Haute école du paysage, d'ingénierie et d'architecture

Techniques de compilation

Compilateur Hepial

B. Coudray, Q. Berthet

2019-2020

Table des matières

1 Introduction

Dans le cadre du cours de techniques de compilation, pour mettre en pratique les notions étudiées en cours, nous avons été amener a créer un compilateur pour le langage Hepial. La réalisation d'un compilateur nécessite plusieurs étapes :

— l'analyse lexicale
— l'analyse syntaxique
— l'analyse sémantique
— l'optimisation/génération du code

Après discussion avec le professeur, l'optimisation du code a été rendu optionnelle. Enfin, le compilateur Hepial s'occupe de générer du code intermédiaire (*bytecode*) Java via le code Hepial écrit par un programmeur. Ce code intermédiaire est ensuite compiler en Java.

1.1 Le langage Hepial

Le langage Hepial est un langage de programmation simple créé dans le but de comprendre la chaîne de compilation d'un langage.

1.1.1 Exemple de programme

```
1  programme Demo2
2      entier x;
3      entier i;
4      constante booleen afficher = vrai;
5  debutprg
6      si afficher alors
7          ecrire "Hello World";
8      finsi
9      pour i allantde 0 a 5 faire
10         x = i*i;
11         ecrire x;
12     finpour
13 finprg
```

FIGURE 1 – Exemple de programme Hepial

Ce programme Hepial affiche *"Hello World"* dans la console, calcule $i * i$ et affiche le résultat dans la console, pendant six itérations.

2 Analyse lexicale

L'analyse lexicale est faite via le logiciel JFlex et le fichier "Scanner.flex" qui contient cette analyse nous a été donnée par le professeur dans le but d'avoir une base pour commencer le projet. Cette partie s'occupe de définir la liste des symboles qui contient :

— les mots-clefs du langage
— les identifiants
— la ponctuation
— les opérateurs arithmétiques

— les littéraux

Cette liste de symboles est ensuite utilisée pour l'analyse syntaxique.

3 Analyse syntaxique

L'analyse syntaxique vérifie que les règles grammaticales qu'on a définit sont respectées par le programmeur en analysant le programme qui l'a écrit. Cette étape est effectuée par le logiciel Cup [1] et on définit les règles grammaticales dans le fichier "Parser.cup".
Le point important dans cette partie est de créer l'arbre de syntaxe abstrait permettant ensuite de faire l'analyse sémantique et la génération du code. En nous basant sur la structure du code déjà fournie, nous avons complété ce qu'il manquait et nous obtenons une vue d'ensemble de l'arbre donnant :

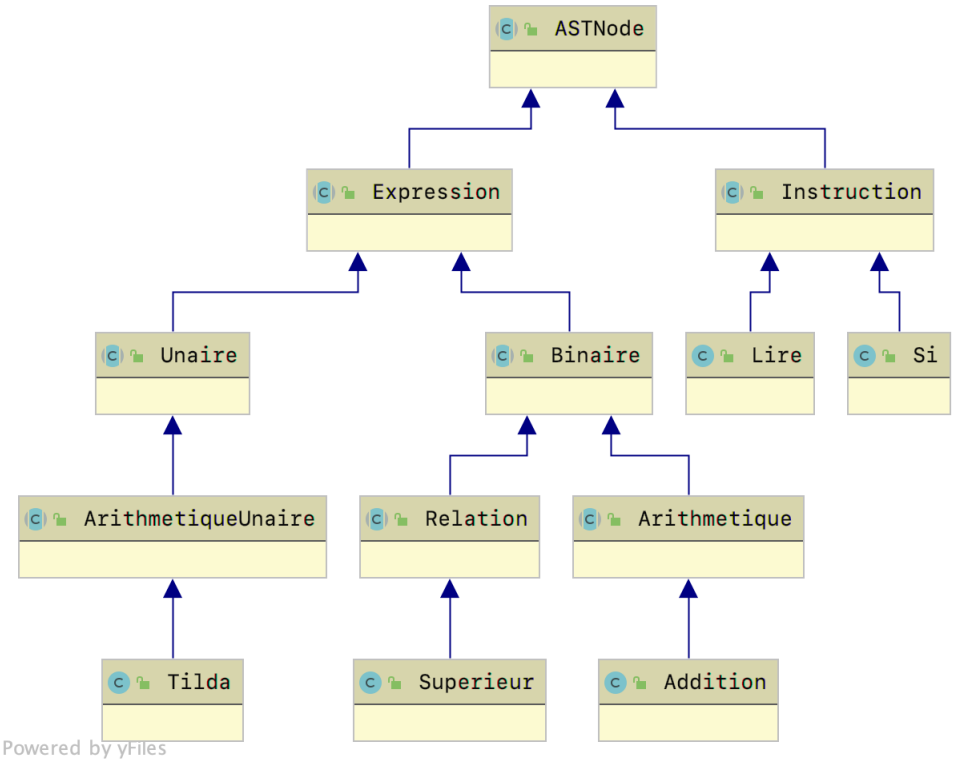

FIGURE 2 – Vue d'ensemble de l'arbre syntaxique

L'arbre est structurée de façon à pouvoir comprendre facilement dans quelle catégorie une expression ou une instruction se trouve. On peut donc voir qu'une expression est divisée en deux catégories, les expressions unaires et les expressions binaires. Dans la première catégorie, on retrouve l'arithmétique unaire qui correspond à l'opération non bit à bit (Tilda) et la négation (Non). Dans la seconde catégorie, on a de nouveau deux sous catégories. La catégorie "relation" comprends les opérateurs de comparaison comme par exemple le supérieur égal, supérieur, et logique, ou logique etc. Dans la catégorie arithmétique binaire, il y a l'addition, la soustraction, division et la multiplication. Enfin, la catégorie instruction comprend la boucle pour, tantque, la lecture/écriture depuis/dans le terminal, la structure conditionnelle etc.

1. *Construction of Useful Parsers*

Ainsi, en implémentant cette structure de données, il devient très facile d'ajouter des opérations arithmétiques unaires/binaires, des instructions ou autre. Par exemple, pour ajouter l'opération "modulo", il suffit de créer une classe "Modulo" qui hérite de la classe "Arithmetique" et qui implémente les méthodes définissant le comportement de l'opération. Finalement, la définition du modulo dans le fichier JFLex ("Scanner.flex") pour l'analyse lexicale et dans le fichier Cup ("Parser.cup") pour cette analyse est nécessaire pour prendre en compte cette nouvelle opération.

4 Analyse sémantique

Durant l'analyse sémantique nous effectuons le contrôle des types de chaque nœud (soit booléen, soit entier) en utilisant le patron de conception Visiteur. Grâce à lui, cette analyse se fait très facilement en parcourant les nœuds de l'arbre syntaxique. A l'aide de la classe AnalyseurSemantique, nous vérifions pour chaque nœud le type des opérandes pour que par exemple, une addition s'opère que entre deux entiers et non entre un entier et un booléen. En cas d'erreur, ce dernier nous retournera une erreur et terminera le processus de compilation.

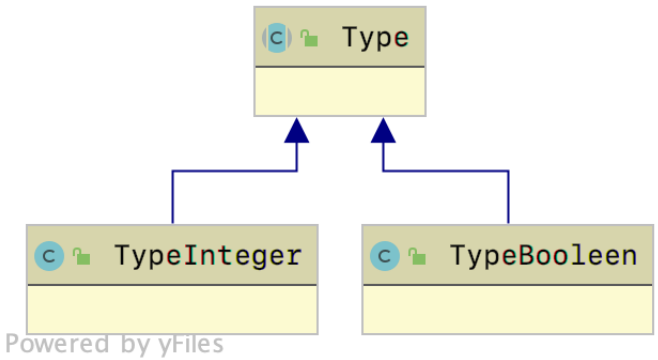

FIGURE 3 – Caption

De plus, il est possible de créer de nouveau type en héritant de la classe abstraite "Type" que nous avons créée et utilisée pour le type entier et booléen. Le nouveau type pourra être utilisé dans l'analyse sémantique pour vérifier que les expressions/instructions voulues correspondent au nouveau type. Il faut évidemment définir le symbole du type dans le fichier "Scanner.flex" et ajouter le type de base dans le fichier "Parser.cup". Finalement, lorsque cette étape est franchie, le code fourni par le programmeur peut être généré.

5 Génération du code

La génération du code se fait via la classe *CodeProduction* qui hérite de la classe *ASTVistor*. En effet, comme pour l'analyse sémantique, celle-ci visite chaque nœud de l'arbre syntaxique mais cette fois-ci génère le *bytecode* Java correspondant au code Hepial écrit par le programmeur. Le *bytecode* Java généré est écrit dans un fichier texte qui sera ensuite compilé en Java via le logiciel Jasmin.

5.1 Exemple

```
1 programme Demo3
2 debutprg
3     ecrire "Hello World";
4 finprg
```

FIGURE 4 – Le fameux programme "Hello World!" de "demo3.hepial"

Pour comprendre, le processus de génération du code, nous avons programmé le fameux *"Hello World!"* en Hepial. Après la génération du code, nous pouvons voir à la

```
1  .class public Demo3
2  .super java/lang/Object
3  .method public static main([Ljava/lang/String;)V
4  .limit stack 20000
5  .limit locals 10000
6  getstatic java/lang/System/out Ljava/io/PrintStream;
7  ldc "Hello World !"
8  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
9  return
10 .end method
```

FIGURE 5 – Génération de "demo3.j" via la classe *CodeProduction*

ligne 1 que le nom du programme Hepial correspond au nom de classe Java qui sera générée. Puis, nous créons la méthode *Main* qui comprendra le code Hepial définit par le programmeur. A la ligne 7, nous mettons sur la pile la constante *"Hello World!"* qui sera ensuite dépilée par la méthode *println* de Java pour afficher le texte dans la console.

```
1 public class Demo3 {
2     public static void main(String[] var0) {
3         System.out.println("Hello World !");
4     }
5 }
```

FIGURE 6 – Décompilation du fichier "Demo3.class"

Enfin, après la compilation du *bytecode* via Jasmin, nous pouvons décompiler le fichier .class pour lire le code Java correspondant au fichier généré. Nous pouvons voir que le code Java correspond bien au code Hepial écrit précédemment.

6 Conclusion

Finalement, bien que ce travail pratique soit difficile à prendre en main, il nous a permis de mieux comprendre le chaînage des analyses pour arriver jusqu'à la génération du code. Ainsi, nous avons réussi à créer l'arbre syntaxique via Cup puis à exploiter cet arbre pour faire l'analyse sémantique, produire le code et le compiler en Java via Jasmin pour qu'il puisse être exécuter dans une JVM [2].

2. *Java Virtual Machine*

Table des figures