Lambda calcul et programmation fonctionnelle

mini-projet : générateur de texte

Principe général

Nous allons créer un outil permettant de générer automatiquement du texte à partir d'un modèle probabiliste.

L'outil aura deux utilités. Premièrement, il permettra de faire de l'autocomplétion de texte, à la manière de ce qui existe sur les smartphones : en fonction de mots qui viennent d'être tapés par l'utilisateur, l'outil proposera trois mots, les plus susceptibles d'arriver ensuite. Par exemple, si j'ai tapé « Bonjour, je suis », l'outil proposera les mots « très », « arrivé » et « bientôt » qui sont les trois mots qui suivent en général « suis » dans un texte. L'algorithme que nous allons implémenter est celui qui est utilisé dans les smartphones (à quelques détails près).

Deuxièmement, l'outil permettra de faire de la génération automatique de texte, à partir d'un premier mot, il sera capable de générer tout un texte, qui n'aura probablement pas réellement de sens, mais qui en donnera l'illusion. C'est le précurseur de ChatGPT que nous allons écrire ici, mais évidemment, en beaucoup, beaucoup moins performant (mais beaucoup, beaucoup plus simple à écrire).

Dans les deux cas, il y aura deux étapes.

La première étape est une étape d'apprentissage. Il faut apprendre à l'algorithme quels sont les mots qui arrivent après un mot donné. Pour cela, nous allons lui donner en entrée un long texte, qui permettra à l'outil de construire une table de correspondance permettant de dire « après tel mot, les mots qui apparaissent le plus sont... »

La deuxième étape est l'étape de génération. On donne un mot, et l'outil propose ensuite un mot qui va suivre ce mot, en utilisant la table construite à la première étape.

Première partie : l'apprentissage

Imaginons que nous apprenions à partir de la phrase suivante :

« Je suis en train de découvrir le langage de programmation Haskell, et je suis très content. Je pense que Haskell est le meilleur langage de programmation au monde. »

Dans ce texte, les mots suivants apparaissent : je, suis, en, train, de, découvrir, le, langage, programmation, Haskell, et, très, content, pense, que, est, meilleur, au, monde. Notez que certains mots apparaissent plusieurs fois, d'autres une seule fois.

Pour chaque mot, voici la liste des mots qui le suivent :

- je : suis, suis, pense
- suis : en, très
- en : train
- train : de
- de : découvrir, programmation, programmation
- découvrir : le

- le : langage, meilleur
- langage : de, de
- programmation : Haskell, au
- Haskell : et, est
- et : je
- très : content
- content : je
- pense : que
- que : Haskell
- est : le
- meilleur : langage
- au : monde
- monde : <fin>

Le but de ce TP sera donc de construire cette table de correspondance.

Étape 1

Il faut d'abord récupérer tous les mots à l'intérieur d'une chaîne de caractère. Nous allons donc écrire la fonction suivante :

```
tousLesMots :: String -> [String]
```

De telle sorte que `(tousLesMots "Je suis en train de découvrir le langage de programmation Haskell, et je suis très content. Je pense que Haskell est le meilleur langage de programmation au monde.") == ["je", "suis", "en", "train", "de", "découvrir", "le", "langage", "de", "programmation", "haskell", "et", "je", "suis", "très", "content", "je", "pense", "que", "haskell", "est", "le", "meilleur", "langage", "de", "programmation", "au", "monde"]`

Pour écrire cette fonctions, nous allons avoir besoin de plusieurs fonctions :

- la fonction prédéfinie `words :: String -> [String]` qui renvoie tous les mots d'une chaîne de caractères. Mais attention : elle ne prend que les espaces et les retours à la ligne en compte, pas les signes de ponctuation :

```
(words "Je suis en train de découvrir le langage de programmation Haskell, et je suis très content. Je pense que Haskell est le meilleur langage de programmation au monde.") ==
["Je","suis","en","train","de","découvrir","le","langage","de","programmation","Haskell,","et","je","suis","très","content.","Je","pense","que","Haskell","est","le","meilleur","langage","de","programmation","au","monde."]
```

Notez que la fonction a par exemple identifié le mot "Haskell," en conservant la virgule. Ce n'est pas ce que nous voulons : nous ne voulons pas faire la différence entre "Haskell,"et "Haskell" : ce sont les mêmes mots ! Nous devons donc supprimer tous les signes de ponctuation, et ne conserver que les lettres et les espaces.

Notez également que les majuscules vont nous poser problème : "Je" et "je" ne sont pas les mêmes mots, et en l'occurrence, cela va nous poser des problèmes. Nous devons donc mettre tous les mots en minuscules.

Heureusement, le module Data.Char contient des fonctions qui vont nous être utiles. Il faut l'importer :

```
import Data.char
```

Les deux fonctions qui nous intéressent sont :

```
isAlpha :: Char -> Bool

toLower :: Char -> Char
```

`isAlpha` renvoie vrai ssi le caractère passé en paramètre est un caractère alphabétique. Par exemple, `(isAlpha 'J') == True` et `(isAlpha '.') == False`

`toLower` convertit une lettre majuscule en son équivalent en minuscule. Si le caractère passé en paramètre n'est pas une lettre majuscule, il est renvoyé tel quel. Par exemple, `(toLower 'J') == 'j'` et `(toLower '.') == '.'`

Écrivez la fonction `tousLesMots :: String -> [String]` en vous aidant des fonctions `words`, `isAlpha`, `toLower`, ainsi que des habituelles `map` et `filter`.

Étape 2 : construire la table de correspondance

Nous allons construire la table de correspondance que nous avons présentée plus haut. Nous voulons associer, à chaque mot trouvé dans la liste, la liste de tous les mots qui le suivent. Nous allons donc avoir une liste de mots, et pour chacun de ces mots, une autre liste de mots (ceux qui suivent).

Chaque élément de notre liste sera donc constitué de deux éléments : le mot qui nous intéresse, et une liste de mots. Nous allons donc utiliser un tuple : `(String, [String])`

Par exemple, puisque le mot « je » est suivi des mots « suis », « suis » et « pense », nous allons générer le tuple `("je", ["suis", "suis", "pense"])`

Pour le texte ci-dessus, nous allons donc générer la table suivante :

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

```
 ("au", ["monde"]),
 ("monde", [])]
```

Notez que cette table est bien de type `[(String, [String])]` comme convenu.

Nous allons donc écrire la fonction suivante, qui va générer la table associée à une liste de mots :

```
genererTable :: [String] -> [(String, [String])]
```

Comment allons-nous la créer ? L'algorithme général (que vous allez devoir écrire) est le suivant :

Parcourir la liste de mots donnée en entrée. À chaque mot, regarder si on l'avait déjà vu auparavant (s'il existe déjà dans la liste de sortie). S'il existe, on ajoute au tuple qui lui est associé le mot suivant dans la liste. S'il n'existe pas encore, on crée une liste qui contient uniquement le mot suivant, et on l'ajoute à la fin de la table.

Cette fonction est loin d'être facile à écrire. N'hésitez surtout pas à la décomposer et à écrire plusieurs fonctions plus simples que vous appellerez dans votre fonction principale.

Étape 3 : programme principal

Enfin, nous allons tester notre programme, étape indispensable pour savoir s'il fonctionne ou non.

Recopiez le programme principal suivant, qui va afficher votre table après l'avoir générée :

```
main :: IO()
main = do
     source <- getContents
     print (genererTable (tousLesMots source))
```

`getContents` va lire du contenu sur l'entrée standard, jusqu'à rencontrer le caractère EOF (fin de fichier). Il y a deux manières de procéder :

Soit vous tapez le texte d'entrée au clavier, ou vous le copiez-collez, et vous tapez sur Ctrl-D une fois que vous avez terminé.

Soit vous passez un fichier en paramètre, précédé du signe <, et le programme va lire le contenu de votre fichier au lieu de lire la saisie au clavier :

```
./generateur < fichier.txt
```