École **N**ationale **S**upérieure d'**I**nformatique pour l'**I**ndustrie et l'**E**ntreprise

IPRF
Projet d'analyse d'un fichier texte en OCaml

Année universitaire 2015-2016

2ème année

Réalisé par : Romain EPIARD

Table des matières

Introduction

Dans le cadre du cours de programmation fonctionnelle, nous devions réaliser un projet qui permet l'analyse d'un texte issu d'un fichier. Pour réaliser ce projet, nous devions réaliser quelques fonctions basiques au départ. Puis un code nous a été fourni, pour nous permettre de lire dans un fichier texte. Enfin, nous avons du coder des fonctions un peu plus compliquées.

Chapitre 1

Constitution et utilisation du projet

1.1 Arborescence du projet

Dans l'archive du projet, en vous plaçant à la racine, vous trouverez trois répertoires. Le répertoire src contient les sources du projet. Ce répertoire contient le fichier d'interface .mli (pour le compilateur) et le fichier d'implémentation .ml, des fonctions définies dans le fichier d'interface.

Il y a un répertoire _build qui ne sert qu'à contenir les fichiers objets .cmi et .cmo pendant la compilation. Une fois que le Makefile a généré les fichiers .cmi et .cmo, il les déplace dans le répertoire _build, avant de compiler le .cmo, pour générer l'exécutable. Le déplacement de ces fichiers objets me permet d'éviter d'avoir des fichiers objets qui se baladent au milieu des sources.

Enfin, dans le répertoire test se trouvent les trois fichiers .txt que j'utilise pour réaliser les tests de mon code. C'est aussi dans ce répertoire qu'est placé l'exécutable généré à la compilation. L'exécutable utilise donc les fichiers .txt placés dans le même répertoire que lui, pour appliquer les fonctions du code. J'ai placé les fichiers texte dans le répertoire test, car je ne voulais pas non-plus me retrouver avec des fichiers .txt au milieu des sources.

1.2 Makefile et interpréteur

Pour tester le projet, vous pourrez soit utiliser le Makefile et exécuter le binaire généré. Soit, vous pourrez aussi utiliser l'interpréteur emacs. Si vous utilisez l'interpréteur, une partie du code dans le fichier .ml ne vous sera pas utile, car cette partie du code ne comprend que les Printf qui permettent de voir les résultats des tests de fonctions.

Dans le cas où vous utiliseriez l'interpréteur tuareg dans emacs, les fonctions qui utilisent les fichiers textes ne fonctionneront pas. Car les fichiers en question se trouvent dans le répertoire test, il est toujours possible de les déplacer pour vérifier que cela fonctionne.

Chapitre 2

Analyse d'un fichier texte

2.1 Code fourni

Pour nous aider à prendre en charge l'analyse d'un fichier texte, un code nous a été fourni. Ce code est constitué de différentes fonctions.

2.1.1 Fonction get_words

La fonction get_words est celle que nous utilisons dans notre code. Elle prend une chaîne de caractères en entrée, qui correspond au chemin vers le fichier. En sortie, on récupère une liste de mots, correspondant à la liste de tous les mots présents dans le fichier texte.

Dans la fonction get_words, après avoir récupéré une liste de mots, applique un filtre, pour passer tous les mots en minuscule, et aussi leur appliquer le filtre de la fonction is_valid, pour ne garder uniquement que les mots valides. Ainsi, la liste de mots récupérées ne comprends que des mots valides, du point de vue de notre alphabet.

Cette fonction fait dans un premier temps appel à la fonction handle_file. En lui passant une fonction à appliquer à tous les éléments rencontrés dans le fichier.

2.1.2 Fonction handle_file

Cette fonction handle_file permet d'obtenir une variable qui pointe vers un fichier, en utilisant la fonction open_in. Avant de refermer le fichier, on récupère tout son contenu pour le stocker dans la variable res. Dans le même temps où l'on récupère son contenu, on applique la fonction update_acc sur tous les éléments rencontrés dans le fichier.

2.1.3 Fonction update_acc

La fonction update_acc est appelée sur chaque élément rencontré dans le fichier. L'élément en question est stocké dans une liste. Peu importe si cet élément est un caractère de notre alphabet valide ou non. Pour l'instant, on crée juste une liste contenant tous les éléments du texte, en appelant la fonction get_char pour récupérer les caractères un par un.

2.1.4 Fonction get_char

Cette fonction prend une valeur qui pointe vers le fichier en entrée. Elle récupère un caractère et renvoie ce caractère si ce n'est pas le caractère de fin de fichier (EOF ou End Of File).

2.1.5 Fonction words

Une fois que l'on a récupéré une liste correspondant au texte contenu dans le fichier, la fonction get_words fait appel à la fonction words, afin de découper cette liste de caractères récupérée, en mots. Les mots sont découpés par rapport aux espaces. J'ai apporté une petite modification au code fourni. En effet, en testant la première fois, je me suis aperçu que certains mots, proches des retours à la ligne, n'étaient pas présents dans la liste de mots récupérées. J'ai donc modifié la fonction words, pour qu'elle découpe la liste de caractères en mots, suivant les retours à la ligne et les tabulation, aussi, en plus des espaces.

2.2 Fonction count_words première approche inefficace

La fonction count_words nous permet de connaître le nombre de mots valides et le nombre de mots valides différents dans un fichier texte. La façon dont nous utilisons notre structure de données pour l'instant fonctionne pour des petits fichiers, mais si on la teste sur de gros fichiers (par exemple un fichier de plus de 2 mégaoctets), nous arrivons sur une erreurs de type Stack Overflow. Effectivement, gérer des listes de plus de 20 000 ou 30 000 éléments peut s'avérer gourmand en termes de mémoire. Dans mon code, j'ai volontairement commenté l'utilisation de count_words sur le fichier lourd (nommé « lorem.txt »), car quand on compile, on arrive sur un stack overflow, mais le programme s'arrête. Comme je n'ai pas réussi à catcher l'exception stack_overflow, pour éviter que mon programme ne s'arrête, j'ai préféré la commenter. Si on veut le tester en interpréteur, il suffit de dé commenter la ligne.

2.3 Utilisation d'un trie

Pour remédier au problème de stack overflow, nous utilisons donc un trie. Ce trie va nous permettre de gérer le texte dans un arbre. Pour les questions 10 et 11, je n'ai pas vraiment eu de problème. Cependant, sur la question 12, j'ai eu un problème de stack overflow, une nouvelle fois. En effet, la structure utilisée pour stocker le texte était bien un trie, mais nous passions tout de même par une liste, pour construire notre trie. Et donc forcément, sur de gros fichiers, nous arrivions tout de même à une erreur de type stack overflow. C'est pourquoi j'ai repris la fonction words fournie, pour l'adapter et lui faire utiliser la nouvelle structure de trie, pour éviter de passer par les liste. Ainsi, la fonction trie_words n'est qu'une adaptation de la fonction get_words. La différence est que nous récupérons un trie contenant les mots du texte. Et pour construire ce trie, nous n'utilisons presque pas de listes.

2.3.1 Question 16 et fonctions difficiles

A partir de la question 16, les choses ce sont vraiment corsées. Les deux premières fonctions, c'est-à dire le mot le plus utilisé, ainsi que son nombre d'occurrence, et le mot le plus avec sa taille ont été difficiles. J'avais deux options :

— Soit utiliser la fonction Map.Bindings pour récupérer les liens de l'arbre, et les utiliser,
— Soit utiliser la fonction Map.fold.

J'ai choisi d'utiliser Map.fold. En effet, Map.bindings nous renvoie tous les liens d'un arbre, mais sous forme d'une liste de couples. Hors sur de gros arbres, je pense que cela peut être problématique. J'ai donc utilisé Map.fold, auquel je passe une fonction qui descend jusqu'à la dernière feuille dans chaque branche, et qui, dès qu'elle détecte une valeur v pour le mot le plus utilisé ou que son compteur est supérieur à celui qu'elle avait avant, renvoie un couple (mot, taille ou nombre d'occurrence). Cet algorithme comportait un avantage. Imaginons si je cherche le mot le plus utilisé, je regarde à chaque noeud, la valeur v qui lui est associée. Si cette valeur est supérieure à la précédente, je renvoie la valeur v dans le couple, sinon je laisse la valeur qui y est déjà. Une fois que j'ai détecté ma valeur, l'algorithme remonte de façon récursive jusqu'à la racine de l'arbre, et en faisant ça, il ajoute toutes les clés rencontrées au mot. Ainsi, le mot, qui est donc une liste de caractères est construit, en partant de la fin, ce qui nous permet de renvoyer le mot directement, sans avoir besoin d'utiliser la fonction List.rev, pour le remettre à l'endroit.

Pour les trois dernières fonctions de la question 16, j'ai eu beaucoup de mal. Je me suis vite aperçu que je pouvais très difficilement faire une adaptation de l'algorithme que j'avais utilisé juste avant. J'ai donc du parcourir l'arbre, en construisant les mots à chaque fois, j'ajoute la clé rencontrée au mot en cours, et en même temps que je descend dans l'arbre, je regarde si la condition que je cherche à vérifier, donc si le mot est répété plus de k fois, qu'il a plus de n lettre, ou qu'il a bien le préfixe donné. Si la condition cherchée est vérifiée, alors j'ajoute le mot courant à ma liste de mots, sinon je n'utilise pas le mot.

Conclusion

Ce projet nous a permis de comprendre l'importance des langages fonctionnels, surtout dans l'utilisation des structures de données que sont les arbres. Ces langages sont vraiment très puissant pour utiliser les arbres, et faire des programmes très efficaces. On s'aperçoit assez rapidement que les listes ont leurs limites.