Projet de Compilation

Clément Hard et Markus Puura

Pour rappel, le projet de compilation consiste à concevoir un compilateur pour un sous-langage du C. L’objectif est de développer un compilateur qui accepte un code C simplifié en entrée et qui génère une représentation intermédiaire sous forme de graphes, également connue sous le nom d'arbres abstraits. Le langage d'entrée simplifié est appelé miniC, tandis que le langage de sortie est appelé DOT. Nous avons travaillé ensemble sur le projet et fait de notre mieux tout au long du semestre pour fournir un résultat représentatif de notre équipe, un résultat dont nous pouvons être fier.

Nous avons débuté le projet après avoir terminé les premiers travaux dirigés sur l'analyse lexicale, puis nous avons travaillé sur l’analyse syntaxique, sémantique et finalement sur la génération de code. Nous avons travaillé étape par étape, avant de passer à la suivante nous devions nous assurer que la précédente fonctionnait bien, il n’était pas question d’aller trop vite et de se perdre dans du code ne fonctionnant pas et mêlant différentes parties du projet.

Aujourd’hui à la date du rendu final, malgré un projet bien développé et bien avancé, quelques problèmes non résolus persistent, notamment pour l’analyse sémantique et pour la génération du code.

Afin de développer la partie sur l’analyse lexical, nous avons utilisé le fichier Lex fourni avec l’archive du projet, modifié (les mots qui ne font pas partie du langage miniC enlevés).

Concernant l’analyse syntaxique, après avoir dessiné l’arbre au brouillon et réfléchi sur la méthode de sa programmation, nous avons créé un nouveau fichier ANSYNyacc.y. Nous avons écrit la grammaire du miniC dans ce fichier, elle fonctionne parfaitement.

L’analyse sémantique quant à elle est également sur ce fichier, elle décore l’arbre syntaxique. Nous avons créé un fichier Structure.h qui comporte notre structure de données utilisée pour les actions sémantiques : Des listes chaînées. Une liste permet de stocker les fonctions déclarées, avec des informations importantes sur eux (comme le nombre de paramètres, s’il attend quelque chose en sortie...). Une autre liste stock les variables et tableaux déclarés par bloc. En effet, lorsque nous entrons dans un nouveau bloc du fichier que l’on compile, un nouvel élément bloc est créé. Il contient un tableau de caractères (où seront stockés les déclarations), lorsqu’on la quitte, l’élément est supprimé. Donc quand il y a une appelle ou création de fonction on parcourt la liste chaînée des fonctions où on trouve toutes les informations nécessaires sur les fonctions ; et pareil pour les variables et tableaux, où on peut parcourir l’élément tout en haut de la pile pour éviter une double déclaration, et les éléments d’en dessous si on l’utilise dans une affectation ou en paramètre pour trouver sa déclaration.

Malheureusement un problème est apparu, parfois le parseur traverse le code dans un sens où cette méthode ne convient pas parfaitement, par exemple lors d’une déclaration de fonction le parseur parcours d’abord l’intérieur du bloc de la fonction avant de parcourir le nom de la fonction avec ses paramètres. Pour pallier cela nous a créé des tableaux de caractères globales où l’on peut stocker des noms de variables pour qu’ils soient contrôlés plus tard. Il nous aura également fallu créer d’autres tableaux et variables globales pour stocker des informations de la même manière.

Malgré notre détermination, comme dit précédemment, quelques erreurs persistent :

* Lorsqu’une fonction est récursive, le parseur parse d’abord l’appel de fonction à l’intérieur du bloc de fonction qui n’est pas encore enregistré dans la liste chaînée. Solution possible : créer une nouvelle liste globale où nous enregistrons les informations sur chaque fonction qui n’est pas déclaré comme nous l’avons fait pour les variables, mais cela nécessite beaucoup de travail en plus, il doit y avoir une meilleure approche, plus optimisé.
* Lorsqu’une erreur est détectée, la ligne de l’erreur qui est indiqué en retours est parfois incorrecte. Il s’agit principalement des informations enregistrées dans les tableaux globales de déclarations pour qu’ils soient contrôlés plus tard. L’erreur est envoyée au moment où le tableau est contrôlé, quand toutes les informations nécessaires sont récoltées, donc à un autre endroit que là où on utilise la variable non déclarée.

La partie qui s’occupe de la génération de code est à la suite de la partie de l’analyse sémantique, nous avons créé d’autres structures pour enregistrer en mémoire ce que nous devrons générer. Il s’agit encore de listes chaînées, une pour enregistrer les sommets qui doivent être crées pour le Dot, qui contient ses informations (label, couleur, identifiant utilisé pour attribuer un nom). Une autre, pour enregistrer les arrêtes, avec l’identifiant du sommet de départ et l’identifiant du sommet pointé.

Ces structures seront ensuite parcourues à condition qu’aucune erreur ne soit générer dans la création du fichier DOT. La plus grande difficulté rencontrée fut l’ordre à laquelle nous créons les sommets, il dépend de l’ordre à laquelle elles seront disposées en dessous d’une sommet mère qui pointe vers plusieurs sommets. Puisque le parseur parcourt la grammaire de manière ascendante, elle ne crée parfois pas les sommets dans l’ordre voulu.

Pour résoudre ce problème nous avons créé une autre pile pour les sommets, afin de les enregistrer à l’intérieur quand cela est nécessaire. Il nous faudra ensuite les mettre dans la pile principale (à la fin) avec une fonction pour qu’ils soient dans un ordre différent. Notre solution ne permet pas de résoudre l’entièreté du problème, donc les sommets fils sont encore parfois à l’envers lorsqu’il s’agit une opération, d’une comparaison ou d’un switch. Les fils d’un sommet BLOCK sont tout le temps à l’envers.

Un autre problème que nous avions c’était lorsqu’il y a une opération entre parenthèses, nous créons parfois un sommet vide qui ne sert à rien entre deux sommets, nous voulions l’effacer dans la fonction qui génère le fichier Dot mais nous n’avons pas réussi.

Dans le sujet il n’était pas indiqué comment représenter un « - » devant des parenthèses (ex : var = -(1+2)). Dans le fichier DOT il sera représenté comme si c’était 0 – la parenthèse, (var = 0 – (1+2)). Nous faisons de même lorsqu’il s’agit d’un identificateur, d’un tableau ou un nombre au négatif.

Pour finir, nous allons aborder le sujet de la répartition des tâches dans notre équipe. Markus est très vite devenu le réfèrent du groupe étant donné notre grande différence de compétences. Nous avons essayé d’équilibrer les taches un maximum en fonctions de nos compétences respectives mais sans s’y limiter, pour que chacun d’entre nous travaille sur un peu toutes les parties. Le but étant d’y passer un nombre d’heure équivalent. Markus avançant plus vite il a évidemment fait plus de code fonctionnel, présent dans le rendu final. Principalement, Clément c’est plutôt concentré sur certaines fonctions en C, toujours avec l’aide de Markus qui a su être très pédagogue et très patient (je l’en remercie d’ailleurs). Markus quant à lui à plutôt travaillé sur la logique, sur les structures et sur le reste. Nous nous sommes rencontrés plusieurs fois à l’université et nous avons organisé des réunions Discord pour partager notre écran et réfléchir ensemble.

Malgré des difficultés rencontrées nous avons tous deux donnés notre maximum, nous avons appris énormément et nous avons fait de très grand progrès dans le domaine de la compilation.

Il est a noté que nous n’avons pas utilisé d’aide externe.

Nous sommes dans l’ensemble très fière du travail rendu, malgré les quelques petites erreurs qui persistent (qui peuvent être régler avec encore plus de travail ).