Le 06/11/18

LIVRABLE 2:

Colin EPALLE, Eléonore BARGES

Pour ce second livrable, nous choisissons de poursuivre une trame semblable au premier :

Ainsi, dans un premier, nous expliciterons notre démarche, notre modélisation du problème et l'organisation de notre travail. Puis nous détaillerons un peu plus les problèmes que nous avons rencontrés ainsi que les objectifs que nous avons atteints dans cette première étape et les choses qu'il nous reste à faire. 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.

Pour ce second incrément, nous avons modifié notre manière de procéder afin de gagner en efficacité.

Nous avons donc au préalable longuement parlé du problème et déterminé la façon dont nous souhaitions l'aborder. Nous avons effectué un travail papier poussé, sur lequel nous avons rédigé les principales étapes de la grande machine à états qui devait nous permettre de remplir nos listes, mais aussi précisé les différentes fonctions qu'il nous faudrait réaliser pour structurer notre code et l'alléger. Nous avons également réfléchi ensemble à certains outils que nous pourrions utiliser lors du codage, comme la fonction fscanf, strtol ou encore le principe d'union taguée. Enfin, nous nous sommes mis d'accord sur les informations que nous souhaitions stocker dans nos listes, et sur la mise en place de certaines variables supplémentaires, comme la variable etiq_def, qui doit indiquer si l'étiquette stockée dans une liste (.bss ou .data) a déjà été définie dans la table des symboles, ou encore la variable previous_type_lexem, qui nous permettra de résoudre le problème de la considération du "moins" pour les valeurs numériques des lexèmes.

Concernant la division du travail, nous avons procédé différemment que pour le premier incrément, pour lequel nous travaillions essentiellement ensemble, côte à côté, et donc très lentement. Notre familiarisation avec GitHub nous permet désormais d'avancer de façon beaucoup plus efficace chacun de notre côté, avec à l'appui notre travail écrit préalable. Nous avons principalement avancé pendant les vacances, et nous sommes vu à la fin de celles-ci, pour clore le projet, discuter du livrable, de nos derniers objectifs, et mettre au clair certains points plus flous ou difficiles à comprendre, déjà codés ou non. Cette division du travail a nécessité de notre part une plus grande rigueur dans la rédaction de notre code,

notamment pour faire des choix de noms de variables/fonctions suffisamment clairs pour être facilement interprétés, mais également en terme de rédaction de commentaires, utiles pour la lecture de nos lignes de code par notre binôme.

Cette méthode de travail a été assez fructueuse. Cependant, nous estimons que pour le prochain incrément, nous tâcherons de définir en amont, ensemble, les noms de fonctions et de structure, pour parer à tout problème de redondance.

Pour ce deuxième incrément, nous avons fait certaines erreurs et rencontré quelques problématiques que nous nous proposons de résumer brièvement ci-dessous:

- Au niveau des étapes de l'analyse syntaxique, nous avons d'abord pensé effectuer un premier parcours de la liste des lexèmes pour mettre à jour la table des symboles grâce aux définitions d'étiquettes. Nous espérions éviter d'avoir à mettre à jour une variable qui serait liée aux appels d'étiquette et indiquerait si elles ont déjà été définies à cet instant de parcours de liste. Cependant, nous avons finalement abandonné cette idée elle nécessitait dans tous les cas de calculer les valeurs de décalage, et donc d’analyser la totalité de nos lexèmes (chose que nous aurions dû faire également dans le second parcours de liste) ! Ainsi, il s’est avéré être plus logique de choisir de remplir les listes (.bss, .data, instructions, table des symboles) dans un premier temps et de vérifier la définition des étiquettes appelées ensuite.

- Au niveau des instructions, ces dernières étaient stockées dans le dictionnaire en majuscules (ou du moins, sont mises en majuscules avant d’être chargé) . Cependant, il fallait pouvoir les reconnaître, peu importe leur type de lettres (majuscule ou minuscule). Nous avons finalement décidé de convertir les lexèmes (de façon locale), avant de les chercher dans le dictionnaire, en les mettant totalement en majuscules. Nous avons créé la fonction put_in_uppercase et choisi de travailler à partir de la table ASCII pour passer les lettres en minuscules en majuscules, sans toucher les autres.

- Comme dit précédemment, il nous fallait considérer les lexèmes de type MOINS lors de la lecture du lexème suivant. Nous avons d'abord songé à créer un état S=MOINS dans la machine à états. Cependant, cela "écrasait" l'état S précédent qui indiquait notamment s'il s'agissait d'une instruction ou d'une directive. Finalement, il nous a semblé plus judicieux et facile de simplement créer la variable previous_type_lexem, qu'il nous suffit de prendre en argument dans nos fonctions de stockage de données dans les listes, et d'effectuer l'affectation du signe juste avant le stockage.

- Enfin, sachant que nous devions contrôler les types de données avant de les stocker dans les listes .bss et .data, nous souhaitions les convertir (de chaînes de caractères à nombres). En effet, pour l'heure tous nos lexèmes étaient stockés sous forme de char*. Il nous utilisé la fonction strtol pour faire cette conversion.

- Enfin, suite à la conversion précédemment explicitée, nous avons donc du nous atteler au principe de l'union taguée afin de pouvoir créer une fonction d'ajout dans la liste générique, soit applicable à tous les types de lexèmes. C'était une difficulté principalement au niveau de la compréhension du fonctionnement de telles unions, mais à force de tests, nous réussîmes. Cela donné lieu à la fonction : fill_val_op appelée dans la fonction add_to_current_list .

Les objectifs que nous avons atteints sont les suivants :

- Notre code remplit avec succès les listes suivantes : list_data, list_bss, list_instr et symb_table selon les choix que nous avions effectués.

- Notre code nous semble assez robuste, c'est-à-dire capable de repérer un certain nombre d'erreurs dès cette première étape.

Choses à faire encore concernant cet incrément :

- Faire les free correspondant à notre allocation de la mémoire
- Stocker les adresses de décalages des appels d'étiquettes avec les arguments correspondants dans les listes .bss et .data (pour l'instant nous ne stockons que la chaîne de caractère du nom de l'étiquette)
- Le second passage pour vérifier si les étiquettes non définies au 1er passage le sont maintenant ne semble pas fonctionner
- Les différentes listes créées se retrouvent « à l'envers » car on fait un ajout en tête. Il faudra faire un ajout en queue avec des files que l'on transformera en liste à la fin (comme pour la liste des lexèmes)
- Il y a encore dans le code quelques éléments à finir de coder, notés par un (ou plus...) TODO. Notamment, pour l'instant, il n'y pas de vérification sur la valeur numérique des arguments (exemple : .byte 500 devra donner une erreur car 500>128)

Enfin, concernant les fichiers tests que nous avons réalisés:

- Nous bien sûr dû créer des fonctions qui affichent chacune des listes (dont celle créée par la lecture du dictionnaire) et leurs arguments principaux. Nous avons choisi d'écrire une fonction par type de liste, car cela nous semblait plus clair en termes de lecture du code mais aussi en termes d'affichage. Cependant, leurs structures se ressemblent énormément.

- Nous avons également réalisé une petite fonction de test afin de tester *look_for_inst* , qui nous a simplement permis de nous assurer que cette fonction permettait bien de chercher une instruction dans le dictionnaire.

- La grosse difficulté de cet incrément n'était pas tellement due à une complexité "technique", mais plus à la longueur de la machine à état mise en jeu. Pour la tester pas-à-pas, nous avons travaillé sur un fichier de test .s, que nous rallongions au fur et à mesure que nous étions capables de traiter des cas nouveaux. Grâce à ce fichier de test, nous avons pu également constater que notre détection était probante.

Pour finir, concernant la détection des erreurs, nous avons choisi de réutiliser ERROR_MSG qui prend en paramètre un message personnalisé au type d'erreur à afficher, et qui l'affiche de façon claire et percutante. Avant chacun de ces appels à ERROR_MSG, nous avons rajouté un affichage de la ligne dans le code .s où se trouve l'erreur considérée pour faciliter le débogage.