

Rapport de compilation

Auteurs: Clavel Julien, Rugengande Ihimbazwe Jaenai, Anderssen Constantin, Bonneau Audric

Sommaire

I. Introduction	1
II. Description Technique	1
La grammaire	1
La table des symboles	2
Génération de code intermédiaire	3
Génération de code MIPS	3
Gestion des erreurs	3
III. Tests	4
Les tests des implémentations	4
Les tests des erreurs	4
IV. Défis rencontrés	5
V. Limitations et points non implémentés	5
VI. Conclusion	5

I. Introduction

Dans le cadre de la matière de compilation, nous avons pu développer un compilateur du langage métier CoRec (Computed Recurrences). Ce projet est à réaliser en groupe de 4. Pour la génération de code nous avons le choix entre générer du code MIPS ou RISC-V. Etant donné que nous avons déjà tous travaillé sur du code MIPS dans notre cursus, nous avons choisi d'utiliser celui-ci.

II. Description Technique

La grammaire

Pour commencer le projet, nous nous sommes d'abord concentrés sur la grammaire. A partir de la grammaire donnée, nous avons rajouté des actions pour mieux convenir aux besoins de notre compilateur. Mais avant cela, nous avons effectué quelques modifications sur celle-ci.

Tout d'abord, nous avons essayé de résoudre les ambiguïtés de la grammaire qui emmené une erreur avec yacc: le ID dans LCALL et le OP.

Ensuite, nous avons pris ID et nous l'avons transformé en symbole non-terminal. ID correspond soit à ID1, soit à ID2. ID2 est défini en flex comme permettant d'avoir une combinaison de lettres et de chiffres comme mot. Cependant, il permet au mot de commencer par un chiffre. Cela n'est possible que pour le nom du programme. Pour le reste, nous allons l'interdire, d'où l'intervention de ID1 qui ne permet pas au mot de commencer avec un chiffre. ID1 est utilisé comme définition de nom pour tout. Pour le programme, nous utiliserons soit ID1 soit ID2 qui est défini par la règle de ID.

Nous avons ensuite modifié LCALL. LCALL peut être vide. Cela permet de faire appel à des fonctions sans arguments. Dans CALL, nous gardons donc ID1(LCALL)|ID1. ID1 nous permet d'appeler une variable.

Cela nous a donc permis d'enlever dans I, print(ID1). Ainsi, le print est généralisé. CALL permettra de soit print une variable ou le résultat d'un appel de fonction.

Nous avons également changé la règle pour COND. Elle est composée de CONDCHECK THEN B ELSE B. CONDCHECK est une nouvelle règle qui correspond à E OPREL E. THEN est une règle aussi qui correspond à ? et ELSE à ':'. Diviser en de multiples règles, nous permet de mieux lire et de rajouter les conditions plus facilement. Les conditions sont des labels en MIPS. Avec notre grammaire, nous pouvons sauvegarder les conditions et les replacer de la bonne manière en MIPS.

La table des symboles

La table des symboles est un pilier principal du projet. Nous avons décidé de reprendre la table faite en TP et d'y ajouter de nouvelles fonctionnalités.

La structure de la table contient la capacité maximale de la table, c'est-à-dire le nombre de symboles qu'il est possible de stocker. Nous initialisons la table à une capacité de 1024 symboles, et l'agrandissons au besoin.

Ensuite nous avons 4 attributs qui fonctionnent comme des compteurs. Afin de mieux comprendre le fonctionnement de ces compteurs, il est important de noter que nous utilisons deux structures de pile globales pour pouvoir compiler les conditions et la section dom des fonctions. Pour en revenir à la structure de la table des symboles, le premier compteur nous sert à nommer les variables temporaires. Nous utilisons des variables temporaires dans certaines opérations comme dans l'affectation de valeur dans les tableaux. Afin de ne pas avoir de conflits entre affectations, nous utilisons plusieurs variables temporaires distinctes, que nous nommons "temp" puis le nombre dans ce compteur. Le compteur est incrémenté à la suite de la création de la nouvelle variable temporaire. Ensuite un compteur "condition" qui nous traque la taille de la pile de conditions. Puis le compteur "dom" nous donne la taille de la pile dom. Enfin le compteur size qui compte le nombre de symboles contenus dans la table des symboles.

Pour terminer, cette structure contient un tableau de structures "symbol" dans laquelle nous gardons le type, la valeur et la portée de chaque symbole.

Génération de code intermédiaire

Nous avons écrit une fonction gencode qui crée une structure "code" et y ajoute des structures "quad". La structure code est comparable à la table des symboles qui stocke des symboles, hormis que la structure code stocke des structures "quad".

Dans la structure quad, avons un type d'opération (addition, soustraction etc.) et 3 symboles. Cette structure nous permettra par la suite de générer du code mips. Comme nous avons du code 3 adresses ou moins en mips, nous aurons au maximum 3 symboles.

La structure code nous permet d'avoir un tableau de structures quad, la taille de ce tableau et l'index du prochain quad à allouer. Nous allons mettre le quadruplet généré par gencode dans le tableau de la structure code et on mettra à jour la taille et la l'index dans le tableau du prochain quadruplet qui sera généré.

La fonction gencode est appelée dans le fichier yacc à chaque itération pour générer une structure code 3 adresses pendant la lecture pour chaque type de cas ou opération que l'on rencontre.

Pour pouvoir écrire le code intermédiaire en entier, on fait appel à code_dump qui parcourt la structure code et pour l'écrire selon chaque cas.

Génération de code MIPS

Nous avons écrit une fonction, code_mips_dump. Celle-ci, nous permet de traduire une structure quad en instructions MIPS en fonction du type d'opération et des 3 adresses dans

la structure quad. Ainsi, nous pouvons écrire dans un fichier output (ou dans le terminal si le fichier n'est pas donné) le code MIPS correspondant. Mais avant le parcours de la structure code, nous essayons de préparer des variables dans la section `.data` dont nous aurons besoin plus tard en faisant un parcours également. Nous faisons donc deux parcours distincts.

Chaque cas fait appel à sa fonction spécifique pour écrire le code MIPS.

Gestion des erreurs

Dans le fichier `yacc`, nous vérifions les cas d'erreurs possibles pour chaque règle. Ainsi, notre compilateur produit une erreur spécifique si nous rencontrons un non-respect de la règle.

Quand nous rencontrons une erreur, nous affichons l'erreur dans la sortie d'erreur `stderr` et nous faisons en sorte de sortir de manière sûre grâce à la fonction `exit_safely`. Avec celle-ci, nous libérons la table des symboles, le tableau contenant les dimensions des tableaux, les piles des conditions initialisées et tout ce qui aurait pu être initialisé par le `lex` ou le `yacc`.

III. Tests

Les tests des implémentations

Pour être sûr de notre programme, nous avons fait des tests simples pour tester chaque fonctionnalité. Ainsi, cela nous permettait d'implémenter chaque fonctionnalité une par une sans avoir beaucoup de conflits. Les tests effectués étaient les suivants:

1. Déclaration des variables entières et assignation des valeurs.
2. Opération sur les variables entières.
3. Déclaration des tableaux locaux à une dimension et assignation des valeurs.
4. Opération sur les tableaux.
5. Opération sur les flottants.
6. Déclaration des tableaux locaux à N dimensions et assignation des valeurs.
7. Opération sur les tableaux à N dimensions.
8. `if else` et vérification des conditions.
9. Lire un nombre entier et un nombre flottant.
10. La boucle `for` dans `do`.
11. Appel de fonction sans arguments.
12. Appel de fonction avec des arguments `"in"`.
13. Appel de fonction avec des arguments `"out"`.
14. Appel de fonction avec des arguments `"inout"`.

Les tests des erreurs

Afin de vérifier le bon fonctionnement de la gestion des erreurs nous avons fait les tests suivants:

1. Une fonction sans section Rec.
2. Une fonction avec la section Rec vide.
3. Une variable locale déjà déclarée.
4. Utilisation d'une variable en dehors de sa portée.
5. Nom de fonction déjà déclaré.
6. Opération non reconnue.
7. Utilisation d'un tableau non déclaré.
8. Utilisation d'un tableau en dehors de sa portée.
9. Accès à la valeur d'un tableau dont l'index est plus grand que le tableau.
10. Accès à la valeur d'un tableau avec un index flottant.
11. Déclaration d'un tableau à taille flottante.
12. Déclaration d'un entier en lui assignant une valeur flottante (seuls les tableaux peuvent contenir des flottants).
13. Utilisation d'une opération flottante à une variable de type entier.
14. Assigner une valeur flottante à une variable de type entier.
15. Assigner une valeur flottante à une variable entière dans la section Loc.
16. Assigner une variable qui n'est pas de type tableau un tableau.
17. Accéder à un index supérieur à la taille d'un tableau multidimensionnel.
18. Accéder à un index dans un tableau multidimensionnel, en omettant une dimension.
19. Accéder à un index dans un tableau multidimensionnel, en ajoutant une dimension.
20. Appeler une fonction en omettant un argument (Loc).
21. Appeler une fonction en omettant un argument (Call).

IV. Défis rencontrés

Le plus grand défi que l'on a rencontré était la grammaire. Il y a eu beaucoup de changements qui ont été faits dans la grammaire pour simplifier et éviter la confusion. Cela a été fait pour les appels des fonctions, les conditions, le print et les ID.

La deuxième difficulté a été la collaboration. Construire un compilateur est un projet assez conséquent, ce qui veut dire que le projet contient énormément de code. Pour faire chaque tâche, il fallait comprendre le code d'autrui. Cela requiert de se mettre d'accord sur la forme mais également de documenter de manière conséquente. Ce n'était pas exactement une difficulté mais une tâche en plus.

V. Limitations et points non implémentés

Pour ce qui est des fonctionnalités non implémentées, il y a les arguments pour les fonctions: in (fonctionne sauf pour les variables de type tableau), out et inout, et aussi l'appel récursif de fonctions.

Nous avons décidé au début de notre projet de faire les fonctionnalités liées aux fonctions à la fin du projet. Nous avons ensuite pris du retard sur d'autres fonctionnalités et avons sous-estimé le temps nécessaire pour implémenter l'appel de fonctions et les fonctionnalités dérivées de celle-ci.

Nous n'avons pas non plus eu l'occasion d'optimiser le code généré. L'ajout des fonctions aurait été une opportunité pour améliorer le code généré.

VI. Conclusion

Ce projet nous aura permis de mieux comprendre le fonctionnement interne d'un compilateur et de mettre en pratique les concepts et méthodes vues en cours. Le travail en équipe de plus de 2 personnes est aussi très différent de beaucoup d'autres projets, mais nous a pas empêché d'atteindre nos objectifs.