ENSIBS – 3ème année, spécialité Informatique et Cybersécurité (parcours Cybersécurité du logiciel, FISE) – Programmation fonctionnelle

TD/TP 4b : programmation fonctionnelle, OCaml

TRÈS IMPORTANT : vérification des programmes

Testez systématiquement tous les exemples OCaml qui ont été donnés. Testez, avec encore plus de soin, les petits programmes OCaml que vous avez vous-même écrit. Vous devez considérer comme faux un programme que vous n'avez pas testé.

IMPORTANT :

- Le travail lors de ce TD/TP est individuel : la communication avec vos collègues, les échanges de fichiers, le copier-coller d'un travail qui n'est pas le vôtre est interdit. Si une triche est détectée lors du TP ou de la correction, une note sanction sera attribuée à tous les élèves participant à cette triche (y compris les élèves ayant réalisé le travail à partir duquel la copie a été faite).

PRÉPARATION :

- Télécharger l'archive « `TPToursHanoi.zip` ». Désarchiver ce fichier. Cette archive contient trois fichiers qui ne doivent pas être renommés :
  - Un fichier « `Makefile` » qui est utilisé pour compiler votre code et générer le fichier exécutable nommé « `test` ». Ce fichier peut être consulté, mais pas modifié !
    - Pour compiler votre code et générer le fichier exécutable vous devez utiliser la commande : « `make` ».
      - Attention, pour cela vous devez utiliser le terminal et vous devez vous placer dans le répertoire « `TPToursHanoi` » qui contient les trois fichiers téléchargés.
    - Pour exécuter votre code vous devez exécuter la commande « `./test` ».
    - Pour effacer les fichiers générés automatiquement, vous devez exécuter la commande « `make clean` ».
    - Si vous n'arrivez pas à exécuter la commande « `make` » dans votre terminal (par exemple, vous utilisez l'environnement Windows), vous devez exécuter la suite de commandes :
      - `ocamlc -c TPToursHanoi.ml TestToursHanoi.ml`
      - `ocamlc -o test.exe TPToursHanoi.cmo TestToursHanoi.cmo`
      - `./test.exe`

      à chaque fois que vous souhaitez compiler et exécutez votre programme.
  - Un fichier « `TestToursHanoi.ml` » que vous utiliserez pour tester les fonctions/prédicats que vous allez développer tout au long de votre TP. Le code

de ce fichier ne doit pas être modifié, vous pouvez uniquement enlever des commentaires au fur et à mesure de votre avancement. Par exemple, vous avez développé une fonction « `check` » et vous souhaitez la tester. Pour cela, il faudra :

- enlever dans le fichier « `TestToursHanoi.ml` » les commentaires pour la partie des tests nommée « `TEST : CHECK` » ;
- compiler les fichiers contenant le code
  - soit à l'aide de la commande « `make` » (Linux, Unix, Mac),
  - soit à l'aide de la séquence des commandes (Windows) :
    - `ocamlc -c TPToursHanoi.ml TestToursHanoi.ml`
    - `ocamlc -o test.exe TPToursHanoi.cmo TestToursHanoi.cmo`
- exécuter le fichier généré
  - soit à l'aide de la commande « `./test` » (Linux, Unix, Mac),
  - soit à l'aide de la commande « `./test.exe` » (Windows).

- Un fichier « `TPToursHanoi.ml` » que vous devrez compléter en répondant aux questions des exercices ci-dessous.
  - Attention, n'oubliez pas de supprimer les commentaires qui entourent une fonction ou un prédicat que vous avez développé pour pouvoir le/la tester.
  - Pour répondre à certaines questions vous serez probablement amenés à développer des fonctions supplémentaires, n'hésitez pas à les créer.

FINALISATION AVANT LE DEPÔT :

- Le code réalisé lors du « TD/TP 4b » doit être compilable (ne doit pas contenir d'erreurs de compilation). Si des erreurs de compilation sont présentes, le travail ne sera pas corrigé et la note 0 sera attribuée à ce travail.
  - À noter que le fichier « `TPToursHanoi.ml` » peut contenir un code inachevé, ce code doit être commenté pour ne pas gêner la compilation. Ce code ne sera pas noté.
- Avant de rendre le travail réalisé, vous devez :
  - nettoyer le répertoire, c'est-à-dire supprimer tous les fichiers générés automatiquement
    - soit à l'aide de la commande « `make clean` » (Linux, Unix, Mac),
    - soit à l'aide de la commande « `rm -rf test *.cmi *.cmo *~` » (Windows).
  - archiver le répertoire « `TPToursHanoi` » contenant trois fichiers.
- L'archive doit être déposée dans le dépôt prévu à cet effet.

Partie 1 : Tours de Hanoï (enregistrements et listes)

Le jeu « Tours de Hanoï » est un jeu de réflexion consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes :

- Nous ne pouvons déplacer plus d'un disque à la fois.
- Nous ne pouvons placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.
- Nous supposons que cette dernière règle est également respectée dans la configuration de départ.

Dans cet exercice, nous nous proposons de calculer une séquence de coups optimale permettant de déplacer les disques de la tour de départ à la tour d'arrivée. Chaque tour est représentée par une liste d'entiers, la valeur d'un entier représentant la taille d'un disque.

Exercice 1.1 (préliminaires)

1. Définir un type enregistrement `hanoi` à trois champs `t1`, `t2` et `t3`, tous trois de type `int list`, représentant les trois tours du jeu.

2. Créer les données suivantes :
   - `tour1` : une liste d'entiers contenant dans l'ordre les valeurs `1` et `4` ;
   - `tour2` : une liste d'entiers contenant la valeur `2` ;
   - `tour3` : une liste d'entiers contenant dans l'ordre les valeurs `3`, `5` et `6` ;
   - `tours3bis` : une liste d'entiers contenant dans l'ordre les valeurs `5`, `3` et `6` ;
   - `jeu1` : un jeu qui contient les trois tours Hanoi : `tour1`, `tour2` et `tour3`. Pour cela vous devez utiliser le type `hanoi` défini dans la question 1 de cet exercice.
   - `jeu2` : un jeu qui contient les trois tours Hanoi : `tour1`, `tour2` et `tour3bis`. Pour cela vous devez utiliser le type `hanoi` défini dans la question 1 de cet exercice.
   - `jeu3` : un jeu qui contient les trois tours Hanoi : `[1;2;3;4]`, `[]` et `[]`. Pour cela vous devez utiliser le type `hanoi` défini dans la question 1 de cet exercice.

3. Écrire une fonction récursive `check : 'a list -> bool`, qui renvoie un booléen indiquant si la liste passée en paramètre représente une tour valide. Les entiers de la liste représentant la taille des disques, il faut s'assurer que pour deux entiers consécutifs de la liste, le premier est strictement plus petit que le second.

4. Dans cette question vous allez apprendre et utiliser la fonction `assert : bool -> unit` de la bibliothèque standard. Elle permet de s'assurer de la validité d'une assertion ou d'un prédicat. Supposons qu'une fonction doit augmenter de 2 la valeur de son paramètre si sa valeur est un nombre pair, dans le cas contraire elle doit générer une exception. Nous pouvons faire cela en utilisant la fonction `assert` comme suit :

```
let pair x = x mod 2 = 0;;
let test x = assert (pair x); x+2;;
```

Résultat d'exécution de la fonction `test` :

```
# test 2;;
- : int = 4
# test 1;;
```

```
Exception: Assert_failure ("//toplevel//", 1, 13).
```

En utilisant la fonction `check` précédemment définie et la fonction `assert`, écrire une fonction :
`make : int list -> int list -> int list -> hanoi`
telle que `make p1 p2 p3` vérifie que les paramètres `p1`, `p2` et `p3` représentent bien des tours valides, et renvoie le jeu composé de ces tours.

5. Écrire une fonction récursive `make_list : int -> int list`, qui à partir de la valeur de son paramètre `n` permet de générer une liste de valeurs entières comme suit :
   - si la valeur du paramètre est inférieure ou égale à 0, elle génère une liste vide ;
   - sinon elle génère la liste contenant toutes les valeurs entières de l'intervalle `[1;n]` dans l'ordre. Par exemple, `make_list 5` génère la liste `[1; 2; 3; 4; 5]`.

Exercice 1.2 (affichage)

Pour répondre à cette question nous allons utiliser le module « `Option` » du langage OCaml (voir https://v2.ocaml.org/api/Option.html). Ce module permet d'indiquer explicitement l'absence ou la présence d'une valeur en utilisant deux options : `(None)` et `(Some valeur)`. Nous utiliserons ce module pour réaliser l'affichage d'un jeu « Tours de Hanoï ». Par exemple, pour réaliser l'affichage suivant :

```
          3
   1      5
   4   2  6
============
```

nous devons transformer le jeu « Tours de Hanoï » représenté par trois listes `[1; 4]`, `[2]`, `[3; 5; 6]` en une liste de triplets `[(None, None, Some 3); (Some 1, None, Some 5), (Some 4, Some 2, Some 6)]` où chaque triplet décrit les disques se trouvant à chaque niveau (de haut en bas). Par exemple, le premier triplet `(None, None, Some 3)` indique que la première et la deuxième tour n'ont pas de disque au niveau 3 (valeur `None`) par contre la troisième tour a un disque de taille 3 au niveau 3 (valeur `Some 3`).

1. Réaliser une fonction :
```
combine : 'a list -> 'b list -> 'c list ->
          ('a option * 'b option * 'c option) list
```
   telle que :
   - `combine [] [] [] = []`
   - `combine [1] [2] [3] = [(Some 1, Some 2, Some 3)]`
   - `combine [1; 4] [2] [3; 5; 6] = [(None, None, Some 3); (Some 1, None, Some 5); (Some 4, Some 2, Some 6)]`

   Pour cela vous devez :
   - d'abord réaliser une fonction auxiliaire récursive
```
combine_aux : 'a list -> 'b list -> 'c list ->
              ('a option * 'b option * 'c option) list
              ('a option * 'b option * 'c option) list
```
     qui construira le résultat en utilisant un 4ème paramètre-accumulateur telle que :
     - `combine_aux [] [] [] [] = []`
     - `combine_aux [1] [2] [3] [] = [(Some 1, Some 2, Some 3)]`

- `combine_aux [1; 4] [2] [3; 5; 6] [] = [(None, None, Some 6); (Some 4, None, Some 5); (Some 1, Some 2, Some 3)]`

Pour la réalisation de la fonction `combine_aux` vous devez utiliser la fonction `hd_tl_opt` expliquée ci-dessous.

- ensuite, réaliser la fonction `combine` en vous aidant des fonctions `combine_aux` et `List.rev` (expliquée ci-dessous). En effet, il suffit de remarquer que si vous appelez la fonction `combine_aux` avec les trois premiers paramètres qui sont des listes dans lesquelles l'ordre des éléments a été inversé, vous obtenez directement le résultat souhaité pour la fonction `combine`.

Deux fonctions utiles pour la réalisation de l'exercice :

- la fonction `hd_tl_opt : 'a list -> 'a option * 'a list` qui à partir d'une liste donnée `lst` construit une paire de valeurs `(x,l)` où
  - `x` est le premier élément de la liste `lst` transformé en une option : soit `(None)` si la liste est vide soit en `(Some valeur_du_premier_élèment)` et
  - `l` est la liste `lst` privée de son premier élément.

```
let hd_tl_opt lst =
   match lst with
   | []     -> (None, [])
   | x::l   -> (Some x, l)
```

Exemples d'application de la fonction `hd_tl_opt` :

```
# hd_tl_opt [1;2;3];;
- : int option * int list = (Some 1, [2; 3])
# hd_tl_opt [];;
- : 'a option * 'a list = (None, [])
```

- la fonction `List.rev : 'a list -> 'a list` de la bibliothèque « `List` » qui permet d'inverser l'ordre des éléments de la liste donnée en entrée.

Exemple d'application de la fonction `List.rev` :

```
# List.rev [1;2;3];;
- : int list = [3; 2; 1]
```

2. Pour répondre à cette question vous devez utiliser la fonction `print_option : int option -> unit` donnée ci-dessous :

```
let print_option opt =
   match opt with
   | None   -> Printf.printf "   "
   | Some i -> Printf.printf "%3i" i
```

qui permet d'afficher une option.

Exemples d'application de la fonction `print_option` :

```
# print_option None;;
   - : unit = ()
# print_option (Some 5);;
  5- : unit = ()
```

En utilisant la fonction récursive `print_option`, écrire une fonction
`print_options_list : (int option * int option * int option) list -> unit`
telle que

```
print_options_list [(None,   None,   Some 3);
                    (Some 1, None,   Some 5);
                    (Some 4, Some 2, Some 6)]
```

affiche :

```
          3
   1      5
   4   2  6
```

3. En utilisant les fonctions `combine` et `print_options_list` précédemment définies, écrire une fonction :

```
print_hanoi : hanoi -> unit
```

telle que

```
print_hanoi { t1 = [1; 4];
              t2 = [2];
              t3 = [3; 5; 6] }
```

affiche :

```
          3
   1      5
   4   2  6
 =============
```

4. En utilisant la fonction `print_hanoi` précédemment définie, écrire une fonction récursive :
   `print_hanoi_list : hanoi list -> unit`
   telle que
   ```
   print_hanoi_list [      {t1 = [1;2;3;4]; t2 = []; t3 = []};
                           {t1 = [2;3;4]; t2 = [1]; t3 = []}   ]
   ```
   affiche :
   ```
      1
      2
      3
      4
   ============

      2
      3
      4    1
   ============
   ```

Exercice 1.3 (résolution)

1. En utilisant la fonction `move : 'a list -> 'a list -> 'a list * 'a list` donnée ci-dessous :
   ```
   let move tour1 tour2 =
      match tour1 with
      | []    -> (tour1,tour2)
      | x::t1 -> (t1,   x::tour2)
   ```
   qui permet de déplacer un disque de la tour `tour1` vers la tour `tour2`, et la fonction `make` précédemment définie, écrire une fonction :

   ```
   play : hanoi -> int -> int -> hanoi
   ```

   telle que `play hanoi src dst` renvoie le jeu `hanoi` dans lequel le disque au sommet de la tour `src` a été déplacé au sommet de la tour `dst`. Si `src` ou `dst` ne sont pas des indices de tour valides, le jeu est renvoyé inchangé.

   ```
   -  play { t1 = [1; 4];
             t2 = [2];
             t3 = [3; 5; 6] } 1 2
      =
      { t1 = [4];
        t2 = [1; 2];
        t3 = [3; 5; 6] }


   -  play { t1 = [1; 4];
             t2 = [2];
             t3 = [3; 5; 6] } 2 3
      =
      { t1 = [1; 4];
        t2 = [];
        t3 = [2; 3; 5; 6] }

   -  play { t1 = [1; 4];
   ```

```
        t2 = [2];
        t3 = [3; 5; 6] } 5 6
=
{ t1 = [1; 4];
  t2 = [2];
  t3 = [3; 5; 6] }
```

2. Écrire une fonction
   `compute : int -> 'a -> 'a -> 'a -> ('a * 'a) list = <fun>`
   telle que `compute n src dst aux` renvoie la liste des séquences de jeu de Hanoi permettant de déplacer une tour de `n` disques de la position `src` à la position `dst` en passant par la position intermédiaire `aux`. À noter que tous les paramètres de la fonction sont des entiers.

   Par exemple, pour déplacer 4 disques, la fonction `compute 4 1 3 2` produira le résultat :
   `[(1,2);(1,3);(2,3);(1,2);(3,1);(3,2);(1,2);(1,3);(2,3);(2,1);(3,1);(2,3);(1,2);(1,3);(2,3)]`.

   Pour faire cela, nous remarquerons que déplacer une tour de `n` disques de `src` vers `dst` en passant par `aux` c'est :
   - déplacer les `n-1` premiers disques de `src` vers `aux` ;
   - déplacer le disque restant de `src` vers `dst` ;
   - déplacer les `n-1` disques de `aux` vers `dst`.

   Vous devez implémenter cette stratégie dans la fonction récursive auxiliaire
   `compute_aux : int -> 'a -> 'a -> 'a -> ('a * 'a) list -> ('a * 'a) list`
   qui utilise son 4ème paramètre comme accumulateur pour stocker le résultat. Cette fonction permettra ensuite de réaliser la fonction `compute`.

3. En utilisant la fonction `play` précédemment définie, écrire une fonction
   `apply : hanoi -> (int * int) list -> hanoi list`
   telle que `apply h ml` renvoie la liste des états successifs des trois tours de Hanoï pour passer de l'état initial à l'état final. Ici vous devez également utiliser une fonction auxiliaire `apply_aux : hanoi -> (int * int) list -> hanoi list -> hanoi list`.

   Exemple d'appel de la fonction `apply` :

```
# apply (make [1;2] [] []) (compute 2 1 3 2);;
- : hanoi list = [{t1 = []; t2 = []; t3 = [1; 2]};
                  {t1 = []; t2 = [1]; t3 = [2]};
                  {t1 = [2]; t2 = [1]; t3 = []}]
```

Cet exemple peut être visualisé comme suit :

```
  1
  2
============

  2  1
============

     1  2
============

        1
        2
============
```

4. En utilisant les fonctions `make`, `make_list`, `print_hanoi`, `compute`, `apply`, `print_hanoi_list`, `List.rev` précédemment vues ou définies, écrire une fonction `hanoi_towers : int -> unit`
   qui :
   - à partir de la valeur entière passée comme paramètre `n`, crée la première tour `tour1` du jeu ;
   - crée le jeu `hanoi` à partir des listes suivantes `tour1`, `[]`, `[]` et affiche ce jeu ;
   - calcule la séquence de mouvements qui doivent être réalisés pour déplacer les disques du jeu ;
   - applique cette séquence à `hanoi` et affiche le résultat des déplacements.

   Cette fonction doit permettre notamment de passer par exemple de `{ t1 = [1; 2]; t2 = []; t3 = [] }` à `{ t1 = []; t2 = []; t3 = [1; 2] }`. L'affichage attendu pour cet exemple est :

```
  1
  2
============

  2  1
============

     1  2
============

        1
        2
============
```