Rapport Compilation

Navarro Robin, Vergé Romain, Boitelle Adrien et Blanchet Yann

Mai 2019

Contents

1 Introduction

Dans ce projet, nous devons analyser un langage impératif, dans le but de produire le code intermédiaire à trois addresses.

2 Plan du projet

Dans un premier temps, nous avons commencé par mettre au point un Scanner permettant d'analyser tous les caractères possibles du language LEA, et renvoyant les Token correspondants.

Ensuite, nous avons du compléter la grammaire fournie dans l'archive originelle du projet, recevant les Token du scanner afin de réaliser l'analyse sémantique. La grammaire devra également nous permettre de trouver les erreurs sémantiques au cours de l'analyse de l'entrée.

Une fois ces deux étapes complétées, nous avons utilisé chaque classe Java du projet, afin de générer un arbre sémantique correct.

Pour continuer, nous avons du écrire les fonctions "generateIntermediateCode()" de chaque classe Node afin de pouvoir générer un arbre de code intermédiaire.

Enfin, dans un soucis de perfection, nous avons décidé de réutiliser la fonction "toDot()" fourni dans le projet afin de pouvoir générer un visuel de l'arbre de code intermédiaire ainsi construit.

3 Problèmes rencontrés

Nous avons eu quelques problèmes avec le Scanner. Le premier était du au "moins unaire", c'est-à-dire la gestion des nombres négatifs. Nous avons rencontré le même problème pour le NOT.

Un autre problème rencontré, cette fois d'ordre organisationnel, était de coder à plusieurs sur un seul fichier (exemple: Parser). Coder à plusieurs sur un seul et même fichier cause des incompréhensions ainsi que des "merge" que nous avons du gérer.

Un des plus gros problèmes que nous avons pu rencontrer concerne le choix d'implémentation de l'arbre du code intermédiaire.
Dans un premier temps, nous avions choisi d'ajouter une classe NodeStm similaire à la classe NodeExp. Nous stockions le code généré dans le champ Stm de NodeStm ou dans le champ Exp de NodeExp.
Avec cette implémentation, nous avons été freiné par le stockage des ExpList et

StmList qui n'héritaient pas de NodeExp et NodeStm.

Ensuite, nous avons choisi de renvoyer directement le code intermédiaire généré sans utiliser de listes mais imbrication de Seq.
Dans cette solution, nous avons été bloqués à cause d'un problème de typage dans NodeList (On ne peut mettre des Exp dans Stm).

Après discussion avec d'autres groupes, nous avons décidé de stocker des StmList et ExpList directement dans NodeStm et NodeExp.
Encore une fois, cela a généré des problèmes (accès en continu à l'élément 0) et par soucis de propreté, nous avons préféré, après réflexion, repartir sur une implémentation propre et renvoyant directement le code intermédiaire.

Un autre problême que nous avons rencontré est l'implémentation des structures un peu plus complexes, telles que NodeIf, NodeWhile.

4 Solutions apportées

Afin de résoudre le problème généré par les nombres négatifs, nous avons eu recours à l'utilisation de unary minus avec beaver.
Pour simplifier la gestion du moins unaire, nous n'avons pas utilisé le constructeur de NodeOp ne prenant qu'un seul argument.

Par exemple, pour -5, nous faisons en réalité 0 - 5.

Concernant le problème causé par le "unary NOT", nous avons également trouvé une autre manière de l'écrire, pour pouvoir utiliser le constructeur de NodeRel à deux arguments.
Nous voyons en effet le NOT comme une égalité avec faux.

Ces deux solutions nous permettent également de simplifier la génération de code intermédiaire à trois addresses.

Afin de résoudre le problème du codage simultané, nous avons utilisé un plugin de Visual Studio Code permettant de coder à plusieurs en direct sur un PC hôte (LiveShare)

Concernant le problème du choix de l'implémentation de la génération de l'arbre du code intermédiaire, nous avons choisi de renvoyer récursivement le code généré sous forme de Seq.
Le problème principal de ce choix était la présence d'expression dans la génération récursive de list sous forme de Seq.
Nous avons réglé ce problème en retournant, dans les expressions pouvant apparaître dans NodeList, un code intermédiaire héritant de Stm. Nous stockons tout de même l'expression dans l'attribut exp de nodeExp.

Ainsi, lorsque nous avons un élément dont nous sommes sûrs qu'il est une expression, nous pouvons utiliser la méthode getExp() pour accéder à l'expression et non au Stm.

Après réflexion, nous aurions sûrement pu utiliser Eseq pour résoudre de manière plus correcte ce problème.

Pour l'implémentation des structures plus complexes, nous nous sommes fortement inspirés de la page 150 de "modern compiler implemetation in Java". Il était ensuite assez simple d'adapter l'arbre pour générer les différentes structures.

5 Pistes d'améliorations

Nous pouvons tout d'abord améliorer notre projet en poussant plus loin la gestion des erreurs dans la grammaire.
En effet, nous effectuons une vérification des erreurs dans notre Parser, mais nous ne vérifions pas les erreurs liées à l'usage des pointeurs.

Nous pourrions améliorer la lisibilité du code en ajoutant une interface "Dotable", nous permettant de produire le graphe pour tout objet l'implémentant.
Nous pouvons également améliorer l'affichage de l'arbre du code intermédiaire, car celui-ci est "à l'envers".
Voici, par exemple, l'arbre de code intermédiaire que nous obtenons pour un programme assez simple contenant un "if/then/else".

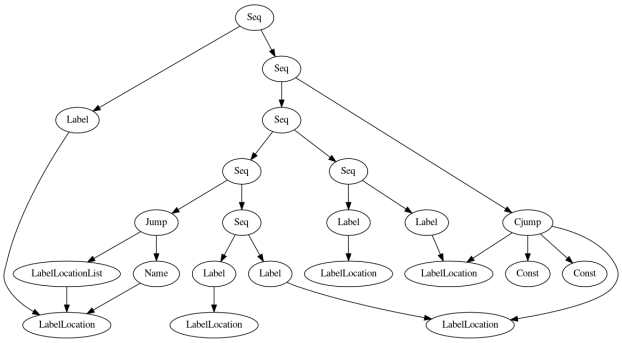

Figure 1: figure
Seq(Seq(CJump(Const(0) 2 Const(1), L0, L1), Seq(Seq(Seq(Label(L0), Label(print)), Jump(Name(L2), LabelLocationList(L2))), Seq(Label(L1), Label(print)))), Label(L2))

6 Conclusion

Ce projet nous a permis de mettre en application les différentes connaissances acquises durant ce dernier semestre de licence.

De plus, cela nous a permis d'apprendre à comprendre et analyser un code fourni et le compléter.

Ce projet nous a aussi appris à persévérer et à garder la motivation malgré les premières difficultés de compréhension.