Techniques de Compilation, Rapport de projet

Dagier Thomas
Bernasconi Dorian

24 Janvier 2021

1 Introduction

Dans le cadre du cours de Techniques de Compilation, nous devons créer notre propre compilateur. Ce projet de semestre réalisé entre le 17 Novembre 2020 et le 24 Janvier 2021 permet de compiler un programme écrit avec le language hepial. A travers divers processus expliqués tout au long de ce rapport, nous devons utiliser certains outils mis à disposition comme "java-cup.jar" et "jasmin.jar" afin de générer un éxécutable (java ByteCode) que nous pouvons lancer pour tester notre programme.

Ce rapport décrit le travail réalisé par Dorian Bernasconi et Thomas Dagier étape par étape. Nous avons travailé ensemble sur chacune des parties de ce projet afin de comprendre le déroulement dans son intégralité. Nous allons aborder dans ce rapport chaque grande partie de ce projet en expliquant leurs interêts, la manière dont elles sont implémentées, quelques points clés importants et les difficultés rencontrées.

2 Structure du projet

Commencons par la structure du projet. On pourrait la représenter par la figure suivante :

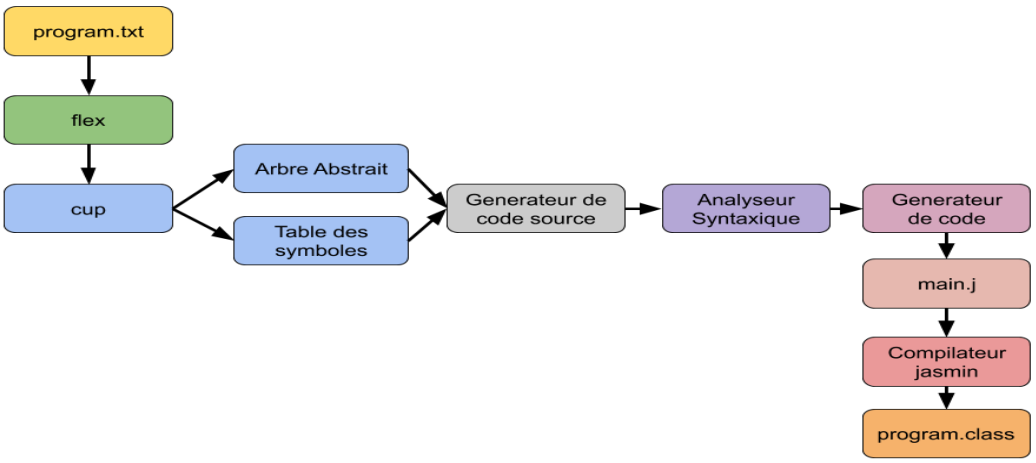

Figure 1: description de la structure du projet

"program.txt" est notre fichier contenant le code à compiler et "program.class" est le fichier bytecode qui résulte de notre compilateur. Nous allons voir chacun des blocs pour comprendre comment s'organise le projet.

3 Flex

3.1 Objectifs

Le Flex est la partie lexicale du projet, c'est le dictionnaire de notre code qui se trouve dans le fichier texte d'entrée. Les choix des mots que l'on utilise pour écrire un programme hepial sont régis par le Flex. C'est une grammaire contenant un certains nombre de règles devant êtres appliquées afin de réduire l'ensemble de notre programme en une liste de symboles. Cette dernière sera utilisée par le CUP afin de vérifier que la syntaxe est bonne.

3.2 Fonctionnement

Le fichier d'entrée est lu caractère par caractère. A chaque lecture, on essaye de trouver une règle permettant de créer un symbole. Chaque élément lu peut être transformé en un symbole s'il existe une règle le permettant. Afin de renforcer notre grammaire, nous utilisons également des expressions régulières qui peuvent aboutir sur la création d'un symbole. Si un élément ne s'applique à aucune règle, on génère une exception due à une erreur lexicale. Dans le cas contraire, on possède, à la fin de la lecture lexicale, une liste de symboles qui sera utilisée par CUP.

3.3 Implémentation

Cette partie du projet est fournie entièrement afin que l'on se concentre sur le reste. Nous avons simplement à retirer certaines règles comme les appels de fonctions.

```
1  ident = [a-zA-Z][A-Za-z0-9]*
2  constantInt = [0-9]+
3  constanteStr = \"([^\"]|\"\")*\" // double " to add in string
4  semicolon = ;
5
6  %%
7  ### rules ###
8
9  # règles conséquences d'expressions régulières :
10 si on trouve un mot du style ... alors on créé le symbole
11 {constantInt}   { return new Symbol(sym.INTEGERCONST, yyline, yycolumn,
   ↪  Integer.parseInt(yytext())); }
12 {constanteStr}  { return new Symbol(sym.STRINGCONST, yyline, yycolumn,
   ↪  yytext()); }
13 {semicolon}     { return new Symbol(sym.SEMICOLON, yyline, yycolumn); }
14 {ident}         { return new Symbol(sym.IDENT, yyline, yycolumn, yytext()); }
15
16 # règles directes : si on trouve le mot ... alors on créé le symbole
17 programme       { return new Symbol(sym.PRG, yyline, yycolumn); }
18 debutprg        { return new Symbol(sym.STARTPRG, yyline, yycolumn); }
19 finprg          { return new Symbol(sym.ENDPRG, yyline, yycolumn); }
20 constante       { return new Symbol(sym.CONSTANT, yyline, yycolumn); }
21 "<>"            { return new Symbol(sym.DIFF, yyline, yycolumn); }
22 ==              { return new Symbol(sym.EQUALS, yyline, yycolumn); }
```

Voici une petite partie du Flex illustrant la manière dont un mot est transformé en symbole. Cela peut être fait soit par utilisation d'expressions regulières (lignes 1 à 4 puis 9 à 14) soit par détéction simple d'un mot (ligne 16 à 22).

3.4 Difficulté rencontrée

La seule vrai difficulté que nous avons rencontré durant le Flex est de s'adapter au lexique du language hepial. Le flex est à première vue très simple mais c'est à double tranchant car nous avons eu plusieurs fois des erreurs sans réelement comprendre d'où elles venaient avant de nous rendre compte qu'elles étaient dûes au Flex (par exemple l'opérateur différent qui s'écrit "<>" au lieu de "!=").

4 Cup

4.1 Objectifs

Une fois que nous avons récupéré la liste des symboles générée par Flex, nous devons vérifier que l'ordre dans lequel ils apparaissent soit cohérent. Ainsi, on adopte la même démarche que pour le flex c'est-à-dire que l'on prend symbole par symbole les éléments de la liste des symboles du Flex jusqu'a pouvoir réduire complètement notre code en un seul objet.

Si la réduction est possible, cela signifie que la syntaxe est bonne donc que le programme suit un ordre logique. Dans le cas contraire, on observe une erreur qui n'est, cette fois-ci, pas due au flex mais à l'agencement des symboles dans le programme (par exemple : entier 4 = resultat; génère une exception alors que entier resultat = 4; n'en génère pas).

Afin de profiter de la réduction pour la suite du travail, nous réalisons en même temps que le cup, un Arbre Abstrait et une table des symboles. Ces deux éléments serviront pour le reste du projet afin de réécrire le code dans le terminal, l'analyser sémantiquement et produire le code cible qui nous permettera de l'éxécuter.

4.2 Fonctionnement

Dans le cup, nous possèdons toute une liste d'objet qui sont le résultat d'une réduction préalable. Voyons l'exemple de l'un de ces objets :

```
1 access ::= IDENT:id {: RESULT = new Idf(id, "", idleft, idright); :};
2
3 assign ::= access:dest EQUAL:e expr:src SEMICOLON {: RESULT = new
  ↪  Assignment(dest, src, "", eleft, eright); :};
```

Ici, on veut créer un objet "assign". Pour ceci, nous avons besoin de 2 autres objets : "access" et "expr". L'objet "access" se créé à partir d'un symbole "IDENT" que l'on s'attend à trouver dans la liste des symboles du flex.

On suppose que "access" est réalisé avec succès. Dans ce cas, on peut continuer la création de l'objet "assign". On s'attend alors à tomber sur un symbole "EQUAL" dans la liste du Flex puis on voudra créer un objet "expr" et ainsi de suite... L'objet "assign" sera utilisé plus tard de la même manière que l'objet "access" jusqu'a créer un objet final qui est le fruit de la réduction de l'ensemble de notre code (dans notre cas, c'est l'objet "program").

4.3 Approche technique

Le cup qui nous a été initialement fournis disposait déjà de la majeur partie des objets nécessaires à la réalisation de la première étape de notre analyse syntaxique. Nous devions simplement décider quelles actions faire une fois que l'objet est créé. Dans notre cas les actions sont d'instancier les classes correspondantes pour construire l'arbre abstrait. Dans le même temps, lorsque l'on déclare des variables et des constantes, nous les ajoutons dans la table des symboles.

4.4 Difficultés rencontrées

Le cup est, selon nous, la partie la plus longue du projet. Elle demande beaucoup de temps afin de bien comprendre le fonctionnement. Nous avons mis un bon mois pour comprendre entièrement le cup afin de réaliser un arbre cohérent et fonctionnel ainsi qu'une table des symboles pratique.

5 Arbre abstrait

L'arbre abstrait correspond à l'ordre dans lequel les éléments sont lues. par exemple, une addition est une expression binaire qui est contenue dans un bloc qui est lui-même contenu dans un program ... Nous instancions les classes dans le même ordre que la réduction faite par cup. Pour la création des classes il suffit d'avoir des méthodes permettant de modifier le contenu des champs ou alors de les récupérer. Il faut en créer beaucoup (une trentaine) afin d'avoir un arbre bien construit et cohérent. Voici donc sa structure :

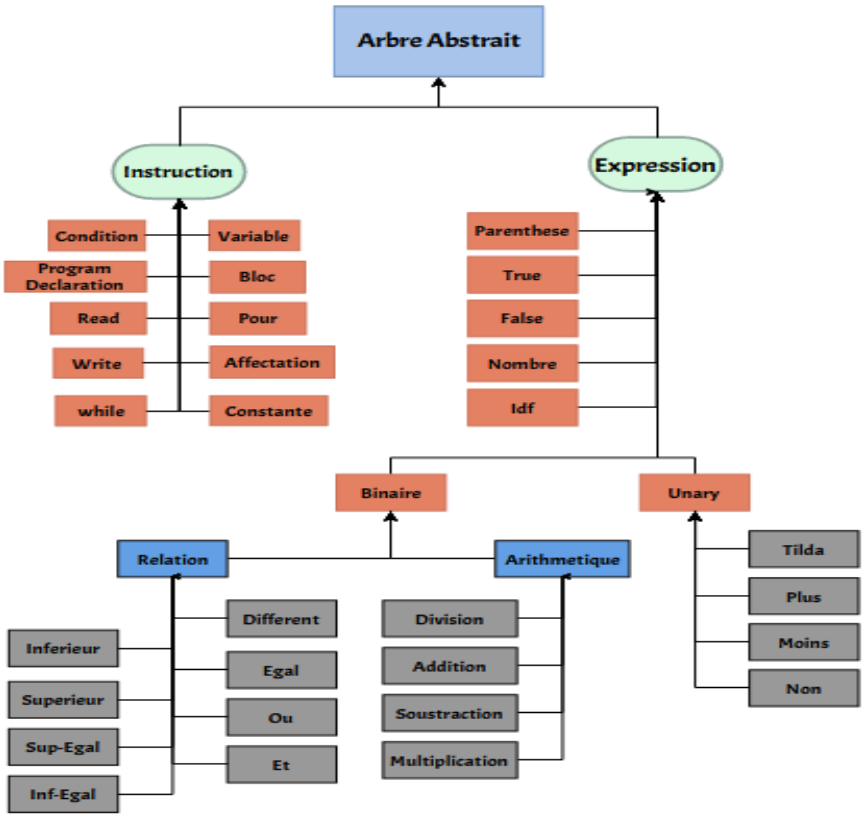

Figure 2: arbre abstrait complet

6 Table des symboles

La table des symboles est également concurente de l'implémentation du cup et de l'arbre abstrait. Dans notre programme, il est possible de déclarer des variables. Elles peuvent être "entier" ou "booleen" et "constante" ou pas. Elles possèdent un nom ("Idf") et une valeur si elles sont déclarées comme constante. Tout autant d'informations dont nous avons besoin pour analyser le code sémantiquement, générer le code source afin de l'afficher ou encore le code cible nécéssaire à l'éxécution.

L'objectif est donc d'ordonner les variables déclarées dans une Hashmap (en java) afin d'accèder rapidement aux données en fonction du nom. Notre hashmap fait le lien entre une clé unique (qui correspond au nom de la variable déclarée) et une un objet instancié de la classe Symbole. Ainsi, en indiquant le nom, on récupère un objet nous donnant accès au type, si c'est une constante ou pas et si s'en est une, sa valeur.

7 Générateur de code source

7.1 Objectif

L'objectif du générateur de code source est simplement d'afficher le code qui a été lu. C'est un très bon moyen de vérifier que notre cup est cohérent et que notre arbre abstrait l'est également.

7.2 Fonctionnement

La génération du code source est basée sur le design pattern visitor. L'objectif est de visiter l'objet qui réduit tout le code. Dans ce cas la notre objet est "declarationProgram". Dès lors, on visite les champs de cet objet qui sont le bloc de déclaration et le bloc d'instructions et ainsi de suite jusqu'a parcourir l'ensemble de l'arbre. On se base sur le concept d'héritage afin de se déplacer efficacement parmis tous les objets.

Par exemple, lorsque l'on visite un objet "Addition", on va visiter les champs de l'addition soient l'opérande de gauche, l'opérateur "+" et l'opérande de droite. En récupérant ces données, il nous est possible de faire de l'analyse sémantique, de la production de code ou simplement d'afficher le contenu. Ici, afficher le contenu nous permet de vérifier le bon fonctionnement de design pattern.

```
1  public Object visit(Addition node){
2      node.getLeft().accept(this);    #récupération de l opérande de gauche
3      code += " " + node.operator() + " "; # ajout de l opérateur
4      node.getRight().accept(this);   #récupération de l operande de droite
5      return node;
6  }
```

Une autre manière d'afficher (que nous avons aussi implémenté) est de redéfinir la méthode toString() dans chacune des classes.

8 Analyseur Semantique

8.1 Objectif

L'objectif de l'analyseur sémantique est de vérifier que le code soit cohérent. Si on est maintenant sûr que l'agencement des symboles est logique grâce au cup, il nous faut quand même vérifier que l'on n'affecte pas un entier à un booleen par exemple. On pourrait aussi devoir vérifier que l'on ne modifie pas une constante ou encore que l'on fait des opérations sur des variables de même type.

8.2 Fonctionnement

Le fonctionnement est exactement le même que celui du générateur de code source. On utilise le design pattern visitor. Cette fois-ci, au lieu d'affiche du contenu, on vérifie sa cohérence (sémantique du code).

8.3 Implémentation

D'un point de vu global, 90 % du travail est de vérifier que les types sont cohérents. pour cela, il faut connaitre le type retourné par une expression, une parenthèse... Pour cela, nous avons créé 2 fonctions qui font la majeure partie du travail. La première vérifie que 2 opérandes sont du même type, la seconde retourne le type des classes les plus basses (les feuilles de l'arbre abstrait).

Par exemple, la class "Numbe" ne possède pas de type, pourtant nous savons que c'est un "entier", il faut donc le signifier pour vérifier la cohérence des types. Si on fait "resultat = 4 - 2" par exemple, "2" est un "Number" que l'on veut interprêter comme un entier. On doit alors vérifier que l'on fait une addition entre 2 entiers et que l'affectation se fait dans un entier qui aura été déclaré précédement.

```
1    public Object visit(Addition node){
2        return checkIfTypesAreEquals((Binary)node);
3    }
```

Ici, on sait qu'une "addition" est avant tout un objet "binary" on peut donc faire un cast sans soucis afin de pouvoir accèder aux méthodes de la class "binary". La fonction checkIfTypesAreEquals() permet de comparer le type l'expression de gauche avec celui de notre expression de droite. Si tout est bon on passe à la suite (qui devrait sans doute être une affectation). Dans le cas contraire, on lève tout de suite une exception.

8.4 Difficultés rencontrées

Cette partie est sans doute la partie la plus compliquée du projet. Il faut prendre en compte tout les cas possibles, ce qui demande du temps mais surtout d'avoir une bonne implémentation ce qui n'est pas vraiment notre cas. Nous avons souvent recourt à des cast ce qui n'est pas une très bonne pratique. Si nous avions eu plus de temps, nous aurions sans dout refait toutes nos classes depuis le début afin d'avoir une implémentation plus propre. Ne gérant pas tout les cas tout de suite, nous revenons régulièrement sur cette partie du projet afin d'ajuster certaines parties.

9 Générateur de code

9.1 Objectif

Le générateur de code est la partie finale du TP. A l'aide de tout les blocs précédents, nous avons vérifié le lexique, la syntaxe et la sémantique. Nous sommes donc à peu près sûr que le code est juste. On ne sait pas encore s'il fait ce que l'on veut par contre on sait qu'il fait quelque chose. On peut donc générer un fichier jasmin contenant les instructions de notre programme.

9.2 Fonctionnement

A partir de notre fichier texte on créé le code jasmin. Ce code est crée à l'aide du design pattern visitor. Prenons le cas d'une addition :

```
1    public Object visit(Addition node){
2        node.getLeft().accept(this);
3        node.getRight().accept(this);
4        cible += "\niadd";
5        return null;
6    }
```

Pour faire une addition avec jasmin, on doit ajouter l'opérande de gauche sur la pile (avec un "ldc") puis celle de droite et enfin écrire l'instruction "iadd". Dans le cas simple où les opérandes sont des nombres et non pas des expressions plus complexes, la ligne node.getLeft().accept(this); aura pour effet d'écrire le contenu de l'opérande de gauche sur la pile. La même chose sera faite pour l'opérande de droite avant de faire un "iadd".

9.3 Difficultés rencontrées

La génération du code n'a pas été la partie la plus difficile car nous avions déjà fait la majorité du TP et résolu la majorité des problèmes. Grâce à la doc nous décrivant quoi faire en fonction des objets sur lesquels on tombe, il n'a pas été très long d'implémenter cette partie du projet.

10 Démonstration

Afin de voir comment se comporte notre programme, voyons ce qui se passe depuis le début. Nous avons un programme qui réalise la suite de fibonacci. L'utilisateur rentre le nombre d'éléments qu'il veut dans sa suite. Ensuite, nous générons les nombres puis nous les affichons. Voici le code complet :

```
1   programme fibonacci
2   entier i, index, next, crt, entier result;
3   debutprg
4           ecrire "Entrez l'indice du numéro de fibonacci: ";
5       lire index;
6           ecrire "\n";
7           crt = 1;
8           next = 1;
9           pour i allantde 0 a index faire
10                  result = crt + next;
11                  crt = next;
12                  next = result;
13                  ecrire next;
14                  ecrire ",";
15          finpour
16          ecrire "\n";
17  finprg
```

Ce code est d'abbord lu par le flex. Aucune erreur n'est détéctée. On envoie donc la liste des symboles dans cup. En plus de ne détécter aucune erreur de syntaxe, on génère la table des symboles et l'arbre abstrait. Afin de vérifier que notre implémentation est jusque là cohérente, on affiche le code qui a été lu.

Ensuite, on vérifie que la sémantique est bonne. Une fois de plus pas d'erreurs à déplorer. Dans ce cas, on estime que le code peut compiler. On crée alors le fichier jasmin contenant notre liste d'instruction. L'étape finale est de créer un fichier class à partir du fichier jasmin. Ainsi, notre fichier de sortie se nomme "fibonacci.class". Il ne nous reste plus qu'à l'éxécuter.

Voici le résultat de notre compilation :

```
# Fichier testé : input.txt

Table des symboles :
        entier next = 0; false
        entier result = 0; false
        entier crt = 0; false
        entier i = 0; false
        entier index = 0; false

Lecture de l'arbre abstrait :
programme fibonacci
        entier i;
        entier index;
        entier next;
        entier crt;
        entier result;
debutprg
        ecrire "Entrez l'indice du numéro de fibonacci:";
        lire index;
        ecrire "\n";
        crt = 1;
        next = 1;
        pour i allant de 0 a index faire
            result = crt  +  next;
            crt = next;
            next = result;
            ecrire null;
            ecrire ",";
        fin-pour
        ecrire "\n";
finprg

Analyse Sémantique : OK

Production du Code : Generated: fibonacci.class
```

Figure 3: vérification du bon fonctionnement du compilateur

Afin de simplifier toutes ces étapes, nous déléguons le travail à un Makefile. en utilisant la commande make dans le terminal, on génère automatiquement un fichier .class. Il nous suffit d'utiliser la commande "java fibonacci.java" pour tester notre programme :

```
thomas@thomas:~/Documents/GIT/tcp/Projet$ java fibonacci
Entrez l'indice du numéro de fibonacci: 10

2,3,5,8,13,21,34,55,89,144,233,
thomas@thomas:~/Documents/GIT/tcp/Projet$
```

Figure 4: éxécution du programme

11 Conclusion

En conclusion, nous avons appris dans les détails comment se comporte un compilateur java. C'est un travail qui a été très interessant d'autant plus qu'il met en application les notions que nous avons abordées tout au long du semestre. Ce fut un projet très long et très compliqué, nottament car la majeur partie du travail aura été de comprendre le travail que nous devions faire. Nous sommes très content du résultat. De plus nous avons eu le temps de terminer complètement le projet ainsi que tester notre compilateur sur des programmes plus complexes.