UCLouvain

École polytechnique de Louvain

LINFO1104 : Projet - Rapport

« Qui OZ-ce ? »

PAQUET Robin – 02192001

HOEBAER Antoine – 91242000

Mai 2021

1. Structure du programme

Notre programme est réparti en plusieurs fonctions de façon que chacune effectue une action en ce qui concerne la construction de l'arbre avec la base de données.

Dans un premier temps, il fallait extraire les questions pour chacun des personnages de la base de données. Pour cela nous avons utilisé la fonction « Record.arity » qui récupérer l'entièreté des arguments. Si nous voulions récupérer seulement les questions sans le nom du personnage, il fallait ajouter « .2 » à la suite. Ce qui a été notre cas.

```
{Record.arity Database.1}.2
```

Lorsque nous avions récupérer ces questions, il était alors possible d'effectuer les actions sur la base de données et donc de créer notre arbre. L'arbre que nous avons mis en place est écrit sous la forme :

```
question(BestQuestion true:{TreeBuilder BranchTrue} false:{TreeBuilder BranchFalse})
```

a. Fonction pour parcourir chaque personnage

Dans cette première fonction, le pattern matching est utilisé pour parcourir chaque personnage un à un. Lorsque l'on arrive sur un premier personnage, un appel à une seconde fonction est fait pour récupérer la meilleure question en fonction du calcul fait sur chaque question. Grâce à cette deuxième fonction, il nous est possible de construire la liste triée par ordre de meilleure question.

b. Fonction pour parcourir chaque question

Cette seconde fonction est utile dans le cas où elle permet de récupérer la question la plus discriminante. Pour ce faire, lorsque la réponse d'une question est vraie, nous incrémentons une variable mise à disposition dans l'argument de la fonction et nous faisons de même dans une autre variable quand la réponse est fausse.

Lorsqu'il n'y a plus de question, nous retournons alors le résultat de l'une des deux variables que nous avons incrémentées auparavant. Il s'agit de retourner celle ayant le plus petit score en premier pour pouvoir les ordonner dans la liste des meilleures questions par la suite.

c. Fonction pour répartir de l’arbre

La troisième fonction permet de diviser l’arbre en sous-branche. De nouveau grâce au pattern matching, nous parcourons toute la base de données avec ses questions et lorsque la réponse est vraie ou fausse, une séparation en sous-branche est faite avec une fonction récursive. Lorsqu’une question vient d’être parcourue, elle est supprimée de la liste des questions pour ainsi pouvoir passer la meilleure question suivante et continuer à séparer l’arbre.

Une fois que toute la base de données a été parcourue, deux variables contenant les branches vraies et fausses sont retournées. Elles seront utilisées dans une prochaine fonction pour la construction de l’arbre et ses sous-branches.

d. Fonction pour créer la liste des personnages

Cette fonction aussi petite soit-elle est rendue utile pour récupérer la liste des personnages et ainsi afficher au moment propice la réponse dans un « leaf ».

Pour créer cette fonction, nous avons fait du pattern matching pour effectuer la récursion qui parcourt la base de données et nous retourne chaque nom.

e. Fonction pour construire l’arbre

L’une des principales fonctions utiles à la construction de l’arbre est la fonction « TreeBuilder ». Dans cette fonction, plusieurs appels à d’autres fonctions sont nécessaires pour la bonne construction de l’arbre final. Comme l’appel à la fonction « Loop » citée précédemment qui retourne la liste triée par ordre de meilleure question. L’appel à la fonction « SplitTree » qui divise l’arbre en sous-branche lorsque les réponses sont vraies ou fausses. Et aussi l’appel à la fonction « Leaf » qui nous donne la liste des noms de personnage lorsqu’il n’y a plus de question au sein de la liste des meilleures questions et que l’on arrive sur une feuille de l’arbre.

f. Fonction pour récupérer la réponse

Cette petite fonction permet de récupérer la réponse que le joueur va donner pour ainsi pouvoir évoluer et se diriger dans l’arbre. Grâce à un pattern matching, si une réponse est vraie pour une question Q, nous nous propagerons dans le « GameDriver » vers une sous-branche T. Si une réponse est fausse, nous nous dirigerons alors vers une sous-branche F. Et lorsque l’on tombera sur un « leaf », nous aurons une réponse N que le joueur pourra visualiser.

g. Fonction pour convertir de la réponse

Initialement, nous recevions une réponse finale sous forme de liste, mais il était demandé de retourner des strings séparés par des virgules. Nous nous sommes donc intéressés à la fonction développée dans le fichier « write_example.oz » pour l'adapter au code de notre programme et recevoir une réponse du type demandé.

Grâce à la fonction « GameDriver », le jeu prend tout son sens. Cette fonction permet l'interaction entre le système et le joueur. C'est elle qui retourne les questions et les résultats au joueur.

Pour terminer, un dernier petit changement était nécessaire. Il fallait ajouter l'argument « ans » dans le record « getArgs » pour pouvoir proposer un fichier « answer » lorsqu'on le souhaitait lors du lancement du jeu en ligne de commande.

2. Décisions de conception

Dans un premier temps, nous pensions qu'il serait correct de compter l'ordre d'importance des questions sans tenir compte des personnages. Nous avons alors pensé à parcourir chaque question et l'incrémenter dans une variable lorsque la réponse était vraie et dans une autre lorsque c'était faux. Cependant lorsque nous avions le résultat, ce dernier n'était pas lié directement à la question. Il était alors difficile d'effectuer des actions, nous avons alors dû réfléchir à une autre solution.

Notre deuxième solution était alors la bonne. Cette solution est celle décrite pour les deux premières fonctions un peu plus haut. C'est ensuite que nous avons pu commencer a effectuer un tri sur les questions et que nous avons commencé la construction de notre arbre.

3. Limitations et problèmes connus

Notre première grosse limitation et donc notre premier gros problème est survenu lorsqu'il était question d'incrémenter un score pour chaque question. Ce score qui sera rendu utile plus tard pour trier les questions par ordre de la plus discriminante. Pour ce faire, nous avons d'abord commencé par implémenter un code avec des cellules que l'on additionnerait lorsqu'une réponse serait vraie ou fausse.

Cependant par la suite, nous nous sommes rendu compte que notre programme devait être déclaratif et donc que l'utilisation de cellule n'était pas autorisée. Nous nous sommes donc tournés vers la solution que nous utilisons actuellement qui est l'incrémentation dans les arguments d'une fonction. Ces arguments sont ensuite retournés lorsqu'il n'y a plus de question à parcourir et c'est comme cela que nous obtenons le résultat de l'incrémentation.