I53 – Projet de fin de semestre Conception d'un compilateur ALGO/RAM Licence 3 – 2022/2023

January 8, 2024

LATHUILE Anne-Sophie et LOPEZ Carlos January 8, 2024

Contents

1 Introduction

Le but de ce projet est de construire un compilateur qui prend en charge le langage algorithmique et produit un code en langage machine RAM. Pour cela, nous avons utilisé Flex et Bison permettant l'analyse lexicale et l'analyse syntaxique de nos algorithmes.

Nous avons rencontré pas mal de difficultés au début du projet, car nous devions nous familiariser avec un nouveau type de code avec notre fichier lexer.lex et parser.y. Mais nous avons pu réaliser une grande partie du projet qui est de sortir du code RAM qui soit correct. Donc notre projet permet de lire et d'analyser des algorithmiques contenant :

- Des instructions d'affectation du style : $a \leftarrow 1 + toto$ ou bien $toto \leftarrow toto + titi$, etc.
- Des instructions SI, TQ et POUR
- Des instructions RENVOYER du style : `RETOURNER toto` ou bien `RETOURNER 3`
- Des instructions LIRE et ECRIRE

2 Partie: asa et codegen

2.1 Entrées/Sorties

Pour l'instruction RENVOYER, je n'étais pas sur de ce qu'elle pouvait faire à part stocker une variable à son adresse mémoire ou un NB. Je pouvais aussi ne pas lui attribueer de code.

2.1.1 Instruction READ, WRITE

Je n'ai pas très bien compris ce que devais faire les instructions LIRE et ECRIRE, donc je suis partie du principe que LIRE devais READ une valeur dans la bande d'entrer et la stocker et ECRIRE devais récupérer l'adresse de l'élément et la WRITE.

2.2 Gestion mémoire, adressage

Le plus important dans le compilateur est qu'il puisse génère un code qui soit exécutable (ici du code RAM). Sachant que le langage RAM accèpte les instructions JUMP donc des saut à des adresses mémoires il faut pouvoir les gerer. Pour cela, chaque noeud créé pour chacune de nos instruction possède une taille de code propre. Par exemple l'instruction:

- ALGO: codelen = 4
- AFFECT: codelen = 1
- DECLA VAR: codelen = 2
- OP (+, -, *, /): codelen = 7
- OP (sup, inf, =): codelen = 9
- NB: codelen = 1
- ID: codelen = 1
- SI: codelen = 1
- TQ, POUR: codelen = 2

initialisation de variables: Pour me simplifier la tâche au niveau des adresses de chaque variables, je lui ai directement attribué son adresse mémoire du code RAM. Donc je suis passée par une variable qui s'appelle **pile_ram** que j'incrémente en fonction de la taille du code de chaque instruction. Cette variable qui est globale est initialisée à 4.

adresses mémoire de 4 à 32: Les adresses de 4 à 32 sont utilisées pour nos déclarations de variables et elles n'occupent que 28 adresses mémoire car je suis partie du principe que nous ne demanderons pas de générer du code avec autant de variables.

adresses mémoire à partir de 32: Au tout début de l'exécution du fichier codegen, nous partons obligatoirement du typeAlgo. Dans cette partie là j'initialise l'adresse 2 à la valeur 32 car c'est elle qui va nous permettre de faire tous nos calculs SUB, ADD, ..., MULT

2.2.1 coment gérer les sauts au bon endroit avec JUMP, JUMZ, JUML et JUMG ?

Pour les sauts à des lignes de notre code RAM j'ai utilisé une fonction **parcours_asa(asa* p)** qui parcours notre ARBRE ABSTRAIT en profondeur de façon récursive pour calculer notre taille de code à l'endroit où nous sommes. Mais il se pose toujours le problème de: comment aller à la bonne instruction ? Prenons l'exemple du code algo suivant:
PROGRAMME ()
VAR i
DEBUT
`i ← 2`
TQ i = 2 **FAIRE**
`i ← i + 1`
FTQ
RETOURNER i
FIN
Donc nous aurons:
du code RAM
code i = 2
JUMZ n
code i ← i + 1
JUMP x
n = taille du code de: i ← i + 1 soit dans codegen.c ligne 464:
codelen = parcours_asa(p → tq.instruct) + nb_ligne;

3 Partie: sémantic

3.1 Fonctionnement du programme

Le programme se découpe en 4 fichiers principaux:

- asa.c et asa.h: s'occupe de générer l'arbre synthaxique abstrait permettant de savoir si le code algo est bien lu à l'aide de paser.y et lexer.lex.
- semantic.c semantic.h: qui permettent d'analyser le code et savoir si nos variables sont bien initialiser.
- codegen.c et codegen.h: qui permettent de générer le code RAM

3.1.1 Ce qui est pris en compte par le programme:

Le code exécute du code RAM correcte pour des instructions: SI, TQ, POUR et des AFFECTATIONS Il gère aussi les appels de fonctions au noveau dela grammaire mais ne peux produire du code correcte car je n'ai pas eu le temps de correctement mieux m'occuper des adresses.

3.2 Exemple de code pris en compte par le compilateur

Un certain nombre d'exemple sont fourni dans le dossier test. A l'exécution du code pour faciliter la lecture dans l'odre d'exécution du programme il ya des affichages pour chaque partie: codegen et semantic.

4 Conclusion

Ce projet a été compliquer et long à réaliser mais le plus enrichissant. Il y a eu pas mal de moment où j'avais l'impression d'avancer puis de reculer surtout à la fin dû à la fatigue. Certaine choses n'ont pas pû etre ajouté comme les appels de fonctions que j'aurai bien voulu traiter mais j'ai préféré me concentrer sur ce qui avait été produit pour que tout fonctionne quelque soit l'exemple possible. Avec du recule la partie la plus longue du projet n'est pas la fin mais le début. Il faut le temps de s'adapter à un tout nouveau langage et synthaxe d'écriture que Bison et Flex peuvent reconnaitre. Au final, ce projet m'a appri: à être rigoureux dans son écriture de grammaire et à comprendre à qquoi elle sert exactement, qu'une solution n'est pas forcément la bonne ni la meilleure, que certe nous avons des objectifs mais qu'il faut aussi savoir quand s'arrêter, m'a aussi appri à utiliser correctement des structures en C, à écrire sur une feuille avant de coder et surtout comment fonctionne un compilateur peut importe le langage avec biensûr ses propres subtilités.