DEBAIN Yann
LETOUZE Clément

Projet informatique : assembleur MIPS
Livrable 2

2017-2018

SOMMAIRE:

Introduction

Lors de l'incrément deux nous nous sommes concentrés sur l'analyse grammaticale du code assembleur MIPS. En effet, lors du premier incrément nous avons réalisé l'analyse lexicale du code, nous pouvons maintenant procéder à l'analyse grammaticale qui consiste à repérer si les instructions et directives indiquées dans les différentes sections du code MIPS sont bonnes ou non. Ici nous vérifierons donc que chaque directives ou instructions sont acceptés dans notre code et si le nombre d'opérande correspondant à chacune d'elles est correct. Nous ne nous intéresserons pas ici à la validité du type des opérandes des instructions, cela sera l'objet de l'incrément numéro trois (analyse syntaxique).

I.1) Analyse grammaticale

Pour réaliser cette analyse grammaticale, nous avons implémenté comme pour l'incrément numéro un, un automate à états finis. Ce dernier a été un petit peu plus compliqué que le précédent puisqu'il comprend plus d'état intermédiaire. En effet, nous pouvons distinguer trois grandes branches pour cet automate. Chacune correspond à une des trois sections de notre code MIPS que nous pouvons rencontrer : ".text", ".data" et ".bss". Chacune de ces branches est ensuite sous-divisé en sous états.
Dans chacune de ces branches nous nous assurons donc que les directives sont valides d'une part, par exemple la section ".bss" ne peut accepter que des directives .space donc l'automate nous retourne une erreur si jamais une directive différente que .space est dans la section ".bss". Cela fonctionne selon le même principe pour la section ".data".
Parlons rapidement de la section ".text" puisqu'elle fait intervenir également une recherche dans un dictionnaire d'instruction que nous avons créée. Ce dictionnaire regroupe toutes les instructions susceptibles d'être rencontrée dans notre code. Il indique pour chaque instruction son nom, le nombre d'opérande qu'il est supposé avoir ainsi que la catégorie de l'instruction (R, I ou J).
Si, lors de l'exécution de l'automate dans la section ".text", l'instruction ne se trouve pas dans le dictionnaire, il retourne une erreur.

Le dictionnaire se présente sous cette forme :

```
30
ADD 3 R
ADDI 3 I
SUB 3 R
MULT 2 R
```

-------> Nombre total d'instructions
-------> Nom de l'instruction, nombre d'opérande que prend l'instruction et type de l'instruction

D'autre part cet automate s'assure que le nombre d'opérande est valide pour chaque instruction.
Enfin, un problème que nous ne gérons pas encore avec cette automate est le problème d'un dépassement de mémoire, par exemple dans le cas où l'opérande est trop grand pour la capacité de l'ordinateur : .space 10^{500}

I.1) Stockage des résultats

Pour pouvoir être réutilisé par la suite, il est impératif de stocker les résultats pour qu'ils soient facile d'y accéder ensuite, notamment pour l'analyse syntaxique qui va suivre.C'est pourquoi nous créons trois files ainsi qu'une table des symboles (ie étiquettes) à l'issus de l'analyse grammaticale :

- Une file BSS qui contient sa directive, le décalage par rapport à la section, le numéro de ligne ainsi que sa valeur si jamais l'automate ne rencontre pas d'erreur et ce pour chaque occurrence de la section ".bss"
- Une file DATA qui contient toutes les directives acceptées rencontrées, le décalage par rapport à la section, le numéro de ligne ainsi que leur valeur et ce encore une fois pour chaque occurrence de la section considérée
- Une file TEXT qui contient toutes les instructions rencontrées, avec le nombre d'opérande que cette dernière doit prendre ainsi que le décalage par rapport au début de la section, le numéro de la ligne en question ainsi que les opérandes effectives.
- La table des symboles est en fait un tableau dont chaque case contient : le nom de l'étiquette, la section dans laquelle elle se trouve, le décalage par rapport au début de cette section et la ligne ou elle se trouve. Nous avions dans un premier temps essayé de remplir les trois files précédentes ainsi que cette table en même temps, mais nous avions des fautes de segmentation. C'est pourquoi nous effectuons deux lectures du code assembleur. Une première (avant l'exécution de l'automate) pour stocker le nom et la ligne de l'étiquette, et une deuxième (pendant l'exécution de l'automate) pour remplir la section et le décalage par rapport à cette dernière.

A noter que c'est également pendant la deuxième lecture et donc pendant l'exécution de l'automate que nous remplissons les trois files précédentes.

```
.set noreorder
.text
    LW $t0, compteur
    LW $t1, fin_compteur

boucle:
    BEQ $t0, $t1, end
    NOP
    ADDI $t0, $t0, 1
    J boucle
    NOP
end:
    J end
.data
compteur:
    .word 1
fin_compteur:
    .word 100
```

```
---------------------------- TABLE -------------------------
[ section = .text
 nom = boucle
 decalage = 8
 ligne n°8 ]

[ section = .text
 nom = end
 decalage = 28
 ligne n°14 ]

[ section = .data
 nom = compteur
 decalage = 0
 ligne n°17 ]

[ section = .data
 nom = fin_compteur
 decalage = 4
 ligne n°19 ]
```

I.3) Retour sur l'incrément 1 et gestion d'erreur

Durant cet incrément, nous avons également amélioré notre gestion d'erreur. En effet, nous avons fait en sorte que les erreurs n'arrêtent pas le programme et pour qu'il soit plus facile de corriger notre programme nous les avons fait afficher avec leur numéro de ligne correspondante avec un commentaire. Cela permet une visualisation rapide de l'endroit de l'erreur et une correction plus facile.
Nous avons également retravaillé un petit peu l'incrément numéro un sur les points qui avaient été souligné lors du retour fait par les encadrants.

I.4) Test du deuxième livrable

Le code rendu avec ce rapport compile et exécute le code sans erreur. Il retourne trois files qui contiennent les directives associées aux sections ainsi que leur décalage par rapport au début de la section, leur numéro de ligne ainsi que le/les opérandes effectives.

Nous avons testé ce code sur plusieurs fichiers test qui regroupent les différents cas d'erreur que nous sommes susceptibles de rencontrer : instructions invalides, nombre d'opérandes invalides etc… Encore une fois, les problèmes de dépassement de mémoire par rapport aux opérande ne sont pas traités ici.
Voici un exemple de ce que le code nous renvoie :

```
.set noreorder
.text
    LW $t0, compteur
    LW $t1, fin_compteur

boucle:
    BEQ $t0, $t1, end
    NOP
    ADDI $t0, $t0, 1
    J boucle
    NOP
end:
    J end
.data
compteur:
    .word 1
fin_compteur:
    .word 100
```

```
---------------------------- FILE D'INSTRUCTIONS -------------------
[ nom = LW
 nombre d'operande = 2
 decalage = 0
 numero de ligne = 5
 -> operande n°1 = $t0
 -> operande n°2 = compteur
 ]

[ nom = LW
 nombre d'operande = 2
 decalage = 4
 numero de ligne = 6
 -> operande n°1 = $t1
 -> operande n°2 = fin_compteur
 ]


---------------------------- FILE DE DATA -------------------------
[ nom = .word
 etat = DECIMAL
 decalage = 0
 numero de ligne = 18
 valeur = 1 ]

[ nom = .word
 etat = DECIMAL
 decalage = 4
 numero de ligne = 20
 valeur = 100 ]


---------------------------- FILE DE BSS --------------------------
FILE VIDE
```