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

TD/TP 4a : 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.

PRÉPARATION :

- Télécharger l'archive « `TPFamille.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 « `TPFamille` » 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 TPFamille.ml TestFamille.ml`
      - `ocamlc -o test.exe TPFamille.cmo TestFamille.cmo`
      - `./test.exe`

      à chaque fois que vous souhaitez compiler et exécutez votre programme.
  - Un fichier « `TestFamille.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 le prédicat « `a_un_pere` » et vous souhaitez le tester. Pour cela, il faudra :
    - enlever dans le fichier « `TestFamille.ml` » les commentaires pour la partie des tests nommée « `TEST : A_UN_PERE` » ;
    - 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).

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

Partie 1 : Famille (listes)

Rappel

- `List.length` – longueur d'une liste ;
- `List.hd` – premier élément de la liste ;
- `List.tl` – liste privée de son premier élément ;
- `List.map` – `List.map f [a ;b ;... ;c]` calcule `[f a; f b; ...; f c]` ;
- `List.fold_left` – `List.fold_left f a [b1; b2; ...; bn]` calcule `f (...(f (f a b1) b2)...) bn`.

Vous pouvez également consulter la bibliothèque « `List` » sur le site de OCaml (https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html).

Exercice 1.1 (une affaire de famille(s))

Dans son étude sociologique de la société Fang (*Balandier, Sociologie actuelle de l'Afrique noire, PUF, 1955*), G. Balandier montre l'importance de la notion de lignage dans la structuration des relations sociales. Deux hommes sont du même lignage s'ils peuvent identifier un ancêtre masculin commun. Connaître sa généalogie est donc indispensable, ce qui introduit des difficultés puisque les noms de famille n'existent pas (chaque personne a un nom original) et qu'il n'y a pas d'enregistrement écrit. Pour résoudre cette difficulté, les Fangs ont inventé une technique de construction des noms qui favorise la mémorisation de la généalogie. Le nom d'une personne est composé de deux éléments, le premier élément est original, le second est le premier élément du nom du père. Une généalogie a donc la forme suivante :
Etogo Sé, fils de Sé Mafo, fils de Mafo Ekwikwi, fils de Ekwikwi Zogo, fils de Zogo Mboyen.

Le problème que nous nous posons est le suivant. Nous disposons d'une liste de noms. Cette liste a été construite en compilant des sources diverses (chroniques, registres, recensements, …). Elle représente les habitants, vivants ou non, d'une certaine région. En fonction de ces informations et en faisant l'hypothèse qu'un même élément de nom ne peut pas se trouver dans deux lignages différents :

- Peut-on déterminer si deux personnes sont « frères de lignage » ?
- Peut-on connaître le nombre de lignages ?

Modélisation

Une **personne** sera représentée par un couple formé de deux noms. Par exemple, `("Etogo","Se")` ou `("Se","Mafo")` ou `("Mafo","Ekwikwi")` ou `("Ekwikwi","Zogo")` ou `("Zogo","Mboyen")`.

Nous appellerons « **population** » l'ensemble des personnes que nous considérerons. Elle sera représentée par une liste de personnes. Par exemple, `[("Se","Mafo"); ("Togo","Bogo"); ("Zogo","Mboyen")]`.

Nous appellerons « **lignage** » une suite de noms obtenus à partir d'une personne et en remontant ses ancêtres.

L'exemple précédent produit le lignage suivant à partir de la personne `("Etogo","Se")` : `["Etogo"; "Se"; "Mafo"; "Ekwikwi"; "Zogo"; "Mboyen"]`.

Il sera donc possible de répondre aux deux questions si nous savons construire la chaîne relative à une personne.

- Pour la première (*Peut-on déterminer si deux personnes sont « frères de lignage » ?*), il suffit de comparer les derniers éléments des deux lignages.
- Pour la seconde (*Peut-on connaître le nombre de lignages ?*), il suffit de compter le nombre de derniers éléments différents dans l'ensemble des lignages de toutes les personnes.

Question 1 : Création des données

Créez :

- trois personnes
  - `p1 : ("Etogo","Se");`
  - `p2 : ("Se","Mafo");`
  - `p3 : ("Togo","Bogo");`
- deux populations
  - `pop1` contenant les personnes suivantes : `("Se","Mafo"), ("Togo","Bogo"), ("Zogo","Mboyen");`
  - `pop2` contenant les personnes suivantes : `("Etogo","Se"), ("Se","Mafo"); ("Mafo","Ekwikwi"), ("Ekwikwi","Zogo"), ("Zogo","Mboyen");`
- une liste des lignages
  - `liste_lignages1 :`

```
[["Etogo"; "Se"; "Mafo"; "Ekwikwi"; "Zogo"];
 ["Se"; "Mafo"; "Ekwikwi"; "Zogo"];
 ["Mafo"; "Ekwikwi"; "Zogo"];["Ekwikwi"; "Zogo"];
 ["Zogo"]] .
```

Question 2 : Prédicat « `a_un_pere` »

Écrivez un prédicat « `a_un_pere` » de type « `'a*'b -> ('b*'c) list -> bool` » qui teste si une personne possède un père dans la population, concrètement, s'il existe une personne dans la population dont le premier élément est égal au second élément de la personne dont on cherche à savoir si elle a un père.

Question 3 : Fonction « `pere` »

Écrivez une fonction « `pere` » de type « `'a*'b -> ('b*'c) list -> 'b*'c` » qui retourne le père du premier argument. Si le père n'est pas présent dans la population donnée, le message d'erreur "La population ne contient pas de pere." est généré.

Question 4 : Fonction « `lignage` »

Écrivez une fonction « `lignage` » de type « `'a*'a -> ('a*'a) list -> 'a list` » qui calcule, pour une personne et une population donnée, la chaîne des noms des ancêtres de la personne (elle comprise). Un lignage comporte au minimum deux éléments : les deux éléments du couple représentant la personne donnée en premier argument.

Vous penserez à utiliser les deux fonctions précédentes.

Question 5 : Fonction « dernier »

Écrivez une fonction « `dernier` » de type « `'a list -> 'a` » qui donne le dernier élément d'une liste. Vous utiliserez la fonction « `List.rev : 'a list -> 'a list` » qui renverse une liste.

Question 6 : Prédicat « frere »

Écrivez un prédicat « `frere` » de type « `'a*'a -> 'a*'a -> ('a*'a) list -> bool` », où liste est une population, qui rend « `true` » si deux personnes sont de « frères de lignage », c'est-à-dire, s'ils possèdent un(des) ancêtre(s) commun(s).

Question 7 : Fonction « liste_lignages »

En utilisant la fonction « `List.map` », écrivez la fonction « `liste_lignages` » de type « `('a*'a) list -> 'a list list` » qui, étant donné une population, calcule la liste des tous les lignages associés aux personnes de la population.

Question 8 : Fonction « liste_ancetres »

En utilisant la fonction « `List.map` », écrivez la fonction « `liste_ancetres` » de type « `'a list list -> 'a list` » qui donne la liste des derniers éléments de chaque lignage.

Question 9 : Fonction « sans_double »

Le résultat de la fonction « `liste_ancetres` » comporte a priori plusieurs occurrences d'un même nom.
Écrivez une fonction « `sans_double` » de type « `'a list -> 'a list` » qui retourne une liste identique à celle donnée en argument dans laquelle on a supprimé les occurrences multiples.
Exemple :

```
sans_double [1; 2; 1; 3; 3; 1];;
- : [2; 3; 1]
```

Question 10 : Retour vers les questions

A l'aide des fonctions écrites ci-dessus, répondez aux questions suivantes :

- Déterminer si deux personnes `p1` et `p2` sont « frères de lignage » dans la population pop1. A noter que `p1`, `p2` et pop1 ont été définies dans la question 1.

- Combien de lignages avons-nous pour les populations :
  -
```
pop1=[("Se","Mafo");
      ("Togo","Bogo");
      ("Zogo","Mboyen")]
```
  -
```
pop2=[("Etogo","Se");
      ("Se","Mafo");
      ("Mafo","Ekwikwi");
      ("Ekwikwi","Zogo");
      ("Zogo","Mboyen")]
```