Lambda calcul et programmation fonctionnelle

mini-projet : générateur de texte – partie 2

Ultime transformation

Au cours de ce TP, nous allons tenter de donner, pour chaque mot, le mot le plus susceptible de le suivre, comme une application d'autocomplétion pour smartphone.

La dernière fois, nous avons produit une liste de type `(String, [String])`. C'est idéal pour générer automatiquement du texte (prochaine partie), mais pas terrible pour ce que nous voulons faire : nous avons besoin, pour chaque mot de la liste, de son nombre d'occurrences, afin de garder les deux plus fréquents.

Prenons par exemple le tuple suivant :

```
["découvrir", "programmation", "programmation"]
```

Ce que nous voulons, en réalité, c'est :

```
[("découvrir", 1), ("programmation", 2)]
```

Nous devons donc écrire une fonction `listeVersNbOccurs` qui transforme une liste de mots en le nombre d'occurrences de chacun des mots.

```
listeVersNbOccurs :: [String] -> [(String, Int)]
```

Pour écrire cette fonction, il est recommandé d'utiliser la fonction `sort :: [a] -> [a]` qui permet de trier par ordre alphabétique les mots d'une liste de mots. Pour l'utiliser, vous devez ajouter en début de programme :

```
import Data.List
```

Il sera plus facile de travailler avec une liste triée.

Une fois que la fonction fonctionne correctement, écrivez la fonction `genererTable2` qui transforme la table fournie par la fonction `genererTable` en une liste d'éléments du type évoqué ci-dessus.

```
genererTable2 :: [(String, [String])] -> [(String, [(String, Int)])]
```

Mot suivant

Maintenant que nous disposons de cette liste, il est aisé de trouver le mot le plus susceptible de suivre un autre mot : il suffit de prendre le mot ayant la plus grande valeur associée. Par exemple, imaginons que nous ayons la liste suivante :

```
[("je", [("suis", 2), ("pense", 1)]),
 ("suis", [("en", 1), ("très", 1)]),
 ("en", [("train", 1)]),
 ("train", [("de", 1)]),
 ("de", [("découvrir", 1), ("programmation", 2)]),
 ("découvrir, [("le", 1)]),
 ("le, [("langage", 1), ("meilleur", 1)]),
 ("langage, [("de", 2)]),
 ("programmation", [("haskell", 1), ("au", 1)]),
 ("haskell", [("et", 1), ("est", 1)]),
 ("et", [("je", 1)]),
 ("très", [("content", 1)]),
 ("content", [("je", 1)]),
 ("pense", [("que", 1)]),
 ("que", [("haskell", 1)]),
 ("est", [("le", 1)]),
 ("meilleur", [("langage", 1)]),
 ("au", [("monde", 1)]),
 ("monde", [])]
```

Le mot qui suivra « je » sera « suis », celui qui suivra « de » sera « programmation », et celui qui suivra suis sera « en » ou « train » (tous les deux apparaissent le même nombre de fois, on peut donc choisir n'importe lequel des deux).

A priori, on souhaite donc écrire une fonction

```
motSuivant :: [(String, [(String, Int)])] -> String -> String
```

Mais quel sera le mot après « fonctionnel » ? Ce mot n'apparaît pas dans la liste, mais il faut bien que notre fonction renvoie quelque chose ! Pareil avec le mot « monde », qui est associé à une liste vide.

Comme certains mots n'ont pas de successeur, il faut que notre fonction puisse renvoyer la valeur « rien ». On va utiliser le type Maybe vu en cours :

```
motSuivant :: [(String, [(String, Int)])] -> String -> Maybe String
```

Ainsi, imaginons que dans la variable table j'aie la liste donnée ci-dessus. On aura donc :

```
motSuivant table "et" == Just "je"

motSuivant table "monde" == Nothing

motSuivant table "de" == Just "programmation"
```

Écrivez enfin un programme principal qui permet de vérifier que tout fonctionne correctement.