Université catholique de Louvain

[LINFO1104]
Projet : Qui OZ-ce ?

Simon Cornelis
simon.cornelis@student.uclouvain.be

Vincent Buccilli
vincent.buccilli@student.uclouvain.be

May 6, 2021

1 Fonctionnement de notre code

Pour ce projet nous avions pour objectif de recréer le jeu « Qui est-ce » en *Oz*. Afin de nous aider, nous avons eu accès à quelques fichiers d'exemple ainsi qu'une librairie *ProjectLib* comprenant un ensemble de fonctions utiles au bon déroulement du jeu ainsi qu'une interface graphique permettant un plus grand confort pour le joueur. Cependant, il nous était malgré tout demandé de faire fonctionner le programme en ligne de commande pour permettre l'automatisation des tests.

Notre approche a donc été de partir de ces exemples, de créer un code fonctionnel et ensuite de l'optimiser et d'implémenter toutes les fonctionnalités demandées ainsi que les extensions. Pour la structure de notre code, nous avons favorisé la création de sous-fonctions simples et courtes, plutôt que peu de fonctions « all-in-one ». Ainsi nous avons les fonctions principales suivantes, dans l'unique fichier *Main.oz*, composées de sous-fonctions dans leurs fermetures :

- *NextQuestion* : retourne, parmi les données restantes, la meilleure question à poser pour diviser le groupe en deux parts les plus égales possibles.
- *TreeBuilder* : fonction produisant un arbre assez équilibré (probablement pas optimal dans tous les cas) de questions à poser au joueur.
- *GameDriver* : fonction parcourant l'arbre passé en argument de façon récursive en posant des questions au joueur pour le choix du sous-arbre à suivre.

1.1 TreeBuilder

Commençons par explorer la fonction *TreeBuilder*. Cette fonction se charge de produire un arbre respectant la définition ci-dessous (structure hybride entre un arbre et une liste). Avec Q qui est la question à poser au joueur et **true**, **false** et unknown qui sont les trois chemins possibles, en fonction de la réponse qu'il choisit. Le chemin unknown ne réduit pas la taille de la liste de personnages possibles, elle ne contient qu'une version où la question actuelle a été retirée de chaque personnage.

```
<Tree T> ::= <List L> % liste de personnages possibles (nil si aucun)
           | question(Q true:<Tree T> false:<Tree T> unknown:<Tree T>)
```

Pour les chemins **true** et **false** on retrouve, dans la fermeture du *TreeBuilder*, la fonction *Split* prenant comme argument une liste de personnages ainsi que la question actuelle. Son rôle est de diviser la liste en deux parties en fonction de leur réponse à la question actuelle dans la base de données.

```
56 Ask = fun {$ E} E.Question end
57 {List.partition Data Ask T F}
```

Pour cela on utilise la fonction *List.partition* qui sépare *Data* en T et F en fonction de leur réponse à la *Question*. La fonction *Split* renvoie ensuite un arbre faisant un appel récursif au *TreeBuilder* avec les listes de personnes restantes réduites et où on a retiré la question qui vient d'être posée à l'aide de *Record.substract*.

```
58 RemoveQ = fun {$ E} {Record.subtract E Question} end
59 question(Question
60         true:{TreeBuilder {Map T RemoveQ}}
61         false:{TreeBuilder {Map F RemoveQ}}
62         unknown:{TreeBuilder {Map Data RemoveQ}})
```

Comme mentionné précédemment, *TreeBuilder* est une fonction récursive. Le cas de base est lorsque la liste est vide puisqu'il ne reste plus de personnages. On ne devrait, cependant, jamais croiser de liste vide (nil) dans l'arbre si on veut respecter la consigne nous demandant de poser un minimum de questions possible. Sinon on fait un appel à la fonction *NextQuestion* qui nous renvoie nil si il n'y a plus de question à poser. Dans ce cas on transforme notre liste de records pour n'avoir plus qu'une liste de personnages. Si *NextQuestion* nous renvoie une question, on fait un appel à *Split* avec cette question.

```
case {NextQuestion Data}
   of nil then {Map Data fun {$ E} E.1 end} % keep only persons list
   [] NextQ then {Split Data NextQ}
end
```

1.2 NextQuestion

L'objectif de *NextQuestion* est de retourner la prochaine question à poser au joueur de façon à minimiser le nombre de questions à poser. L'approche que nous avons choisie est de trouver la question qui divise le groupe de personnages en deux parts les plus égales possibles. Pour cela, nous fixons un idéal qui est la moitié du nombre de joueurs et trouvons la question la plus proche de cet idéal.

```
Ideal = Persons div 2
fun {Dist A} {Abs Ideal - A} end
```

On a pour commencer un record score (ci-dessous) contenant les questions restantes avec chacune une valeur de 0. On génère ce record en prenant la première personne de Data, on crée une liste de ses features et on retire la première qui est celle contenant le nom du personnage. On crée pour chaque question un tuple pour finalement utiliser la fonction *ToRecord* pour transformer tout ça en un record de type score(q1:0 q2:0 ... qn:0) avec q1, q2, ... qui sont les questions.

```
Start={List.toRecord score {Map {Arity Data.1}.2 fun {$ E} E#0 end}}
```

La fonction *ScoreQuestions* est utilisée pour mettre a jour le score actuel (dans l'accumulateur Acc) en ajoutant 1 pour chaque question où la réponse est **true** pour le personnage Elem actuel.

```
fun {ScoreQuestions Acc Elem}
   {Record.mapInd Acc fun {$ Q E} if Elem.Q then E+1 else E end end}
end
```

La fonction *GetBestScoredQ* parcoure le record dans Scores et garde la question avec un nombre de **true** le plus proche de l'idéal à l'aide de la fonction *Dist*. La comparaison est faite avec un *FoldL* où l'accumulateur est la question avec le meilleur score selon la comparaison avec la fonction *IsBetterThan*.

La variable AllEqual vaut **true** si tous les personnages restants répondent la même chose à toutes les questions. Il ne sera donc pas nécessaire de poser de questions supplémentaires, on a notre liste de réponses. Cela se produit lorsque toutes les questions ont un score nul ou maximal.

```
AllEqual = {Record.all Scores fun {$ E} E == {Length Data} orelse E
   == 0 end}
```

On renvoie donc la question avec le score le plus proche de l'idéal, sauf si il n'y a plus qu'une personne, qu'il n'y a plus de questions à poser ou que toutes les personnes répondent la même chose sur les questions restantes. Alors on renvoie nil pour le signaler au *TreeBuilder*.

1.3 GameDriver

L'implémentation du *GameDriver* est récursive, plus exactement la sous-fonction *Next* l'est. Elle prend en argument l'arbre actuel ainsi qu'une liste des arbres précédents (du plus récent au plus ancien).

La fonction *Next* a deux cas de base. Si l'arbre est nil, c'est qu'on est tombé sur une liste vide donc il n'y a pas de personnage qui a cette combinaison de réponses aux questions et on doit donc se rendre (ce cas ne devrait idéalement jamais arriver). Le deuxième cas serait une liste non vide, dans ce cas on a notre réponse !

La récursion se fait lorsqu'on tombe sur un arbre qui est a nouveau une question (plutôt qu'une liste). Alors on pose la question au joueur et on fait l'appel récursif en fonction de sa réponse. Et on ajoute l'arbre actuel à l'historique des arbres (le second argument) pour l'extension oops.

Le *GameDriver* fait donc un seul appel à *Next* (qui se réapelle tout seul jusqu'à obtenir le résultat de la partie). Le résultat est **false** lorsque la liste ne contenait pas le personnage auquel le joueur pensait, dans ce cas, on se rend. Sinon on a soit une liste de personnages, soit un unique personnage (atome).

1.4 Version en ligne de commande

Pour la version en ligne de commande, tout ce qui était nécessaire, c'est de se charger de la sortie du résultat dans le terminal. Pour cela, nous nous sommes basés sur le fichier *write_ example.oz*. Notre but est d'écrire dans le *standard output* donc on commence par ouvrir le fichier *stdout* et à la fin, juste avant l'appel à *Application.exit*, on le referme. Nous avons également défini une fonction qui permet d'imprimer des *virtual strings* pour n'avoir qu'à faire un appel à *FPrint*.

```
9 FPrint = proc {$ S} {File write(vs:S)} end
```

2 Extensions

2.1 Incertitude du joueur (Extension ***)

L'objectif de cette première extension était de permettre à l'utilisateur de répondre *Je ne sais pas* lorsque qu'il ne connaît pas la réponse à une question. Dans cet objectif, nous avons rajouté un atome supplémentaire *unknown* dans l'objet *question* qui permettra de souligner l'inconnue lors de la réponse. L'arbre prend désormais en compte cet atome et sélectionne également tout les champs pour les questions pour lesquelles la réponse est *unknown*

2.2 Bouton « oups » (Extension ***)

L'objectif de cette deuxième extension était de permettre une gestion des erreurs par l'utilisateur. En effet dans un premier temps nous avons implémenté la fonctionnalité qui permettait à l'utilisateur de revenir une question en arrière via le bouton *oops*. Via le positionnement de *oopsButton:true* dans les options du jeu, nous avons pu :

- Améliorer notre fonction *Next* qui prend désormais un référence au développement de l'arbre de la question précédente
- Introduire une nouvelle condition dans le cas où l'utilisateur appuie sur le *button oops*. Lorsque la fonction *askQuestion* retournera un atôme *oops*, alors la condition renverra la question précédente à l'utilisateur.

Dans un second temps, nous avons amélioré la fonctionnalité en permettant à l'utilisateur de revenir en arrière jusqu'à la première question du jeu.

3 Problèmes-limitations

Actuellement, l'ensemble des fonctionnalités demandées dans l'énoncé sont représentées et fonctionnelles. Que ce soit en ligne de commande ou via l'interface graphique, les différents possibilités de jeu sont réalisables.