FOURNIER Augustin
GARRIGOU Théo

Rapport 2 : Analyse Grammaticale

Introduction :

Ce second rendu concernait la partie de l'analyse grammaticale : on rappelle que cette partie permet de classer les lexèmes dans les sections de mémoires .data .bss et .text, elle permet également de répertorier les différentes étiquettes et on vérifie également certains éléments de la syntaxe du code assembleur.

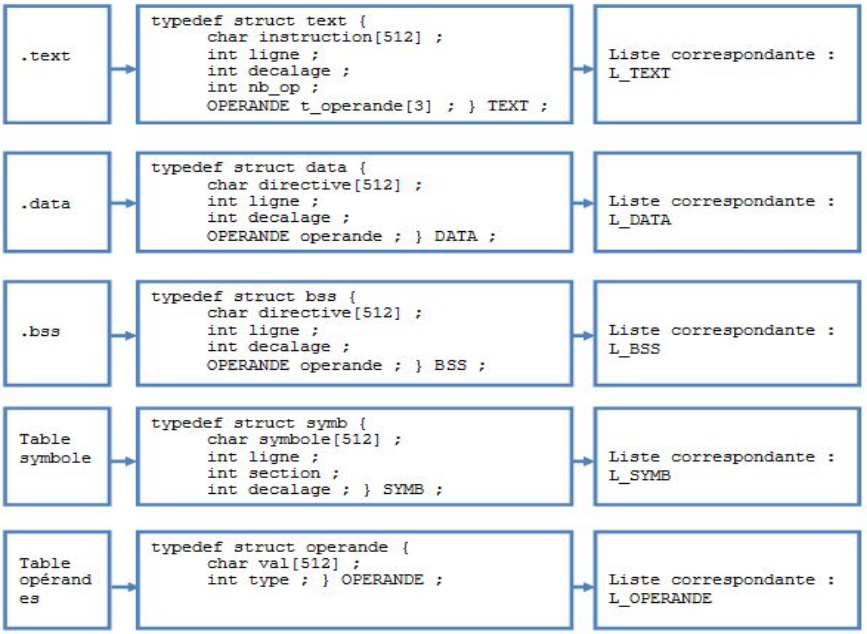
I) La démarche :

L'organisation pour cette seconde partie de projet a été similaire à la première partie. Tout d'abord nous avons lu et décortiqué bien attentivement les conseils du tutorat n°2 ainsi que le dossier puisque cette partie nous a semblé plus complexe à coder. Nous avons commencé par définir les types de structures de donnée que nous utiliserons (gestion des sections, recherche d'existence de commande, gestion des étiquettes). Ensuite nous avons schématisé l'automate sur papier, et enfin nous avons écrit quelques programmes de tests pour vérifier le fonctionnement global du code, bien sur il y a eu des petits tests simples au fur et à mesure de l'avancement du code. Nous avons géré les erreurs à la fin de la partie quand l'automate était terminé. Nous avons également utilisé des listes pour la gestion des trois sections et la gestion des étiquettes, ensuite nous avons utilisé un dictionnaire pour répertorier les différentes instructions assembleur.

II) L'organisation pour cette partie :

Nous avons quelque peu inversé les rôles par rapport au premier rendu. Théo s'est chargé de coder l'automate et de définir les structures (types) dont il avait besoin pour coder l'automate, et Augustin a codé l'ensemble des structures de données associées à ces types : codage de toutes les liste, réalisation du dictionnaire. Enfin Augustin s'est chargé d'écrire les programmes de test assembleur puis Théo a testé l'automate. Nous avons tous les deux interprété les résultats obtenus. Concernant l'organisation des fichiers, nous avons utilisé des .h et des .c en utilisant ce que les tuteurs nous ont conseillé. Pour partager le code nous utilisons désormais gitub pour plus de simplicité.

III) Détail des structures :



IV) Le dictionnaire :

Nous avons choisi d'utiliser un dictionnaire pour décoder et vérifier les instructions (R, I, J) de la section .text. Pour coder nous avons opté une table de hashage, les instructions de référence sont présentées dans un fichier texte sous la forme [nom instructions nombre opérandes type opérandes(R, I, J]. A noter que le nombre d'instructions est noté sur la première ligne du fichier, ensuite on a une seule instruction par ligne du fichier. Les données sont chargées dans la table de hashage entrée dans un tableau de listes (gestion des collisions) du type INSTRUCTION. La fonction de hashage est un modulo entre la somme des numéros ascii des caractères des instructions et la longueur du tableau. Cette dernière est nettement améliorable mais nous avons choisi la simplicité étant donné que le nombre d'instructions pris en compte dans ce projet est assez réduit. Enfin une fonction permettant d'initialiser le dictionnaire et de rechercher la présence d'un élément est disponible. La dernière fonction renvoie tous les paramètres de l'élément si il est reconnu en renvoyant un INSTRUCTION*.
typedef struct

```
{
    char nom_inst[512];
    int nb_op;
    char type_instruction;
} INSTRUCTION;
```

V) Organisation du code pour cette seconde partie :

analyse_grammaticale.h : Contient la définition des types utilisés pour les section .data, .bss, .text et symboles. Elle contient également les en-têtes de ces fonctions.

dictionnaire.h : Contient tous les types et les en-têtes crées spécialement pour le dictionnaire.

analyse_grammaticale.c : Ici sont contenus toutes les fonctions principales de l'analyse grammaticale.
dictionnaire.c : Ici sont contenues toutes les fonctions relatives à la création et à la recherche dans un dictionnaire.

fonction_affichage.c : Ce fichier contient l'ensemble des fonctions permettant d'afficher les listes L_BSS, L_DATA, L_TEXT, et L_SYMB.

liste_directives.c : Contient l'ensemble des fonction des différents types de listes à savoir L_BSS, L_DATA, L_TEXT, L_SYMB et L_OPERANDE.

instruction_dictionnaire.txt : Contient les instructions à charger dans le dictionnaire.

VI) Automate :

Pour réaliser l'analyse grammaticale, on utilise un automate qui va parcourir la liste de lexèmes du précédent incrément.
L'automate est constitué de plusieurs fonctions :

- *init* permet de diriger l'analyse du lexème en cours. A la fin de cette analyse, on appelle init de nouveau
- dans le cas d'une directive, on va appeler une fonction en fonction de la valeur de cette dernière
- dans le cas d'une instruction, on vérifie dans le dictionnaire si la celle-ci existe, puis on charge les opérandes en fonction du nombre demandé
- dans le cas des étiquettes, on utilise d'abord une liste d'attente d'étiquettes pour connaître leur décalage dans la section correspondante (car cette information peut arriver plus tard dans le

code). Une fois le décalage connu, on ajoute l'étiquette à la liste, qui sera à la fin convertie en tableau.

Dans chaque appel de fonction, il est nécessaire de rentrer le décalage en cours pour chaque section : on utilise alors un tableau de taille 3 de int* qui permet d'accéder aux décalages des 3 sections.

On utilise aussi dans presque chaque appel de fonction la section en cours ainsi que les pointeurs vers toutes les listes susceptibles d'être appelées.

Beaucoup de messages pour renseigner l'utilisateur de ses erreur ont été mis en place dans l'automate.

VII) Tests réalisés :

Pour vérifier notre automate, on réalise différents tests avec des fichiers tests plus ou moins long que l'on adapte pour essayer de couvrir un maximum d'erreur. Beaucoup d'erreur ont été corrigée avec cette méthode.

VIII) Objectifs atteints et ce qu'il reste à faire :

Ce qu'il reste à faire aujourd'hui dans cet incrément :

- prendre en compte le type des opérande (J,R…)
- finir le tableau de symbole et son affichage
- prendre en compte les étiquettes (ex: J boucle où boucle est une étiquette)
- finir de gérer les signes (un problème vient du fait que nous avons un lexème signe ne comportant que "-")
- corriger certains messages d'erreurs (notamment pour les lignes)
- corriger un problème qui survient dans l'analyse lexicale avec le 1er lexème lu

Le programme est fonctionnel mais il reste perfectible : nos testent donnent de bons résultats sur un code assembleur qui ne contient pas d'erreurs mais lorsque l'on glisse quelques erreurs en assembleurs certains cas de figure sont traité mais il est relativement facile de buggé le programme. Il faudra donc par la suite au fur et à mesure de l'avancement et des autres tests ajouter d'autres cas de figure de correction d'erreur. Concernant grammaire, nous ne sommes pas encore capable de détecter les erreurs d'opérandes des instructions dans .text à l'aide du dictionnaire comme convenu dans le sujet cela sera traité dans la troisième partie de projet, comme d'autres points de grammaire.

Dans les points améliorés nous avons un code notamment pour l'automate plus simple, mieux découpé et plus clair (nous avons également simplifié l'automate de la partie I) ce qui consolide le code et en facilite la correction. De même nous avons utilisé plus de fonctions prédéfinies lors du projet comme strcpy ou strcmp ce qui simplifie et consolide le code

Un point important dans la suite, il nous arrive encore d'avoir des surprises avec l'analyse grammaticale malgré l'applications des recommandations de notre tuteurs après le premier rendu. Il arrive que l'analyse lexicale plante encore même si nous avons corrigé plusieurs bugs il n'est pas exclu d'en découvrir de nouveaux. Ce point sensible devra absolument être encore consolidé.

On pourra améliorer la structure de dictionnaire notamment la fonction de hashage qui n'est pas efficace si l'on voulais coder un compilateur avec un très grand jeux d'instructions

Conclusion :

Notre code est donc fonctionnel mais perfectible : il se casse relativement rapidement lorsque l'on ajoutes des erreurs dans le code assembleur. Aussi nous avons appliqué les recommandations de notre tuteurs à la suite du livrable I : le code du livrable I a été corrigé en appliquant les conseils, ces conseils ont égalment été appliqué sur la partie II (simplification de l'automate, découpage du code, utilisation de fonctions prédéfinies). Néanmoins nous continuerons à améliorer continuellement notre code au fur et à mesure des problèmes que l'on identifie.