TP n°3 - Correction

Un générateur de spam en OCaml

La generation de texte pseudo-aleatoire est un probleme bien connu des producteurs de spams. Une methode pour generer ce type de texte consiste a partir d'un texte reel (*e.g.* le contenu d'une ou plusieurs pages web), et de construire pour ce texte une *table de succession*, sous forme de liste d'association.

Table de succession

Pour chaque mot `m` du texte, une table de succession contient un couple de la forme `(m, ls)`, ou `ls` est une liste de mots contenant, pour chaque occurrence de `m` dans le texte, un exemplaire du mot qui suit cette occurrence. Voici par exemple un texte dû a Aristote :

> *"Le faux et le vrai ne sont pas dans les choses, comme si le bien était le vrai et le mal, en lui-même le faux, mais dans la pensée, et en ce qui regarde les natures simples et les essences, le vrai et le faux n'existent pas même dans la pensée."*

En convertissant les majuscules en minuscules, et considerant l'espace comme l'unique separateur de mots, voici le debut sa table de succession :

```
[("le",   ["faux"; "vrai"; "faux,"; "mal,";
           "vrai"; "bien"; "vrai"; "faux"]);
 ("faux", ["n'existent"; "et"]);
 ("et",   ["le"; "les"; "en"; "le"; "le"]); ...]
```

Le mot *"le"* a huit occurrences dans le texte. Trois d'entre elles sont suivies par le mot *"vrai"*, qui apparaît donc trois fois dans sa liste associee. Noter que le mot *"pensée."* a une seule occurrence, et celle-ci est la derniere du texte : le couple correspondant dans la table est donc `("pensée.", []).`

Génération du texte

La generation d'un texte aleatoire a partir d'une table de succession se fait par la methode suivante : le premier mot du texte genere est choisi au hasard dans la table. A chaque etape, on prend la liste associe au mot courant dans la table, et le mot suivant est choisi au hasard dans cette liste. Si le mot courant est le dernier, le mot suivant est a nouveau choisi au hasard dans la table.

Noter que plus un mot a tendance a suivre le mot courant dans le texte initial, et plus il a de chances d'être le mot suivant. Le resultat ressemble donc a l'original, tout en etant di erent :

> *"pas même dans la pensée, et le bien était le mal, en ce qui regarde les choses, comme si le faux n'existent pas même dans la pensée, comme si le vrai et les essences, le bien était le vrai"*

On considerera l'espace comme l'unique separateur des mots d'un texte. Il peut avoir un ou plusieurs espaces entre deux mots, ainsi qu'avant le premier et le dernier mot d'un texte.

Exercice 1 Les fonctions utilitaires suivantes sont deux a deux independantes.

{ Ecrire une fonction `simplifier : 'a list -> 'a list` renvoyant, a partir d'une liste quelconque, la liste obtenue en ne gardant qu'une seule occurrence de chaque element. Exemple :

```
simplifier  ["ab"; "cd"; "ab"; "ab"; "ef"] = ["ab"; "cd"; "ef"]
(* ou ["ef"; "cd"; "ab"] selon l'implementation *);;
```

Correction :

```
let rec simplifier l = match l with
    [] -> []
  | x::l' ->
      (if List.mem x l' then [] else [x])@
        (simplifier l');;
(* ou, avec List.fold_left : *)
let simplifier l =
  List.fold_left
    (fun l x -> if List.mem x l then l else x::l) [] l
;;
```

{ Ecrire une fonction `successeurs : 'a -> 'a list -> 'a list` telle que `successeurs x l` renvoie la liste des elements situes apres chacune des occurrences de `x` dans `l`. Exemples :

```
successeurs "cd" ["ab"; "cd"; "ab"; "ab"; "ef"] = ["ab"]
successeurs "ab" ["ab"; "cd"; "ab"; "ab"; "ef"] = ["cd"; "ab"; "ef"]
successeurs "ef" ["ab"; "cd"; "ab"; "ab"; "ef"] = []
```

Correction :

```
let rec successeurs x l = match l with
  y::z::l' ->
    (if x = y then [z] else [])@(successeurs x (z::l'))
| _ -> []
```

{ A partir des fonctions prede nies `String.contains`, `String.index`, `String.sub`, ecrire une fonction `decouper : string -> string list`. Appliquee a un texte, cette fonction doit renvoyer la liste formee de la suite des mots du texte, dans l'ordre ou ils apparaissent. Exemples :

```
{ decouper " abc abc   def " = ["abc"; "abc"; "def"]
{ decouper " abc" = ["abc"]
{ decouper "" = []
```

Correction :

```
let rec decouper s =
  if s = "" then [] else                                    (*1*)
    if not (String.contains s ' ') then [s] else            (*2*)
  let n  = String.index s ' ' in                            (*3*)
  let s' = String.sub s (n + 1) (String.length s - (n + 1)) in  (*4*)
    if n = 0 then decouper s' else                          (*5*)
      (String.sub s 0 n)::(decouper s')                     (*6*)
;;
(* si s est la chaine vide, la liste obtenue est vide (1).*)
(* sinon, et si s ne contient aucun espace (2), la liste  *)
(* des mots de s est reduite a [s]. sinon, on cherche la  *)
(* position du premier espace dans s (3). on extrait de s *)
(* la sous-chaine s' commencant immediatement apres cet   *)
(* espace (4). si l'espace est en debut de chaine, on     *)
(* l'oublie, on relance le decoupage sur s' (5). sinon,   *)
(* on extrait de s son premier mot, entre le debut de la  *)
(* chaine et le premier espace (6), et on relance sur s'. *)
```

Exercice 2 En vous servant de `simplifier`, `successeurs` et `List.map`, ecrire une fonction :

`table : string list -> (string * (string list)) list`

telle que si `lm` est un texte decoupe en liste de mots, `table lm` renvoie la table de succession de ce texte.

Correction :

```
let table lm =
  List.map (fun m -> (m, successeurs m lm)) (simplifier lm)
;;
(* on simplifie lm pour obtenir le lexique du texte  *)
(* (la liste de mots du texte sans repetitions). un  *)
(* map tranforme ensuite chaque mot m de lm en le    *)
(* couple forme de m et de la liste des successeurs  *)
(* des occurrences de m dans lm.                     *)
```

Exercice 3 Appliquee a un entier positif `n`, la fonction `Random.int` renvoie un entier compris entre `0` et `n-1` inclus. A partir de cette fonction, de `List.length` et `List.nth`, ecrire :

`piocher : 'a list -> 'a`

telle que pour toute liste `l` non vide, `piocher l` renvoie un element choisi au hasard dans `l`.

Correction :

```
let piocher l =
  List.nth l (Random.int (List.length l))
;;
```

Exercice 4 A partir de `piocher` et de `List.assoc`, ecrire une fonction :

`generer : int -> (string * (string list)) list -> string`

telle que si `n` est un entier et `t` est une table de successions, `generer n t` renvoie a partir de cette table et suivant la methode decrite dans l'introduction, un texte aleatoire a `n` mots, separes par des espaces. Vous pouvez ecrire une ou plusieurs fonctions auxiliaires.

Correction :

```
let generer n t =
  if n = 0 then "" else
  (* i  : compteur de mots *)
  (* sc : le mot courant   *)
  let rec aux i sc =
    if i = 1 then sc else
    (* la : liste de successeurs associee a sc    *)
    let la  = (List.assoc sc t) in
    (* sc' : le mot suivant *)
    let sc' =
      (* cas ou la liste des successeurs est vide *)
      if  la  = [] then (fst (piocher t))
      (* cas ou elle est non vide                 *)
      else piocher la
```

```
    in sc^" "^(aux (i - 1) sc')
  in
  (* pioche du premier mot, et lancement *)
  (* de la fonction auxiliaire :         *)
  aux n (fst (piocher t))
;;
```

Exercice 5 En deduire une fonction

`spam : int -> string -> string`

telle que si `n` est un entier et `s` est un texte, `spam n s` renvoie un texte aleatoire de `n` mots separes par des espaces, construit a partir de `s`. Pour de meilleurs resultats, commencez par convertir le texte initial en minuscules, a l'aide de `String.lowercase`.

Correction :

```
let spam n s =
 generer n (table (decouper (String.lowercase s)))
;;
```