Projet informatique SICOM
CAMUS-KAPLAN

Livrable n°1 : Analyse lexicale

Introduction

Ce projet a pour but de réaliser un compilateur de programmes écrits en langage assembleur MIPS. Ce programme, que l'on appelle Assembleur et qui est beaucoup plus simple qu'un compilateur de langage de haut-niveau, sera écrit en langage C.

La première étape du travail consiste à lire les instructions, non compilées dans un fichier, et à les classer dans différentes catégories définies (INIT , DECIMAL_ZERO, DEBUT_HEXA, HEXA, LETTRE_MIN, etc...) : c'est l'analyse lexicale. Pour cela, on doit d'abord lire les différents types lus via une machine à états finis. Ensuite il faut les classer dans des tableaux particuliers, nommés des piles, pour pouvoir les lire plus facilement lors de la phase suivante : l'analyse syntaxique.

1. Définition de la machine à états finis

Une machine à états finis (ou FSM pour finite state machine en anglais) décrit les états dans lesquels un automate est autorisé à être. Une FSM décrit aussi quoi faire lorsque l'on atteint un état donné, ou même ce qu'il faut faire lorsque l'on passe de tel à tel autre état.

Ici notre machine à état consiste à lire une chaîne de caractères sans espace, caractère par caractère, pour déterminer son type. La structure que nous élaborer se présente de la manière suivante, d'après la figure 1 :

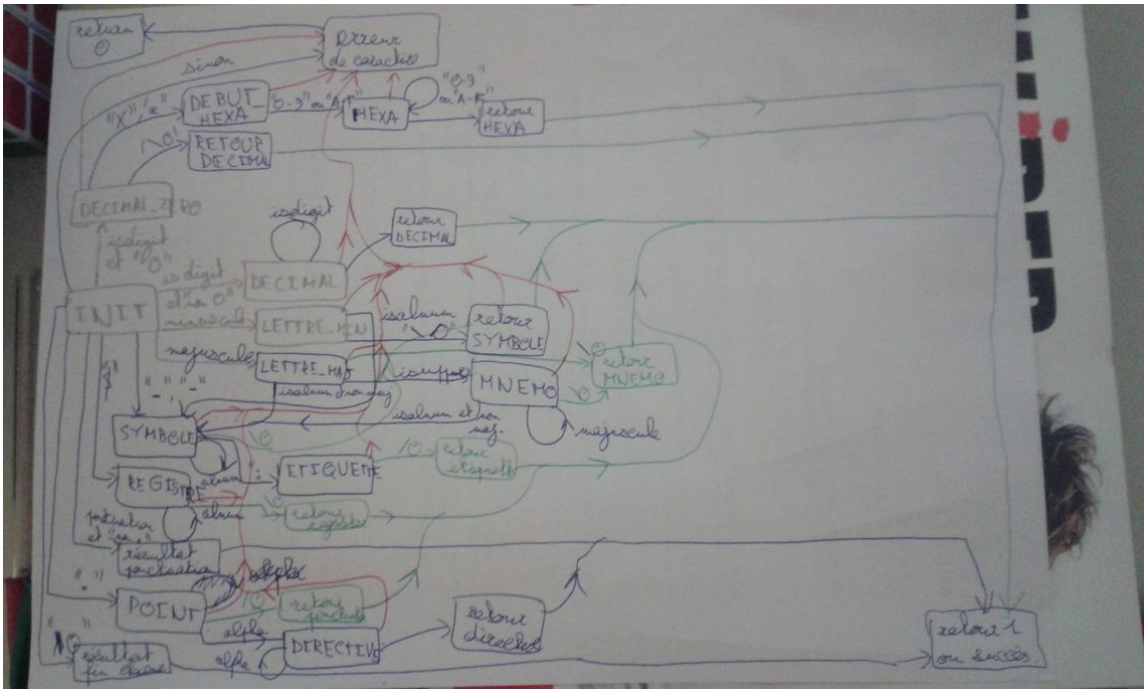

Figure 1 : définition de la machine à états finis

Pour le premier caractère lu, on part de la case INIT. puis on charge l'une des cases définies selon les différentes conditions réalisées. On utilise la bibliothèque <ctype.h> permettant de reconnaître les différents types de caractères. Les commentaires et les expressions entre guillemets

ne sont pas pris en compte dans l'automate, mais dans la fonction newGetNextToken, permettant de segmenter une ligne en plusieurs mots pour la machine à état.

2. Gestion des tests et des erreurs

Principe

Au début du programme, on avait des fonctions de base pour lire les fichiers, les lignes et les mots dans le programme *lex.c* du répertoire *step0*, avec notamment la fonction GetNextToken incomplète. Il manquait la fonction de la machine à états, *machine_etat*, dans cette fonction permettant de créer des chaînes de caractères, sans espace. Les premiers tests ont été fait dans un dossier différent du compilateur *test_nGNT.*

Pour cela, on a d'abord fait, dans un enum dans lex.h, la liste de tout les types de lexèmes dont on avais besoin (*enum {INIT , DECIMAL_ZERO, DEBUT_HEXA, HEXA, LETTRE_MIN, LETTRE_MAJ, DECIMAL, REGISTRE, SYMBOLE, MNEMO, ETIQUETTE, DIRECTIVE, POINT} ;*). Ensuite on a programmé la machine à état, d'après les descriptions sur la figure 1. On l'a implémenté dans la nouvelle fonction *newGetNextToken* dans le fichier c *supertestlex.c*.

Résultats des tests séparés

Pour tester la nouvelle fonction , avant de la mettre dans le programme du compilateur, on a fait un test sur trois chaînes de caractères, "#Newbie\n" (commentaire), "Toto: ADD $1, $2" (message normal) et ".sh JAP 5,6" (séparation sans espace avec la virgule). Le résultat détaillé donne pour la troisième :

```
start:.sh  JAP 5,6
end:  JAP 5,6
etat= 0
etat= 12
etat= 11
etat= 11
[DIRECTIVE      ]          .sh
.sh
start:JAP 5,6
end: 5,6
etat= 0
etat= 5
etat= 9
etat= 9
[MNEMO          ]          JAP
JAP
start:5,6
end:,6
etat= 0
etat= 6
[DECIMAL        ]          5
 5
5
start:6
end:
etat= 0
etat= 6
[DECIMAL        ]          6
 6
6
start:
end:
```

Les variables start et end correspondent aux bouts de chaîne découpés dans *newGetNextToken.* Le mot sans espace analysé après est start-end. Les numéros de la variable état correspondent à la position d'une constante en particulier dans l'enum. On a de façon non détaillée :

```
[COMMENT   ]#Newbie

[ETIQUETTE    ]        Toto:
Toto:
[MNEMO        ]        ADD
ADD
[REGISTRE     ]        $1
$1
[REGISTRE     ]        $2
$2
[DIRECTIVE    ]        .sh
.sh
[MNEMO        ]        JAP
JAP
[DECIMAL      ]        5
5
[DECIMAL      ]        6
6
```

On a donc un bon affichage des résultats escomptés par rapport à la présentation du projet.

Résultats implémentés au compilateur

Par la suite, nous avons implémenté notre programme dans les fichiers pour construire l'assembleur. Une fois celui-ci assemblé, on obtient les résultats suivants pour le fichier *miam_sujet.s* en extrait *:*

```
[COMMENT   ]# TEST_RETURN_CODE = PASS
[COMMENT   ]# allons au ru
[DIRECTIVE    ]        .set
.set
[SYMBOLE      ]        noreorder
noreorder
[DIRECTIVE    ]        .text
.text
[SYMBOLE      ]        Lw
Lw
[REGISTRE     ]        $t0
$t0
[SYMBOLE      ]        lunchtime
lunchtime
[MNEMO        ]        LW
LW
[REGISTRE     ]        $6
$6
```

Les résultats ressemblent à ceux du répertoire *test_NGNT, ce qui indiquent que le programme marche bien. Si on fait tourner Valgrind en même temps pour le contrôle de la mémoire, on a :*

```
==3731== HEAP SUMMARY:
==3731==     in use at exit: 243 bytes in 46 blocks
==3731==   total heap usage: 49 allocs, 3 frees, 5,915 bytes allocated
==3731==
==3731== LEAK SUMMARY:
==3731==    definitely lost: 243 bytes in 46 blocks
==3731==    indirectly lost: 0 bytes in 0 blocks
==3731==      possibly lost: 0 bytes in 0 blocks
```

```
==3731==    still reachable: 0 bytes in 0 blocks
==3731==         suppressed: 0 bytes in 0 blocks
```

On n’a pas de fuite de mémoire. Donc le programme est stable. Cela proue que l’analyse s’exécute bien. Cependant certains objectifs sont à relever pour la prochaine fois.

3. Bilan et objectifs à venir

L’analyse lexicale se fait bien. La prochaine phase sera l’analyse syntaxique où il faudra charger un dictionnaire des instructions de l’assembleur MIPS, créer une file pour stocker les lexèmes de l’analyse lexicale et les comparer à ce dictionnaire. Cela est en cours de réalisation pour l’instant, mais pas encore au point.

On a eu quelques difficultés durant ce de projet. On était incertain par rapport à la réussite du programme final. On a donc créé notre répertoire à part de tests pour éviter les dérives, quitte à perdre du temps.