École Nationale Supérieure
d'Électrotechnique, d'Électronique, d'Informatique, d'Hydraulique et des Télécommunications

Rapport de projet

Projet de Sémantique et Traduction des Langages
Compilation du langage Micro-Java

Philippe Leleux
Quentin Glinel-Mortreuil
Arthur Manoha

10 juin 2013,

Résumé : Le but de ce projet est d'écrire un compilateur pour le langage Micro-Java. Pour cela nous devons étudier plusieurs partie dépendant les unes des autres :

- Définition et test de la syntaxe du langage
- Gestion de la table des symbole ou traitement des identificateurs
- Contrôle du type
- Génération du code pour la machine TAM

Table des matières

I Présentation du projet

Pour ce projet nous avons considéré que les connaissances acquises en cours sont connues et donc ne seront pas ré-expliquées dans le rapport comme par exemple le principe de la Table des Symboles. Ainsi nous présenterons, de manière plus ou moins détaillée, les choix que nous avons fait et ce que nous avons implanté

I.1 Introduction

Le but de ce projet est d'appliquer les principes vus en Tds et Tps pour écrire un compilateur pour le langage Micro-Java, sorte de langage Java simplifié. Pour effectuer cela, nous avons utilisé l'outil EGG que nous verrons plus loin.

Nous devons nous baser sur les techniques vues en cours :

- gestion de la table des symboles
- contrôle de type
- génération de code assembleur

I.1.1 Principe du compilateur

Ce compilateur doit respecter le principe des compilateur normaux, c'est à dire vérifier la syntaxe et la sémantique du langage spécifié et signaler les erreurs grâce à un message donnant assez d'informations pour pouvoir débugger le programme. La sortie du compilateur doit également fournir un fichier en langage assembleur.

Le principe du compilateur est simple : en entrée on prend un fichier écrit en langage Micro-Java et le résultat doit être un fichier contenant le code assembleur utilisable par une appropriée. Nous nous contenterons dans ce projet de générer du code spécifique à la machine TAM tout en gardant à l'esprit que l'application doit être portable et donc pouvoir être adaptée à un autre assembleur facilement

I.1.2 Fournitures

Tous les outils mis à notre disposition sont ceux déjà utilisés pendant les Tps :

- la grammaire LL(3) de Micro-Java contenue dans le fichier MJAVA.egg à compléter
- le couplage de EGG et java ainsi que le guide utilisateur de EGG
- la machine TAM avec sa documentation et le débeugueur itam
- un plugin EGG pour l'IDE eclipse (dont nous ne nous sommes pas servis en raison d'une obscure erreur interne empêchant le téléchargement sur les ordinateurs)
- un fichier makefile pour compiler l'application

I.1.2.a Fonctionnement de EGG :

L'outil EGG fourni prend en entrée un fichier décrivant la grammaire du compilateur à générer ainsi que les classes java que nous lui fournissons pour la gestion des types et de la table

des symboles ainsi que la génération de code. EGG crée un analyseur lexical et un analyseur syntaxique et sémantique descendant. Il suffit pour cela d'écrire la grammaire attribuée, actions à effectuer : de la gestion des identificateurs à la génération de code.

I.1.2.b Langage Micro-Java :

La langage Micro-Java est un langage Java simplifié. Dans ce langage :

- toutes les classes seront écrites dans un même fichier
- toutes les classes utilisées seront supposées définies plus haut dans le fichier, il n'y aura pas de référence en avant.
- La redéfinition d'une méthode dans une sous-classe doit être possible, mais la surcharge d'une méthode n'est pas à prendre en compte.
- Aucune visibilité (public, private, protected) des méthodes, classes et attributs n'est définie. Ainsi tout est considéré comme public.
- Ce sont les différences principales dites dans le sujet mais on peut également ajouter l'absence d'interface ou de classe abstraite etc...

Exemple : Factorielle

```
class Fact {
	int fact(int n) {
		if(n <1) {
			return 1;
		} else {
			return n * fact(n-1);
		}
	}
}
class TestFact {
	void main() {
		Fact f = new Fact();
		int x = f.fact(6);
	}
}
```

Toutes les contraintes de bases demandées et celles que nous avons choisies de traiter sont présentées dans la sections suivantes.

I.2 Spécifications

I.2.1 Spécifications minimales

La grammaire du langage Micro-Java à compiler est donnée (ou en tout cas une grammaire de base que l'on peut compléter ou dans laquelle on peut enlever des règles) et il faut la traiter entièrement. Nous devons nous occuper de la gestion de la table des symboles, du contrôle de type et de la génération de code de cette grammaire.

Il y a cinq points qui forment les concepts Java minimaux demandés dans ce projet :

- La définition d'une classe et du type associé

- L'héritage et le sous-typage associé
- Quelques opérations arithmétiques et booléennes
- L'accès aux attributs et méthodes d'une instance de classe
- L'appel de méthodes par liaison tardive

I.2.2 Spécifications traitées

Nous avons pu traiter les 5 points de base demandés dans ce projet, en tout cas pour ce qui est de la gestion de la table des symboles et du contrôle de types. Cependant, il y a des points sur lesquels nous avons un peu différé de cette configuration minimale :

- les mots-clés this et super ne seront pas pris en compte dans la syntaxe du langage.
- L'instruction tant que est prise en compte par le langage (bien que non testée sur la génération de code)

I.3 Structure de l'application

Le projet est divisé en plusieurs parties, toutes réunies dans le dossier tdl. On retrouve tout d'abord le dossier ex_mj où nous avons mis tous nos tests, nous le verrons dans la section spécifique aux tests. Ensuite, on trouve le dossier src qui contient tous les outils et sources de l'application :

- eggc.jar, l'exécutable contenant l'outil EGG lui-même.
- Le dossier properties : contient tous les messages d'erreur possibles pouvant être lancés lors de la compilation.
- MJAVA.egg : le fichier contenant les règles de production et la grammaire attribuée du langage Micro-Java.
- le dossier mjc contenant tous les package Java nécessaires :
  - mjc.compiler : contient tout le nécessaire à la génération du compilateur lui-même (main utilisé etc..)
  - mjc.egg : contiendra la traduction du fichier MJAVA.egg en langage Java. Il est généré automatiquement par EGG durant la compilation.
  - mjc.gc : Classes java utilisées pour a génération de code. C'est ici que l'on trouve l'AbstractMachine et la classe TAM. C'est ici que l'on rajoutera une classe si l'on veut générer du code pour un autre assembleur que TAM (il y a d'ailleurs SPARC et X86, qui ne sont dans cette application que des copies de TAM pour éviter les erreurs de compilation).
  - mjc.tds : Ce package contient toutes les classes que nous avons implantées pour la gestion de la Table des Symbole.
  - mjc.type : Idem mais pour la gestion des types.

II Conception et classes java

II.1 Choix généraux de conception

Au tout début de ce projet, avant même d'écrire la moindre ligne de code, il faut se demander comment on va représenter le java avec les outils que nous avons étudié durant l'année. Comment la table des symboles va-t-elle prendre en charge classes, attributs et variables ? Comment les types seront représentés et surtout le type d'une classe déclarée

dans le fichier ? Comment permettre à l'application de rester portable à d'autres langages que la machine TAM ? Un objet qui hérite d'un autre possèdera-t-il tous les champs ou un pointeur vers un objet de type parent ?

Pour répondre à la première question, il nous a paru logique et nous avons décidé à l'unanimité de **séparer l'idée de table des symboles en deux**. D'un côté nous aurons la **table des classes (TDC)**, contenant les classes, elle-même composées d'attributs et de méthodes, et de l'autre une **table des symboles (TDS)** contenant les variables du fichier. De plus **chaque bloc possédera sa table des symboles** tandis que la table des classes est partagée par tout le fichier, cela changera donc les fonctions de recherche de chaque table. Ainsi il n'y aura qu'une unique fonction de recherche au sein de la table des classes et deux fonctions (locale et globale) pour la table des symboles, comme vu en TP/TD.

Ensuite, les types seront représentés comme nous l'avons toujours fait en TP par **un objet DTYPE dont hériteront les types concrets**. Les types des classes déclarées dans le fichier quant à eux seront représentés par **un type POINTER héritant de DTYPE** et qui aura **en attribut la classe de l'objet**. On reprend ainsi le principe de Java lui-même où pour ainsi dire tout est objet et en réalité tout est pointeur.

Les Diagrammes UML représentant les classes pour les types et pour la table des symboles seront donnés plus loin.

La génération de code se présente (déjà dans les classes fournies en début de projet) comme une **classe abstraite Abstract Machine implantée par 3 autres classes : TAM, SPARC et X86** représentant les 3 langages assembleurs que l'on aurait pu nous demander d'implanter. En fait, la classe AbstractMachine représente une machine de génération de code utilisant un langage abstrait. C'est **un objet de ce type que l'on utilisera dans MJAVA.egg** pour générer du code, ainsi pour changer de langage assembleur, il suffit de créer un objet AbstractMachine de type réel TAM pour du TAM, SPARC pour du SPARC etc... Nous n'implanterons cependant dans notre projet que la classe TAM.

Nous avons choisi au niveau de l'**héritage**, de **copier toutes les méthodes et attributs de la classe héritée** au moment d'enregistrer la classe comme parente dans la nouvelle classe déclarée. Ainsi on s'assure de pouvoir accéder à tous les attributs et toutes les méthodes et il faudra également s'assurer que l'**on pourra remplacer une méthode de la classe parente mais que la surcharge n'est pas prise en compte**. De plus, la **compatibilité des types** est **gérée sur les héritages en chaîne**.

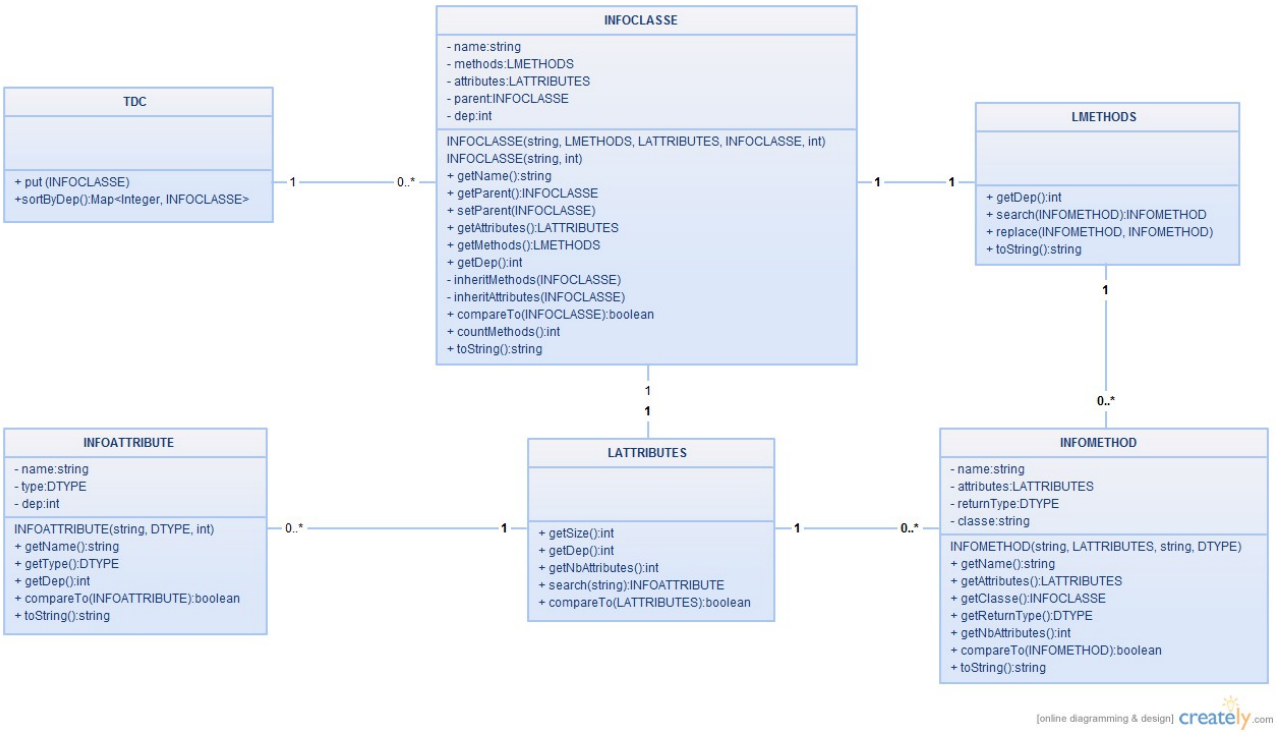
Il y a quelques autres points qui nous ont paru être objet à débat mais qu'il a fallu imposer :

- Un unique main doit être défini dans tout le programme.
- On doit toujours vérifier le retour des fonctions.

II.2 Table des Classes

II.2.1 Diagramme

Dans ce paragraphe, nous allons nous intéresser plus précisément à l'implantation de la table des classes, sans pour autant expliciter les structures Java sous sus-jacentes. Voici le diagramme des classes la représentant :



Il est logique de dire qu'une classe est composée d'attributs et de méthodes, elles-même composées d'"attributs" – qui sont en fait les paramètres de la méthode – et c'est ainsi que logiquement le diagramme se construit.

Les classes ont chacune des attributs spéciaux (ex : la classe parente parent pour une classe) et des méthodes propres cependant elle partagent toute un point commun : la méthode compareTo.

II.2.2 Fonction de comparaison

La méthode compareTo renvoie vrai si deux objets sont comparables mais à partir de quand peut-on dire qu'ils le sont ?

Comparaison entre méthodes : cette comparaison est largement simplifiée dans le fait que l'on autorise pas la surcharge (auquel cas il faudrait comparer les paramètres de deux méthodes pour pouvoir les comparer) mais ici il suffit de comparer leur nom.

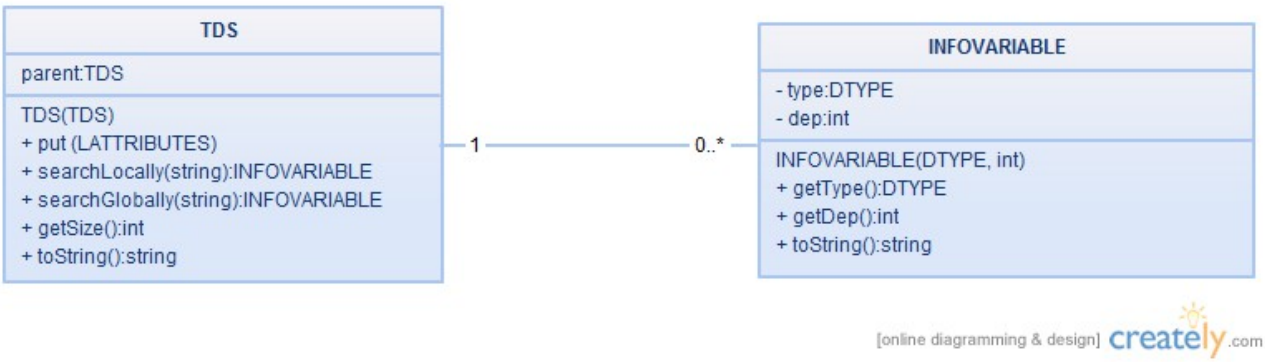
Comparaison entre attributs : Ici, même si nous n'avons pas encore parlé de type, la comparaison se fera plutôt sur ce point lorsque l'on parle de paramètres de méthode, en effet le nom du paramètre donné peut ne plus rien avoir à voir avec le nom lors de la déclaration, seul le type compte. Par contre, lorsqu'il s'agira de rechercher un attribut, c'est une autre histoire et ce sera sur le nom cette fois que l'on se concentrera, en effectuant une recherche type de table des symboles.

Comparaison entre classes : Pour les classes nous n'allons regarder que le nom, en effet le nom d'une classe est unique et cela suffit. Cependant se rajoute un problème qu'il faut prendre en compte dès maintenant : l'héritage. On voudra dans la suite pouvoir affecter à un objet d'une classe A une instance d'une classe B héritant de A (A a = new B();). Ainsi il va falloir chercher dans tous les parents de la classe B si A est présent ou non. C'est cette recherche en cascade qui sera spéciale

dans la règle de comparaison des classes. **Attention !** Cette règle, au contraire des autres, n'est plus commutative et il faudra être prudent dans la suite.

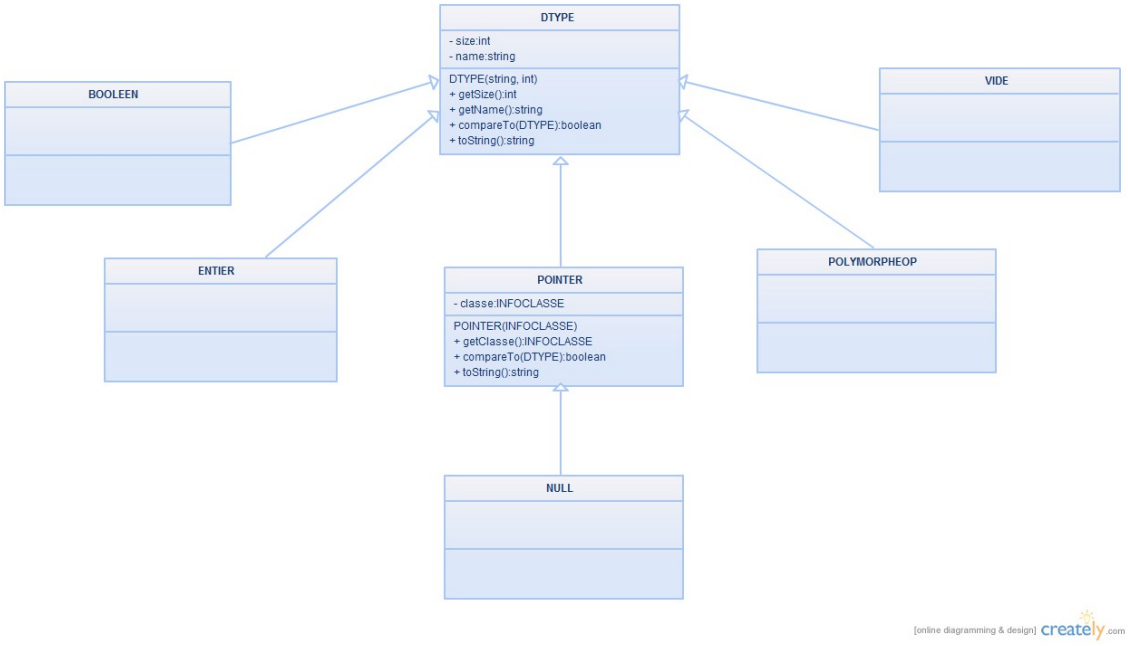
II.3 Table des Symboles

La Table des Symboles a une structure bien moins complexe que la table des classes : c'est une table qui va stocker des variables et permettre de recherche une variable dans elle-même ou bien dans ses parentes en remontant en cascade. Ainsi on stocke la table parente dans la table actuelle. Voici le diagramme de classe représentant cette relation :



II.4 Contrôle de types

Pour le contrôle de type nous avons donc repris le principe vu sur le langage BLOC en Tds et Tps : une classe abstraite DTYPE qui sera implantée par les classes de types concrets définie toutes les méthodes utiles et notamment la fonction de compatibilité entre types. Cette méthode sera valable pour tous les types sauf un : le type POINTER qui redéfinie la méthode pour prendre en compte les contraintes liées aux classes déclarées dans le fichier. Le diagramme suivant représente les relations entre les différentes classes du système de type :



La plupart des classes de ce système de typage sont très simples et ne font qu'appeler le constructeur de la classe mère (DTYPE) en donnant le nom du type (ex : "booleen" pour la classe

BOOLEEN) sauf la classe POINTER. Mais il y a une autre classe, bien que simple, dont il faut préciser le pourquoi du comment.

A quoi sert donc la classe POLYMORPHEOP ? Cette classe est en fait là pour représenter le comportement de certains opérateurs qui permettent des relations entre plusieurs types différents au contraire des opérateurs définis sur un seul type comme par exemple l'opérateur && défini seulement pour les booleens. = ou != font partie de ces opérateurs dits polymorphes puisque l'ont peut tester l'égalité de deux objets du moment qu'ils ont le même type. Ces type est pris en compte dans la fonction de compatibilité des types dans DTYPE.

Intéressons-nous maintenant à la classe POINTER. Cette classe fait deux choses. D'abord elle stocke la classe de l'objet dont elle représente le type, ainsi on crée un type spécial pour cette classe. Il faut alors ensuite redéfinir la méthode de comparaison compareTo pour tester que le deuxième objet est bien un pointeur et que, si c'est le cas, les deux classes pointées sont bien identiques, auquel cas les deux types sont compatibles.

II.5 Génération de code

Concevoir la génération de code c'est avant tout imaginer ce que pourra donner le code généré automatiquement d'une classe. Ce fut l'une des étapes les plus compliquées du projet puisque prendre un programme et tenter de le traduire en TAM ne suffisait pas à pouvoir appliquer des règles automatiques de génération de code. Il nous a fallu nous baser sur ce que nous avions fait en cours et réfléchir à comment l'adapter au Micro-Java, sans oublier que nous devrons ajouter une table d'indirection que nous verrons un peu plus loin, pour implanter la liaison tardive.

Ainsi à force de persévérance nous avons pu nous décider pour une certaine structure de code généré : tdv – Supermain – npe – methodes/constructeurs/méthode main

Exemple : Factorielle

```
; Table des virtuelles : tableau d'indirection des méthodes des classes
        ; INFOCLASSE Fact
        LOADA Fact_fact_entier

        ; Appel de la fonction main
        CALL (LB) TestFact_main
         LOADL "FIN DU PROGRAMME !\n"
        SUBR SOut
        LOADL "VALEUR DE RETOUR : "
        SUBR SOut
        SUBR IOut
        LOADL "\n"
        SUBR SOut
        HALT

; En cas d'exception
_NPE_
        LOADL "Null Pointer Exception"
        SUBR SOut
        HALT
```

;Méthodes et constructeurs

```
        ; Constructeur par défaut de la classe Fact
Fact_defaut
        code du constructeur par défaut
        RETURN (1) 0

        ; Méthode fact de la classe Fact
Fact_fact_entier
        code de la fonction fact (comportant une structure si)
        RETURN (1) 2

        ; BLOC SINON
_X_0
        code du bloc sinon
        RETURN (1) 2

        ; FIN SI
        ; Constructeur par défaut de la classe TestFact
TestFact_defaut
        code du constructeur par défaut
        RETURN (1) 0

; Méthode main de la classe TestFact
TestFact_main
        code de la méthode main (avec appel à la méthode fact)
        RETURN (1) 1
```

Il nous a fallu alors définir dans AbstractMachine (toujours pour garder une séparation avec le code TAM lui-même) les entêtes des fonctions dont nous aurions besoin sans oublier les 2 méthodes fournies writeCode et getSuffixe :

a) Une méthode pour la génération de la Table des Virtuelles, table d'indirection des méthodes, dont on reparlera plus loin.

b) Une méthode pour appeler la méthode main qui génère en fait une sorte de super classe main, commune à toutes les applications. Cette partie du code va démarrer le programme, appeler la classe main et afficher la fin du programme ainsi qu'une valeur de retour (pas forcément significative) à la fin de la classe main.

c) Trois méthodes pour générer le code, le retour et l'appel d'une méthode.

d) La génération du constructeur par défaut qui sera appelé par chaque constructeur non trivial. Il faut une méthode pour générer ces constructeurs également (en effet, la génération est différente) et enfin une méthode pour générer l'appel à un constructeur.

e) La génération de la vérification de Null Pointer Exception – NPE - (on vérifie que l'objet que l'on a créé et dont on récupère l'adresse n'est pas null) ainsi que la génération du code de la section qui sera appelée en cas de NPE.

f) La génération de code pour l'affectation, la déclaration de variable ainsi que la libération de mémoire.

g) La génération d'adresse pour les variables, paramètres et attributs ainsi qu'une méthode spéciale pour l'empilement de l'adresse de l'objet courant.

h) Génération de code pour l'accès à une variable, un paramètre ou un attribut.

i) L'écriture et la lecture en mémoire (sans oublie qu'une donnée qu'on lit que être une valeur ou une adresse)

j) Génération de code pour les structures de contrôle (basé sur de la génération d'étiquette et leur appel en TAM).

k) La génération de valeurs basiques : null, vrai, faux ou encore un nombre entier.

l) enfin la génération des différents opérateurs applicables que l'on générera en utilisant les routines déjà créées en TAM.

« Oui mais c'est bien beau tout ça mais la spécification dans tout ça ? » ET bien en fait, sans nous en rendre compte avec ces bases là, tout ou presque est fait et il n'y a plus qu'à appeler tout ça dans la Grammaire Attribuée. Mais regardons un peu ce qu'il reste à faire pour répondre aux 5 points de la spécif.

II.6 Respect de la spécification

II.6.1 Définition d'une classe et du type associé

Et bien cette question est en réalité presque réglée : pour définir une classe, il suffit de stocker dans la table des classes un nouvel objet INFOCLASSE, en enregistrant toutes ses méthodes et ses attributs. Quant au sous typage associé, il suffit de donner comme type POINTER avec en attribut la classe stockée dans la Table des Classes. Ce n'est pas plus compliqué que ça.

Il va falloir cependant faire passer certains attributs sémantiques que nous verrons dans la prochaine partie.

II.6.2 Héritage et sous-typage associé

L'héritage est également en grande partie réglé et tient principalement en deux points.

D'abord, comme on l'a déjà dit plus tôt, on copie toutes les méthodes et les attributs de la classe dont on hérite dans la nouvelle classe, ainsi on aura pas plus de problème à accéder à un attribut de la classe hérité qu'à un attribut de la classe courante. De cette manière on effectue l'héritage effectif.

Le deuxième point, le typage a en réalité été réglé grâce à la fonction de compatibilité des types. En effet, lors d'une affectation, une déclaration on vérifie que l'objet donné est bien du même type que l'objet affecté ou alors qu'il est compatible vis-à-vis de l'héritage.

Dans la grammaire attribuée, il n'y aura alors rien à ajouter de plus que la vérification du typage normale.

II.6.3 Opérations arithmétiques et booléennes

Pour ce qui est de la gestion des opérations arithmétiques et booléennes, on a déjà réglé les problèmes de fonction de compatibilité et de types car les opérateurs auront également un type. Mais comment ? Il va en réalité falloir faire passer un attribut sémantique « type » à presque tous les objets pour pouvoir géré tout cela.

Quant à la génération de code des opérateurs, elle correspond en réalité à l'utilisation

des Subroutines de TAM que l'on appellera directement (ex : addition ==> SUBR Iadd).

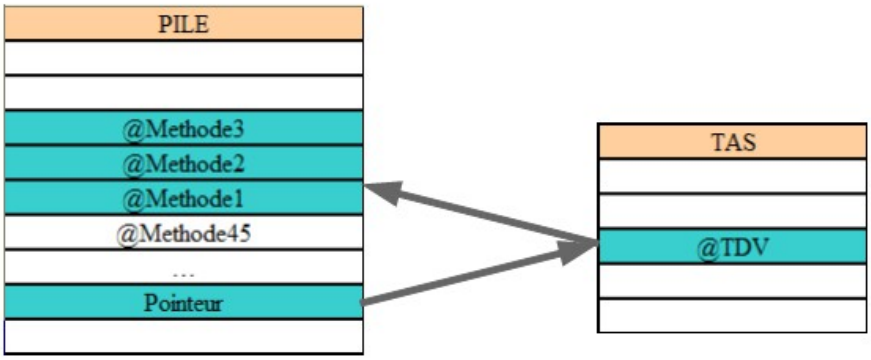
II.6.4 Accès aux attributs et méthodes d'une instance de classe

Pour ce qui est de l'accès aux attributs et méthodes d'une instance de classe, tout va se jouer par passage d'attribut sémantique là encore. En effet, il va falloir pouvoir savoir quel est l'objet courant pour avoir son type et tester la présence des attributs et méthodes, or on peut changer de classe en cascade par appel de méthode. Cette gestion va être compliquée et nécessiter une étude détaillée que nous verrons dans la prochaine section.

II.6.5 L'appel de méthodes par liaison tardive

Qu'est-ce que l'appel de méthode par liaison tardive ? Lorsque l'on déclare un objet, il est impossible de savoir de quel type il sera plus loin dans le programme à cause de l'héritage. Ainsi il se pose le problème de savoir quel méthode on doit appeler, est-ce la méthode de la classe parente ou celle de la classe fille ?

On va donc devoir donner son type réel à l'objet au fur et à mesure du déroulement du programme pour savoir quelles méthodes on appellera et on fera appelle à une Table des Virtuelles : table d'indirection des méthodes. Cette table est enregistrée dès le départ et contient les étiquettes des méthodes d'une classe unique. L'adresse de cette table est enregistrée dans chaque objet créé et sera changé au besoin.



Lors de l'appel d'une méthode, on commence par empiler le pointeur duquel on applique la méthode, ainsi que tous les paramètres de la méthode a l'envers afin de les dépiler a l'endroit. Ensuite, on récupère l'adresse de la Table des Virtuelles, qui est l'adresse de la base de la classe dans le tas. La lecture de la valeur stockée à cette adresse nous donne l'adresse de la première méthode de la classe dans la pile, et en additionnant cette l'adresse avec l'indice de la méthode (déplacement de la méthode) on obtient l'adresse réelle de la méthode.

III Grammaire Attribuée

Nous ne nous intéresserons pas exprès à certains points en détail, ceux-ci ayant été vu et revu en Tds/TPs comme par exemple la gestion des identificateurs ou le contrôle de type. Nous n'en parlerons alors que de manière concise à moins que ce ne soit des éléments spécifiques au Micro-Java.

III.1 Gestion des identificateurs

La gestion des identificateurs se fera exactement de la même manière que nous l'avons fait avec le langage BLOC :

- Chercher les règles de production référençant des identificateurs
- Déterminer les infos à associer à ces différents noms : présenté dans la partie 1.
- Attributs sémantiques et actions sémantiques créant et mettant à jour ces infos

On ajoute plusieurs attributs sémantiques pour garder les informations d'une règle de production à l'autre :

- **Tdc** : on passe la table des classes à toute l'application.
- **Classe** : de même on va garder partout une trace de la classe courante.
- **parent_classe** permet de transmettre la classe parente pour enregistrement.
- **Method** : de même que la classe, on va garder partout une trace de la méthode dans laquelle on se trouve.
- **Params** : transmission des paramètres lors de la déclaration.
- **Tds** : on passe la tds à tout moment, en en créant une pour chaque bloc.

Lors d'une **déclaration**, on va vérifier que l'élément que l'on crée (variable, classe, attribut ou encore méthode, affilié à la bonne table ou classe) n'est pas présent. Pour les classes, une classe de même nom de doit pas être présente du tout, pour les variables, elle ne doit pas être ni une variable de la TDS actuelle, ni d'aucune TDS parente, ni un paramètre, ni même un attribut de même nom.

Lors d'une **affectation**, on doit cette fois vérifier la présence de la classe ou la variable.

- Ajouter les actions sémantiques de transmission

Remarque : C'est par ces principe simple que l'on effectue la définition d'une classe et l'enregistrement des attributs et méthodes.

III.2 Contrôle de type

Ici aussi, nous avons suivi la méthodologie du cours :

- Faire la liste de tous les types : présenté dans la partie 1.
- Définir une fonction de compatibilité des types : idem.
- Définir les règles de contrôle et de construction des types.

Les règles de contrôle de type que l'on va vérifier sont en réalité définies dans les properties avec les messages d'erreurs. Ainsi, on va par exemple vérifier les compatibilités de type lors de déclarations, lors d'opérations ou encore lors d'affectations. Il faudra également vérifier le type de retour des méthodes et le types des paramètres en appel. Nous en profiterons pour vérifier la présence d'un retour si besoin est et que ce retour est unique.

On donne le type à une variable lors de sa déclaration simplement en observant le type déclaré ou alors celui qu'on lui donne si il y a un cas spécial d'héritage.

- Écrire une grammaire attribuée associée

Définissons les attributs sémantiques dont nous avons besoin :

- type : passé à tous les éléments dont on peux vouloir le type, c'est à dire toute expression ou type (E, ER, FACTEUR, TYPE, F etc...)
- op_type : type de l'opérateur défini par OPREL, on a besoin de cet attribut car le type de l'opérateur peut-être différent de celui des membres gauches et droits (l'opérateur peut être POLYMORPHEOP)
- returnType : donne le type de retour d'une méthode
- hasReturn : compte au fur et à mesure du code le nombre de return rencontrés, cela permettra par la suite de vérifier le retour.

III.3 Opérations arithmétiques et booléennes

Pour les opérations arithmétiques et booléennes, nous avons alors déjà tout défini au niveau du contrôle de type. Il suffira de rajouter l'appel aux méthodes de génération de code correspondant directement à ces opérations.

III.4 Génération de code

On rajoute des attributs sémantiques spéciaux pour la génération de code :

- dep : calcule et transmet le déplacement de certains non-terminaux.
- Machine : AbstractMachine passée à toutes les règles de production, c'est elle qui contient toutes les méthodes de génération de code.
- Est_valeur : caractérise le fait qu'une expression contient une valeur ou seulement une adresse (type de base ou classe construite).
- Code : Le code transmis que l'on transmet pour concaténation et écriture dans le fichier de sortie.

On a une action sémantique de génération de code (#gc) pour chaque règle de production même si c'est seulement pour renvoyer du code vide et en réalité on se base sur une compréhension logique du code Micro-Java pour appeler telle ou telle méthode dans tel ou tel ordre.

Exemple : DEF → void ident paro PARFS parf BLOC #gc ;
On a affaire à une définition de procédure, on va alors logiquement faire appelle à la méthode genMethode qui génère le code d'une méthode et enregistrer le résultat dans l'attribut code que l'on transmet:

#gc : DEF^code:= DEF^machine.genMethode(DEF^^classe.getName(), methode, BLOC^code) où methode est l'info de la méthode à générer et BLOC^code le code du corps de cette méthode.

III.5 Appel de méthode par liaison tardive

Pour l'appel de méthode par liaison tardive, comme on l'a expliqué, il va nous falloir un attribut sémantique représentant l'adresse de la Table des Virtuelles et que l'on calculera au fur et à mesure de la compilation. Cet attribut sera l'entier tdv que possèderont les non-terminaux DEFCLASSE et CLASSES.

III.6 Accès aux attributs et méthodes d'une instance de classe

Les règles de production qui définissent l'accès à un attribut ou une méthode sont toutes celles concernant le non terminal FX ainsi que ARGS et ARGSX.

Rien que pour ces règles, il faut ajouter des attributs et actions sémantiques spéciales :

- hclasse : pour FX, classe à laquelle on accède.
- param_new : pour ARGS et ARGSX, liste des paramètres de la méthode.
- prec_ident : pour FX, identifiant auquel on accède dans la règle de production précédente, il est l'ident de l'objet dans lequel on accède à un attribut ou une méthode.
- attr_dep : pour ARGS et ARGSX, déplacement du paramètres (numéros du paramètres dans la liste de paramètres d'appel)
- hcode : code hérité par FX.

IV Tests

Nous nous sommes efforcés au long de trouver les erreurs et de les régler, dans tout le rapport on peut retrouver ce que nous avons traité ou non de manière détaillé. Les tests présentés ici sont différents tests mettant en lumière les aspects fonctionnel de notre projet.

De plus, les fichiers sources ne seront pas inclus dans ce rapport, seulement le but et les résultats de ces tests qui peuvent être trouvés dans le dossier ex_mj du projet.

IV.1 Table des Symboles et Contrôle de Type

Exemple 1 : Factorielle (working/Fact.mj)

Cet exemple nous était fourni en sujet du projet. Nous avons gardé ce test comme test de base pour vérifier le fonctionnement du remplissage de la TDS et détection du typage. Tout se passe bien avec celle-ci et même pour la génération de code que nous verrons plus loin.

Exemple 2 : Détection d'erreurs dans la gestion de la TDS (error/error_tds.mj)

Ce test a pour but de montrer l'efficacité de la détection d'erreurs dans la gestion de la table des symbole. Il regroupe toutes les erreurs détectables par notre projet dans ce cadre :

- classe, identificateur non définie
- classe, attribut, paramètre, variable, méthode déjà défini
- accès à un attribut d'un type basique (absurdité s'il en est)
- un constructeur ne porte pas le nom de la classe
- un constructeur, une procédure a un retour
- retour de méthode absent

Exemple 3 : Détection d'erreurs dans le typage (error/error_type.mj)

De même qu'avec les erreurs liées à la TDS, dans ce fichier on va vérifier la détection des erreurs liées au typage :

- type affecté différent, même après prise en compte de l'héritage
- type de retour de méthode invalide
- les membres gauche et droit d'un opérande n'ont pas le même type
- le type du membre gauche et/ou du membre droit est incompatible avec le type de l'opérande.

Exemple 4 : Test de l'héritage (working/Musicien.mj)

Dans cette classe, nous testons l'héritage, c'est-à-dire la copie des attributs et méthodes de la classe mère et attribution du type réel, et tout se passe bien à priori. On observe bien que les méthodes sont héritées par la classe fille.

IV.2 Génération de code

Nous ne nous sommes pas réellement concentré sur le debuggage de la génération de code, préférant utiliser le temps qu'il nous rester pour améliorer le reste.

Exemple 1 : Factorielle (working/Fact.mj)

Le code généré avec cette classe marche.

Exemple 2 : Factorielle améliorée (working/Fact_mem.mj)

Nous avons voulu, en partant d'un exemple dont on est sûr qu'il marche tester plus d'opérations et pour cela, nous avons modifié la factorielle. Pour cela, une nouvelle classe a été ajoutée avec en attribut le résultat de la factorielle pour pouvoir voir le résultat du calcul. Nous avons ainsi ajouté l'appel de fonction.

Le résultat obtenu est mitigé. On obtient un code presque fonctionnel. « Presque » dans le sens où il comporte quelques petits problèmes d'adresse d'accès dans la pile. Ainsi il faut modifier à 3 endroits l'adresse à laquelle on accède.

Nous avons par contre observé un problème de rafraîchissement au niveau de l'affichage de la pile. Avec un code fonctionnel pour cet exemple, le résultat n'est réellement observable qu'en étape par étape sous itam. Si on lance l'exemple d'un coup, itam affiche un peu n'importe quoi.

V Conclusion

V.1 Limitations et améliorations possible

Après plusieurs tests, nous nous sommes rendu compte que la liaison tardive n'est traitée que lors de la déclaration et non d'une affectation, il nous faudrait plus de temps que nous n'en disposons pour régler ce problème. Il faudrait alors faire en sorte de changer le type de l'objet à une affectation où on affecte un objet de type différent.

Nous pensons qu'il y a des erreurs auxquelles nous n'avons pas pensé, bien que nous ayons traité les plus importantes, ainsi certaines erreurs courantes telles que la division par zéro ne sont pas prises en compte.

Le plus gros regret que nous garderons c'est de ne pas avoir le temps de pouvoir réellement débugger la génération de code qui reste peu fonctionnelle. Autant nous avons réussi à générer un code fonctionnel pour la factorielle, autant pour n'importe quelle autre classe il reste des problèmes

de décalage du code qui nous font appeler les mauvaises adresse et d'autres problèmes qui mériteraient encore quelques dizaines d'heures de travail.

Pour ce qui est des améliorations possibles, il faudrait commencer par autoriser la surcharge, on n'aurait qu'à changer la recherche de méthodes pour autoriser l'ajout d'une méthode avec un nom identique et des paramètres différents.

De plus, on pourrait imaginer l'ajout d'un autre fichier .egg pour pouvoir parcourir plusieurs fois le fichier durant la compilation, on pourrait ainsi faire de la référence en avant. Et tant qu'à effectuer plusieurs parcours du fichier, autant découper la compilation en plusieurs étapes pour rendre le code plus modulables et surtout plus clair ! En effet comment débugger un code qui fait plus de 2000 lignes ? Cela devient très long et fastidieux.

On pourra également imaginer faire plus de traitement au sein des classes Java et moins au sein du fichier .egg pour simplifier la lecture et surtout pouvoir modifier le fonctionnement plus efficacement.

V.2 Conclusion générale

Ce projet était dans la continuité de ce que nous avons vu en cours, nous avons pu reprendre toutes les notions et les ré-appliquer sur un autre langage, le langage Micro-Java. D'une manière plus générale, ce cours était très intéressant : c'est bien la première fois que l'on nous a donné l'occasion de réellement savoir ce qu'il se passe derrière la console lorsque l'on compile un fichier. L'étude des sémantiques est quelque chose d'essentiel pour tous ceux qui voudront s'orienter vers la recherche et 'informatique théorique.

Si nous avions juste une remarque de plus à faire, c'est le regret de ne pas avoir pu travailler à ce projet tout au long de l'année au fur et à mesure que l'on voyait les notions et que les appliquions en TP. D'un côté cela nous à permis de tout reprendre pour l'examen et de tout se remettre en tête mais d'un autre côté, nous avons eu moins de temps de réflexion. Au contraire de prendre du recul sur les compilateurs, il nous a semblé qu'avec le temps certaines notions étaient devenues floues alors que nous aurions pu les ancrer en les appliquant directement.

Ce projet nous a permis de mener à bien le développement d'une application en équipe où chaque partie est courte et simple mais où l'ensemble devient très complexe - ce que nous vivrons très très prochainement dans le monde professionnel. Ça a été très compliqué de se partager le travail de manière efficace et surtout sans s'embrouiller. Nous avons pour la gestion du versionnage utilisé la plate-forme BitBucket avec un dépôt Mercurial.