1 Algorithmes élémentaires sur les listes

Le but de cet exercice est de vous familiariser avec le schéma récursif sur les listes. Certaines de ces fonctions existent déjà en bibliothèque ; ce serait naturellement ces versions qu'il conviendrait d'utiliser dans la pratique.

Réécrivez les fonctions suivantes vues en TD :

- length : $'a \overset{l}{list} \rightarrow$ int qui donne la longueur d'une liste
- map : $('a \overset{f}{\rightarrow} 'b) \rightarrow 'a \overset{l}{list} \rightarrow$ 'b list qui applique la fonction f à tous les éléments d'une liste

2 Une affaire de famille(s)

Enoncé du problème

Dans son étude sociologique de la société Fang (*in* 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 l'on se pose 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

Un nom (donc une personne) sera représenté par un couple formé de deux éléments de noms. Nous appellerons " *chaîne* " une suite d'éléments de noms obtenus en remontant les ancêtres. L'exemple précédent produit la chaîne suivante :

```
[ "Etogo" ; "Sé" ; "Mafo" ; "Ekwikwi" ; "Zogo" ; "Mboyen"].
```

Il sera donc possible de répondre aux deux questions si on sait construire la chaîne relative à une personne. Pour la première, il suffit de comparer les derniers éléments des deux chaînes (pourquoi ?), pour la seconde, il suffit de compter le nombre de derniers éléments différents dans l'ensemble des chaînes de toutes les personnes (pourquoi ?)

Nous appellerons " *population* " l'ensemble des personnes que nous considérons. Elle sera

représentée par une liste de noms.

Fonction a_un_pere

Ecrivez un prédicat de profil

```
a_un_pere : 'a* 'a -> ('a*'a) list -> bool
```

qui teste si une personne possède un père dans la population ; concrètement, s'il existe un couple dans la population dont le premier élément de nom est égal au second élément de nom du premier argument.

Fonction père

Ecrivez un prédicat de profil

```
pere : 'a* 'a -> ('a*'a) list -> 'a*'a
```

qui retourne le père du premier argument. On suppose que ce père existe.

Fonction chaine

Ecrivez une fonction de profil

```
chaine : 'a* 'a -> ('a*'a) list -> 'a list
```

qui calcule pour une personne et une population, la chaîne des éléments de noms. Une chaîne comporte au minimum deux éléments : ceux formant le nom de la personne. Vous penserez à utiliser les deux fonctions précédentes.

Fonction dernier

Ecrivez une fonction de profil

```
dernier : '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.

Prédicat frere

Ecrivez un prédicat de profil

```
frere : 'a*'a -> 'a*'a -> ('a*'a) list -> bool
```

qui rend `true` si deux personnes appartiennent au même lignage.

Fonction liste_chaines

En utilisant la fonction `List.map`, écrivez la fonction de profil

```
liste_chaines : ('a*'a) list -> 'a list list
```

qui, étant donné une population, calcule la liste des toutes les chaînes associées aux personnes de la population.

Fonction liste_ancetres

En utilisant la fonction `List.map`, écrivez la fonction de profil

```
liste_ancetres : 'a list list -> 'a list
```

qui donne la liste des derniers éléments de chaque chaîne.

Fonction sans_double

La fonction `liste_ancetres` comporte a priori plusieurs occurences des mêmes éléments de nom. Ecrivez une fonction de profil

```
sans_double : 'a list -> 'a list
```

qui, étant donné une liste, retourne la liste des valeurs présentes dans l'argument en supprimant les occurences multiples.

Exemple :

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