Technique de compilation - HEPIAL

El Kharroubi Michaël - Fleury Noé

Sommaire

Introduction

Ce rapport a comme utilité d'expliquer le projet *HEPIAL*, un compilateur simpliste.

Tout d'abord, *HEPIAL*, qu'est ce que c'est ? C'est un language de programmation qui se résume à quelques instructions seulement, mais assez pour créer un grand nombre de programmes.

Le but de ce projet a été de se familiariser avec la façon dont un compilateur (tel que GCC, JAVAC, etc) fonctionne, et de créer notre propre version afin de pouvoir compiler du *HEPIAL*.

Procédures

JFlex

JFlex est un générateur d'analyse lexical, en d'autres termes c'est un programme permettant de passer sur un fichier texte et d'en analyser le contenu en fonction de certains patterns ou "regex". En fait, c'est là qu'on définit explicitement ce que l'on souhaite considérer comme un nombre, une chaîne de caractère, un signe +, etc. Cela permet de n'accepter que des mots clés du langage. Une fois que l'on à géré tous nos symboles, on indique à JFlex qu'il doit connecter sa sortie pour être compatible avec CUP.

CUP

CUP (Construction of Useful Parsers) est un logiciel qui permet de parser la sortie de JFlex, donc d'interpréter les différents symboles qu'on lui passe. Le but ici est de faire une analyse syntaxique, c'est à dire qu'on va vérifier que notre fichier à compiler est bien écrit, et respecte la grammaire imposée. Plus précisément, on va vérifier que chaque mot clé du langage est utilisé quand il le faut, et où il faut. Prenons par exemple une condition, elle devra contenir une condition, des instructions à effectuer dans le cas où cette dernière est validée, et optionnellement des instructions dans le cas où elle ne le serait pas.

À l'aide de CUP, notre but est de créer un arbre abstrait qui va représente le code à compiler.

Mise en place d'un arbre abstrait à l'aide du polymorphisme

Comment pouvons nous représenter un code sous forme d'arbre? En fait, la technique utilisée est d'opter pour un arbre abstrait. Chaque élément de l'arbre est abstrait, sauf ses feuilles. Chaque nœud va donc hériter de son élément parent, jusqu'à remonter à la racine qui est fait notre programme.

Étant donné que JFlex procède à une analyse dite gloutonne, il va automatiquement englober le plus gros élément reconnu. De plus, à l'aide de CUP, dès la détection d'un potentielle objet, par exemple une suite d'addition, tel que `n = 1 + 2 + 3;`, cela aura comme conséquence : premièrement d'englober `2+3` dans une expression, puis créer une seconde expression `1 + expression(2+3)`, c'est à dire une expression qui possède un entier comme opérande de gauche, et une expression comme opérande de droite. Par ce biais, lorsqu'on va parcourir notre arbre, on va pouvoir procéder indépendamment au traitement de chacun de nos objets, afin de créer le code assembleur Jasmin correspondant.

Pour donner un exemple de la création de cet arbre, voici le code *HEPIAL* suivant :

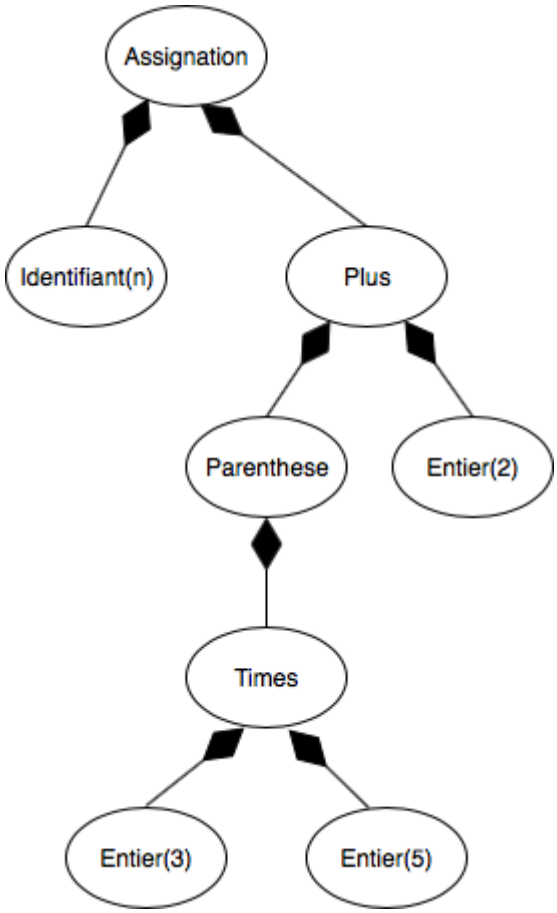
```
programme ProgrammeHepial

entier n;

debutprg

  n = (3 * 5) + 2;

  si n == 17 alors
    ecrire "Calcul OK !";
  sinon
    ecrire  "Calcul NOK !";
  finsi

finprg
```

Dans ce code, on commence par déclarer notre variable `n` comme étant un entier. Ensuite, on assigne à la variable n la valeur `(3 * 5) + 2`. Si la valeur de la variable est bel et bien `17`, on affiche que le calcul est juste, sinon qu'il est faux. Nous allons étudier comment est traité ce code par notre compilateur.

Tout d'abord, lorsqu'on déclare `n`, cela aura pour conséquence de passer dans la règle CUP *declar_var*. Par ce biais, on va vérifier que cette variable n'existe pas déjà dans notre table de symbole, afin de l'ajouter. On va stocker cette variable dans une *HashMap* qui fait correspondre un identifiant (String) à son type (Type). Cela va nous permettre par la suite, c'est à dire lors du passage en Jasmin, de s'assurer qu'on n'utilise pas cette variable pour contenir autre chose qu'un entier.

Pour poursuivre, lors de l'assignation de la valeur de `n`, on peut remarquer qu'on utilise des parenthèses, cela nous permet ne nous assurer de la priorité de l'opération `3 * 5`. CUP va générer un objet *Plus*, qui hérite de *OpBinEntier*. Ce dernier va avoir une partie gauche, qui pointera vers un objet *Parenthese*, qui va contenir une expression, constituée d'un objet *Times*, avec comme partie gauche *Entier(3)*, et comme partie droite *Entier(5)*. Ensuite, CUP va nous ajouter une partie droite, qui pointera elle vers un objet Entier(2).

Voici un schéma permettant de comprendre à quoi ressemblerai l'instruction `n = (3 * 5) + 2;` dans l'arbre abstrait :



Nous arrivons maintenant au niveau de notre condition *si*. Ici, CUP va reconnaître à l'aide des règles de syntaxe que nous somme face à une condition. Pour commencer, on va créer un objet *Cond*. Ce dernier, doit contenir différentes choses. On lui donne une expression, c'est elle qui doit être vérifiée afin que notre condition le soit. Ici, elle vaut `n == 17`. Elle est donc, elle-même composé de l'objet *Equals* qui hérite de la classe abstraite *Expression*, qui lui contient `n` comme opérande gauche, et Entier(17) comme opérande de droite. Ensuite, notre condition ne s'arrête pas là, en effet, elle doit contenir un corps à exécuter lorsque la condition sera validée, et optionnellement un corps lorsque la condition n'est pas validée. Ces corps sont en fait des Objets, plus particulièrement des `Queue<Instruction>`. Ici, les queues ne contiendront qu'un objet, puisque nous n'avons qu'une seule instruction par corps. Par exemple, pour le cas du `alors`, cette instruction est analysée par CUP, qui va nous permettre de générer un objet *WriteStringConst*, qui hérite de *Write*, et finalement d'*Instruction*. Notre objet va contenir ici une chaîne de caractère, `"Calcul OK !"`, qu'il faudra afficher lorsque la condition sera validée. On procède de la même manière pour la partie `sinon`.

Conversion de l'arbre abstrait en Jasmin

Afin d'obtenir un exécutable java, nous avons convertis notre arbre abstrait, ainsi que la table des symboles en code Jasmin.

Pour cela nous avons implémenté le pattern visiteur. Notre implémentation est composé d'une interface "*Visiteur*", d'une interface "*Nœud*" et d'une classe concrète "*VisiteurJasmin*" implémentant l'interface correspondante.

Les classes concrètes de l'arbre abstrait implémente l'interface *"Noeud"*. Lors de la production du code Jasmin, on vérifie la cohérence des types par exemple, on ne peut pas additionner des booléens, ou vérifier l'égalité entre un entier et un booléen. On vérifie également que les variables et constantes utilisées ont bien été déclarées.

La table des symboles est séparées en deux HashMap, la première associe à un identifiant donné un type. La seconde associe à un identifiant donné un entier. Cet entier est utilisé lors de la production du code Jasmin. En effet dans le code Jasmin, chaque variable est numérotée, ce numéro permet d'y accéder plus tard pour la lecture et l'écriture.

Une fois les variables déclarées, on parcourt chaque instruction du programme que l'on converti en code Jasmin.

Voici la liste des nœuds de l'arbre abstrait traité par le pattern voyageur.

- For
- While
- Cond
- Read
- WriteExpression
- WriteStringConst
- Assignation
- And
- Booleen
- DiffEqual
- Divide
- Entier
- Equals
- Identifiant
- Inf
- InfEqual
- MinusBin
- MinusUna
- Not
- Or
- Paranthese
- Plus
- Sup
- SupEqual
- Tilda
- Times

Usage :

Pour créer le compilateur, on utilise la commande :

```
make
```

Pour générer le programme "*ProgrammeHepial*" :

```
make ProgrammeHepial
```

Cette commande généra un fichier "*ProgrammeHepial.j*", puis ce programme sera compilé en "*ProgrammeHepial.class*" et finalement exécuté.

De manière analogue, il est possible d'exécuter un programme de test, permettant de s'assurer du bon fonctionnement de l'ensemble des instructions.

```
make HepialTest
```

Ce programme de test à été réalisé en partenariat avec nos collègues Pirkl Théo et Pallud Lucas.

Attention notre compilateur possède une petite spécificité, si l'on utilise l'instruction "*lire [variable]*;", il faut entrer les valeurs en anglais, c'est à dire "true" ou "false";

Conclusion

Pour conclure, ce projet nous a permis de comprendre comment fonctionne un compilateur. La compilation nécessite différentes phases. Premièrement, on commence par faire une analyse lexicale du code source, ensuite, on procède à une analyse syntaxique, avant de convertir notre code en un arbre abstrait simple à parcourir. Pour finir, on visite chacun de nos éléments de l'arbre dans l'ordre (DGR), afin de produire du Jasmin, code assembleur pour la machine virtuelle Java (JVM).

D'après nos protocoles de test l'ensemble de nos instructions fonctionnent. On peut dès lors dire que les objectifs de ce travail pratique, ont été remplis à nos yeux.