UCLouvain - EPL

LINFO1104

Paradigmes de programmation

Rapport du projet

Professeur :
Van Roy Peter

Assistants :
Kabasele Gorby
Wirtgen Thomas

Auteurs :
Ben Haddou Mehdi - 19912000
Hennebo Eliot - 43762000

1 Introduction

Dans le cadre du cours de Paradigmes de programmation, il nous a été demandé d'utiliser nos connaissances en Oz afin d'implémenter le fonctionnement du jeu "Qui est-ce ?". Concrètement, nous avons une librairie à notre disposition et il nous est demandé de lire une base de données, de la transformer en arbre de décision avec un algorithme efficace, et ensuite de pouvoir poser des questions à un joueur afin de trouver à quel personnage il pense.

Il y a en plus de cela la possibilité d'implémenter des extensions supplémentaires à la version basique du programme, les extensions sont caractérisées par une difficulté calculée en nombre d'étoiles, et il nous est demandé d'acquérir au minimum 4 étoiles.

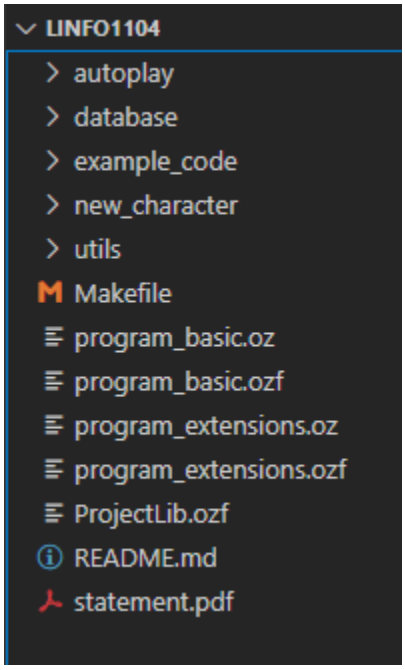
➔ Nous avons donc implémenté la version basique du programme reprenant les 3 grandes étapes, lire la base de données, construire l'arbre efficace, jouer au jeu.
➔ Nous avons également implémenté des extensions pour un total de 7 étoiles, reprenant donc les 3 extensions suivantes :

- Incertitude dans la base de données : 1*
- Incertitude du joueur (true/false/I don't know) : 3*
- Bouton "oups" (revenir à la question précédente) : 3*

2 Structure du projet

Pour structurer au mieux le projet, nous avons décidé de placer les différentes parties dans des dossiers, et laisser les principaux fichiers .oz dans le répertoire principal.

2.1 Dossiers & fichiers



- autoplay : Ce dossier contient simplement des fichiers textes avec des personnages ainsi que leurs questions/réponses que l'on peut passer au programme pour fonctionner en mode autoplay.
- database : Ce dossier contient les différentes bases de données qui nous ont été fournies et que nous avons ajoutées.

- example_code : Ce dossier contient le code d'exemple qui a été fourni lors de l'initialisation du projet, il permettait notamment d'illustrer le fonctionnement de la librairie ProjectLib.ozf.
- new_character : Ce dossier contient un fichier que l'on peut utiliser pour ajouter un personnage via l'extension appropriée, que nous n'avons pas développé.
- utils : Ce dossier contient un fichier avec toutes les méthodes utilitaires que nous avons créé à un moment donné du projet et que nous avons réussi à supprimer en optimisant notre fichier principal de façon à ne plus en avoir besoin.
- program_basic.oz : Fichier source de notre version basique sans extension.
- program_extensions.oz : Fichier source de notre version avec extensions.
- *.ozf : Fichiers .oz compilés reprenant nos fichiers sources compilés ainsi que la librairie fournie par les assistants.

2.2 Makefile

Nous avons essayé d'utiliser le Makefile au maximum pour faciliter le développement et les tests en le modifiant à nos besoins. C'est ainsi que nous avons réparti les points d'entrée en :

- ➔ all : qui permet de compiler tous les fichiers .oz du répertoire.
- ➔ clean : qui permet de supprimer tous les fichiers .ozf compilés par le Makefile auparavant.
- ➔ run_basic : qui permet de lancer la version basique de notre programme.
- ➔ run_extensions : qui permet de lancer la version avec extensions de notre programme.
- ➔ run_example : qui permet de lancer la version d'exemple fournie par les assistants.

La compilation via Makefile passe donc par des arguments que nous initialisons au début de ce Makefile en modifiant :

- ➔ DBPATH : Chemin vers la DB.
- ➔ NOGUI : Lancer sans l'interface graphique ?
- ➔ ANS : Chemin vers le fichier de réponse pour le mode autoplay.

```
run_basic: all
	@echo RUN program_basic.ozf
	@$(OZENGINE) program_basic.ozf --db $(DBPATH) $(NOGUI) --ans $(ANS)

run_extensions: all
	@echo RUN program_extensions.ozf
	@$(OZENGINE) program_extensions.ozf --db $(DBPATH) $(NOGUI) --ans $(ANS)
```

2.3 Entrées et sorties

Les arguments qui sont passés à l'exécution du Makefile ou lorsqu'on exécute directement en ligne de commande sont récupérés par le code suivant, avec des arguments par défaut en cas d'absence de l'argument dans la commande.

```
Args = {Application.getArgs record(
	'nogui'(single type:bool default:false optional:true)
	'db'(single type:string default:CWD#"database/database.txt")
	'ans'(single type:string default:CWD#"autoplay/test_answers.txt"))}
```

On récupère également un flux vers stdout afin de pouvoir y imprimer les résultats à la fin de l'exécution du programme.

```
Output = {New Open.file init(name: stdout
            flags: [write create truncate text])}
```

3 Implémentation : Version basique

Pour l'implémentation basique, nous avons commencé par établir certaines fonctions utilitaires afin de nous faciliter la vie pour effectuer les trois tâches de base. Parmis ces méthodes on retrouve de simples fonctions utiles tels que `Reverse`, `DeleteInd`, `Contains` ou encore des fonctions plus spécifiques au problème telle que `ComputeCounters` qui renvoie une liste de compteurs contenant la différence entre le nombre de réponses true/false par question dans la base de donnée et pour chaque question.

Une fois que nous avions ces différentes méthodes implémentées, cela nous a allégé d'un gros poids pour la construction de l'arbre. Nous avions plusieurs stratégies en tête et nous avons opté pour une méthode récursive se basant sur la méthode `ComputeCounters` citée ci-dessus. A chaque étape, notre algorithme récupère donc une liste des différences de true/false pour chaque question dans la base de donnée et supprime l'élèment le plus petit de cette liste, c'est à dire la question pour laquelle il y a presque autant de personnages qui répondent true ou false ce qui correspond à la question la plus discriminante actuellement. On récupère donc cette question et on la supprime de l'ensemble de questions qu'il reste.

Ensuite, parmis tous les personnages restants, nous récupérons dans deux listes les personnages qui répondent true ainsi que ceux qui répondent false à cette dernière question. Nous recalculons également à nouveau les compteurs de true/false pour chacune des deux listes et nous pouvons enfin construire l'arbre récursivement en envoyant les nouvelles listes adéquates dans les méthodes récursives appelées dans les champs 'true' et 'false' de l'élément 'question' courant.

Nous nous assurons également d'avoir des conditions de sortie afin de ne pas boucler indéfiniment ainsi que de retourner les leafs quand il faut. Les leafs sont retournées lorsque soit il n'y a plus qu'une question, soit il ne reste plus qu'un personnage ou encore lorsque la plus petite question actuelle n'est pas du tout discriminante (i.e. que des réponses true ou des réponses false).

Voici le code en question ;

```
fun {TreeBuilderAux Database Characters Counters Questions}
    if {Or {Length Questions} < 1 {Length Characters} <2 } then leaf(1:Characters)
    else
        local
        Min Question CharactersTrue CharactersFalse
        NewCountersTrue NewCountersFalse NewQuestions
        in
            Min = {MinPos Counters}
            Question = {Nth Questions Min}
            if {Nth Counters Min} == {Length Characters} then leaf(1:Characters)
            else
                NewQuestions = {DeleteInd Questions Min}

                CharactersTrue = {GetCharactersOnQuestion
                    Database Question true Characters}
                CharactersFalse = {GetCharactersOnQuestion
                    Database Question false Characters}

                NewCountersTrue = {ComputeCounters Database
                                        CharactersTrue NewQuestions}
                NewCountersFalse = {ComputeCounters Database
                                        CharactersFalse NewQuestions}

                question(
```

```
                    1:Question
                    true:{TreeBuilderAux Database CharactersTrue
                            NewCountersTrue NewQuestions}
                    false:{TreeBuilderAux Database CharactersFalse
                            NewCountersFalse NewQuestions}
                    )
            end
        end
    end
end
```

En ce qui concerne la méthode GameDriver, nous faisons un simple parcours d'arbre en fonction des résultats obtenus aux questions.

4 Implémentation : Extensions

Pour la partie extensions, nous avons su garder la même implémentation et structure de base en ne faisant que très peu de modifications sur celle-ci. En effet, les extensions sont construites de façon modulaire ce qui permet d'ajouter ou de supprimer aisément du code sans avoir d'effets de bords, ce qui est très pratique.

4.1 Incertitude dans la base de données (1*)

Pour implémenter cette extension, nous faisons en sorte de garder les personnages qui n'ont pas de réponse pour la question dans les 2 (ou 3 avec 'unknown') branches de l'arbre. Ainsi, l'utilisateur peut jouer avec une base de données qui comporte des personnages qui n'ont pas toutes les mêmes questions et on ne discrimine pas une personne pour une question qu'il ne possède pas.

4.2 Bouton "oups" (3*)

Pour le bouton oups qui permet de revenir en arrière à une question, le plus dur fut de choisir la bonne stratégie à adopter. Pour implémenter cette extension, nous avons donc choisi de construire un tableau de réponses true/false au fur et à mesure que l'utilisateur répond aux questions. Ainsi, lorsque l'utilisateur utilise l'option 'oops' nous n'avons qu'à re-parcourir l'arbre sur base des réponses incluses dans ce tableau.

Pour ce faire, avons donc implémenté une méthode `GetTree` qui prend en arguments un tree ainsi qu'un tableau de réponses et renvoie un nouvel arbre. Afin de ne pas reconstruire un deuxième tree en partant de zéro à chaque fois, nous passons une copie du 'tree source' à la fonction `GameDriverAux` (qui est utilisé par `GameDriver`) afin de toujours avoir l'arbre complet sans devoir le reconstruire. Cela demande un certain coût en mémoire mais permet d'économiser beaucoup d'opérations inutiles pour la reconstruction de l'arbre.
Voici le code de `GetTree` ;

```
fun {GetTree Tree Reps}
    Condition
in
    case Reps
    of nil then Tree
    [] H|T then
        case Tree
        of leaf then leaf
        [] question(1:Q true:TreeTrue false:TreeFalse unknown:TreeUnknown) then
            Condition = {Nth Reps Q}
            if Condition == 'unknown' then {GetTree TreeUnknown T}
            elseif Condition then {GetTree TreeTrue T}
```

```
            else {GetTree TreeFalse T}
            end
        end
    end
end
```

4.3 Incertitude du joueur (3*)

L'incertitude du joueur, ou la réponse 'unknown', permettant de continuer à avancer dans les questions lorsqu'on ne sait pas la réponse a su se greffer efficacement à notre implémentation de base. En effet, nous avons commencé par modifier la construction de notre arbre en ajoutant deux lignes, la première pour mettre à jour la liste de compteurs en supprimant le plus petit élément et la deuxième pour ajouter à la 'question' courrante un champs 'unknown' qui appelle récursivement la fonction `TreeBuilderAux`.

Ensuite, une fois que l'arbre comprend la branche 'unknown', il ne reste plus qu'à adapter la méthode `GameDriver` pour utiliser cette branche lorsque le joueur répond 'unknown'. Nous devons également ne pas oublier de faire attention à l'extension 'oops' qui doit être adaptée aussi (voir code 4.2) étant donné que la structure de l'arbre a changé.

5 Gestion du projet

5.1 Organisation

Nous avons globalement rencontré quelques difficultés à nous organiser au début mais nous avons fini par planifier différentes sessions de travail de façon à pouvoir terminer toutes les spécifications à temps. Nous n'avons pas mis en place une méthodologie de gestion de projet à proprement parler, nous avons simplement défini différentes deadlines à travers le temps indiquant des dates auxquelles nous devions avoir fini certaines parties.

5.2 Implémentation

L'implémentation a été une bonne expérience pour apprendre à créer un projet en utilisant uniquement le paradigme fonctionnel, on a pu se rendre compte que cela rendait le projet assez différent des projets en orienté objet que nous avons l'habitude de faire, en effet, même si cela devient de plus en plus naturel, utiliser la récursion terminale de façon quasi systématique est une chose que nous avons définitivement appris à faire durant ce projet.

Nous avons également essayé de structurer au maximum notre code en répartissant au mieux les tâches dans les différentes fonctions, cela nous a permis d'implémenter des extensions de façon très simple en n'ayant parfois à rajouter que quelques lignes de code afin de les faire fonctionner.

6 Conclusion

En conclusion, sur base de la librairie fournie nous avons su implémenter le jeu 'Qui est-ce ?' en n'utilisant que le paradigme fonctionnel en Oz. Notre jeu a une version de base qui permet de répondre à une suite de questions ainsi qu'une version plus élaborée qui permet ; d'avoir des personnages avec des questions manquantes, de revenir en arrière à une question si nous nous sommes trompés ou encore de répondre 'je ne sais pas' à une question. Finalement, ce projet nous a permis d'en apprendre plus sur Oz, tout en étant original, ce qui nous a permis de voir ce langage sous un autre angle moins formel.