Projet de compilation : Du TPC au NASM

LE COQUIL - TOUSSAINT
Marc – Lesly Jumelle
Université Gustave Eiffel
Semestre 6 – 2023/2024

Table des matières

Introduction

Plan du rapport

Bonjour et bienvenue dans ce compte rendu de production du projet de Compilation. Nous aborderons ici les points clés de la réalisation de ce logiciel développé par Lesly Jumelle TOUSSAINT et Marc LE COQUIL. Nous présenterons brièvement le travail à réaliser, puis nous passerons aux étapes de sa production. Nous terminerons par la répartition des tâches, les difficultés rencontrées et une conclusion de ce travail. Sur ce, passons à la présentation.

Sujet du projet

Il nous a été demandé de coder en langage C un programme écrivant en langage d'assembleur (convention AMD64 little endian) la traduction littérale d'un code écrit en TPC (une version simplifiée du langage C). Ce programme peut s'apparenter à un compilateur. Il est à noter que contrairement à « clang » ou « gcc », celui-ci n'assemble pas lui-même le fichier .asm produit. La difficulté de ce travail réside dans les points suivants :

- La grammaire imposée devait être analysée minutieusement par un analyseur syntaxique (parser) et toute erreur syntaxique (qui entre en conflit avec cette grammaire) devait être rapportée et non compilée.
- Lors de l'analyse, un arbre abstrait de toute instruction, expression ou clause devait être construit. Aucune ambiguïté n'était permise car c'est sur cet arbre que la suite se basait.
- Traduire l'arbre, ce qui implique une compréhension de la base du langage NASM (manipulation des registres et de la pile) mais aussi de la convention AMD64.
- Détecter les erreurs sémantiques (qui ne peuvent être compilées raisonnablement, ex : utilisation de variable inconnue).
- Et bien sûr, une maîtrise totale du langage C est absolument requise.

Le projet devait se présenter dans un certain format. Pour plus de détails, nous vous invitons à consulter le sujet du projet décrivant plus en détail les attentes quant au travail à réaliser. Nous pouvons maintenant explorer les étapes de réalisation.

Réalisation

Préparation

Ce projet faisait suite à celui d'analyse syntaxique (création de l'arbre et analyse de la syntaxe), une partie du code était donc déjà réalisée et devait être réadaptée pour ce nouveau projet. Nous avons commencé par réorganiser nos dossiers, nos fichiers, et planifier la suite :

- Nous devions apprendre le langage et, ne pouvant pas aller plus vite que les cours magistraux ou les TP, nous avons dès le départ décidé d'attendre d'avoir les connaissances requises pour commencer à coder.
- Nous avions dans l'idée d'avoir les fichiers suivants :
  - `writter.c` : Principal fichier de cette seconde partie, `writter.c` devait contenir tous les accès au fichier .asm produit par le compilateur, mais ne devait pas s'occuper de la gestion des erreurs sémantiques.
  - `checker.c` : Devait contenir toutes les fonctions en relation avec les erreurs sémantiques et leur détection. Il serait le leveur d'exception et informerait l'utilisateur de ses erreurs.
  - `main.c` : Fichier principal, `main.c` devait permettre de lancer le programme. La fonction main se situant initialement dans l'analyseur syntaxique, il a fallu la déplacer.
  - `symbol.c` : Nous avions eu notre premier cours nous prévenant que la production de tables de symboles devrait être la première étape de notre projet. Nous avons donc anticipé cela avec ce fichier.

Nous nous sommes alors éloignés de la compilation, nous appliquant à faire rigoureusement les TP afin de nous préparer à la production finale.

Table de symbols et ré-organisation

Vint le jour où nous nous sentîmes prêts à commencer ce travail, près de trois semaines avant la date de rendu. Le rythme allait s'intensifier. Nous écrivîmes nos premières traductions dans `writter.c`, mais il ne fait aucun doute que les tables de symboles devaient être la première tâche à réaliser. Toute traduction dépend du corps de la fonction contenant les instructions à traduire, et donc des variables locales, globales et arguments auxquels ces instructions peuvent se référer. Nous entreprîmes donc de produire ces tables, ce qui nécessita une réorganisation immédiate.

En effet, les tables de symboles détectent des erreurs sémantiques dès leur production (redéfinition de variable, utilisation de type inconnu, etc.) et nous commencions à questionner l'utilité de `checker.c`. De plus, il nous faudrait bien plus que `symbol.c` pour décrire toutes les tables nécessaires. Nous avons donc choisi de diviser ce fichier en `progTable.c` (table du programme entier, ses variables globales et ses fonctions), `functionTable.c` (table des fonctions, chaque fonction possédant sa table d'arguments et de variables locales) et `symbolTable.c` (simple table associant un nom à un type et une adresse). Une partie de la consigne fut alors rapidement remplie : la création et l'affichage des tables de symboles étaient terminés, ainsi que la détection des premières erreurs sémantiques.

Traduction de base et structure de writter.c

Fonctions courtes

Nos symboles sont maintenant accessibles dans des tables bien organisées, nous pouvons passer à la traduction elle-même. Pour nous faire la main et ne pas risquer de perdre du temps, nous commençons par les traductions simples : `writeNum()`, qui pousse un entier sur la pile, en est un bon exemple.

Le pivot du programme

On se rend alors compte que l'ordre des appels de nos fonctions suivait l'arbre abstrait des instructions. Vint alors le point central, le carrefour de notre programme : `writeInstr()`, qui contient un simple switch redirigeant la traduction vers la bonne fonction en fonction du label de chaque nœud de l'arbre. Nous pouvions alors écrire nos fonctions à l'avance, les placer dans `writeInstr()` et les compléter au fur et à mesure. Ceci créant un grand “arbre” d’appel récursif entre chaque fonction.

Structure conditionnelle

Viennent ensuite les structures conditionnelles. Boucles et conditions utilisant les opérations booléennes ont dû suivre un chemin similaire : en NASM, on utilise des étiquettes pour « sauter » à plusieurs endroits dans le code, ce qui est notamment utile pour revenir au début d'une boucle ou aller directement au `else` d'un `if`. Vint alors un concept clé de notre projet : la numérotation des étiquettes. Chaque `if`, chaque `while`, chaque `true` ou `false` était accompagné de son numéro et incrémentait la valeur associée afin qu'aucune confusion ne s'opère. Nous avions donc quatre variables globales traquant l'évolution du nombre de ces structures.

Opérations de base

Pour finir avec les instructions simples, nous écrivons de quoi additionner, soustraire, multiplier et diviser. Nous terminons aussi les opérations booléennes. Le reste des opérations « simples » allait nous demander d'abord de nous pencher sur l'utilisation des variables (que nous appellerons identifiants pour faire référence aux noms de fonctions, variables globales, locales ou arguments).

Traduction des identifiants

Les identifiants représentent le plus grand défi jusqu'alors. Ils sont la plus grande source d'erreurs sémantiques et peuvent se référer à énormément de choses. On les utilise pour assigner des valeurs, récupérer leur valeur, les placer dans des opérations, dans des valeurs de retour ; ils sont les noms des fonctions, variables globales et locales, et arguments. Il va sans dire que la confusion n'est pas permise entre « a = b » et « b = a ». De nombreuses erreurs ont été commises durant cette phase de travail, et elle fut parmi les plus chronophages alors que la date butoir approchait.

Nous finissons par régler les erreurs liées aux types et par corriger de nouvelles erreurs sémantiques (opérations sur des valeurs sans type, assignation d'entier à un caractère, passage de variable en paramètre de fonctions qui n'en prennent pas, etc.). Il est maintenant temps de constater si nos efforts ont payé : nous allons pouvoir attaquer les entrées-sorties qui nous permettront de savoir si le code produit fonctionne effectivement.

Fonctions par défaut, entrée et sortie

Quatre fonctions par défaut nous étaient imposées : `getChar()` et `putChar()` pour respectivement récupérer sur l'entrée standard et placer sur la sortie standard un unique caractère, `getInt()` et `putInt()` leurs homologues pour les entiers. Leur réalisation fut très pénible malgré les indications présentes dans le cours. La création de `__getCharAux__()` fut même requise pour que le tout fonctionne correctement. Ce n'est pas la partie dont nous sommes le plus fiers, mais il est certain que sa réalisation nous a permis d'avancer un peu plus vers la fin de production.

En effet, c'était l'occasion de constater que les entiers négatifs n'avaient pas été correctement implémentés, que les `if` et `else` se mélangeaient illégalement et que les retours de fonctions ne se produisaient pas correctement. Nous étions donc repartis pour plusieurs jours de débogage, et nous n'étions pas au bout de nos peines.

Tableaux et pointeurs

Nous avons gardé les tableaux pour la fin volontairement. Nous avions anticipé que ce serait un gros travail et nous préférions finir le principal avant d'attaquer celui-ci. Mais nous y voilà, et sans surprise ce fut le plus grand défi de toute la réalisation. Les tableaux étaient une toute nouvelle manière de traiter les identifiants, de les analyser, de les passer en paramètres, d'assigner des valeurs. Les erreurs de segmentation furent nombreuses, ce qui nous a permis de nous familiariser avec le logiciel de débogage gdb mais aussi d'échanger avec nos camarades qui rencontraient les mêmes difficultés que nous. C'est à partir de cette étape que nous nous fîmes entendre sur l'espace de tchat de la promotion afin de pouvoir avancer au mieux.

Deux jours avant la date de rendu, nous finissions les tableaux. Il nous restait deux jours pour écrire nos tests sémantiques, le présent rapport de production, mais aussi tester notre code sur le banc de test mis à disposition et documenter notre code. Les heures étaient comptées.

Test et correction

Nous commençons par tester notre programme sur les bancs de test, et c'est un euphémisme que de décrire cela par une déception tant la note affichée pouvait être basse comparée aux efforts que nous avions fournis. Il s'avéra que notre programme ne renvoyait pas les bons codes d'erreurs. Sans plus d'explication, nous devions remonter une note à 60/100 pour la sémantique et 32/100 pour l'exécution. En sachant seulement que nous ne renvoyions pas les bons codes d'erreurs. Nous avons alors supposé que notre code était trop restrictif sémantiquement. Nous avons donc laissé passer ce que nous prenions pour des erreurs comme des cas normaux d'utilisation et nos notes sont passées à 75/100 pour la sémantique et 36/100 pour l'exécution. Contents d'avoir trouvé une partie du problème, cela en confirmait un autre : nous n'avions pas pleinement compris ce qui était considéré comme une erreur de sémantique et ce qui ne l'était pas. Le sujet est minimaliste quant au détail de la grammaire et nous laisse une grande liberté dans son interprétation, mais les tests automatiques n'étaient pas de cet avis. Nous devions trouver une solution.

Nous nous sommes alors lancés dans la quête de produire un maximum de tests sur des erreurs sémantiques, et cela s'est avéré fructueux. Quelques fonctionnalités avaient effectivement été oubliées ou ne fonctionnaient pas bien. Certaines erreurs passaient et certains codes corrects non. Cette phase de test fut un réel pas en avant et nous a permis une réelle progression dans nos notes : 77/100 pour la sémantique et 52/100 pour l'exécution. Malheureusement, nous étions dans la dernière nuit avant le rendu, l'heure n'était plus aux tests mais bien à la documentation de ce projet.

Fin de production

Nous prîmes la décision commune d'écrire la documentation en anglais puisque nos fonctions ont déjà leurs noms en anglais. Le rapport pouvant être long et utilisant un langage plus élaboré, nous avons préféré le garder en français pour plus de lisibilité. Nous éliminons alors les fichiers inutilisés, notamment `checker.c`, qui ne fut finalement jamais implémenté. Finalement, nous écrivons ce rapport et rendons notre travail.

Bugs subsistants

Malheureusement, un problème n'a pu être résolu totalement : les arguments passés lors des appels de fonctions ne peuvent dépasser le nombre de 6 sans causer de bug. Ce cas est explicitement donné comme devant être géré et ne l'est pas. Tout autre bug provient de cas d'utilisation imprévu. Nous ne considérons pas notre programme infaillible ; il est robuste là où nous avons pensé à le rendre robuste.

Réparititon du travail et difficultées rencontrées

Répartition du travail

Le travail n'a pas été facile à répartir. Nos compétences étaient inégales et il a fallu se répartir les tâches intelligemment. Premièrement, toutes les actions concernant une correction ou une modification de l'arbre abstrait étaient réalisées par Lesly Jumelle, qui avait une bien meilleure compréhension de l'analyseur syntaxique. Bien que ce travail fût réalisé par nous deux, il est clair que l'un de nous était plus à l'aise dessus que l'autre ; ce n'était que du bon sens de prendre cette décision. Quant à la traduction, même si le début (tables de symboles et premières fonctions de traduction) était un travail commun, Marc a repris peu à peu la majorité du travail lorsque les fonctions et raisonnements devenaient plus complexes. En effet, même si le niveau de Lesly Jumelle était à la hauteur d'un élève de 3e année en programmation en C, Marc codait plus vite, et ce simple détail lui donna plus de responsabilité sur cette partie à la vue de la date butoir approchant. Les tests ont été faits ensemble ; nos esprits pensant différemment, cela nous a permis de détecter de nouveaux problèmes plus rapidement. La documentation a également été faite ensemble, chacun écrivant les docstrings associées aux fonctions qu'il avait codées.

Difficultées rencontrées

Les difficultés rencontrées ont déjà été décrites plus haut, mais voici un résumé sous forme de liste :

- Une mauvaise organisation dès le départ due à l'entrée dans un environnement nouveau.
- La programmation des fonctions par défaut en raison de leur caractère unique et de leur écriture « brute » en NASM.
- L'implémentation des tableaux qui modifiait notre code déjà établi.
- La recherche des raisons des échecs des bancs de test due à l'incompréhension apparente de la grammaire attendue.

Conclusion

En conclusion de ce rapport, nous aimerions exprimer notre gratitude envers les enseignants qui ont répondu à nos questions jusqu'à la fin. L'esprit d'entraide entre les élèves et la coopération à travail égal avec un binôme sont des choses qui se sont avérées rares durant ce parcours scolaire. C'est, non sans un peu d'émotion, que nous rendons ici notre dernier projet avant l'obtention, nous l'espérons, de notre licence. Malgré le fait qu'il ne soit pas parfait, ce projet illustre bien la difficulté du chemin parcouru jusqu'ici.

Merci pour la lecture de ce rapport, au revoir.