Rapport
Projet Informatique : Assembleur MIPS

Livrable 1

Année Universitaire 2018 - 2019

BRITTON Giovani - Giovani-Crasby.Britton-Orozco@grenoble-inp.org
WRIGHT Gilles - gilles.wright@grenoble-inp.org

Professeur encadrant : Katell Morin-Allory - katell.morin-allory@imag.fr

Filière : SEI

I. Démarche Générale

Le but de ce projet informatique est de réaliser un assembleur MIPS, codé en C, capable de compiler un programme écrit en assembleur. La première étape consiste à réaliser la lecture et l'identification des différents lexèmes ainsi que leurs classement par type : c'est l'analyse lexicale. Pour ce faire, trois débuts de fonctions permettant d'une part la lecture d'un fichier texte - *lex_load_file(char *file, unsigned int *nlines)* - , et d'autre part le découpage du texte en lexème - *getNextToken(char** token, char* current_line)* - et l'identification des lexèmes lus - *lex_read_line(char *line, int nline)* -, étaient à disposition. Les fonctions étant imbriqués, il faut d'abord bien réaliser le découpage du texte en lexèmes afin que ces derniers puissent ensuite être analysés afin de déduire leur nature. Le programme devra aussi gérer les erreurs, c'est-à-dire repérer lorsque la syntaxe du lexème lu ne correspond pas au format admissible en langage assembleur MIPS.

II. Modélisation du problème

Afin de réaliser la découpe des mots en lexème, il a d'abord fallu répertorier l'ensemble des catégorie de mot du langage assembleur que l'on peut rencontrer : [COMMENTAIRE], [SYMBOLE], [DIRECTIVE], [REGISTRE], [DEUX-POINTS], [VIRGULE], [DECIMAL], [HEXADÉCIMAL]. Le retour à la ligne n'a pas été pris en compte parmis les états. Ainsi la fonction de découpage en lexème n'en tient pas compte. Cependant, afin de sauvegarder la ligne d'un lexème pour la gestion des erreurs, un décompte des lignes est réalisé.

Dans un premier temps, la fonction *getNextToken()* découpe les lexèmes selon le premier caractère du mot, en prenant en comptes les différents séparateurs (l'espace, la virgule, le retour à la ligne, les deux-points, le diez, les parenthèses et les guillemets.)

Dans un deuxième temps, les lexèmes sont récupérés dans la fonction *lex_read_line()* pour déterminer leur nature. De nouveaux états ont été introduits pour une bonne analyse des lexèmes. Cela peut être modélisé par la machine à états présenté en annexe.

III. Organisation du travail

Au début, il a fallu étudier les différentes catégories de lexèmes du langage assembleur MIPS. Ensuite, du temps a été consacré à la compréhension des fonctions de départ. Une fois cela fait, nous nous sommes alors focalisés sur la réalisation de la machine à états de l'analyse lexicale. Le travail a ensuite été réparti comme suit:

Tâches	Semaine 1	Semaine 2	Semaine 3
- Analyse du sujet et des fonctions fournis	X X		
- Réalisation de la machine à états	X X		
- Codage de la fonction getNextToken + tests		X	
- Codage de la machine à états + tests		X	
- Réalisation et codage des structure de files + test		X	X
- Rédaction du rapport			X X

Légende :

réalisé par : Giovani X Gilles X

Figure 1 : Répartition et organisation du travail

Afin de pouvoir travailler simultanément sur le programme, et communiquer nos avancés, le site Github a été utilisé. Le codage des deux fonctions *getNextToken()* et *lex_read_line()* à donc été réalisé et testé séparément avant que le travail ne soit regroupé pour tester le bon fonctionnement de l'analyse lexicale complète. Ce n'est que par la suite que le stockage des lexèmes n'a été abordé. Finalement, l'analyse lexicale s'est terminé par des essaies sur les différents fichiers de test en langage assembleur, afin de vérifier le bon fonctionnement du programme..

IV. Méthode de gestion des erreurs

Pour la gestion des erreurs, nous avons préféré ne pas utiliser un état d'erreur, car il faudrait par la suite identifier cette erreur pour en informer l'utilisateur. Nous avons choisi une méthode similaire au compilateur en C qui n'arrête pas la compilation à la première erreur rencontrée, mais qui analyse tout le code et qui renvoie à l'écran toutes les erreurs rencontrées. Pour réaliser cela, lorsqu'on est dans un état et que le caractère suivant engendre une erreur, un *printf()* est réalisé, renvoyant la ligne d'erreur, le lexème posant problème, ainsi que l'état dans lequel se trouvait le lexème au moment de l'erreur.

V. Tests réalisés

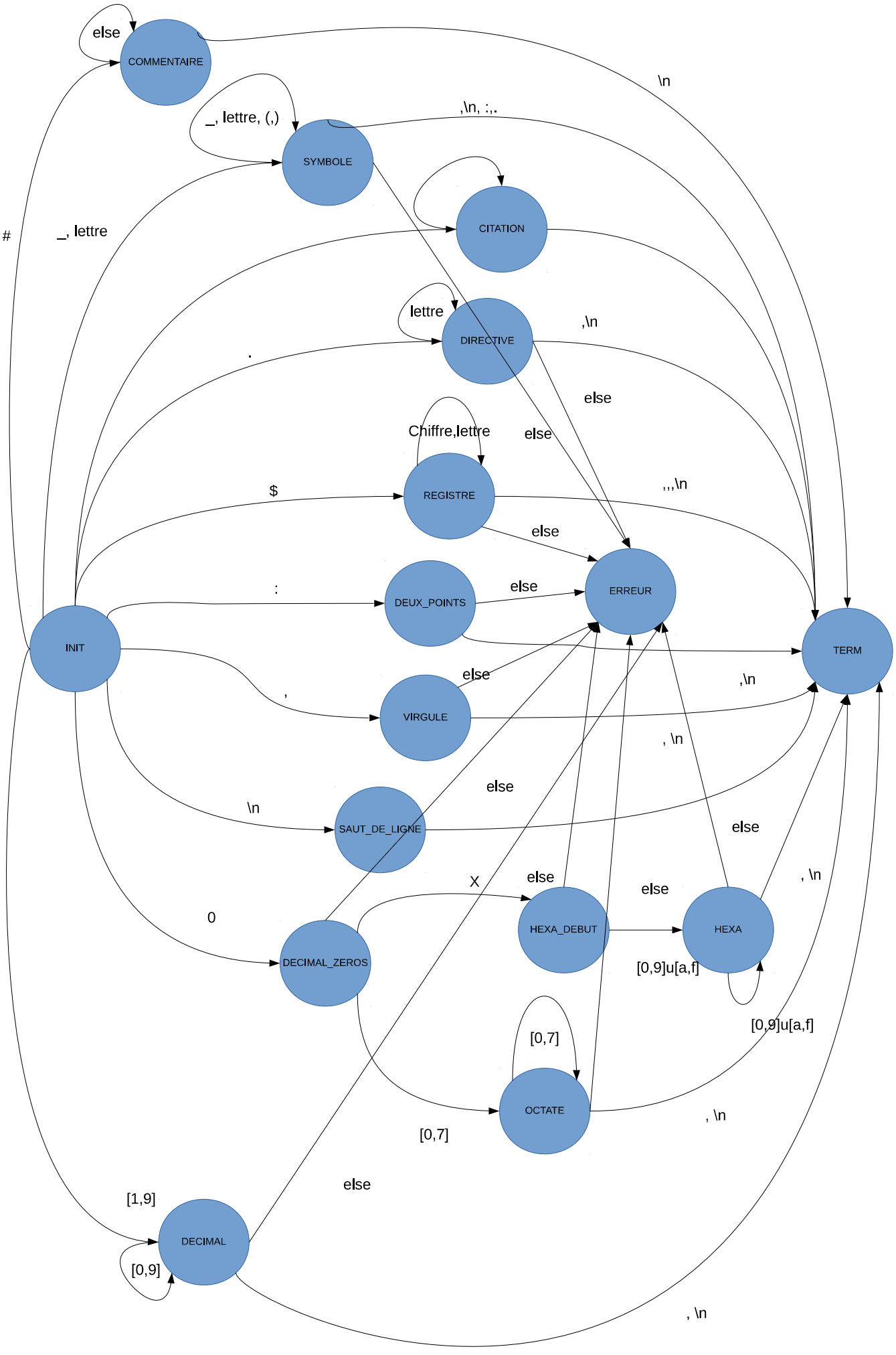
Pour tester le programme complet, nous avons utilisé le code en assembleur qui était fourni. Nous avons par la suite, écrit un deuxième code en assembleur regroupant une bonne partie de la syntaxe utilisé en langage assembleur mais aussi des chaînes de caractères sans espaces, ou pouvant générer des erreurs, pour voir si le programme découpe bien tous les lexèmes, et les identifie bien. Nous avons vérifié nous même si les mots étaient bien découper, et bien associé à leur type. Par la suite, lorsqu'il a fallu utiliser les files pour stocker les lexèmes, nous avons réalisé un programme de test qui vérifie le bon fonctionnement de chaque fonction. Cependant notre dernier fichier test pour les fonction sur les file ne fonctionne pas encore correctement.

VI. Problèmes rencontrés et travail restant à faire

En ce qui concerne le travail restant à faire, une amélioration de la gestion des erreurs est encore possible. Il serait intéressant par exemple d'identifier les possibles causes d'erreurs et pas uniquement l'état posant problème. D'autre part, le choix de la structure pour stocker les lexème, ainsi que la réalisation de cela, a requis beaucoup de temps. Les fonctions sur les files posent encore problème. Lors des tests la création des files fonctionnait bien, mais les chaînes de caractères ne sont pas bien recopiés. Ainsi nous avons uniquement renvoyer l'analyse lexicale grâce à des printf. Nous reviendront sur le stockage par la suite. Enfin la complexité du programme pourrait être améliorée, car le code en assembleur est parcouru deux fois : lors du découpage du lexème et lors de l'identification de sa catégorie. Néanmoins cela ne devrait pas poser problème tant la taille du code en assembleur reste raisonnable.

VII. Conclusion

Pour conclure, un des facteurs les plus importants de l'analyse lexicale est de bien découpé les lexèmes pour qu'ils puissent être identifiés par la suite. De plus il faut stocker ces informations de telle façon qu'elles puissent être utilisées par la suite lors de l'analyse grammaticale.


else
COMMENTAIRE
\n
_, lettre, (,)
,\n, :,.
SYMBOLE
#
_, lettre
CITATION
lettre
DIRECTIVE
,\n
.
else
Chiffre,lettre
else
$
REGISTRE
,,,\n
else
ERREUR
:
DEUX_POINTS
else
INIT
TERM
,
VIRGULE
else
,\n
, \n
\n
SAUT_DE_LIGNE
else
else
X
else
else
, \n
0
HEXA_DEBUT
HEXA
DECIMAL_ZEROS
[0,9]u[a,f]
[0,9]u[a,f]
[0,7]
OCTATE
, \n
[0,7]
else
[1,9]
DECIMAL
[0,9]
, \n