Rapport de projet: compilateur

Chaolei CAI

December 31, 2019

Contents

1 Introduction

Ceci est mon rapport pour le projet du cours de compilation, il est assez courte car la plupart des fonctions sont explicites. Voici le lien vers github du projet:
https://github.com/bk211/MyCompiler

2 001.liec

Ajout du type de la variable après ':'
Fonction print_num qui n'a quasiment pas changée par rapport à ce que nous avons fait en cours, il supporte aussi la fonction print_int qui fait la même chose.

3 001-e1.liec

Affiche l'erreur due au différence de type.

4 002.liec

Prise en charge des operations arithmetiques de base comme + - * /.

4.1 mise en place

En ce qui concerne l'addition ou le soustraction, il y a déja l'instruction add et sub dans MIPS.
Pour la multiplication et la division, l'instruction existe aussi en MIPS, à la différence près qu'il faut rajouter une instruction mflo pour avoir le résultat.
Pour la division, il retourne 0 s'il s'agit d'une division par zero, et fait troncature du résultat afin d'avoir un résultat entier.

5 002-e1.liec

L'ordre des priorité est respectée, il y a aussi la possibilité d'ajouter des parenthèses pour influencer l'ordre de calcul.

6 003.liec

Prise en charge du type booleen, qui pour moi, est juste un entier comme les autres.

6.1 mise en place

Les chaine de charactere "True" et "False" sont lue par le lexer comme des token Lboolean (qui contient la valeur 1 ou 0). Puis au niveau du parseur, il est convertit en Pboolean, enfin, dans analyser, il est reconvertit à la sortie comme un entier normal.

7 004.liec

Prise en charge des opérateurs de comparaison.

7.1 mise en place

L'opérateur '¡' est déja implementé dans MIPS via l'instruction slt (set on less than). Par extension, si vous inverser l'ordre des expression dans le parseur, vous pouvez facilement obtenir l'opérateur '¿'.
Pour l'egalite, c'est un peu plus complex, les arguments sont charges dans les registre t0 et t1. Puis, on effectue t0 ¡ t1 et t1 ¡ t0 par l'instruction slt, les résultats sont stockés dans t2 et t3. Enfin, on fait l'operation binaire 'or' et 'Xori' avec 1.
Cela viens du faite que slt met 0 dans le registre dst s'il y a une égalite, dans ce cas alors, t2 et t3 ont pour valeurs 0, un 'or binaire ne change pas le résultat, v0 recoit alors 0, le Xori le "ou" binaire exclusif avec 1 permet d'inverser le résultat. On obtient ainsi 1 si egalite, 0 sinon.
Dans le cas ou les expressions ne sont pas égaux, l'un des deux slt retourne necessairement 1, 1—0 = 1, 1 xor 1 = 0, on obtient alors 0 dans v0.

8 005.liec

9 006.liec

Prise en charge de la fonction str_len

9.1 mise en place

```
0 (cons 'str_len
1     (list
2           (Lw 'a0 (Mem 'sp 0))
3           (Li 't0 0)
4           (Print-label)
5           (Lb 't1 (Mem 'a0 0))
6           (Beq-lbl-e 't1 'zero)
7           (Addi 't0 't0 1)
8           (Addi 'a0 'a0 1)
9           (Jlbl)
10          (Print-label-e)
11          (Move 'v0 't0)
12          (Inc-uniq-lbl )
13          (Inc-uniq-lbl-e )
    ))
```

ligne 2: charge la chaine de charactere dans a0 ligne 3: met le compteur t0 a 0 ligne 4: crée une lbl unique de depart ligne 5: charge un octect dans t1 ligne 6: si t1 est le dernier character, brancher vers le label de sortie ligne 7-8: Sinon, incrémenter le compteur t0, et avancer d'un octet ligne 9: aller vers le label de depart ligne 10: crée une lbl unique de sortie ligne 11: place le résultat dans v0 ligne 12-13: ne sont pas des instruction compilées à proprement parler, car mips-printer incremente juste les compteurs associés, il ne printf rien. cela me permet d'avoir des label unique.

10 007.liec

Je n'ai pas réussit a compiler cet exemple, en théorie, il faut convertir la liste en pair de pair de pair

11 008.liec

read_int marche mais pas les conditions if else native

12 008-e1.liec

Seul la condition if marche, else n'est pas implementé.
Il faut ajouter une instruction "endif" pour indiquer la fin de la condition if

12.1 mise en place

Au niveau de parseur, le matching suivant cree une structure Pif

```
(Lif expr Lcolon)                    (Pif \$2 \$1-start-pos)\]
```

Au niveau de l'analyseur, on verifie si l'expression de condition est bien du type entier.
AU niveau du compilateur, similairement a la compilation d'une expression, "compilation-if" ajoute une instruction (Beq-if 'v0 'zero).
Ce dernier est une pseudo instruction pour (Beq $v0 $zero (label if uniq))
Enfin, "endif" ajoute le label unique pour servir de sortie si la condition est fausse, si la condition est vrai, le branchement n'est pas prise et les instructions dans la portée du if sont alors executés.

Un problème majeur à ce systeme est qu'il ne marche pas pour les if inbriqué, car endif incremente forcement de 1 le compteur de label uniq

```
If True: # debut de premier If -> branchement vers le label_0
    blabla
    If False: #debut du deuxieme If -> branchement vers le label 0
    endif #fin du deuxieme If -> label_0 et incrementation du compt
endif # fin du premier If -> label 1
```

Or la bonne ordre serait "branchement vers label 0, branchement vers label 1, label 1, label 0
Neanmois, il marche pour les If successif comme vous pouvez le voir dans cet exemple.

13 009.liec

N'est pas fonctionnel, j'ai tenté de faire la manipulation proche de if mais cela crée des problèmes de mémoire. Par exemple, la variable n dans le test et le decrementation n'est pas pareil au niveau de la memoire.