Projet Informatique : Assembleur MIPS

LIVRABLE 1

Colin EPALLE, Eléonore BARGES

Pour aborder ce projet, nous avons adopté une certaine démarche que nous nous proposons de détailler dans ce livrable.

PLAN :
I. Dans un premier, nous détaillerons notre démarche, comment nous avons modélisé le problème et l'organisation de notre travail.
II. Puis nous détaillerons un peu plus les problèmes que nous avons rencontrés ainsi que les objectifs que nous avons atteints dans cette première étape.
III. Nous terminerons par les différents tests que nous avons effectués afin de faire un suivi de notre projet, et les choix que nous avons fait concernant les erreurs.

I. Notre Démarche et organisation de travail

Pour commencer, nous avons dû cerner le problème posé. Après avoir effectué une analyse personnelle des objectifs et des moyens dont nous disposions pour les achever, nous avons fait une mise en commun afin de nous accorder sur la stratégie à mettre en œuvre. Ainsi, nous avons d'abord tenté de nous familiariser avec le projet en rédigeant notre propre code.

Les étapes que nous souhaitions que notre code parcourt étaient les suivantes :

1. récupération du fichier
2. lecture de celui-ci ligne par ligne grâce à fgets
3. normalisation de la ligne (voir plus loin ce qu'on entend par là)
4. division de la ligne en token grâce à strtok
5. analyse des types de token grâce à la machine à état
6. stockage des lexèmes, soit des tokens avec leur type associé, ainsi que leur numéro de ligne (utile pour les erreurs)

Concernant les choix généraux de code que nous avons faits, nous avons choisi d'utiliser des files (QUEUE) pour ajouter plus facilement en queue de liste les lexèmes, de sorte à ce que l'ordre de lecture de la liste des lexèmes corresponde à l'ordre de lecture des tokens dans le fichier. Une file n'est rien d'autre qu'une liste bouclée sur elle même : la transformation d'une file à une liste est très facile.

Nous avons également fait le choix de mettre dans la structure Queue un élément de type void* afin de pouvoir réutiliser cette structure de Queue pour stocker d'autre élément. Cependant cet aspect n'est pas encore utilisable car pour l'instant les fonctions traitants les files et les listes (lecture, libération) ne fonctionnent que si l'on stocke des lexèmes.

En outre, en anticipant l'étape de la machine à état, nous avons supposé que la détermination des types des lexèmes serait probablement facilitée si les chaînes de caractère étaient "normalisées" avant de passer par **strtock**. C'est-à-dire si nous isolions les symboles ":", "," et "-" afin de pouvoir, plus tard, facilement identifier les étiquettes, ou encore les différents arguments des instructions, sans que ceux-ci soient, par exemple, regroupés en un seul et même token: ex: "arg1,arg2,arg3" mais plutôt en "arg" "," "arg2" "," "arg3". Cependant, après avoir codé cette fonction, nous avons

estimé qu'il était plus simple et plus correct (pas de modification de chaîne de caractère, …) de séparer les tokens directement dans la fonction GetNextToken (par exemple via des cas particuliers pour les caractères de type ":" , "," et "-" que l'on considère comme séparateur en plus de l'espace). Nous avons donc conclu que la création d'une fonction Normalisation et l'utilisation de **strtok** n'était pas nécessaire.

Une fois ces étapes-ci passées, nous avons récupéré les squelettes de code préconçus car afin de nous assurer de la propreté et de la rigueur de notre code.

Concernant la division du travail, nous avons lors des premières séances travaillé à deux sur un ordinateur, codant ensemble, pour nous familiariser au problème, mais aussi pour nous mettre d'accord sur une "manière de coder" qui nous soit compréhensible pour tout deux. Cependant, cette méthode était peu efficace et très chronophage.

Dans un second temps, nous avons séparé les tâches: l'un de nous faisait certaines "petites fonctions" tandis que l'autre avançait la structure d'une fonction principale. Ou encore l'un de nous travaillait sur les erreurs ou sur les tests, tandis que le second avançait le programme. Nous débriefions souvent pour être au courant du travail de l'autre.

II. Complications et objectifs

Lors de cette première étape, nous avons rencontré certaines **erreurs et problématiques** que nous nous proposons de résumer brièvement ci-dessous:

- Nous avons d'abord songé à ne pas stocker les commentaires et les retours à la ligne lors de la lecture du fichier, pensant d'abord que c'était des informations inutiles pour l'assemblage. Finalement, nous avons estimé que pour le débogage et probablement pour la suite du projet, il était plus judicieux de les stocker néanmoins.

- Nous avons un peu du temps très bêtement. Nous avions écrit :

```
QUEUE Q = calloc (1, sizeof(*QUEUE)) ;
```

Ce qui n'a pas vraiment de sens …

- Pour la machine à état, nous avons décidé de prendre en compte la structure de type : AIBD, soit Adressage Indirect avec Base et Déplacement (par exemple 200($4)). Cette forme nous a posé plusieurs questions: nous avons hésité à mettre un type intermédiaire, DEBUT_AIBD, qui aurait permis de détecter les lexèmes pour lesquels il manquait la parenthèse finale (condition nécessaire à l'état AIBD simple), ou a simplement définir un AIBD comme un lexème qui contient après des chiffres une parenthèse ouverte et dont le dernier élément aurait été une parenthèse fermée. Finalement, nous avons choisi la première option pour la raison suivante : dans le cas d'une erreur de syntaxe de ce type 200($4)500($8), il nous semblait mieux, au niveau du débogage, de renvoyer une erreur stipulant la "présence anormale d'un terme après la première parenthèse fermée" dès l'étape de la machine à état. Cependant, si nous avions choisi la seconde option, l'erreur n'aurait pu être constatée à cette étape, et il aurait fallu attendre l'analyse grammaticale pour déterminer que "$4(500)$8" n'est pas un registre existant.

- Nous avons également rencontré une certaine difficulté dans la fonction GetNextToken afin de permettre la lecture des commentaires et des chaines de caractère comme UN seul et même token. Cependant, nous avons réussi notamment en tenant à jour les variables booléens inString et inComment qui valent TRUE si l'on se trouve dans une string/commantaire, et avec la condition dans le while :

while (*end!='\0' && ((!isblank(*end) && *end!=':' && *end!=',' && *end!='-') || inString || inComment))

Les objectifs que nous avons atteints dans cette première étape sont les suivants :
- Notre code permet de déterminer les types des lexèmes selon les choix que nous avions effectués.
- Notre code nous semble assez robuste, c'est-à-dire capable de repérer un certain nombre d'erreur dès cette première étape, et de gérer tout (?) les cas possibles
- Nos tokens avec leur type et leur numéro de ligne ont été stockés avec succès dans une liste.

III. Tests et gestion des erreurs

Enfin, concernant les **tests** que nous avons réalisés:

- Nous avons écrit une fonction test void type_lex_test() qui permet de vérifier si notre machine à état renvoie bien les bon types pour les lexèmes correspondants. Nous avons tenté de mettre des cas possiblement "pièges" mais faisables (qui ne renvoie pas d'erreur).

- Nous avons également écrit une fonction print_list_lex qui permet de voir ce que contient la liste de lexèmes. Elle est d'ailleurs utilisée dans le programme principale.

- Enfin, nous avons aussi beaucoup travaillé sur le fichier miam_sujet.s, en le modifiant volontairement en y mettant certaines erreurs, afin de tester notre code au fur et à mesure de son avancement.

Pour lancer les tests, nous avons fait le choix de mettre une condition au début du main qui repère si la commande lancée sur le terminal est ./as-mips -t. Dans ce cas c'est les fonctions de test qui sont lancées et pas le programme normal.

Pour finir, concernant la détection des erreurs, nous avons choisi d'utiliser une fonction print_error_lex qui prend en paramètre un message personnalisé au type d'erreur à afficher, et qui l'affiche de façon claire en pointant l'erreur sur la ligne du fichier assembleur. Pour cela, il notamment fallu traîner de fonction en fonction le pointeur sur la ligne du fichier.