REPUBLIQUE FRANÇAISE

*****

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR, DE LA RECHERCHE ET DE L'INNOVATION

*****

UNIVERSITE NICE CÔTE D'AZUR

FACULTÉ DES SCIENCES : INFORMATIQUE

Unité d'enseignement : Compilation

Projet de compilation en licence informatique : strucit, un mini compilateur C

Réalisé par :

Ahmed ADJIBADE (22015938) & Paul ZANAGLIA (21803683)

Superviseur :

Pr Sid TOUATI

Année Académique : 2021-2022

Table des matières

I. Introduction

Dans le cadre de l'apprentissage des notions fondamentales de la compilation en informatique, nous avons été conviés à réaliser un mini compilateur C : strucit.

Ce compilateur doit prendre en entrée du code textuel lexico-syntaxiquement correct du point de vue de notre langage source STRUCIT-frontend et fournir en sortie un code dans un langage STRUCIT-backend dont les spécificités ont été précisées par notre enseignant.

Pour réussir cette traduction, nous avons identifié plusieurs étapes qui se succèdent comme suit :

a) L'analyse lexicale
b) L'analyse syntaxique
c) La mise en place d'une représentation intermédiaire
d) L'analyse sémantique et la génération du code en parallèle.

Dans le processus de réalisation, nous avons finalisé les trois étapes. La dernière étape en revanche, nous l'avons entamée mais nous n'avons pas pu la finir.

II. Déroulement du projet

a. Méthode de travail

Pour mener à bien notre projet nous avons décidé de travailler en présentiel. A chaque séance, on définissait l'ordre du jour et on se répartissait les fonctions à écrire après une analyse au tableau du concept à implémenter.

b. Analyse lexicale

Comme proposé par notre enseignant, nous sommes partis du fichier « ANSI-C.l » qui contenait une description lex généralisée.

Grâce aux tests fournis nous avons pu, grâce à des messages de détections, repérer ce que chaque expression régulière reconnaissait et cela nous a permis d'en apprendre plus sur le langage STRUCIT-frontend.

Nous avons ensuite **retiré les éléments qui n'intervenaient pas** pour la reconnaissance des lexèmes de notre langage.

c. Analyse syntaxique

A cette étape, nous devions **nous assurer que notre grammaire était correcte** et permettait véritablement de **détecter les erreurs syntaxiques** relatives aux lexèmes fournis.

Pour que notre grammaire soit conforme, nous avons dû la modifier à 04 reprises :

1. Pour gérer un shift/reduce provoqué par **l'instruction if {…} else {…}** : on a précisé que le « else » était **non associatif.**
2. Pour les besoins de notre **représentation intermédiaire** : nous avons ajouté un non terminal « program_start » qui se dérive en « program ».
3. Pour **détecter l'entrée dans une nouvelle portée** (un nouvel environnement de symbole) et agir sur notre table des symboles : nous

avons rajouté un non terminal « ACCOO » qui se dérive en '{' et un autre « ACCOF » qui se dérive en '}'.

4. Pour **modifier la déclaration de fonction** : il fallait pouvoir agir sur la déclaration de fonction indépendamment des paramètres, pour cela nous avons ajouté un non terminal « function_init » qui se dérive en « declaration_specifiers declator »

Une fois ce cap franchi, nous avons défini l'interface de la **table des symboles** qui contient la signature des fonctions utiles pour la manipulation de cette dernière. Au départ, après analyse de notre grammaire, nous avons convenu de faire trois tables des symboles à raison d'une pour les variables, une pour les fonctions et une pour les structures.

Mais en avançant, nous avons finalement opté pour **une table des symboles unique** avec des paramètres particuliers visant à détecter la nature du symbole (variable, fonction, structure) et d'autres qui sont spécifiques au symbole.

Notre table de symbole est représentée par une **pile** qui a, à son **sommet**, **l'environnement** le **plus récent** et à sa **base** le **plus ancien**. Nous avons deux types de fonctions de recherches : une fonction d'instance (dans l'environnement courant) et une fonction globale.

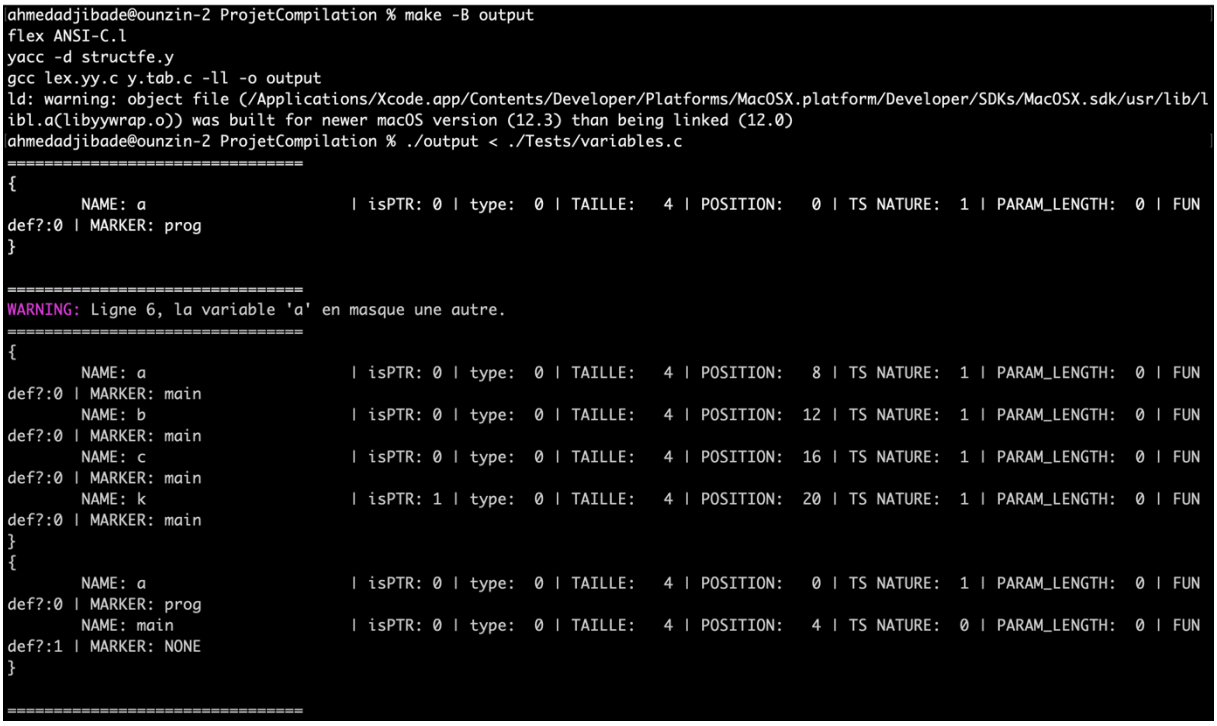

Figure 1 : Affichage table des symboles (variables.c)

Dans le cas du fichier variables.c ci-dessus, nous pouvons apercevoir les spécificités des symboles.

d. Représentation intermédiaire

Nous avons choisi de faire notre **représentation intermédiaire** sous la forme d'un **arbre abstrait**.

Pour cela, nous avons défini une interface pour représenter nos nœuds et la signature des fonctions utiles à la manipulation de ces derniers. Chaque non terminal est un nœud ou une liste de nœuds (notamment pour les listes de paramètres) et **chaque nœud est potentiellement rattaché à un symbole.**

L'arbre se construit au cours de l'analyse syntaxique. Puisque l'analyse est ascendante on part des feuilles pour la racine. La racine ici est représenté par la production syntaxique « program » et c'est cela qui justifie la modification de la grammaire (point c-2 ci-dessus) car nous affichons aussi les arbres de chaque fonction et pour notre grammaire une définition de fonction est considéré comme

un « program ». Tout ceci afin d'éviter un arrêt prématuré de notre fonction de construction d'arbre.

Figure 2 : Représentation intermédiaire (variables.c)

e. Analyse sémantique (incomplète)

A cette étape, nous avons défini une fonction qui prend en paramètre notre arbre abstrait et qui aura pour rôle de s'assurer de la véracité sémantique des instructions décrites par l'arbre. Chaque nœud étant au préalable rattaché à un symbole, la vérification est d'autant plus aisée.

Ici, nous n'avons hélas pas pu traiter toutes les cas possibles.

f. Génération de code (début d'implémentation)

Une fois l'analyse sémantique validée, la prochaine étape consistait à générer du code. Pour cela, nous avons rajouter un attribut « code » et un attribut « val » dans l'interface de nos nœuds afin d'y rattacher le code et la valeur (potentiellement une variable temporelle pour notre code 3 adresses).

Ici également, nous n’avons pas pu aller plus loin que la gestion des variables et les opérations de calculs.

III. Conclusion

Réaliser le présent projet nous a permis de cerner certaines notions sur la compilation mais également sur une notion toute aussi importante : l'organisation. Nous avons crucialement manqué d'organisation ce qui nous a conduit à ne pas finir le projet dans la marge de temps définie. Toutefois, pour pouvoir atteindre le niveau où nous sommes arrivés, nous avons pu compter en partie sur les diverses questions du forum en ligne mais aussi sur les conseils de quelques amis qui ont mieux et vite perçu le bon angle pour comprendre et réaliser le projet.

Au-delà de cela, nous avons pu en apprendre beaucoup plus sur la programmation et ce qui se passe au niveau de la machine. Nous nous étions toujours demandé comment est-ce qu'on arrivait à partir d'un langage textuel compréhensible par l'homme, pour arriver à un langage compréhensible par la machine. Ce projet, et plus globalement cette unité d'enseignement, nous ont introduit aux notions de base de la compilation. Ce domaine génère autant de curiosité que de réflexions et c'est cette dualité qui en fait un thème d'étude intéressant.

Somme toute, ce fut une expérience riche en découverte et qui nous a permis de nous discipliner pour des projets futurs dans le monde de l'informatique.

Nous regrettons tout de même de ne pas avoir pu aller plus loin.

IV. Liste des figures

V. Bibliographie

Romain Legendre, François Schwarzentruber. *Compilation : Analyse Lexicale et Syntaxique du Texte à Sa Structure en Informatique,* Ellipses, 1e édition, 31 mars 2015, 321.

VI. Webographie

- http://www.cs.man.ac.uk/~pjj/cs2111/ho/node8.html , Parse Trees 1
- http://web.eecs.utk.edu/~bvanderz/teaching/cs461Sp11/notes/parse_tree/ , Parse Trees 2
- https://stackoverflow.com/questions/12731922/reforming-the-grammar-to-remove-shift-reduce-conflict-in-if-then-else , Shift/Reduce
- https://pageperso.lis-lab.fr/alexis.nasr/Ens/Compilation/cmX_lex_yacc.pdf , Présentation Lex & Yacc
- https://www.geeksforgeeks.org/ , Les bibliothèques en langage C
- https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm , Pour mieux comprendre l'analyse sémantique