_moi` ?

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

let mon_tel = 0123456789;;
let mon_email = "moi.je@ici.fr";;

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

Comment peut-on la rendre plus intéressante ?

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 `#trace rev`. Essayez avec `rev [1;2;3;4]`.
- Malheureusement l'outil `trace` ne permet pas de donner les valeurs lorsque celles-ci sont des instanciations de valeurs polymorphes (d'où l'indication `<poly>`). Restreignez maintenant 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]

1. **Préparation:**
   Écrivez trois fichiers `plus.ml`, `fois.ml`, `exp.ml` contenant respectivement la définition des fonctions `plus`, `fois`, et `exponentielle` sur les entiers naturels. La fonction `fois` devra utiliser la fonction `plus`, et la fonction `exponentielle` devra utiliser la fonction `fois`. Écrivez également un fichier `main.ml` qui affiche (via `print_int`) le résultat de 2^4.
2. **Compilation manuelle:**
   Compiler votre programme en bytecode (ou code-octet), en utilisant directement les deux commandes suivantes:
   - `ocamlc -c a.ml` qui produit le fichier objet `a.cmo` à partir du fichier source. Si le fichier d'interface `a.mli` n'est pas présent, il produit aussi le fichier d'interface compilée `a.cmi`.
   - `ocamlc a.cmo b.cmo c.cmo -o monprog` qui produit l'exécutable `monprog` en liant ensemble des fichiers objets. On peut ensuite vérifier que `./monprog` exécute bien le programme.
3. Compilez maintenant votre programme en version native, en utilisant `ocamlopt` au lieu de `ocamlc`. Quel est alors le nom des fichiers objets générés ?
4. **Compilation automatisée:**
   La plupart du temps on pourra simplement utiliser la commande `ocamlbuild` qui permet d'automatiser la compilation de n'importe quel fichier en code-octet ou natif, en s'occupant des dépendances et de l'ordre de compilation. Utilisez-le pour générer le code-octet du module `main`, via `ocamlbuild main.byte`, ou bien le programme natif via `ocamlbuild main.native`. Attention à supprimer d'abord tout fichier objet qui existerait au même endroit: `rm *.cm*`.

3 Questions avancées

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`. Idéalement, `a` et `b` devront pouvoir avoir des types différents.

Exercice 10 [Récursion terminale] Écrire une implémentation tail-récursive 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;;
```