Rapport de Projet de programmation fonctionnelle et de traduction des langages

Justin Jourdant, Issam Alouane

10 janvier 2022

Contents

1 Introduction

Ce projet consiste à etendre le compilateur du langage RAT réalisé en TP de traduction des langages pour traiter de nouvelles constructions : les pointeurs, le bloc else optionnel dans la conditionnelle, la conditionnelle sous la forme d'un opérateur ternaire, les boucles "loop" à la Rust.

Cette extension vise à rendre notre première version du compilateur plus complète et intéressante à utiliser en proposant des opérations plus poussées. De plus, l'apparition d'un nouveau type : les pointeurs va nous amener à devoir repenser certaines parties du code déjà écrit. Ce projet n'est donc pas seulement un ajout de portion de code par rapport aux TPs mais égalemment une restructuration du travail déjà fourni.

Nous avons divisé le travail de la manière suivante :
Justin JOURDANT : Les pointeurs
Issam ALOUANE : Les boucles "loop" à la Rust, le bloc else optionnel dans la conditionnelle, la conditionnelle sous la forme d'un opérateur ternaire

2 Traitement des Pointeurs

Tout d'abord, il a été nécessaire de modifier les fichiers lexer et parser pour pouvoir reconnaitre les différents éléments de code propre à l'usage des pointeurs qui ont été ajouté. C'est le cas des signes "&","new", et "null" et leurs tokens correpondant. Des règles de production propres aux pointeurs ont par ailleurs dû être ajouté dans le parser pour respecter la grammaire proposée. Cela se reflète notamment par l'ajout d'un non-terminal aff pour affectable et par l'ajout de règles dans expressions et d'une dans instruction. Ces modifications nous ont par conséquent mené à modifier les fichiers type.ml et type.mli pour ajouter le type "Pointeur of typ" aux types existants et mettre à jour certaines opérations en en tenant compte.

Avant de modifier les passes il a égalemment fallu modifier le fichier "ast.ml" pour expliciter l'évolution des règles ajoutés au fur et à mesure des passes. Pour se faire, nous nous sommes inspirés du travail fait précédemment (notamment en convertissant les string en info). Pour le non-terminal affectable, cela a demandé d'avantages de réflexion car il a fallu juger de son évolution à travers les passes et notamment de sa modification éventuelle pour la passe de placement. Nous avons jugé au départ qu'au vu de sa proximité avec les expressions, il était pertinant de suivre les décisions prises pour cette partie de code ce qui au final nous a guidé pour la suite de l'écriture. Null ne prenant pas d'arguments, nous ne l'avons pas modifié. Concerant new, nous avons considéré qu'il n'était pas nécessaire de créer une info pour conserver le type qui est le seul argument.

Pour les quatres passes, l'écriture du code s'est fait en s'inspirant des morceaux de codes déjà existant pour certaines parties (comme l'ident ou l'affecation classique). De même, new et null n'ont pas posés de problèmes particuliers, les instructions étant simple. Pour adresse et deref, certaines phases se sont avérés plus compliqué comme notamment la génération de code ou la phase de typage.

Cette partie nous a posé quelques difficultés car l'ajout d'un nouveau type et la passe de génération de code se sont révélés complexes.

NB: à la fin du projet, cette partie ne fonctionnent pas, alors on avait décidé de la laisser en commentaires pour que les autres fonctionnalités puissent marcher.

3 Traitement du bloc else optionnel

Cette partie n'était qu'une extension du bloc conditionnel. Il a été traité de la même manière. D'abord on a ajouté la règle suivante dans le parser (pour les instructions) : IF exp=e li=bloc {ConditionnelleOptionnel (exp,li)}, qui explicite la syntaxe : if (expression) { bloc }. Ensuite, on a ajouté dans tous les ast l'instruction : ConditionnelleOptionnel of expression * bloc. Enfin, c'était le même traitement (à peu près) de la conditionnel normale, sauf que le deuxième bloc n'existe pas.

Nous n'avons pas rencontré de difficulté particulière dans cette partie puisque nous pouvions nous s'inspirer du travail réalisé sur la conditonnelle normale.

4 Traitement de la conditionnelle ternaire

Pour implémenter la conditionnelle sous la forme d'un opérateur ternaire, on a commencé par ajouter dans le parser la règle (pour les expressions) : PO e1=e PI e2=e DP e3 =e PF {Conditionnelleternaire(e1,e2,e3)}, en définissant les tokens DP (":") et PI ("?") et en l'ajoutant dans le lexer. Ainsi, dans tous les ast, on a ajouté l'expression : Conditionnelleternaire of expression * expression * expression. Dans le passeTdsRat, on a pas fait de traitement particulier, mais dans le passeTypeRat, il fallait se mettre sûr que le type de la première expression est bool, et que les deux expression de retour ont le même type, sinon en lève des exceptions TypeInattendue. Enfin, dans la passecodeRatToTam, le traitement était semblable au traitement d'une conditionnel normale.

Nous n'avons pas rencontré de difficulté particulière dans cette partie.

5 Traitement de la Loop à la Rust

D'abord, on a décidé qu'on aura besoin de 6 nouveaux types :
- Loop : Pour les loop sans nom
- LoopNom(nom) : Pour les loop avec nom
- Break : Pour la break sans nom
- BreakNom(nom) : Pour le break avec nom de loop à stopper
- Continue : Pour continuer une loop sans nom
- ContinueNom(nom) : Pour continuer une loop qui a un nom

On a ajouté les règles qui définissent ces types dans le parser et le lexer en introduisant les 2 nouveaux tokens : continue et break. Ensuite, on ajouté ces types dans les ast, tel que le paramètre du loop est son bloc, et les paramètres du LoopNom sont son nom et son bloc, Break et Continue sont sans paramètre, et le paramètre des BreakNom et ContinueNom est le string qui indique le nom de la loop concerné.
Ainsi, on a ajouté dans le type Info (dans tds.mli), la règle InfoLoop of string, alors on peut enregistrer les loop dans notre tds.
Pour la passeTdsRat, il n'y avait pas de traitement particulier pour le les Loop, break et continue sans nom. Mais, pour les LoopNom, Il faut chercher globalement l'existence d'une loop avec le même nom, s'elle existe, on print un warning et on complète notre traitement avec les 2 loop à la fois (comme si la première n'existe pas), sinon, on crée une TDSfille, où on ajoute notre loop et on analyse le bloc avec TDSfille comme paramètre, car si on crée pas une TDSfille, si on a une loop l qui se termine et une autre loop l qui commencent après la fin de la première, le compilateur vas afficher un warning même qu'ils ne sont pas imbriquée. Pour le BreakNom et le LoopNom, il faut chercher Globalement l'existence d'une loop avec les noms fournis, s'elle n'existe pas on lève l'exception : MauvaiseUtilisationIdentifiant.

Pour le passeTypeRat et le passePlacementRat, il n'y avait pas de traitement particulier.

Pour le passeCodeRatToTam, on avait l'obligation d'ajouter des fonctions analyse_code_bloc_loop et analyse_code_instruction_loop. Ces fonctions sont appelés pour l'analyse des blocs et des instructions des loop. Elles prennent un paramètre additionnel, une liste des triplets, chaque triplet contient un le nom de la boucle, l'étiquette de début, et l'étiquette de fin. Les Loop sans nom sont enregistrées avec le nom "". Les break et les continue appelent les fonctions auxiliaires finir et continuer qui cherchent le triplet avec le nom du loop fourni et renvoie l'étiquette voulue, pour les loop sans nom on fournit "" comme paramètre.

La grande difficulté dans cette partie était de trouver comment enregistrer une loop avec le même nom qu'une variable ou une fonction. Pour résoudre ce problème, nous avons décidé d'ajouter une apostrophe au début de nom du Loop et qu'on l'enregistre avec ce nom dans la tds, tant qu'on peut pas enregistrer des variables ou des fonctions avec une apostrophe au début.

On avait aussi écrit quelques fichiers de tests en s'inspirant des programmes fourni dans l'énoncé.

6 Conclusion

Pour conclure, l'extension du compilateur du langage RAT réalisé en TP de traduction des langages nous permet de traiter des programmes plus complexes. Il nous a permis de nous intéresser et de comprendre plus en profondeur des portions de code qui n'avait pas spécialement été développer en TP (comme les fichier lexer, parser ou ast).

Les difficultés rencontrées ont été abordées plus spécifiquement pour chaque partie mais globalement la passe de génération de code a propbablement été la plus grosse difficuté à laquelle nous ayons été confrontés. Le nombre important d'instruction utilisable et la grande spécificité de la génération rendent l'écriture du code par comparaison plus complexe.

Pour améliorer le projet, une bonne solution pourrait consister à mener d'avantages de tests sur des cas complexes (pointeurs imbriqués, composition d'instructions...) pour tester les limites du code écrit et corriger les erreurs éventuels qui pourraient survenir.