Rapport du projet de compilation 2019

Compilateur LÉA

Etudiants :
Vincent Ameeuw
Louis-Gabriel Barrère
Maxime Capdordy
Marc Cerutti

Enseignants :
Lionel Clément

3 mai 2019

Table des matières

Première partie

Projet

1 Plan du programme

```
ProjectCompilation2019
.
|-- 5-LR.pdf
|-- AUTHORS
|-- build.xml
|-- cmdHierachie
|-- documentation.pdf
|-- hierarchie.txt
|-- LICENSE
|-- data/
|-- lib/
|-- parser/
|-- scanner/
|-- src/
    |-- environment/
    |-- errors/
    |-- intermediateCode/
    |-- main/
    |-- node/
    |-- type/
```

> Les fichiers à la racine du projet servent à se documenter, se renseigner sur le projet et lancer un script ou une compilation.
> data/ contient les fichiers de test à lire pour vérifier l'attitude du compilateur dans le fichier résultant output
> lib/ contient les bibliothèques nécessaires au bon fonctionnement de Beaver et du compilateur
> parser/ contient la grammaire qu'est censé reconnaître Léa
> scanner/ contient le Lexer qui doit analyser et retranscrire ce qui est lu dans n'importe quel code fait en Léa
> src/ contient tous les fichiers .java qui représentent le coeur du langage Léa :
- la gestion d'environnement
- la gestion d'erreurs
- la génération du code intermédiaire
- les "fichiers de départ" qui permettent de lancer le projet
- les fichiers Node (classe parente) qui compose notre arbre abstrait
- les fichiers Type qui permettent de définir précisément les types de toutes les variables de Léa

2 Difficultés rencontrées

> La première difficulté était de se rendre compte, dans la grammaire fournie, qu'une ambiguïté était déjà présente dans le cas d'un if then else : la règle étant ambiguë, il était possible qu'une partie de code ne soit pas reconnue par la grammaire.
> La documentation pauvre de Beaver et des fonctionnalités obscures au premier abord comme toDot, attestWellFormed, etc. n'était pas explicitées dans le code de départ.
> Le code de départ incomplet, mais sans indication de où.
> Pour s'adapter au langage Léa, l'analyseur lexical a dû être pensé pour prendre tous les cas d'écritures possibles. Le plus dur ici était d'imaginer avec les quelques exemples fournis comment tout prendre en compte, mais surtout de traiter les cas qui n'apparaissaient pas dans les exemples.
> Lors de la création d'environnement pour la pile de variables, une difficulté a été de rajouter du code avant les règles. D'ailleurs, du fait de Beaver, on ne peut toujours pas à l'heure actuelle extraire des informations d'une règle avec une règle vide (comme l'environnement Main) qui nécessite des informations d'un contexte plus élevé.
> La lecture avec le système de base était insuffisante voire inutile, comme on ne pouvait pas voir quelle ligne du fichier dans les inputs était en cause. > Lorsque nous avons voulu implémenter l'affichage des lignes/colonnes en cas d'erreur de compilation, nous avons du changer entièrement le système de Node de départ pour y gérer l'implémentation des lignes et colonnes de la classe parente **Symbol** de Beaver. Cela a changé des constructeurs, des types dans le .grammar, et de ce fait, on a du corriger environ 40 erreurs une par une, changeant la quasi-totalité du programme dans certains dossiers.
> La gestion d'erreur a été complexe à aborder à cause du grand nombre de cas à traiter. Plus la gestion d'erreurs avançait, plus il était dur d'implémenter le traitement des suivants, du fait des cas particuliers.

3 Solutions apportées

3.1 Analyseur lexical : Jflex

La stratégie adoptée pour compléter le .jflex (à ce moment, la grammaire n'était pas encore présente) était de regarder tous les mots utilisés dans les exemples et de leur donner un sens dans le .jflex. Ils ont tous été classés, puis associés à leurs TOKEN respectifs. Les commentaires ont été implémenté facilement : le commentaire court est composé de "//" et d'une suite de n'importe quels caractères jusqu'au retour chariot et le long est une suite de n'importe quels caractères entre "/*" et "*/".

3.2 Analyse syntaxique : Beaver

Pour constituer toute la grammaire de Léa, la tactique était de partir du bas du fichier .grammar (contenant la grammaire de Léa à implémenter) et de remonter dans le but de traiter les cas les plus simples, de comprendre "l'arbre" en le remontant. Le premier ajout fut la règle de grammaire qui permettait de reconnaître et différencier les "if then" et "if then else", l'ambiguïté a donc été levée en ajoutant une priorité.

3.3 Environnements

Les environnements ont été implémenté à base de **HashMap** de **NodeId**. Ainsi il était très facile de vérifier si un identifiant (variable, itemEnum, fonction/procédure, type) n'existait pas ou existait déjà. Un cas particulier fut la manipulation des fonctions lors du remplacement d'un noeud de fonction non défini avec sa définition, et de vérifier si elle était déjà définie pour le faire. La manipulation des variables se fait grâce à une pile de ces environnements, qui sont créés durant l'analyse.
Il y a donc trois environnement principaux : **typeEnvironment**, **procedureEnvironment** et **stackEnvironment**.

3.4 Gestion des erreurs

Ce qui a nécessité le plus d'effort fut la réadaptation du code de départ qui ne prenait pas en compte les lignes et les colonnes du fichier d'analyse. Ainsi tous les constructeurs de classe qui dérivaient de **ClonableSymbol**, et donc de **Symbol** de Beaver, ont été changés une fois que l'arbre abstrait avait déjà été complété dans le .grammar. Lorsqu'il a fallu commencer à vérifier avec les inputs, la lecture des erreurs était compliqué, imprécise et vide de sens.
On a donc, grâce à une classe d'erreurs personnalisée et les modifications des **Nodes**, traqué avec le système d'exception de java là où se déroulait l'erreur dans le fichier d'input. Les tests en ont été grandement facilité, et cela a permis

d'améliorer la vérification des types, beaucoup plus claire avec la location de l'erreur.

3.5 Code intermédiaire

Même si la théorie du code intermédiaire nous paraissait évident, il nous est apparu impossible de pouvoir terminer le code intermédiaire dans les temps. Aucune documentation ou explication de comment le construire n'a pu nous aider dans cette tâche, nous avons donc dû improviser et tout de même tenter quelque chose, au moins sur certaines classes.

4 Tests effectués

En compilant l'intégralité du projet, nous avons testé chaque fichier exemple dans l'ordre en consultant l'output. L'output nous a permit d'avoir d'abord des informations sur la grammaire et de corriger les ambiguïtés apparentes. Puis nous avons vérifiés comment chaque fichier d'entrée était lu, s'ils donnaient le résultat attendu et si des erreurs apparaissaient au moment où nous nous y attendions.