Projet Compilation

Compte rendu

Mattieu Allié
Benjamin Blois
Dylan Darcy
Arnaud Manente

Introduction

Le but de ce projet est la réalisation d'un compilateur. Ce dernier traduit un langage simple orienté objet dans un code proche de l'assembleur. Ce dernier est ensuite Interprété par une machine virtuelle fournie, afin de tester l'exactitude du code produit.

Notre compilateur s'appelle gaouc, pour gaou compiler. Le gaou est le petit du gnou, et au-delà du clin d'œil évident à GNU, ces derniers sont prêts à se protéger sitôt après leur naissance. C'est une attitude que l'on aimerait voir dans notre compilateur : opérationnel sitôt après sa naissance.
Car en effet, notre compilateur gère à peu de choses près toutes les spécificités du langage (sauf peut-être quelques fonctionnalités obscures comme l'appel à un champ statique d'une classe depuis une instance de celle-ci).

Le projet s'est découpé en deux grandes phases, regroupant les différentes étapes clés de la compilation.
La première étape commença par l'élaboration de la grammaire et la définition des tokens à reconnaitre par l'analyseur lexical. Ensuite, nous avons travaillé sur les structures de données contenant le programme, et la manière de les construire avec l'analyseur syntaxique.
Une fois tous ces mécanismes en place, nous avons pu commencer de front la vérification contextuelle ainsi que la génération du code à proprement parler.

Dès le début, nous avons veillé à répartir au mieux les différentes tâches. Ainsi sur la première étape, Mattieu a travaillé sur l'analyse lexicale, Dylan et Benjamin conçurent la grammaire, tandis qu'Arnaud fut affecté aux structures de données et à leur implémentation.

Pour la deuxième étape, deux équipes furent formées, une pour chacune des grosses étapes suivantes. Ainsi, Arnaud et Mattieu firent les vérifications contextuelles, alors que Dylan et Benjamin s'attelèrent à la génération de code.

Nous allons vous présenter ci-après les principaux choix d'implémentations pour chacune des grandes étapes du projet.

Analyse Lexicale

Nous sommes restés dans cette première étape très près des sentiers battus.
Nous avons défini un token par mot clé du langage ainsi que pour chaque opérateur arithmétique ou de concaténation.
Les délimiteurs et les commentaires sont reconnus et échappés.
Les symboles intéressants pour notre langage sont gardés tels quels, tandis que les autres lèvent une erreur lexicale.
Les chaines de caractères doivent correspondre au format de ces dernières dans le langage C.
Ensuite, pour simplifier le travail avec ces prochains et éviter la multiplication des règles, nous avons groupés tous les opérateurs relationnels nous un seul token.
Enfin, toute suite de caractères (éventuellement entrelacés avec des entiers à partir du deuxième caractère), est considérée comme un identificateur (si le premier caractère est une minuscule), ou comme un nom de classe (si le premier caractère est une majuscule).
Ainsi, les identificateurs réservés *result*, *this* et *super* sont considérés comme des identificateurs par la grammaire (ce qui simplifie le traitement). Ces mots-clés spéciaux seront gérés par l'analyse contextuelle.

Analyse Syntaxique

Nous avons dès le début fait le choix de réaliser une grammaire très explicite. Cela s'accompagne d'une multiplication du nombre de règles, mais par une facilité accrue de compréhension, et d'utilisation de cette dernière.
Nous avons cependant essayé de factoriser au maximum les expressions qui pouvaient l'être, afin de ne tout de même pas causer une explosion du nombre de règles.
D'une manière générale, nous avons réutilisés certains patterns :

- NomRègleO : partie optionnelle, de la forme Null | NomRègle
- ListNomRègle : répétition arbitraire d'une règle, de la forme NomRegle | NomRègle « caractère » ListNomRègle. éventuellement sans caractères.

Toute la première partie des règles n'a pas posé de problème, mais les expressions soulevèrent des questions. En effet, certaines expressions peuvent être partie gauche d'une affectation ou d'une sélection, et d'autres ne le peuvent pas. Les résultats d'expressions avec opérateurs, par exemple, ne le peuvent pas. Nous avons donc décidé de découper les expressions en 2 familles, afin d'empêcher ces cas de figures.

Par la suite, nous avons été obligé de faire de petites modifications a la grammaire, afin de faciliter le traitement. Par exemple la règle ClassAlloc, qui sert uniquement a allouer la structure de donnée contenant la description de la classe. Nous avons cependant essayé de garder ces modifications au minimum.

Structures de données

Afin de faciliter les traitements ultérieurs, nous avons décidés de ne pas stocker tout le code sous la forme d'un énorme arbre syntaxique. En effet, certains éléments du langage source comme les classes (ainsi que tous leurs champs et méthodes) ont une portée globale. Nous avons donc décidé de les stocker ces dernières dans des listes de classes, contenant des listes de méthodes (statiques ou non) ainsi que des listes de champs (de classe ou d'instance). Cela nous permet de faire tous les traitements globaux au programme une seule et unique fois, et de les stocker au bon endroit. Cela permet de simplifier énormément certains traitements, comme par exemple faire des vérifications entre une classe mère et sa fille.

Cependant, certains éléments du code, comme les blocs d'instructions (et tous leurs composants) ou les arguments d'envoi de message, ne peuvent pas s'affranchir de la structure d'arbre. Chacune de ces structures contient donc un certain nombre de champs TreeP, contenant par exemple l'initialisation d'une variable, le code d'une méthode…

La seule contrepartie de cette représentation est qu'il faut bien appliquer les étapes suivantes (vérification contextuelle et génération de code) à chacun de ces arbres, et dans le bon sens. Cependant, une fois le sens trouvé et fixé, le gain en complexité contrebalance grandement ce petit défaut.

Vérification Contextuelle

La passe de vérification contextuelle se décompose en plusieurs parties bien distinctes et a de multiples intérêts. En effet en plus de sa fonctionnalité évidente qui est de vérifier les erreurs dans le code n'étant pas détectées par les vérifications préalables, il faut aussi que la mise à jour de tous les nœuds de l'arbre soit faite (notamment leur type).

Voilà comment sont ordonnancées nos vérifications :

- Une passe dans laquelle il faut relier tous les types rentrés en dur (sous forme de chaines de caractères lors de la lecture du fichier) aux bonnes structures de données, et ce pour chaque champ, chaque méthode, dans toutes les classes. C'est une passe nécessaire car il faut attendre d'avoir construit toutes les classes pour l'effectuer. Ici, on écarte toute possibilité d'utilisation d'une classe non déclarée dans le programme.
- Une passe va générer les tables de sauts de toutes les classes. C'est ici que l'on gère l'héritage, le masquage de champs, la surcharge d'opérateurs… Tous les décalages (« offsets ») utilisés par la suite pour adresser les champs et les méthodes sont définis ici.
- Ensuite la vérification se porte sur le code actuel. De toutes les fonctions, de toutes les classes. Pour ce faire on a recours à l'utilisation de tables des symboles dans lesquelles on trouve pour chaque fonction le contexte global de la classe dans laquelle on se trouve, les paramètres de cette fonction et ensuite le contexte local aux différents scopes.
- Enfin c'est le code du « main » qui est vérifié.

Si tout se passe bien, le programme est considéré valide et il ne reste plus qu'à générer le code. De plus tous les nœuds des différents arbres sont mis à jour avec les bonnes valeurs.

Génération de code

On a vérifié que tout était bon dans le programme source et toutes les données nécessaires à cette étape étant renseignées, nous pouvons maintenant générer le code.
Afin de maitriser la pile d'exécution, quelques règles ont été fixées :

- Une expression, quelle qu'elle soit, empile un nombre arbitraire de valeurs pendant son exécution, mais n'en laisse qu'une seule lorsqu'elle est finie (même les appels de fonction sans type de retour).
- A la fin d'une Instruction, la FP pointe sur la même adresse, quel qu'ait été le nombre d'empilement/dépilement effectués au cours de celle-ci.
- Dans une fonction, on a **toujours** l'appelant dans FP-nbParamètres-1, la valeur de retour dans FP-nbParamètres-2, et le paramètre n à $FP - nbParamètres + indexParamètre$, avec l'index qui commence à 0.
- Dans une classe (autre que les types par défauts), on a toujours la table virtuelle de fonction à l'offset 0, puis les différents champs à partir de l'offset 1.

Ces règles permettent également une imbrication des différentes opérations sans risque de comportements non définis.

Une fois ces règles fixées, il ne reste plus qu'à générer les différentes parties du code dans le bon ordre.
On va donc commencer par réserver de l'espace pour les variables statiques de chaque classe, et pour la Table Virtuelle des fonctions de chaque classe (utile pour le super), puis faire un JUMP vers le début effectif du programme (ce qui permet de définir ensuite toutes les fonctions.

On va donc parcourir chaque classe, et écrire le code de chacune de ses fonctions, statiques ou non, précédés du label NomClass_NomFonction (pour le constructeur, uniquement NomClass), et suivi d'un return. On en profite également pour écrire le code des trois méthodes de base des types primitifs, en dur.
Les constructeurs effectuent les actions suivantes, dans cet ordre : récupérer l'espace alloué, appeler le superconstructeur avec les bon paramètres, récupérer la bonne Table Virtuelle des fonctions (override celle du super si besoin), initialisation des champs (dans leur ordre déclaration) et enfin exécution du code du constructeur.

Ensuite, on débute le programme, mais avant de générer le code du bloc principal, il convient d'initialiser les champs statiques (encore une fois dans leur ordre de déclaration) et d'initialiser les Tables Virtuelles des fonctions.
On peut enfin générer le code principal, qu'on suivra d'un STOP.

Pour générer chacun de ces codes, on va faire un parcours récursif du TreeP associé. Les règles indiquées plus tôt nous assurent que les différents nœuds d'imbriqueront bien, et il suffit de jouer sur l'ordre des appels de la fonction de parcours sur les fils afin de gérer la précédence. Le code est maintenant généré, et par défaut, un breakpoint est placé avant chaque appel de fonction ainsi qu'au début (START) et à la fin (STOP) du programme.

Conclusion

Ce projet nous a permis de cerner la difficulté des compilateurs actuels, ainsi que le courage des personnes les maintenant et les faisant évoluer.
Et pour cause, pour faire ne serait-ce qu'un embryon de ces derniers, il nous a fallu plusieurs mois et des heures de réflexion, code et débogage.
Ce fut cependant une expérience très enrichissante, nous permettant de mettre en pratique cette matière parfois très théorique qu'est la Compilation. Et ce fut pour nous un moment d'émotion de réussir à compiler nos premiers programmes.

Mais il resterait si le temps nous le permettait, beaucoup de choses à ajouter ou modifier dans notre compilateur.
En particulier, nettoyer certaines structures de données contenant beaucoup de redondance, et en modifier d'autres (comme nos piles) afin de simplifier le traitement.
Le code généré pourrait lui aussi être un peu optimisé, certaines instructions (comme les POPN 0) étant en effet inutiles, ou certains commentaires peu intéressants ou explicites.