Traduction des Langages
Compilateur MicroJAVA

Fabian Guion   Audric Schiltknecht   Denis Villand

11 juin 2009

Table des matières

1 Introduction **2**

2 Table Des Symboles **2**
2.1 Présentation 2
2.2 Variables 2
2.3 Classes 3
2.4 Liaison tardive 3
2.5 Appel de méthode 5

3 Ce qui a été fait **5**
3.1 Étapes 5
3.2 Fonctionnalités 6

4 Critiques sur la réalisation du projet **7**
4.1 TDS 7
4.2 Surcharge de méthode/constructeur 7
4.3 Mots-clés `this`/`super` 7

5 Tests **8**
5.1 Compilation 8
5.2 Génération de code 8
5.3 Geo.mj 8
5.4 Fact.mj 8

6 Conclusion **8**

1 Introduction

Ce projet vise à l'écriture d'un compilateur pour un sous-ensemble du langage de programmation Java. Ce compilateur peut théoriquement produire du code pour n'importe quelle plateforme, mais nous nous contenterons de la production de code pour la machine TAM.

D'autre part, nous n'effectuerons qu'une passe pour réaliser la compilation. Ainsi, il faudra prendre soin à ne référencer que des données qui sont déjà définies.

2 Table Des Symboles

2.1 Présentation

La première étape du projet consiste à étudier la grammaire et en déduire une structure pour la table des symbole (notée « TDS »). La figure 1 décrit la structure que nous avons choisie.

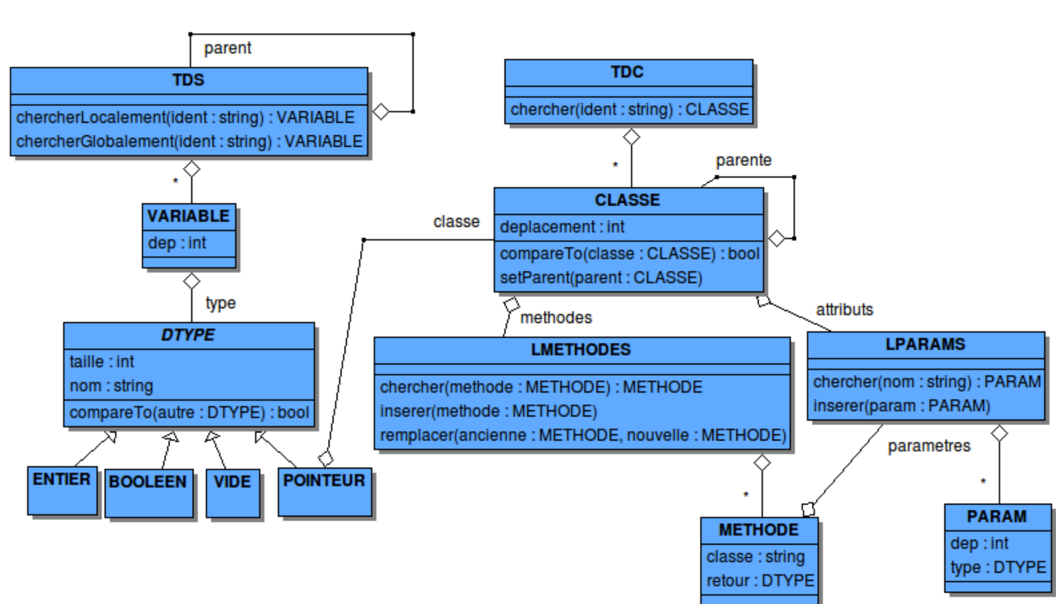

Fig. 1 – Architecture de la TDS

Nous disposons de deux tables : la *TDS*, qui contiendra les locales, et la *TDC*, qui contient l'ensemble des classes définies dans le programme.

Il nous a paru important de bien faire la distinction entre une *classe* et une *instance de classe*. C'est pourquoi variables et classes sont séparées.

2.2 Variables

Les variables (qui représentent les locales), sont simplement définies par un type et un déplacement.

Nous avons définis la classe abstraite DTYPE, qui contient le nom du type, ainsi que sa taille, et créé une fonction de comparaison des types (qui basiquement, va comparer le nom donné au type).

Ensuite, nous avons spécialisé cette classe en les divers types utilisés dans la grammaire : entier, booléen, void, pointeur, etc. Cela nous permet, par exemple, d'ajouter d'autre types primitifs simplement en créant une autre classe héritant de DTYPE, et en redéfinissant éventuellement la fonction de comparaison.

Par exemple, la classe POINTEUR (qui possède un attribut de type CLASSE, indiquant la classe associée à l'objet pointé), redéfinit la fonction `compareTo(DTYPE autre )` en utilisant la comparaison des classes (ce qui permet de gérer les cas d'héritage).

2.3 Classes

Une classe sera donc représentée par le type CLASSE. Lors de sa définition, on ajoutera les divers attributs et méthodes décrits. Ces données seront disponibles dans toutes les règles de productions, puisqu'elles seront transmises via un attribut hérité `classe`.

2.3.1 Constructeurs

Un constructeur par défaut est disponible. On peut redéfinir ce constructeur, ou en ajouter d'autres.

2.3.2 Héritage

Lorsqu'une classe hérite d'une autre, il faut en premier lieu vérifier que la classe parente a bien été définie. Ensuite, on hérite de tous ses attributs ainsi que de ses méthodes[1] grâce à la fonction `setParent`.

On pourra savoir si une méthode est héritée, ou bien redéfinie grâce à son attribut `classe`, qui indique le nom de la classe dans laquelle la méthode a été définie (ou bien redéfinie).

2.4 Liaison tardive

Puisque nous disposons de l'héritage, il se pose le problème de la liaison tardive. En effet, prenons l'exemple du listing 1. Lorsque l'on applique la méthode `uneAutreFonction` sur l'objet p, de type réel PointColoré, il faut pouvoir appeler la méthode définie dans la classe de base (PointColoré).

```
class Point {
    void uneFonction() { ... }
    void uneAutre Fonction() { ... }
}

class PointColore extends Point {
    void uneFonction() { ... }
```

[1] exceptés les constructeurs

```
    int encoreUneFonction() { ... }
}

class Main {
    int main() {
        Point p = new PointColore();
        p.uneAutreFonction(); // Point::uneAutreFonction
        p.uneFonction(); // PointColore::uneFonction
    }
}
```

Listing 1 – Problème de la liaison tardive

Pour ce faire, nous avons utilisé le principe vu en classe : les méthodes d'une classe sont classées en fonction de leur ordre de définition. Lorsque l'on hérite de cette classe, on hérite de l'ensemble des méthodes, toujours triées dans le même ordre. Si l'on redéfinit une méthode héritée, alors on la remplace dans la liste des méthodes héritées. Si l'on ajoute une méthode, elle vient s'ajouter à la suite des autres méthodes.

Puis, on ajoute à une classe un attributs « caché » , qui pointera sur cette *Table des virtuelles*. L'appel d'une méthode se passera de la façon suivante :

```
Empiler la valeur du pointeur (@ de la base de la classe dans le tas = @ de la tdv)
Lire la valeur stockée à cette adresse (@ de la première méthode de la classe)
Ajouter le déplacement ( = le numéro de la méthode a appeler)
Lire la valeur stockée à cette adresse (@ de la méthode a appeler)
Effectuer l'appel
```

La figure 2 décrit la disposition des données dans le tas et la pile.

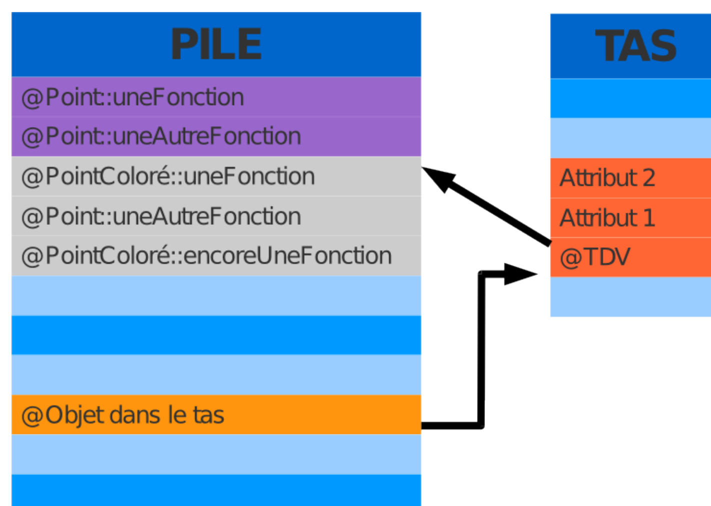

Fig. 2 – TDV

2.5 Appel de méthode

Voici ce qui ce passe lorsqu'une méthode est appelée :
- Le pointeur sur l'objet auquel on souhaite appliquer la méthode est empilé
- Les paramètres sont empilés de telle sorte que le premier paramètre de la fonction soit sur le haut de la pile
- On appelle la fonction (cf 2.4).

3 Ce qui a été fait

On commencera par décrire succinctement les diverses étapes de la réalisation du projet, et on poursuivra par les fonctionnalités de notre compilateur.

3.1 Étapes

3.1.1 TDS et typage

Dans un premier temps, nous avons réalisé le traitement de la TDS. Cette étape consiste à stocker dans les divers conteneurs présentés au schéma 1 les données qui seront utilisées par la suite.

Puis, nous avons pu procéder au typage. Cette étape consiste à vérifier que les types des données utilisées dans les règles de productions sont corrects. La prise en compte de l'héritage se fera grâce aux fonctions de comparaison implémentées dans la classe CLASSE.

Les actions sémantiques réalisant ces opérations sont généralement appelées *tds*, *type*.

3.1.2 Calcul du déplacement

Dans une deuxième étape, nous avons ajouté le calcul du déplacement. Cela permet ainsi de pouvoir récupérer les adresses des données utilisées (locales, paramètres, attributs, pointeurs, ainsi que la création de la TDV), mais aussi de réaliser des opérations de récupération de la mémoire lors de la sortie de bloc d'instructions par exemple.

Les actions sémantiques décrivant ces opérations sont nommées *depl*.

3.1.3 Adresse ou valeur ?

Une des difficultés de ce projet est la suivante : dans une affectation (par exemple), un identifiant peut à la fois être utilisé dans le membre de gauche (en écriture donc), mais aussi dans le membre de droite (en lecture).

Ainsi, il a fallut introduire un attribut sémantique `est_valeur`, qui indique s'il est vrai que l'identifiant ne peut être accédé qu'en lecture, et sinon, en lecture (en récupérant la valeur à l'adresse calculée) ou en écriture.

3.1.4 Génération de code

Enfin, la dernière étape consiste à générer le code exécutable sur la machine cible. On commencera par générer l'ensemble des TDV des classes, puis le code des classes.

Cette génération est effectuée dans les actions *gen*.

3.2 Fonctionnalités

3.2.1 Visibilité des attributs

Les attributs possèdent une visibilité ressemblant à `protected` : ils sont hérités, et non disponibles pour les autres classes.

3.2.2 Redéfinitions de variables dans des blocs

Nous avons permis au programmeur de redéfinir dans des blocs des variables précédemment utilisées. En revanche, une fois sortie du bloc, il ne peut plus accéder aux variables utilisées dans le bloc.

3.2.3 Vérification de la présence/absence de `return`

Une vérification de la présence du mot-clé `return` dans une fonction est faite. Nous avons décidé des règles suivantes :
- Un bloc peut contenir un `return`, auquel cas, la vérification est faite
- Si le bloc *sinon* d'une conditionelle est défini :
  - Et qu'il a un `return`, alors il faut que le bloc *alors* en ait un
  - S'il n'en a pas, on considère que le bloc *alors* n'en a pas non plus.
- S'il n'y a pas de bloc *sinon*, on ne prend pas en compte celui du bloc *alors*
- Un `return` dans le corps d'une boucle ne sera pas pris en compte.

Ces règles nous permettent de nous assurer que dans tous les cas, une fonction renvoie bien une valeur.

D'autre part, nous avons aussi vérifié qu'une procédure ainsi qu'un constructeur ne possédait pas d'instruction `return`.

3.2.4 Constructeur par défaut

Un constructeur par défaut est automatiquement créé. Il s'occupe de réaliser l'allocation de mémoire, ainsi que l'initialisation des attributs (à *NULL* pour les pointeurs, 42 pour les types entiers, et *FAUX* pour les booléens)

3.2.5 Surcharges des méthodes

Il est possible de surcharger des méthodes (ainsi que les constructeurs) : la surcharge se fait au niveau des paramètres (pas du type de retour !). Cependant, l'appel de méthodes ne fonctionne pas totalement (cf 4.2).

3.2.6 Adresse/Valeur

En utilisant l'attribut `est_valeur`, nous pouvons vérifier qu'un affectation est valable. En effet, la grammaire permettait d'écrire des instructions telles que : `(5 = 4)= 3`, ce qui n'est évidement pas valide. De telles erreurs sont maintenant détectées et signalées.

3.2.7 Détection des exceptions sur pointeur

Nous avons rajouté une détection des *Null Pointer Exception*. Il suffit pour cela de rajouter un test lors de l'utilisation d'un pointeur. Si celui-ci n'a pas été initialisé, alors on déroute le programme vers un bout de code signalant l'exception et terminant l'exécution.

3.2.8 Méthode principale

Lors de la compilation, le programme va chercher une méthode possédant la signature suivante : `int main()`. Il va l'executer, puis afficher la valeur de retour.

Nous conseillons de placer la méthode `main` dans une classe annexe. Si elle est placée dans la même classe que les fonctions à exécuter, alors il faut **impérativement** que ces fonctions soient appelées sur un pointeur créé exprès, et non comme des membres de la classe.

3.2.9 Ajout des affichages

Initialement pour des causes de débuggages, nous avons ajouté à la grammaire la règle `INST -> print E`.

4 Critiques sur la réalisation du projet

4.1 TDS

Au final, même si notre organisation n'est pas mauvaise, certains choix peuvent être améliorés ou remis en cause.

En premier lieu, le fait de mélanger méthodes et constructeurs n'est pas conseillé, puisqu'il faut couramment faire la distinction entre les deux.

D'autre part, il aurait peut être été plus judicieux de faire en sorte que toutes les données à identifier (donc des instances des classes VARIABLE, PARAM (en tant qu'attribut/paramètre de méthode)) héritent d'une même classe. En effet, lorsque l'on souhaite vérifier qu'un identifiant existe (ou n'existe pas justement), nous sommes obligés d'effectuer une recherche dans chacune de ces catégories de données.

4.2 Surcharge de méthode/constructeur

La surcharge n'est pas complètement fonctionnelle. En effet, lors de l'appel d'une méthode, on va parcourir la liste des méthodes de la classe. Lorsqu'un méthode »correspond »(ie. même identifiant,type des arguments compatibles), on utilise cette méthode. Ainsi, on ne va pas appeler la bonne méthode, mais la première dont les types sont compatibles. Pour corriger cela, il faudrait que la fonction retourne la méthode dont les types sont les plus proches (au sens de l'héritage) de celle recherchée.

4.3 Mots-clés `this`/`super`

Nous n'avons pas effectué le traitement de ces mots-clés.

Pour `this`, il faudrait le rajouter en tant que paramètre de chaque fonction.
Pour `super`, il faut tester si l'on se situe bien dans la définition d'un constructeur. Puis, il faut chercher le constructeur de la classe parente qui correspond, et effectuer l'appel. Il faudra prendre soin de vérifier qu'il n'y ait pas plusieurs allocations de mémoire (car n'oublions pas que le constructeur par défaut est appelé automatiquement à chaque appel d'un constructeur).

5 Tests

Nous avons réalisés plusieurs tests pour pouvoir éprouver les différentes parties de notre compilateur.
Le fichiers de tests, ainsi que le code engendré (ou la sortie du compilateur) sont fournis en annexe.

5.1 Compilation

Ces premiers tests sont regroupés dans le fichier `Testscompil.mj`. Ils visent à éprouver la partie compilation de notre projet. Ces tests vérifient la véracité du contrôle de type, la vérification adresse/valeur, ainsi que quelques tests sur des fonctions (présence/absence de retour, etc).

5.2 Génération de code

Le fichier `Testsgen.mj` contient l'ensemble des tests utilisés pour vérifier la correction de la génération de code. Entre autre, y sont testés le passage d'arguments, la vérification d'une exception sur pointeur nul, la surcharge de méthodes.

5.3 Geo.mj

Nous avons aussi réussi à exécuter correctement (après adaptation du fichier source à nos contraintes) le fichier de tests fourni.

5.4 Fact.mj

Ce fichier de test, fourni avec le sujet, permet de tester l'appel de fonction récursive (il code la factorielle de façon récursive). En exécutant le code TAM généré, nous obtenons bien le résultat attendu.

6 Conclusion

Nous avons trouvé ce projet très intéressant, tant d'un point de vue théorique que pratique. Ce fut l'occasion de mettre en pratique la théorie vue en cours, et approchée en TP sur un projet de plus grande envergure.

La conception a été facilitée par l'utilisation du plugin EGG pour Eclipse, qui permet une compilation (presque) à la volée, et une détection efficace des erreurs.

Concernant la répartition des tâches, nous avons tous travaillé plus ou moins sur chacune des phases, car les étapes de la conception de ce compilateur restent quand même assez liées.