RAPPORT PROJET COMPILATION

Etudiants: Hugo BOMMART
Miaobing CHEN
Leo GALMANT
Manfred MADELAINE
Cédrick RIBET

Choix d'implémentation

Nous avons dès le départ décidé que les blocs du programme seraient représentés par une structure d'arbre telle qu'elle nous était donnée au début du projet. La première étape a donc été de créer l'arbre représentant le bloc principal. Pour cela, nous avons créé plusieurs étiquettes qui permettent de se retrouver dans l'arbre. Par exemple, pour une affectation, l'arbre résultant possède l'étiquette "AFF", son fils gauche correspond à l'opérande gauche de l'affectation et son fils droit à l'opérande droit.

Nous avons utilisé 5 structures différentes pour stocker nos différents éléments: Class, Method, Attribut, Heritage (elle ne nous sert qu'à la création de classe filles) et la structure "Tree" décrite précédemment.

Dans la structure Class, on stock :

- son nom, sous forme de chaîne de caractère,
- explicitement les paramètres de son constructeur sous forme de pile d'Attribut,
- sa classe mère si elle existe sous forme de pointeur sur Class,
- la liste de ses méthodes sous forme d'une pile de Method,
- la classe suivante, pour former une pile de classe,
- le nombre d'attributs, le nombre de méthodes et,
- le rang de cette classe (utilisé lors de la GCI).

Cette structure permet notamment de représenter tout le programme hormi le bloc principal par la première classe du programme. Depuis cette première classe, modélisée par une structure Class, on peut accéder à toutes les autres (grâce au système de pile) ainsi qu'à l'ensemble de ses éléments.

Pour un Attribut on stock :

- son nom, sous forme de chaîne de caractère
- son type, sous forme de pointeur sur Class
- son expression, sous forme de pointeur sur un arbre
- l'attribut suivant, pour former une pile d'attribut
- son rang

Pour une Method on stock :

- son nom, sous forme de chaîne de caractère
- sa signature, sous forme de pile d'Attribut
- un booléen "redefinition" qui vaut TRUE si la méthode est préfixée par "override"
- ses instructions, sous forme d'arbre

- son type de retour, sous forme de pointeur sur Class
- la méthode suivante, pour former une pile de Method
- son rang

La construction de ces différents éléments se fait à partir d'une variable globale firstClass qui sera utilisé tout au long du programme. Cela se fait via la méthode fillClass() qui va construire notre "environnement".

Les rangs permettent à la génération de code de savoir où stocker les éléments. Par exemple si une variable locale à un bloc est au rang n, on sait qu'il faut se déplacer de n pour obtenir sa valeur.

Ce qui marche

Gestion des types primitifs :

Pour la compilation, nous avons décidé de définir les types primitifs, Integer et String, en dur dans le code. On crée donc artificiellement deux structures Class qui correspondent aux deux types voulus (avec leurs méthodes respectives). Ce sont les deux premières classes de tout programme, mais on ne les voit pas apparaître dans le code. Nous avons bien sûr veillé à ce qu'aucune classe ne tente d'extends un des types primitif.

Lors de la construction des classes, nous faisons quelques vérifications. Celles-ci seront détaillées ci-dessous.

Vérifications contextuelles effectuées

- Vérification de portée : nous parcourons toutes les méthodes de toutes les classes, en prenant en compte un environnement. Dans une classe donnée, on ajoute à son environnement l'environnement de sa classe mère, ainsi que ses attributs. Lorsque l'on rentre dans une méthode, on garde l'environnement de la classe auquel on ajoute localement les paramètres et les éventuelles variables déclarées. Lorsqu'on tombe sur un IDVAR, c'est à dire un identifiant, on vérifie qu'il existe dans l'environnement. Sinon, on prévient l'utilisateur et le compilateur s'arrête.
On effectue cette vérification pour le bloc principal.

- Vérification d'affectation : on vérifie que le type l'opérande droit d'une affectation est égal modulo héritage au type de l'opérande gauche.

- Vérification d'appel : pour un appel de fonction, par exemple on imagine p1 un objet Point qui fait appel à la méthode exemple() : p1.exemple(2, x1, nom);
On vérifie d'abord que la méthode exemple existe dans Point ou dans les classes mères de Point. Si elle existe, on compare les types des arguments modulo héritage. Si un type est incorrect, on prévient l'utilisateur et le compilateur s'arrête.
Cette vérification est faite récursivement et les appels d'appel fonctionnent exactement de la même façon.

- Vérification mots-clé : on vérifie que les mots-clé this, super et result sont utilisés uniquement dans des corps de méthodes. On considère this, à la vérification contextuelle, comme une variable du type de la classe dans lequel il se trouve. Ainsi, si je fais this.exemple2(), on va chercher la méthode exemple2 dans la classe où this est utilisé.

Super est considéré comme une variable de la super-classe, et les vérifications sont les mêmes que pour this. Le programme s'arrêtera donc si un super est utilisé dans une classe qui n'a pas de classe mère.
Result ne peut être utilisé que comme opérande gauche d'une affectation et est considéré comme une variable du même type que le type de retour de la méthode dans lequel il est.

- Vérification redéfinition : le mot-clé override ne doit être utilisé sur une méthode que si et seulement si elle redéfinit une méthode de sa super-classe, à savoir qu'elle comporte le même nom, que son type de retour est le même que celui de la super-classe, et que les types de ses paramètres sont les mêmes que la super-classe.

- Vérification instanciation : Lorsque l'on instancie un objet avec le mot-clé new, par exemple new Point(x, y, name); on vérifie que le constructeur de la classe Point existe et qu'il prend bien le type des paramètres donnés. On considère tout "new Point(x, y, name)" comme une variable de type Point. Si on l'affecte, c'est la vérification d'affectation qui entre en jeu.

Enrichissement de l'arbre

Pour grandement faciliter la génération de code, nous effectuons, durant les vérifications contextuelles, un enrichissement des arbres qui représentent le programme.

Le premier enrichissement se fait à la vérification de portée. Lorsque l'on rencontre un IDVAR, qui est jusqu'ici une feuille de l'arbre représentée par un string, on le remplace par un pointeur vers la structure Attribut qui le représente. Ainsi, pour un IDVAR donné, on peut directement accéder à son type et son nom.

Un second enrichissement est fait à l'appel d'une méthode. Jusqu'ici, lorsqu'un appel était fait, il existait une feuille IDAPPEL qui représentait le nom de la méthode appelée. Désormais, on remplace ce string par la structure Method qui correspond.

Enfin, un dernier enrichissement est fait pour une instanciation : on remplace le string du constructeur par la structure Method qui y correspond.

Génération de code

La génération de code fonctionne actuellement sur les expressions simples comme les opérations arithmétiques et de comparaison.
Le code est aussi généré dans le cadre d'un if then else. Les classes et les méthodes sont bien générées et sont accessibles facilement pour la suite.
La gestion dynamique des appels est implémentée: on passe par les tables virtuelles de chaque classe pour trouver la bonne méthode, s'il y a redéfinition, c'est la méthode redéfinie qui est appelée.
La génération de code fonctionne pour les appels de méthode qui ne comportent aucun argument et qui ne sont pas chaînés.
L'affectation fonctionne aussi.

Ce qui ne marche pas

Dans la génération de code, les appels de méthode avec un argument ou plus ne fonctionnent pas. De même, les appels chaînés ne fonctionnent pas.

Le cheminement est bon mais nous n'avons pas eu le temps de rassembler tous les morceaux pour parvenir à boucler toute la fonctionnalité d'un appel. Il existe donc dans code.c des morceaux de code qui sont bons mais que nous n'avons pas pu implémenter.

Contribution des membres du groupe

Au début du projet, Cédrick s'est occupé de la création des structures dont nous avions discuté et dont nous aurions besoin plus tard. Il a donc créé les structures ainsi que les méthodes qui permettent de les créer/les remplir.
Par la suite, il a participé à la génération de code, ajoutant notamment les rangs dans les différentes structures afin d'allouer la mémoire de chaque objet au bon endroit.

Hugo et Léo se sont occupés au départ de la création de la grammaire, dans le .y. Ils ont ensuite créé les structures du programme : l'arbre principal, les classes, les méthodes, etc… à partir de la grammaire et des méthodes que Cédrick leur a fourni.
Ils ont par la suite effectué les vérifications contextuelles, Hugo se concentrant sur toutes les vérifications de typage hormis les appels de méthode et Léo se concentrant sur la vérification de portée ainsi que les vérifications contextuelles d'un appel. Pour le reste, ils ont souvent fonctionné à deux.

Au début du projet Manfred s'est occupé de l'analyse lexicale et de la création de tous les tokens nécessaires à la création de la grammaire. Il a aussi aidé à la création des structures de classe et de méthode depuis la grammaire.
Il a ensuite participé à l'essentiel de la génération de code : générer le code des expressions, de toutes les méthodes, des classes, des appels de méthode, etc...

Miaobing a principalement aidé Manfred sur l'analyse lexicale. Il a aussi aidé à la création des structures depuis la grammaire.