Projet de Compilation – L3 Informatique UCA

Fait par J. Maxime, K. Romain

strucitc

Rendu le 20/05/2020 à 12h

Pr. Sid Touati

Sommaire

1 : Résumé

Ce projet nous a mis en difficulté du au retard que nous avons pris pour le commencer et à la gestion de projet qui n'était pas efficace. Le projet final ne prend pas en compte la traduction de plusieurs éléments de strucitc-frontend vers strucitc-backend. Lex et Yacc ont été les outils faisant office respectivement de Lexer et Parser. L'analyse sémantique se voit ne pas aussi complète que nous le voulions. La gestion du projet a été effectué avec github.com (outil GIT). Nous estimons que si nous avions commencé au moins 2 semaines avant notre départ, le projet aurait pu être rendu convenablement.

2 : Front-end : Fonctionnement Interne

2.1 Analyse lexicale

L'analyse lexicale permet de construire les mots qui seront ensuite utilisés dans l'analyse syntaxique.

Notre analyseur lexical n'accepte que les caractères de la table ASCII, en excluant les caractères spéciaux, les représentations de nombres qui ne sont pas en base 10, les chaines de caractères, les caractères, les entiers à base exponentielle, ainsi que les opérations d'assignements différents de l'affectation primaire (« = »). Elle construit les tokens en suivant les indications données dans le sujet.

Elle permet aussi de compter les lignes et colonnes indiquant la position du mot dans le fichier d'entrée, pour pouvoir personnaliser les messages d'erreurs dans l'analyseur syntaxique.

Le fichier lex (ANSI-C.l) diffère grandement du fichier lex donné pour que notre analyse lexicale corresponde au sujet, en modifiant certaines expressions régulières, et en ajoutant et supprimant d'autres.

2.2 Analyse Syntaxique et Arbre Syntaxique Abstrait

L'analyse syntaxique a été un passage relativement fastidieux du projet. En effet, la grammaire a elle aussi grandement évolué. Elle correspond aux attentes du sujet, et permet également de bloquer certaines expressions, comme les doubles pointeurs.

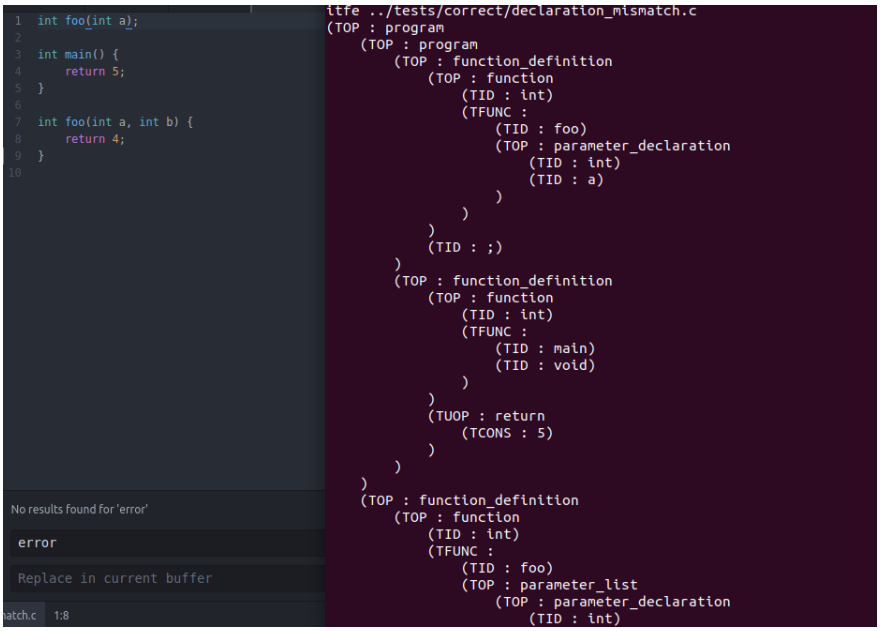
Elle permet aussi de laisser passer certaines expressions plus complexes, comme les déclarations multiples :

```
int a, b = 0, c;
```

Les prototypes, les déclarations dans un for

```
for (int i = 0 ;;);
sizeof(void *);
```

Pendant l'analyse ascendante dans yacc (LALR), un arbre syntaxique est construit, qui permettra par la suite de faire une analyse sémantique. Voici un exemple d'analyse sémantique :



L'arbre syntaxique est représenté par une structure node_t (= node type = « type nœud »). Nous avons décidé de faire une union dans l'arbre pour dissocier différents cas, et pour pouvoir ensuite mieux les traiter, et chaque cas est différenciable par l'attribut type_t type, qui est une énumération de possibilités. Par exemple, un node_t peut être une feuille (cas d'un identificateur ou d'une constante), une opération à une ou deux opérande(s), une fonction etc. Chaque type est lui-même une structure.

Les fonctions utilitaires utilisées pour construire l'arbre sont contenues dans tree.c.

L'arbre syntaxique est ensuite parcouru pour créer la table des symboles. Toutes les fonctions liées à l'analyse syntaxique sont fournies dans symbol_table.c. Nous utilisons deux tables des symboles : une pour les variables globales et une qui détient toutes les déclarations.

2.3 Analyse Sémantique

L'analyse sémantique est également une partie lourdement fastidieuse. Vous pourrez trouver le code de l'analyse sémantique dans le fichier semantical_check.c

L'analyse sémantique permet de trouver :

- La redéfinition d'une fonction.

- Redéfinition d'argument de fonction.

- Un appel incorrect à une fonction (nombre d'arguments / types d'arguments)

- Définition de variables de type void et utilisation de strings.

- Assignation de constantes du type : 3 = (…)

- Utilisation d'une variable non déclarée.

- Mauvaise utilisation du retour de fonction du type : void foo() ; int a = foo() ; De même si les types dans une assignation ou une opération arithmétique diffèrent.

2.4 Génération de code intermédiaire

2.4.1 Correspondance des fichiers sources

Fichier source (*.c)	Description
crpdct.c	Table de correspondance entre le nom front-end et back-end des variables
declarations.c	Fonction pour déclarer une variable/fonction
forstmt.c	Génération du code pour un statement FOR
fundef.c	Génération du code pour une fonction
ifstmt.c	Génération du code pour les if / else
stack.c	Gestion d'une pile et de ses composantes
statements.c	Fonctions pour gérer tous les statements et la génération du code 3 adresses
whilestmt.c	Génération du code pour un WHILE
structstmt.c (non implémenté)	Gestion de la table de correspondance sur les structures

2.4.2 Explications de l'ICG (Intermediate Code Generator)

La génération de code intermédiaire s'est avérée très compliqué du fait de la mauvaise approche du problème. En effet, l'approche utilisée a été « Comment je veux représenter mes données » et non « Comment je veux manipuler mes données ».
Il a été décidé d'utiliser des piles (stack_t) pour gérer une pile d'instruction à traduire. Il été prévu d'avoir une pile pour les statements et une autre pour les déclarations, et que chaque pile de statements contienne une pile pour les déclarations et pour les statements, de ce fait nous pouvions théoriquement gérer les variables locales.

De ce fait, l'ICG marche très bien dans un projet local, sur une pile maitrisée. Mais lors de l'implémentation dans Yacc, la grammaire a dû être pas mal modifiée pour que cela corresponde à peu près pour une utilisation de nos structures et fonctions. Mais lors des tests, il s'est avéré que notre ICG ne gérait pas tous les cas et provoquait parfois des Segmentation Fault (SIGSEGV / Corruption de mémoire / Saut sur une adresse invalide). Au final, beaucoup de choses ne sont pas implémentées dans l'ICG, notamment les structures, pour lesquelles on a les fonctions mais on ne les a pas implémentées par souci de temps (il était prévu une structure sn_t (Structure N Type) pour définir une structure avec juste son nom front-end ainsi que l'élément suivant, et une structure sstmtm_t (Structure Statement Manager Type) qui permettrait de garder en mémoire toutes les structures déclarées ; un dossier indev/ est prévu pour les fichiers des structures en gage de bonne foi).

Par souci de temps, certaines expressions auxquelles notre traducteur n'est pas préparé ne sont pas implémentées, ainsi que les déclarations locales puisque l'ICG a mal été pensé, les variables déclarées dans le langage strucit-frontend sont déclarées globales.

Il y a aussi des bogues, notamment sur la traduction d'expressions qui peut mener à des résultats absurdes suite à une mauvaise gestion de la pile sur des expressions non anticipées, ainsi qu'un bogue troublant dont nous ne connaissons pas la solution, qui nous affiche une « syntax error » sur la déclaration d'une variable globale, qui étrangement disparait si l'on supprime les définitions/déclarations de fonctions dans la règle external_declarations.

Malgré tout, notre ICG implémente une base fonctionnelle sur des programmes basiques. Les fonctions sont nommées par leur nom respectifs, les arguments par « aX », les variables par « vX ». Il était prévu que les structures soient déclarées par « sX ». Les statements d'itérations, de sauts et de sélections fonctionnent. Par ailleurs, nous n'interprétons pas le contenu d'un sizeof, puisque sizeof selon la norme C11 indique que le contenu de sizeof n'est pas interprété, par conséquent nous renvoyons uniquement une constante, nous avons décidé de simuler une architecture 32 bits, sont nous renvoyons toujours 4 (un int est ici de 4 octets, et une adresse aussi).

Les pointeurs sur fonctions sont déclarés sous la forme d'une variable de type void *, c'est ce que laisse penser le code exemple donné en strucit-backend.

3 : Back-end : Fonctionnement Interne

3.1 Analyseur Lexical

L'analyseur lexical du back-end est bien plus petit que celui du front-end dû à la contrainte d'instructions possibles. Comme dans le front-end, il est impossible de rentrer des mots non conventionnés par le langage.

3.2 Analyseur Syntaxique

L'analyseur syntaxique reste assez simple, puisqu'il y a juste le rajout de goto et de label ainsi que la suppression de beaucoup de mots clés, le yacc génère seul l'erreur et ce qu'il attendait, avec ce qu'il a reçu.

4 : Finalité du projet

4.1 Gestion de projet

Pour la gestion de projet, nous avons mis en place un github pour faciliter le travail.
Chaque partie du projet était composé d'un dossier, contenant un Makefile ainsi qu'un script Bash pour automatiser la compilation et les tests, ce qui nous a fait gagner un temps considérable.

Nous nous sommes ensuite réparti les tâches de la sorte :

- Maxime s'est occupé du lex, de la grammaire front et backend, ainsi que de la génération de code et des scripts (bash et Makefile).

- Romain s'est occupé de la construction de l'arbre, la table des symboles et de l'analyse sémantique.

Vous trouverez le lien du github ici : https://github.com/theoricien/strucitc

4.2 Petit guide d'utilisation du mini compilateur

Une fois dans le dossier racine du mini compilateur, si vous n'avez pas les logiciels prérequis d'installés sur votre machine (flex et bison), il vous suffit de taper la commande suivante :

make install

Si vous voulez compiler le projet, il vous suffit d'exécuter la commande suivante :

make

Un message en français devrait apparaitre sur l'organisation des fichiers si la commande réussie. Enfin, si vous désirer re-compiler le projet, exécutez ces commandes :

make clean; make

Pour exécuter le mini-compilateur (aka traducteur), vous avez le fichier « strucitc » à la racine. Il s'agit d'un mini script bash permettant de simplifier le lancement du logiciel. Vous avez le même principe avec le logiciel « check_backend » qui s'occupe de la vérification back-end.

4.3 Perspectives d'évolution/ amélioration

Dans un premier temps, l'analyse sémantique pourrait être complétée avec de nombreux cas, notamment la déclaration multiple dans un même bloc. Les fonctions qui pourraient gérer ces cas sont écrites, mais ne sont pas stable.

```
int b;

int foo(){
  int b;
}

int main() {
  int c;
    {
        int c;
        int a;
        int a;
    }
}
```

Prenons par exemple le cas ci-contre :

- Les déclarations de 'a' ne sont pas correctes et attrapées par la fonction.

- Les déclarations de 'b' sont correctes et pas attrapées par la fonction.

- Les déclarations de 'c' sont correctes (autorisé avec gcc) mais attrapées par la fonction.

Dans certains cas, ces fonctions pourraient aussi boguer quand le fichier contient des prototypes car la table des symboles peut mal se construire (suivant l'arbre).

Nous avons donc décidé de ne pas inclure ces fonctions dans le rendu, même si elles ne sont pas loin de fonctionner.

Le lex et la grammaire front-end sont correctes et ne nécessitent pas, à notre sens, d'amélioration.

Les structures ne sont pas gérées

Pour nous assurer du bon fonctionnement de notre compilateur, nous l'avons testé sur une centaine de fichier tests en entrée.

Dans un deuxième temps, l'ICG pourrait être complété par une remise en en forme globale de l'objectif, une totale reprise de 0.

5 : Conclusion

Dans l'ensemble ce projet a été relativement épuisant. Nous avons été surpris par la quantité de travail et nous n'avons pas pu finaliser le projet comme nous l'aurions voulu. De plus, nous avons perdu du temps avec l'arbre syntaxique, dans la mesure nous avions pris une mauvaise direction quant à la construction de l'arbre.

Pour ce qui est de l'analyse sémantique, les possibilités de dérivation de l'arbre sont tellement importantes qu'il y a une énorme quantité de cas à gérer.

Cependant, nous sommes tout de même fiers de ce que nous avons réalisé dans la mesure où nous avons fourni une quantité de travail considérable.

De plus, pour l'un d'entre nous, il a fallu réapprendre le C, un langage difficile à contrôler entièrement.

Notre projet n'est pas parfait, mais nous avons réussi à parvenir à une finalité convenable.

Pour conclure, nous pouvons dire que ce projet était très intéressant, mais la charge de travail est à notre sens trop grande pour un binôme.