Rapport projet compilation

Florian SPIRE, Elie ANTOINE

19 Mai 2020

1 Structure de données

La structure de donné que nous utilisons pour notre compilateur est une de table de hachage contenant le nom de la variable (char*), son type (int), et une valeur(int) optionnelle qui nous est utile dans certains cas. Pour la créer nous nous sommes basés sur la table que nous avions mise au point pour le TD6 et l'avons modifié pour ajouter les fonctionnalités que nous voulions comme le fait de pouvoir trouver une entrée de notre table à partir du nom de celle-ci.

2 Utilisation des fonctions lex / yacc

• On reprend les pointeurs de fichiers yyin et yyout générés par yacc pour rediriger l'entrée sur le (ou les) fichier(s)indiqué(s) en argument de la ligne de commande. Le yyin est mis à jour en prenant un à un les différents arguments de la ligne de commande. Pour le parseur back-end le pointeur yyout pointe vers un fichier créé après la mise à jour de yyin qui a le même nom que yyin mais avec "_3.c" à la fin à la place de ".c" pour pouvoir le différencier de yyin.

• On utilise l'option yylineno proposée par flex afin de pouvoir afficher le numéro de la ligne en cas d'erreur lors de l'analyse syntaxique et lexicale. On se sert également de ça pour déclarer en début de bloc les noms de variables temporaires induites par la traduction en code 3 adresses (précision dans la partie Structure du code ci-dessous).

• Utilisation de la fonction yacc yyrestart(file) lorsqu'on change de fichier d'entrée afin de passer au suivant pour que yacc réinitialise l'état du parser (pour ne pas rester bloqué sur une erreur syntaxique précédente par exemple). Lors d'un changement de fichier on réinitialise également toutes nos variables pour éviter les bugs et pour reprendre les compteurs de 0 pour notre fichier de sortie.

3 Modifications apportées au lexique de Lex

Front-end :

• Détections de ++ - - (placé en préfixe ou post-fixe) et en supposant leur sens identique c.a.d (x++=++x=x+1) pour simplifier leur traitement. Leur traduction est aussi gérée pour les transformer en code trois adresses équivalentes.

• Détections des opérateurs de “shift” . Leur traduction est aussi gérée pour les transformer en code trois adresses équivalentes.

• Détection des commentaires grâce à une expression régulière, le contenu des commentaires est ignoré et on peut détecter les commentaires mal fermés pour lancer une erreur.

4 Modifications apportées à la grammaire Yacc

Front-end :

• Marqueurs ACT1 à ACT6 qui nous permettent ‘’d’intercaler " des actions sémantiques dans la grammaire à des endroits précis.

5 Tests qui passent/ne passent pas

Le parseur frontend s’exécute correctement sur tous les fichiers de tests. Les tests syntaxiques de notre parseur backend passent sur tous les fichiers générés, nous supposons donc que notre compilateur génère un code correct syntaxiquement. Pour la vérification sémantique manuelle, tous nos fichiers nous semblent valides à l’exception du fichier listes.c où nous avons quelques problèmes avec les structures et les appelons de fonction. Notre compilateur ne fait pas de vérification sur les types donc il se peut que certaines portions de code aient des erreurs sémantiques à cause de cela.

6 Structure du code

Toutes les variables sont déclarées dans le fichier lex et sont récupérées (si besoin) dans le yacc avec extern.

Les variables int CompteurGoTo et CompteurFor permettent de mémoriser le nombre de conditions ou de boucles dans le programme pour avoir un identifiant unique à chaque fois (pour chaque Goto).

On initialise de nombreuses variables telles qu'inFor, instruct ou encore insizeof pour savoir dans quel type de bloc on se trouve. Cela permet d'ajuster des actions très spécifiques qui seraient différentes selon le type de bloc dans lequel on est. De même nous avons créé quelques variables dans le but de retenir le nombre de crochets ouvrants par exemple afin d'avoir un affichage correct en sortie.

N'ayant pas créé une table des symboles spécifiques pour les structures, pour enregistrer l'offset de chacun des champs nous avons créé une variable interne au compilateur pour l'enregistrer. Son nom est de la forme : _struct_nomstruct_nomchamp et sa valeur est l'offset correspondant. Commençant par un _ elle ne peut normalement pas entrer en conflit avec les variables de l'utilisateur.

Utilisation d'un attribut hérité dans le marqueur ACT4 $<symbol Value>-1 qui nous permet de récupérer le nom de la structure que nous sommes en train de créer et l'enregistre dans actstructdef. Nous réutilisons cette variable entre autre pour créer les variables internes précédemment expliquées.

La traduction en code 3 adresses impose de définir de nouvelles variables dans le fichier de sortie. Or, on ne peut pas déclarer la variable ailleurs qu'en début de bloc. Comme notre analyse syntaxique fonctionne en une passe nous avons donc pris la décision d'utiliser l'option yylineno pour retenir le numéro de ligne du début de bloc associé à la déclaration de la variable nouvellement créée et d'insérer la déclaration de la variable en début de bloc. C'est le rôle de la fonction inserttext qui permet d'insérer un texte passé en paramètre à la ligne indiquée elle aussi en paramètre. Pour cela utilise plusieurs techniques comme la création d'un fichier temporaire pour insérer le texte à l'endroit voulu et des renommages de fichier. Cela est particulièrement utile sur un fichier texte comme cond.c qui nécessite la déclaration de nombreuses nouvelles variables sans que l'on sache combien d'avance.

7 Répartition du travail

Nous nous sommes organisé avec Git Hub, ce qui a été particulièrement pratique au vu de la situation actuelle. Nous n'avons pas travaillé ''ensemble " beaucoup dans le sens que nous nous sommes réparti grossièrement le travail pour éviter de travailler sur la même chose puis au fur et à mesure de l'avancée du travail nous avons précisé cette répartition. Au final cela nous a permis de répartir le travail de façon plutôt équitable et d'avancer plus vite et plus efficacement.

8 Conclusions

Elie Antoine :
J'ai eu beaucoup de mal à me lancer dans le projet, principalement car j'avais une assez mauvaise compréhension de Lex Yacc, mais dès que la ''base" c'est à dire le parseur à été fait et que nous avons dû mettre en place les routines sémantique dans le yacc cela à été plus ''facile" et intéressant. La plus grosse difficulté de cette partie à été de bien comprendre comment yacc gérait la grammaire, et le C qui est un langage assez strict et que je n'avais pas pratiqué depuis la L2. Au final ce projet m'a permis de mieux comprendre le fonctionnement interne d'un compilateur qui est une notion fondamentale en informatique et m'a obligé à réutiliser du C ce qui me sera très certainement utile en Master. Tout cela en plus de l'option paradigme m'a ainsi permis d'avoir une vision plus large de l'informatique.

Florian Spire :
Le démarrage du projet m'a paru compliqué et il m'a fallu un certain temps avant de vraiment m'approprier le sujet. Le plus compliqué au départ a été de passer de la théorie enseignée en cours au code lui-même et de bien intégrer comment les fonctions lex et yacc interagissent entre elles et fonctionnent. Une fois cette étape franchie nous avons réussi à avancer le projet beaucoup plus facilement et même si cela prenait beaucoup de temps, le fait de réussir à avancer permettait de rester motivé. Finalement, même si le projet m'a paru globalement compliqué j'ai trouvé intéressant à comprendre plus précisément comment fonctionnait vraiment un compilateur qui est un outil que l'on utilise constamment en tant qu'informaticien sans vraiment s'intéresser réellement à son fonctionnement interne et c'est une bonne chose de pouvoir découvrir cela en tant qu'étudiant car ce n'est pas forcément une occasion que l'on aura plus tard.