Rapport d'activité 1

TELECOM Nancy
Ingénieurs du numérique • Inspiring your digital future

Rapport d'activité 1
PCL1

Professeurs référents : S. COLLIN et S. DA SILVA

TELECOM NANCY
Projet de Compilation
Deuxième année (S7)

Semaine 46
Dernière mise à jour : le 18 novembre 2022

Membres du groupe :

Nicolas FRACHE
Théo GOUREAU
Cyrielle LACRAMPE–DITER
Rida MOUSSAOUI

TELECOM Nancy
193, Avenue Paul Muller
54600 Villers-lès-Nancy
+33 (0)3 72 74 59 00

Table des matières

1 Sommaire du travail réalisé

Pendant cette période du temps, les membres du groupe ont réalisé quatre réunions dont l'objectif principal est la réalisation d'une grammaire fonctionnelle qui respecte le manuel de référence.

- Les membres se sont réunis pour se mettre d'accord sur la méthode de travail et l'organisation des tâches. Il a été décidé de se servir de Trello en tant qu'outils de gestion de projet ainsi que Discord pour la communication entre les membres
- Discussion autour de la grammaire : lecture du manuel de référence et réflexion autour de la grammaire (lexique, syntaxe et sémantique)
- Répartition des parties de la grammaire pour l'implémentation
- Test premier de la grammaire sur le programme d'exemple existant dans le document de référence en utilisant l'outil ANTLR
- Rédaction des fichiers de tests supplémentaires de la grammaire (non correct syntaxiquement, correct syntaxiquement mais pas sémantiquement)
- Utilisation de la librairie JUnit pour mettre en place les tests unitaires de la grammaire (cf. 4)
- Adaptation de la grammaire suite au mail envoyé par le professeur référent : changement de règle lexer pour les chaînes de caractères
- Utilisation de Maven pour l'exécution des tests unitaires en commandes de ligne. Suite aux problèmes liés aux configuration de la librairie JUnit, il a été choisi d'utiliser un outil supplémentaire, Maven qui a été facile à utiliser sur les différents environnements (cf. 4)
- Travail sur la grammaire pour la rendre LL(1) (cf. 3.1)

2 Difficultés rencontrées

Durant le projet, les membres du projets ont été confrontés les difficultés suivantes :

- Avoir de la difficulté pour bien comprendre la syntaxe et la sémantique du langage défini par la grammaire à partir du manuel de référence puisqu'il n'est pas semblable à un langage maîtrisé par les membres du groupe
- Rendre la grammaire LL(1) n'est pas une tâche évidente : il a fallu recommencer plusieurs fois (cf. 3.1)
- Rencontrer des problèmes liés à la configuration des IDE pour mettre en place les tests unitaires
- Un retard relatif pour quelques tâches dû au manque du temps au delà de celui dédié au projet ainsi que la coïncidence avec les périodes des examens

3 Grammaire

3.1 Construction de la grammaire

Départ Notre grammaire part de la grammaire du document *Tiger Language Specification*. Celle-ci contenant des étoiles de Kleene, étoiles positives et étoiles de Kleene avec des séparateurs (point-virgule, virgule) dans certaines productions, nous avons créé des règles pour les gérer. Cette grammaire n'est pas LL(1) et ne gère ni les opérateurs ni le fait que le *else* d'un *if then else* est optionnel.

LL(1) Nous nous sommes ensuite chargés de la factoriser et de supprimer les récursivités gauches afin de la rendre LL(1).

Les opérateurs Nous avons travaillé sur les opérateurs (associativité, priorité, etc.). La grammaire gère désormais les opérateurs et est presque LL(1) (il nous reste quelques exceptions que nous pensons pouvoir régler dans la semaine).

Le *else* Nous avons ajouté le fait que le *else* n'est pas obligatoire. Notre grammaire n'est donc plus LL(1). Nous indiquons donc à ANTLR qu'en cas de lecture d'un *else*, il faut suivre la règle $ElseOpt \rightarrow else\ expr$ plutôt que $ElseOpt \rightarrow \varepsilon$ en jouant sur la priorité.

3.2 Structure de la grammaire

La grammaire est strucutrée de la manière suivante :

Axiome :
$program \rightarrow expr$

Expression :
$exp \rightarrow expNoBinOpNoId\ binOp \mid loop \mid ifThen \mid negation \mid id\ idFacto$
expNoBinOpNoId est comme *exp* mais ne commence pas par un identifiant et ne finit par par un *BinOp* (opérateurs binaires).

Boucles (loop) Les boucles ne posent pas de problème pour la propriété LL(1) de la grammaire :
$loop \rightarrow whileExp \mid forExp$
$whileExp \rightarrow while\ exp\ do\ exp$
$forExp \rightarrow for\ id\ :=\ exp\ to\ exp\ do\ exp$

Conditions (*if then else*) :
$ifThen \rightarrow if\ expNoBinOp\ binOp\ then\ exp\ optElse$
$optElse \rightarrow else\ exp \mid \varepsilon$
Cette règle pose problème et n'est pas LL(1). Cependant, nous indiquons à ANTLR une règle de priorité (*else* est prioritaire sur ε, i.e. un *else* est associé au *if then* le plus proche) pour régler le conflit.

Négation (-) :
$negation \rightarrow -\ exp$

Ce qui commence par un identifiant Les identifiants d'objets et de types étant régis par la même règle lexicale, il est impossible de les distinguer. C'est pourquoi nous avons regroupé toutes les règles qui commencent par un identifiant ($exp \rightarrow id\ idFacto$) :
$idFacto \rightarrow binOp \mid refOrCreate$
refOrCreate gère ensuite les identifaints seuls ($refOrCreate \rightarrow \varepsilon$), les *lValues* (éléments de tableaux, attributs d'un enregistrement, etc.), les appels de fonctions, les assignements et les créations de tableaux ou d'enregistrement (qui commencent par un identifiant de type).
binOp gère les opérateurs binaires.

Etoiles Les étoiles sont gérées par des règles comme celle ci-dessous :
$expCKleeneFacto \rightarrow \varepsilon \mid exp\ expCKleeneFacto$
Ici il s'agit d'une étoile de Kleene d'expressions séparées par une virgule.

4 Maven et mise en place des tests

Contexte Nous nous sommes rapidement rendus compte que, chaque personne ayant une configuration différente sur son ordinateur, il est difficile mettre en place un projet Java ayant des dépendances externes sans perdre beaucoup de temps pour la configuration.

Solution Nous avons reconstruit le projet avec Maven. Cela permet de gérer les dépendances automatiquement. De plus l'utilisation du plugin Maven *antlr4-maven-plugin* permet d'automatiser la création du Parser et du Lexer lors de la compilation du projet. Enfin, un autre plugin nous permet de créer un jar contenant notre compilateur et fonctionnant indépendamment.

Tests unitaire Nous avons mis en place des tests unitaires gérés par Junit. Mais nous nous sommes rendus compte que lors de l'analyse, en cas d'erreurs, ANTLR se contente d'afficher un message dans la console. Pour utiliser ces informations nous avons donc surchargé la gestion des erreurs par défaut d'ANTLR pour contrôler le comportement.