DEBAIN Yann
LETOUZE Clément

Projet informatique : assembleur MIPS
Livrable 1

2017-2018

SOMMAIRE:

Introduction

Le but de ce premier livrable est d'être capable d'effectuer l'analyse lexicale d'un code écrit en langage assembleur MIPS. Cette analyse lexicale passera par une standardisation du texte à analyser, puis par l'analyse lexicale à proprement parlé en catégorisant chaque mot du code assembleur (voir partie XX) et enfin le stockage de ces derniers avec les paramètres qui le représente dans une file.
Nous allons vous présenter brièvement chaque étape du programme que nous avons mis en place pour la réalisation des fonctions décrites précédemment.

I.1) Standardisation du code assembleur

La standardisation du code assembleur est l'étape qui précède l'analyse lexicale. Cette dernière consiste à faire en sorte que l'analyse lexicale se déroule sans erreur. En effet, il est impératif de donner comme argument d'entrée de l'analyse lexicale une chaîne de caractère cohérente et surtout compréhensible par le programme d'analyse lexicale.
La standardisation permet d'ajouter ou de supprimer des espaces et/ou tabulations pour que la chaîne de sortie de ce programme soit lexicalement correcte, par exemple, une étiquette devra être écrite de la manière suivante "etiq:" et non pas "etiq :".

Cette standardisation repose sur l'analyse de tous les caractères spéciaux qui peuvent apparaîtrent dans un lexème du code assembleur, par exemple ":", "-", "$" etc… On effectue alors un test sur les caractères de la chaîne de caractère d'entrée (in[i]). Si le ième caractère est de la chaîne d'entrée ne fait pas parti des caractères spéciaux des lexèmes, on recopie ce dernier dans la chaîne de sortie (out[j]), en revanche si le ième caractère de cette chaîne correspond à un caractère spécial de lexème, on analyse son entourage (in[i+1] et/ou in[i-1]) pour tester par exemple les espaces sont placés correctement. Ici, le point important est de bien tester tous les caractères spéciaux que l'on peut être amené à rencontrer dans un lexème.
Voilà ci-dessous un exemple de test pour le cas où un registre est mal écrit :

```
for ( i= 0, j= 0; i< strlen(in); i++ ) {

    if (isblank(in[i]) && in[i-1]=='$') {
        out[j]=in[i+1];
    }
```

```
in = $ 01
out = $01
Program ended with exit code: 0
```

Exemple de standardisation

I.2) Analyse lexicale

Une fois le texte que l'on cherche à analyser a été standardisé, nous pouvons effectivement en faire son analyse lexicale.
Pour que le programme assembleur fonctionne correctement, il est impératif que tous les mots présents dans ce dernier est une réelle signification pour ce langage. Il est donc impératif d'effectuer cette vérification. Le rôle de cette analyse va être de détecter que tous les mots sont effectivement compréhensibles pour le langage, par exemple, si quelque part dans le code assembleur un nombre est : "0xaafu" cette analyse doit être capable de détecter une erreur car ce mot est un nombre hexadécimal qui n'existe pas.

Pour mener à bien cette analyse, nous avons mis en place un automate.
Le principe de ce dernier est le suivant : on analyse ligne par ligne le code assembleur. Et dans chaque ligne on effectue une analyse mot par mot. Ensuite, on parcourt le mot et à chaque nouveau caractère on effectue un test, chaque test renvoyant à un état différent par l'intermédiaire d'un switch()/case:.

```
5  enum {
6      INIT, MOT_OU_INSTRUCTION, ETIQUETTE, REGISTRE, COMMENTAIRE, DIRECTIVE, SIGNE_MOINS, DECIMAL_ZERO,
           DEBUT_HEXADECIMAL, DECIMAL, OCTAL, HEXADECIMAL, ERREUR, VIRGULE, PARENTHESE
7  } ETAT;
8
```

```
1   10
2   #b
3   a
4   $a
5   .a
6   0
7   0a
8   07
9   0xa
10  0xg
11  #a baba
12  a:
13  a (frfzrf)
14  a, fefz
15
```

Texte assembleur

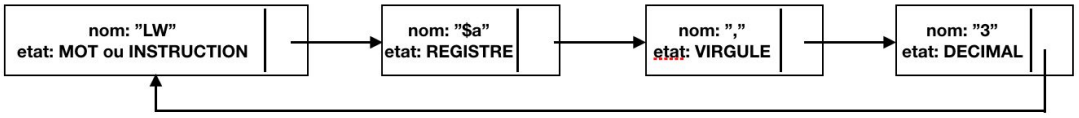
```
10 || c'est un DECIMAL
#b || c'est un COMMENTAIRE
a || c'est un MOT ou une INSTRUCTION
$a || c'est un REGISTRE
.a || c'est une DIRECTIVE
0 || c'est un ZERO
0a || Il y a une ERREUR
07 || c'est un OCTAL
0xa || c'est un HEXADECIMAL
0xg || Il y a une ERREUR
#a baba || c'est un COMMENTAIRE
a: || c'est une ETIQUETTE
a || c'est un MOT ou une INSTRUCTION
( || c'est une PARENTHESE
frfzrf || c'est un MOT ou une INSTRUCTION
) || c'est une PARENTHESE
a || c'est un MOT ou une INSTRUCTION
, || c'est une VIRGULE
fefz || c'est un MOT ou une INSTRUCTION
```

Résultat affiché par la console

I.3) Stockage du code dans une file

Pour les analyses qui suivront dans ce projet, il est impératif de pouvoir avoir accès à chaque mot du code ainsi que le type de lexème qu'il représente. C'est pourquoi nous devons stocker chacun des mots que nous analysons. Cette dernière opération se fera via une structure de file. On peut assimiler cela à une liste chaînée "circulaire" dont chaque cellule est en fait une structure LEXEME qui comporte 2 "cases" : une pour stocker le mot (i.e la chaîne de caractère) et une pour stocker son état (étiquette, registre…), la cellule pointe alors ensuite sur la cellule suivante qui représente le mot suivant dans la ligne de code assembleur. On obtient comme cela une structure file qui contient tous les mots de chaque ligne du code ainsi que leur type et nous pourrons comme cela y avoir accès par la suite. On peut donner le schéma suivant à titre d'exemple si les seuls mots du code assembleur sont les suivant : LW $a , 3

file de l'instruction : LW $a , 3




```
8
9   typedef struct {
10      char* nom;
11      char* etat;
12      int valeur;
13  } LEXEME;
14
15  struct cell {
16      LEXEME val;
17      struct cell* suiv;
18  };
19
20  typedef struct cell* File;
```

structure LEXEME et File

I.4) Test du premier livrable

Etat du premier livrable

Le code fournit avec ce rapport compile et exécute le code sans erreur. Le programme retourne la liste des lexèmes ainsi que son type en fonction de la ligne du code.

Le programme parcours donc chaque ligne d'un fichier assembleur MIPS et analyse chaque mot afin de déterminer son état (en stockant une chaîne de caractères et son état). Le cahier des charges de ce premier livrable est respecté.

Tests effectués

Nous avons testé le code sur différents fichiers assembleurs afin de vérifier le bon fonctionnement du programme. Nous avons pu mettre en lumière des problèmes soit dans la canonisation des phrases, soit dans l'analyse lexicale ou soit dans le stockage des lexèmes et ainsi les corriger dans la phase de débogage.

Point d'amélioration

On peut penser à stocker chaque phrase dans un tableau ou dans une autre file pour avoir accès à tout le code quand on veut.

Exemple d'exécution

```
1  # addition
2
3  .set noreorder
4  .text
5      LW $t0, a
6      LW $t1, b
7      LW $t2, res
8      ADD res, a, b
9  .data
10 a:
11     .word 5
12 b:
13     .word 2
14 res:
15     .word 0
```

Texte assembleur

```
# addition || c'est un COMMENTAIRE
.set || c'est une DIRECTIVE
noreorder || c'est un MOT ou une INSTRUCTION
.text || c'est une DIRECTIVE
LW || c'est un MOT ou une INSTRUCTION
$t0 || c'est un REGISTRE
, || c'est une VIRGULE
a || c'est un MOT ou une INSTRUCTION
LW || c'est un MOT ou une INSTRUCTION
$t1 || c'est un REGISTRE
, || c'est une VIRGULE
b || c'est un MOT ou une INSTRUCTION
LW || c'est un MOT ou une INSTRUCTION
$t2 || c'est un REGISTRE
, || c'est une VIRGULE
res || c'est un MOT ou une INSTRUCTION
ADD || c'est un MOT ou une INSTRUCTION
res || c'est un MOT ou une INSTRUCTION
, || c'est une VIRGULE
a || c'est un MOT ou une INSTRUCTION
, || c'est une VIRGULE
b || c'est un MOT ou une INSTRUCTION
.data || c'est une DIRECTIVE
a: || c'est une ETIQUETTE
.word || c'est une DIRECTIVE
5 || c'est un DECIMAL
b: || c'est une ETIQUETTE
.word || c'est une DIRECTIVE
2 || c'est un DECIMAL
res: || c'est une ETIQUETTE
.word || c'est une DIRECTIVE
```

Exécution du code stockage des lexèmes

```
-------- ligne 1 ---------
# addition est de type COMMENTAIRE

-------- ligne 3 ---------
.set est de type DIRECTIVE
noreorder est de type MOT_OU_INSTRUCTION

-------- ligne 4 ---------
.text est de type DIRECTIVE

-------- ligne 5 ---------
LW est de type MOT_OU_INSTRUCTION
$t0 est de type REGISTRE
, est de type VIRGULE
a est de type MOT_OU_INSTRUCTION

-------- ligne 6 ---------
LW est de type MOT_OU_INSTRUCTION
$t1 est de type REGISTRE
, est de type VIRGULE
b est de type MOT_OU_INSTRUCTION

-------- ligne 7 ---------
LW est de type MOT_OU_INSTRUCTION
$t2 est de type REGISTRE
, est de type VIRGULE
res est de type MOT_OU_INSTRUCTION

-------- ligne 8 ---------
ADD est de type MOT_OU_INSTRUCTION
res est de type MOT_OU_INSTRUCTION
, est de type VIRGULE
a est de type MOT_OU_INSTRUCTION
, est de type VIRGULE
b est de type MOT_OU_INSTRUCTION
```

Exécution du code avec stockage des lexèmes