Université Paris Diderot — Programmation Fonctionnelle Avancée
M1 Informatique — Année 2012-2013

TP n°0 - Correction

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;;
```

Correction :

```
# let x = 2;;
val x : int = 2
# let x = 3
  in let y = x + 1
     in x + y;;
- : int = 7
# let x = 3 and y = x + 1
  in x + y;;
- : int = 6
```

Quand on écrit `let x = toto and y = tata in ...` avec un `and`, les valeurs `x` et `y` sont définies *simultanément*, au lieu de définir d'abord `x` puis `y` ; la nouvelle définition de `x` n'est donc pas encore disponible quand `y` est définie, et la définition de `y` voit donc l'ancienne définition si elle existe.

Ici on définit `x = 3 and y = x + 1` dans un environnement où `x` vaut `2`. `y` vaut donc `2 + 1`, et non `3 + 1` comme dans l'exemple précédent.

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
```

Correction : La définition de f utilise la première définition de x. Redéfinir x dans la suite n'a aucun effet sur la définition de f qui est déjà fixée.

Il faut bien comprendre que les définitions de variables en OCaml ne "modifient" pas une variable, il s'agit uniquement de donner un nom à une valeur (ce qui peut cacher une définition précédente de ce nom). C'est comme en mathématiques où on peut dire "soit x le plus petit réel tel que...", et longtemps après de la fonction qui "à x associe $3x+2$". On ne modifie pas x en l'utilisant pour deux usages différents, on se contente d'oublier l'ancienne définition.

Le concept de mémoire modifiable existe en OCaml (les références), mais il est complètement séparé la notion de variable – contrairement à d'autres langages comme Java qui mélangent les deux aspects. Cela peut sembler plus lourd pour les débutants, mais ça clarifie les concepts et permet de raisonner plus facilement sur le code que l'on écrit.

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
;;
```

Correction : 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]`.
  Correction :
  ```
  let rec append l1 l2 = match l1 with
  | [] -> l2
  | h::t -> h :: append t l2;;
  ```
  OCaml propose déjà l'opérateur infixe `(@)` pour concaténer des listes : `[1;2] @ [3;4]`. Attention, `List.concat` existe mais ne fait pas cela, c'est un synonyme de la fonction `flatten`.
- `flatten` : aplanir d'un niveau une liste de listes.
  Exemple : `flatten [[2];[];[3;4;5]] = [2;3;4;5]`.
  Correction :
  ```
  let rec flatten li = match li with
  | [] -> []
  | h::t -> append h (flatten t);;
  ```
- `rev` : inverse l'ordre des elements d'une liste (une version naïve suffira).
  Exemple : `rev [1;2;3] = [3;2;1]`.

Correction :

```
let rec rev l = match l with
| [] -> []
| h::l -> append (rev l) [h];;
```

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`.
  Correction :
  ```
  let rec somme_liste = function
  | [] -> 0
  | t::q -> t + somme_liste q;;
  let rec fold_left f acc = function
  | [] -> acc
  | t::q -> fold_left f (f acc t) q;;
  let somme_liste li = fold_left (+) 0 li;;
  ```
  La syntaxe `(+)` représente l'opérateur infixe d'addition. Elle est équivalente à `(fun a b -> a + b)`.
  ```
  let scal v v' = List.fold_left (+) 0 (List.map2 ( * ) v v');;
  ```

Attention, comme `(*` et `*)` sont les délimiteurs de commentaires en OCaml, il ne faut pas écrire `(*)` pour l'opérateur infixe de multiplication, car OCaml croit à un début de commentaires. On met des espaces `( * )` pour éviter la confusion.

– Même question en utilisant uniquement `fold_left2`.
Correction :
```
let scal v v' = List.fold_left2 (fun s x x' -> s + (x * x')) 0 v v';;
```

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 ;;
```

Correction :

```
let f x y z =
  if not z then
    (if not x && y then 2 else 3)
  else if not y then 1
  else if not x then 2
  else 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
```

Correction : Cette fonction renvoie toujours `true`, et d'ailleurs OCaml prévient que le troisième cas ne se produit jamais :

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

        Characters 76-77:
  | _ -> false
    ^
Warning 11: this match case is unused.
val est_ce_moi : contact -> bool = <fun>
```

Quand une variable apparaît dans un motif, par exemple `mon_tel` dans `match Tel 123 with Tel mon_tel -> ...`, elle est toujours *liée*, c'est-à-dire qu'elle devient le nom de la partie correspondante de la valeur sur laquelle on effectue le filtrage, ici `123`. Cela se produit même si la variable est déjà définie dans le programme ; il n'y a jamais de test d'égalité avec la valeur de la définition courante – ce à quoi on pourrait s'attendre en lisant ce code.

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 descent 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 bytecode. 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.

  Correction : Quand on écrit du code dans un fichier `toto.ml`, cela crée implicitement un module `Toto` utilisable depuis d'autres fichiers. Si je veux appeler la fonction `f` définie dans `toto.ml` depuis un autre fichier `.ml`, c'est `Toto.f` – tout comme on utilise la fonction `List.rev` de la bibliothèque standard, définie dans les fichiers `list.ml` et `list.mli`
  Par exemple, un code possible pour `main.ml` est le suivant :

```
let () =
  print_int (Exp.exponentielle 2 4);
  print_newline ();;
```

  Il appelle la fonction *exponentielle* du module *Exp* défini par le fichier `exp.ml`.
  La suite de commande à utiliser est la suivante :

```
% ocamlc -c plus.ml
% ocamlc -c fois.ml
% ocamlc -c exp.ml
% ocamlc -c main.ml
% ocamlc plus.cmo fois.cmo exp.cmo main.cmo -o prog
% ./prog # pour exécuter le programme résultant
16
```

  Il faut faire attention à compiler les fichiers dans l'ordre des dépendances (`plus.ml` avant `fois.ml` qui utilise `Plus.plus`, et ensuite à lier les `.cmo` dans le bon ordre. Sinon on obtient des erreurs de

ce genre (après avoir supprimé tous les `.cmo` et `.cmi` existants) :

```
% rm *.cm*
% ocamlc -c fois.ml
File "fois.ml", line 3, characters 7-16:
Error: Unbound module Plus
```

Le fichier `plus.cmi` n'est pas disponible, donc le compilateur n'arrive pas à trouver le module `Plus` quand il travaille sur `fois.ml`. Il faut compiler `plus.ml` (ou `plus.mli`, s'il existe) avant. Ensuite on peut encore se tromper sur l'ordre des `.cmo` au moment de la production de l'exécutable final :

```
% ocamlc -c plus.ml
% ocamlc -c fois.ml
% ocamlc -c exp.ml
% ocamlc -c main.ml
% ocamlc fois.cmo plus.cmo exp.cmo main.cmo -o monprog
File "_none_", line 1:
Error: Error while linking fois.cmo:
Reference to undefined global `Plus'
```

– 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 dépendences et de l'ordre d'éxécution.

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).

```
% rm *.cm* # on nettoie les fichiers compilés existants
% ocamlbuild main.byte
```

3 Questions très 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`.
Correction : Certain-e-s élèves ont proposé des variantes (en général plus compliquées) de la solution suivante :

```
let pair a b = function
  | true -> a
  | false -> b;;

let first p = p true;;
let second p = p false;;
```

Le problème est que les paramètres `a` et `b` sont forcés d'être de même type, car ils sont tous les deux renvoyés par un cas du filtrage. On peut écrire `(1, "toto")` en OCaml, mais avec cette solution ça ne type pas :

```
# pair 1 "toto";;
Characters 7-13:
  pair 1 "toto";;
         ^^^^^^

Error: This expression has type string but an expression was expected of type
         int
```

C'est déjà pas mal, mais ce n'est pas une solution complète. Voici une vraie solution (je vous laisse trouver `first` et `second`...) :

```
let pair a b = fun f -> f a b;;
```

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;;
```

Correction : On commence par ré-écrire la fonction en utilisant un accumulateur. Ce n'est toujours pas tail-récursif, mais on s'en rapproche (un des appels est un appel terminal).

```
let leaves tree =
  let rec leaves acc = function
  | Leaf v -> v::acc
  | Node (a, b) -> leaves (leaves acc b) a
  in leaves [] tree;;
```

On peut maintenant utiliser une pile explicite de sommets à visiter (c'est une technique assez classique pour rendre les parcours d'arbre tail-récursifs dans des langages moins élégants que OCaml).

```
let leaves tree =
  let rec leaves acc to_visit = function
  | Node (a, b) -> leaves acc (a :: to_visit) b
  | Leaf v ->
      let acc = v::acc in
      match to_visit with
      | [] -> acc
      | tree::to_visit -> leaves acc to_visit tree
  in leaves [] [] tree;;
```

Notez qu'on peut en fait ré-écrire cette fonction pour travailler directement sur une liste d'arbre à visiter (une forêt) en entrée, au lieu de séparer l'arbre en entrée de la liste `to_visit`.

```
let leaves tree =
  let rec leaves acc = function
  | [] -> acc
  | Leaf v :: rest -> leaves (v::acc) rest
  | Node (a, b) :: rest -> leaves acc (b::a::rest)
  in leaves [] [tree];;
```

Dans un langage fonctionnel il est possible d'utiliser une technique plus jolie (car on peut passer automatiquement de la version initiale à la version tail-récursive, sans changer d'algorithme ou réfléchir trop), en utilisant des *continuations*. Vous allez en entendre parler plus tard dans le cours, et donc pouvoir bien comprendre le code ci-dessous.

```
(* on définit un opérateur infixe pour l'application de fonction;
   c'est juste cosmétique, on pourrait utiliser des parenthèses
   à la place. Il existe par défaut à partir de OCaml version 4.01. *)
let (@@) f x = f x;;

let leaves tree =
  let rec leaves tree return = match tree with
  | Leaf v -> return [v]
  | Node (a, b) ->
      leaves a @@ fun la ->
      leaves b @@ fun lb ->
      return (la @ lb)
  in leaves tree (fun res -> res);;
```