Sémantique et TDL

Projet de compilation du langage Micro-Java

Julien Noleau
Loïc Ginoux
David Brown

Juin 2008

Table des matières

I. Introduction

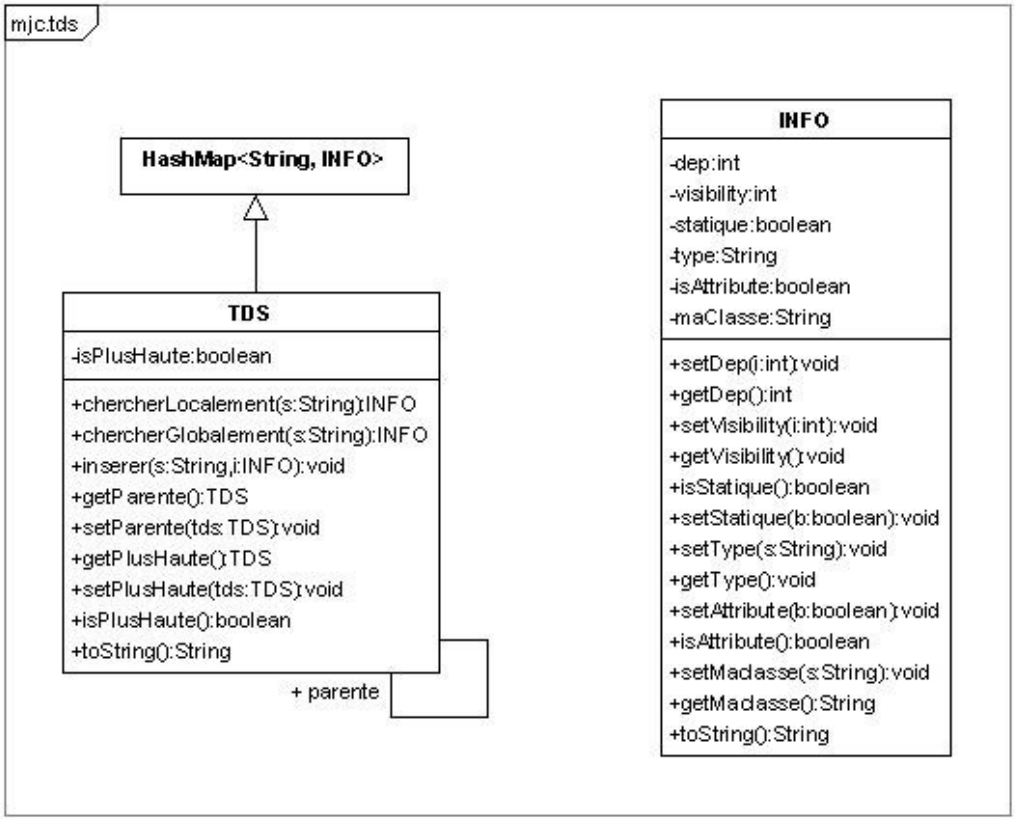
Le but de ce projet était de réaliser un compilateur pour le langage Micro-Java dans le cadre du cours de sémantique et TDL. Ce compilateur devait générer du code TAM interprétable par la machine virtuelle TAM et être assez générique pour pouvoir implémenter rapidement un traducteur dans un autre langage assembleur comme X86 ou encore SPARC.

II. Conception et Gestion des tables

1. Conception du compilateur

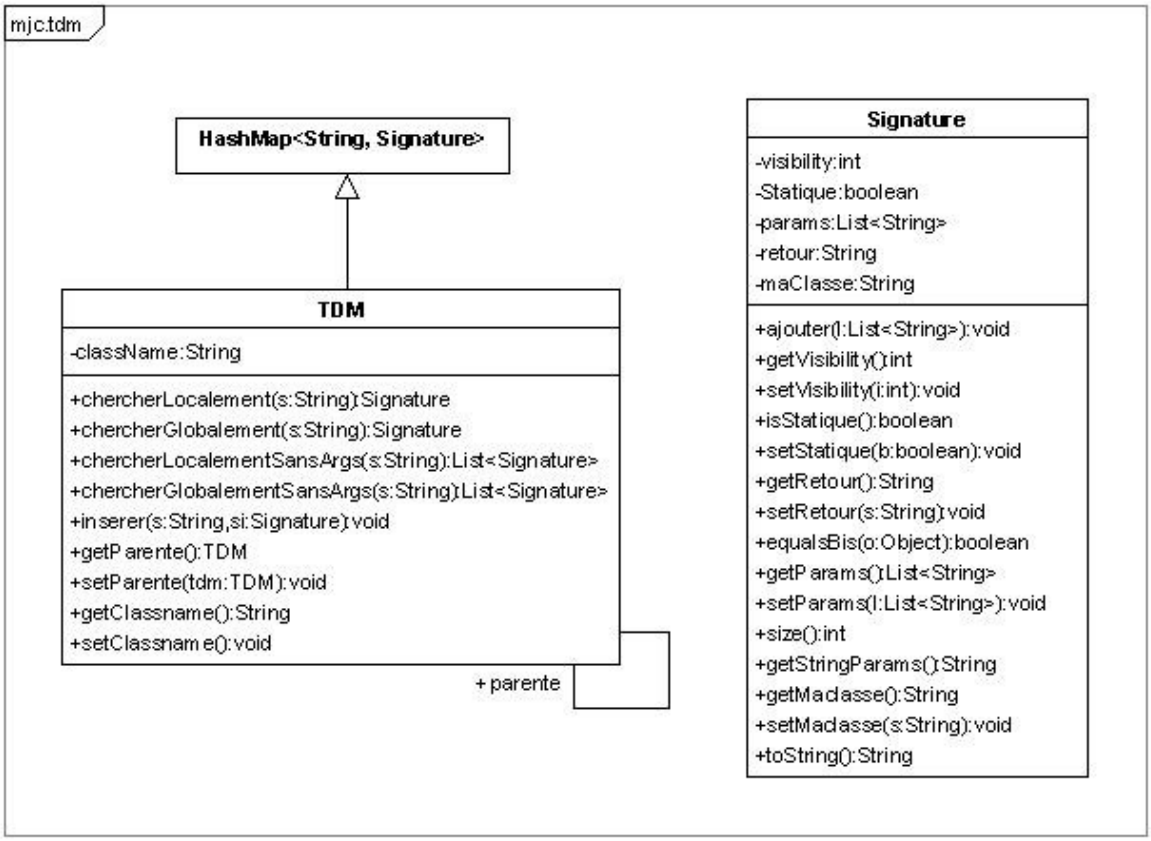
Pour que la génération soit générique, on choisit le langage à la génération avec l'option -m. Une machine est alors créée selon ce langage, réalisant AbstractMachine qui contient toutes les méthodes de génération de code en abstract.

2. TDS : table des symboles



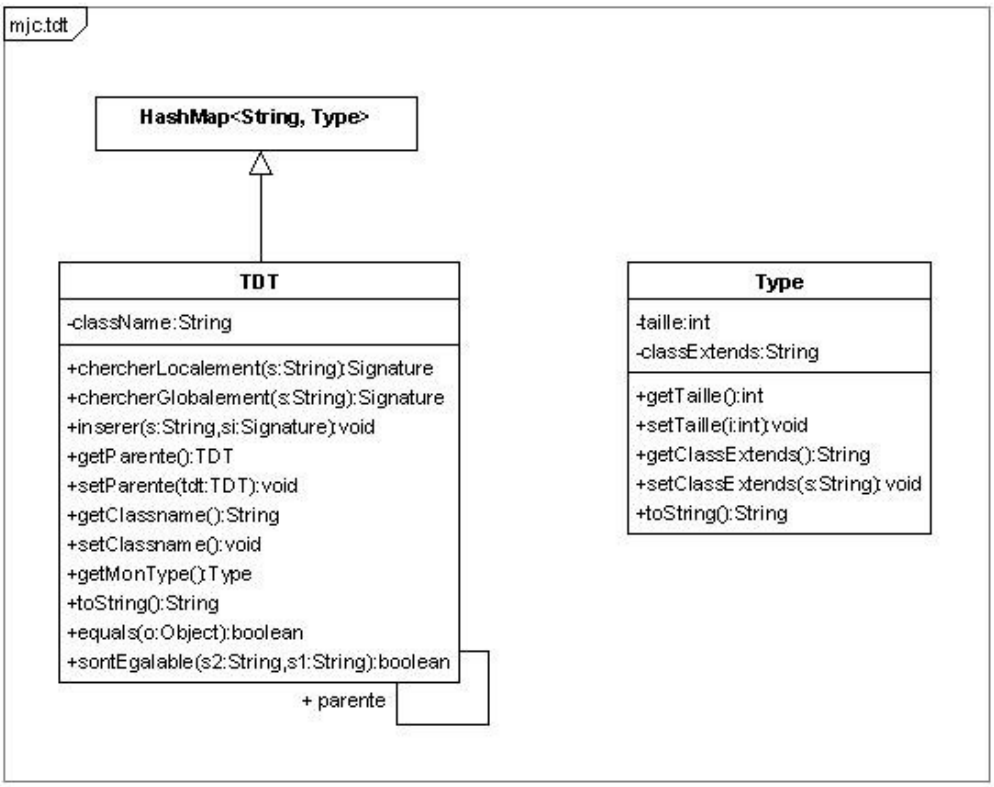
Nous avons adopté le principe de la table des symboles hiérarchique. Ainsi, lorsque nous sommes dans le corps de la méthode, la table des symboles courante a pour table parente celle de la classe où est définie la méthode.

3. TDM : table des méthodes



La table des méthodes a été créée dans l'idée d'accepter la surcharge, c'est-à-dire qu'on enregistre une méthode dont le nom prend en compte sa signature (par exemple : translater_int pour la méthode transater(int dep) de la classe Point).

4. TDT : table des types



Cette classe comprend toutes les classes connues par la classe que l'on compile (c'est-à-dire les classes importées). Nous avons bien différencié l'import de l'include en incluant le code des classes importées au moment du linkage.

III. Ce qui a été réalisé

1. La notion d'importation de classe

L'importation des classes se fait au moment du linkage. En effet, quand on génère le code d'une classe, on l'enregistre dans un fichier « nomClass_code.extension », puis si il existe une méthode main, on inclut récursivement le code de toutes les classes importées et de toutes les classes qu'elles importent à leur tour (un seul import par classe). On écrit le tout dans « nomClass_exe.extension »).

2. Définition d'une classe et du type associé

On a considéré que chaque objet était un pointeur vers un champ du tas. Ainsi, quand on initialise un objet, on fait un malloc de la bonne taille (trouvée dans la TDT quand on rencontre un new) et le constructeur rempli l'objet. Puis quand on appelle une méthode sur cet objet, on met les paramètres dans la pile et on fait un call de l'étiquette auto_tardif qui gère ensuite l'appel à la bonne méthode nomClassObjet_nomMéthode_typeArgsConcaténés.

3. L'héritage et le sous-typage associé

Grâce à l'attribut super que nous définissons comme étant un attribut de la classe héritée, nous avons la notion de sous-typage. La représentation mémoire se fait de la sorte : nomClasse, super (adresse de l'objet parent), attributs. Ainsi, pour accéder à l'adresse réelle d'un attribut x de la classe parente, on transcrira this.super.x.

Cette représentation mémoire implique de redéfinir automatiquement toutes les méthodes parentes lorsqu'elles ne le sont pas dans le code à compiler. Cette méthode systématique sera de la forme :

```
public m(les_param){
        [return] super.m(les_param);
}
```

4. Les opérations arithmétiques et booléennes

Toutes les opérations arithmétiques et booléennes ont été implantées, puisqu'il suffit de charger les valeurs dans le bon ordre puis de faire un SUBR + opération.

5. L'accès aux attributs d'une instance

Comme tout objet est un pointeur vers le tas, pour accéder à un attribut d'une instance, il suffit de connaître le décalage qu'il faut ajouter pour avoir l'adresse du bon attribut. Ce décalage est stocké dans la TDS de la classe associée.

6. L'appel de méthodes par liaison tardive

Dans la fonction auto_tardif, nous concaténons le type de l'objet avec le nom de la méthode appelée, et par un simple switch, nous renvoyons l'adresse réelle dans le code de la méthode.

7. La visibilité des attributs et des méthodes

Tous les fichiers étant dans le même répertoire, on ne gère pas les notions de packages, donc protected = public. On stocke donc dans la TDS la visibilité des attributs et dans la TDM celle des méthodes. Puis si on veut accéder à un attribut ou à une méthode privée, on vérifie qu'on soit dans la bonne classe.

8. L'utilisation du this

On traite le this comme si c'était un paramètre caché de chaque méthode non statique. Si on affecte un attribut du this (this.x = 24), on chargera son adresse dans la pile avant de faire des opérations, ce qui revient au même que x = 24 puisque dans ce cas on recherche l'adresse de l'objet et on la charge. De même pour les appels de méthodes sur le this.

9. L'utilisation du super

On rajoute le super comme étant un attribut de la classe qui étend l'autre. Ainsi, dans le tas, le super est matérialisé par un pointeur vers l'objet parent. Il faut obligatoirement initialiser le super dans les constructeurs des classes étendues de la manière suivante :

```
super = new ClasseParente(params)
```

IV. Les limites et améliorations possibles

1. L'ordre des méthodes et des classes

Le compilateur ne gère pas l'utilisation d'une classe qui n'a pas été compilée ou l'utilisation d'une méthode (ou d'un attribut) qui est définie plus loin dans le code.

On aurait pu ajouter cette fonctionnalité de plusieurs manières : soit en sauvegardant dans une liste toutes les méthodes appelées et les enlever quand elles sont définies, puis si il reste une méthode dans cette liste, on lève une erreur (problème : on n'a plus la ligne de l'erreur). La deuxième façon aurait été de faire deux passages : un pour remplir les tables, et un autre pour générer le code (ou remplir des trous dans le code si on en génère une partie au premier passage).

2. Les attributs statiques

Les attributs statiques ne sont tout simplement pas gérés. Il aurait fallu prévoir un emplacement unique dans le tas vers lequel on pointe à chaque fois qu'on utilise la classe associée.

3. La surcharge

On aurait pu gérer la surcharge en créant une fonction qui renvoie la distance entre deux signatures : si deux méthodes ont le même nombre de paramètres, la fonction renvoie 0 pour chaque paramètre de même type + un coefficient à chaque fois que les paramètres ne sont pas les mêmes.

V. Organisation du travail et conclusion

Pour la réalisation de ce projet, nous nous sommes organisé de la manière suivante :

- au début, nous avons tous réfléchi à la conception de la TDS et aux spécifications globales du projet.
- puis nous nous sommes plus réparti les différentes tâches (Julien pour la TDS, Loïc pour le typage et David pour la génération de code), sachant que chacun à traité des morceaux de différentes partie.

Au niveau des outils, nous avons utilisé le générateur de compilateur EGG ainsi que son plugin pour Java. Pour gérer le travail en groupe, nous avons utilisé un dépôt SVN de GoogleCode.

En conclusion, nous pouvons dire que nous avons réussi à concevoir en un peu plus d'un mois un compilateur fonctionnel, capable de compiler des programmes évolués, utilisant des concepts propres aux langages objets comme la liaison tardive, l'héritage ou la surcharge.

Il resterai quelques points à améliorer sur notre compilateur pour qu'il puisse traiter la totalité des cas du langage Micro-Java et générer un autre langage que TAM, mais ce projet nous a tout du moins permis d'approfondir nos connaissances en sémantique et TDL, de nous rappeler la programmation en assembleur ainsi que d'améliorer notre capacité de travail en groupe.