Réalisation d'un compilateur pour HEPIAL

Maxime Lovino Thomas Ibanez

19 juin 2017

Table des matières

1 Introduction

Nous avons réalisé un compilateur pour un langage de programmation fictive appelé `HEPIAL` qui va produire du byte-code pour pouvoir exécuter notre programme dans la JVM.

Ci-dessous un exemple de programme en `HEPIAL` :

```
1  programme hepial5
2
3  entier n;
4  entier result;
5
6  entier facto(entier k)
7  entier x;
8  debutfonc
9    x=0;
10   si k == 0 alors
11     retourne(1);
12   sinon
13     retourne(k*facto(k-1));
14   finsi
15 finfonc
16
17 debutprg
18   lire n;
19
20   si n < 20 alors
21     result = facto(n);
22     ecrire "factorielle de ";
23     ecrire n;
24     ecrire " est égale    : ";
25     ecrire result;
26   sinon
27     ecrire " votre nombre est trop grand ! ";
28   finsi
29 finprg
```

2 Flex

Dans la partie flex de notre compilateur, nous détectons tous les mots-clés du langage et retournons des symboles à la partie CUP. Dans le cas de la détection d'une string ou d'une constante entière, nous passons également la valeur à CUP.

3 CUP

Dans la partie CUP, nous définissons la grammaire de notre langage à partir du document de spécification fourni. Ici, nous remplissons la table des symboles et l'arbre abstrait.

4 Table des symboles

La table des symboles est une map qui prend comme clé une `Entry` et comme valeur une liste de `SymbolHEPIAL`

Elle prend en compte les différents blocs et lorsqu'on appelle la fonction `identify` à partir d'un bloc, elle nous renvoie le symbole le plus spécifique au bloc actuel.

Elle gère également les doubles déclarations et renvoie une exception en cas de double déclaration d'un même identifiant dans le même scope.

Les différents types du langage sont définis comme des sous-classes de `Type`. Le diagramme UML du package des types se trouve en annexe.

5 Arbre abstrait

L'arbre abstrait permet de representer notre programme comme un ensemble d'objets qu'on va grouper pour permettre la vérification et la génération du code.
Il faut donc créer une classe par instruction ou expression possible du langage qu'on compile. Après l'analyse du programme, il ne reste qu'un objet dans la pile d'arbre, qui est axiome, à savoir le programme complet et les différentes fonctions.
Le diagramme UML de l'arbre abstrait est en annexe.

6 Analyseur sémantique

L'analyseur sémantique, basé sur le pattern visiteur va visiter chaque instruction de chaque bloc, et chacun de composants de ces instructions afin de vérifier la validité de celles-ci. Par exemple, on va vérifier le scope des variables et la validité des paramètres lors de l'appel de fonctions. Il va également réduire les expressions constantes. Par exemple si on prend le programme

```
x = 10 + 2
```

L'analyseur va le transformer en

```
x = 12
```

Cette transformation est également vraie pour les booléens, si une expression "ou" contient un membre vrai, alors elle sera réduite à vrai, au contraire si une expression "et" contient un membre faux, alors elle sera réduite à faux.
L'analyseur va également vérifier que lors d'une opération n'impliquant que des booléens ("et" et "ou" par exemple) les deux opérandes sont de type booléen, au contraire lors d'une opération arithmétique, les deux opérandes doivent être des entiers. Si l'opérateur fonctionne avec les deux types ("==" par exemple) alors l'analyseur va vérifier que les deux types sont compatiables.
Voici quelques exemples d'erreurs possibles : Si on prends un programme avec beaucoups d'erreurs :

```
1 programme errorhepial
2
3 booleen b,c;
4 entier x;
5 entier y;
6
7 entier f(entier k)
8 debutfonc
9   retourne(k + 2);
10 finfonc
11
12 debutprg
13   x = vrai;
14   y = x + b;
15   b = 0;
16
17   si x < b alors
18     x = ~x;
19     y = b(x);
20     y = f(c);
21     y = f(x, y);
22   sinon
23     pour b allantde 0 a 10 faire
24       r = 3;
25     finpour
26   finsi
27 finprg
```

On obtiens :

```
[ERROR]Symbol r not found file:24
[ERROR]Type mismatch: trying to put BOOLEAN in INTEGER file:14
[ERROR]Bad operand : expected integer (b)
[ERROR]Type mismatch: trying to put INTEGER in BOOLEAN file:17
[ERROR]Bad operand : expected integer (b)
[ERROR]b is not a function file:19
[ERROR]Type mismatch: trying to put BOOLEAN in INTEGER file:20
[ERROR]Parameter number 0 type mismatch, expected INTEGER but got BOOLEAN file:20
[ERROR]Too many parameters when calling function f file:21
[ERROR]For variable must be an integer file:26
[ERROR]Type mismatch: trying to put INTEGER in null file:25
```

7 Générateur de Jasmin

Jasmin fonctionne avec une pile d'opérandes servant a faire les différentes opérations et de registres locaux au contexte d'une fonction.

Lorsqu'une variable est utilisée on lui assigne un numéro de registre local afin de savoir où elle est stockée.
Le générateur va parcourir l'arbre et transformer les instructions et expressions dans leurs équivalent jasmin.

7.1 Opérandes relationnelles et labels

Dans le cadre des opérateurs relationnels, leurs fonction visiter va mettre les opérandes sur la pile, ajouter une soustraction puis retourner le test a faire pour sauter (branch) si la comparaison est vraie.
Par exemple dans le cas de l'opérateur "==" :

```
1   @Override
2   public Object visit(Equal e) {
3     visitBinary(e);
4     appendln("isub");
5     return "ifeq";
6   }
```

La fonction retourne "ifeq" car si la soustraction retourne 0, alors le test est vrai, ainsi l'appelant (un "si" par exemple) pourra définir la label oû sauter.
Par exemple dans le cas de l'opérateur "==" :

```
1   @Override
2   public Object visit(Condition c) {
3     Object cnd = c.getCondition().accept(this);
4
5     appendln(((String) cnd) + " if" + c.hashCode() + "_then");
6     appendln("goto if" + c.hashCode() + "_else");
7
8     appendln("if" + c.hashCode() + "_then:");
9     for (Instruction i : c.getThen()) {
10      i.accept(this);
11    }
12    appendln("goto endif" + c.hashCode());
13    appendln("if" + c.hashCode() + "_else:");
14    for (Instruction i : c.get_else()) {
15      i.accept(this);
16    }
17    appendln("endif" + c.hashCode() + ":");
18    return null;
19  }
```

Les if, for et while se servent du `hashCode()` de l'objet pour générer leur labels de saut. Exemple le programme :

```
1 programme hepial1
2
3 entier x;
4
5 debutprg
```

```
 6   x = 10 + 2;
 7   si x == 12 alors
 8     ecrire "YOUHOU";
 9   sinon
10     x = 21;
11   finsi
12   ecrire x;
13 finprg
```

Donne le jasmin suivant :

```
.method public static main([Ljava/lang/String;)V
.limit stack 6
.limit locals 8
ldc 12
istore 0
iload 0
ldc 12
isub
ifeq if1956725890_then
goto if1956725890_else
if1956725890_then:
ldc "YOUHOU"
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
goto endif1956725890
if1956725890_else:
ldc 21
istore 0
endif1956725890:
iload 0
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/print(I)V
return
.end method
```

On peut voir comment le if génère son code et ses labels et que l'analyseur a bien fait son travail car x est directement assigné à la valeur 12 (10 + 2).

7.2 Fonctions

Le programme étant séquentiel, les fonctions sont déclarées static et appelées par l'instruction `invokestatic`.
Elles sont déclarés avant le fonction main dans le fichier jasmin.
Exemple, la fonction factorielle recusive définie plus haut :

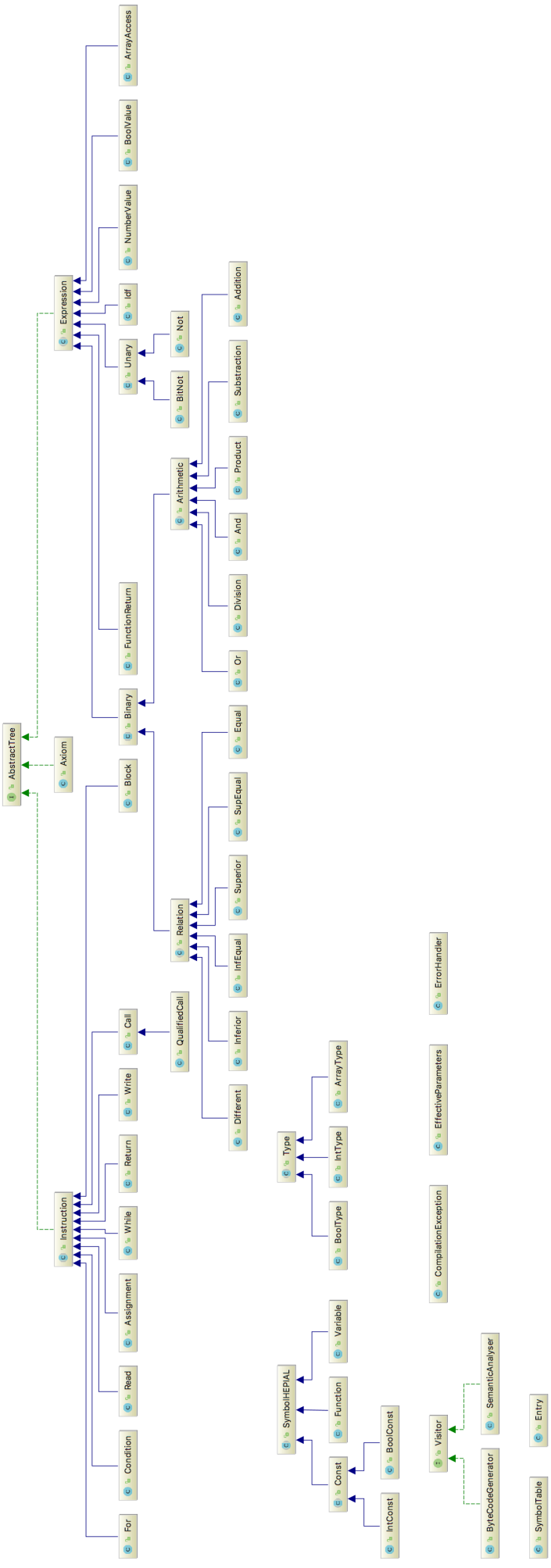
```
.method public static facto(I)I
.limit stack 5
.limit locals 10
ldc 0
istore 1
iload 0
ldc 0
isub
ifeq if1956725890_then
goto if1956725890_else
if1956725890_then:
ldc 1
ireturn
goto endif1956725890
if1956725890_else:
iload 0
ldc 1
isub
invokestatic hepial.facto(I)I
iload 0
imul
ireturn
endif1956725890:
ldc 0
ireturn
.end method
```

8 Conclusion

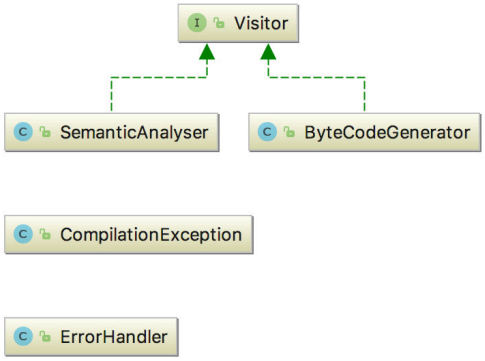
En conclusion, ce projet était assez intéressant à réaliser, malgré le fait que la documentation sur les outils utilisés ainsi que les différentes techniques était assez difficile à trouver. Cependant cela nous a permis de mieux comprendre les fondements de la compilation et de la génération de code.
Nous sommes très contents du résultat car nous pouvons compiler tous les programmes fournis, ainsi que d'autres programmes écrits par nos soins.

Diagrammes UML

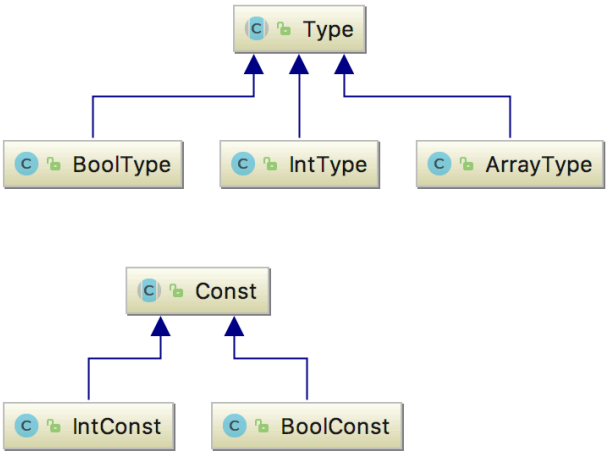
Code complet



Package code



Package type



Package tree