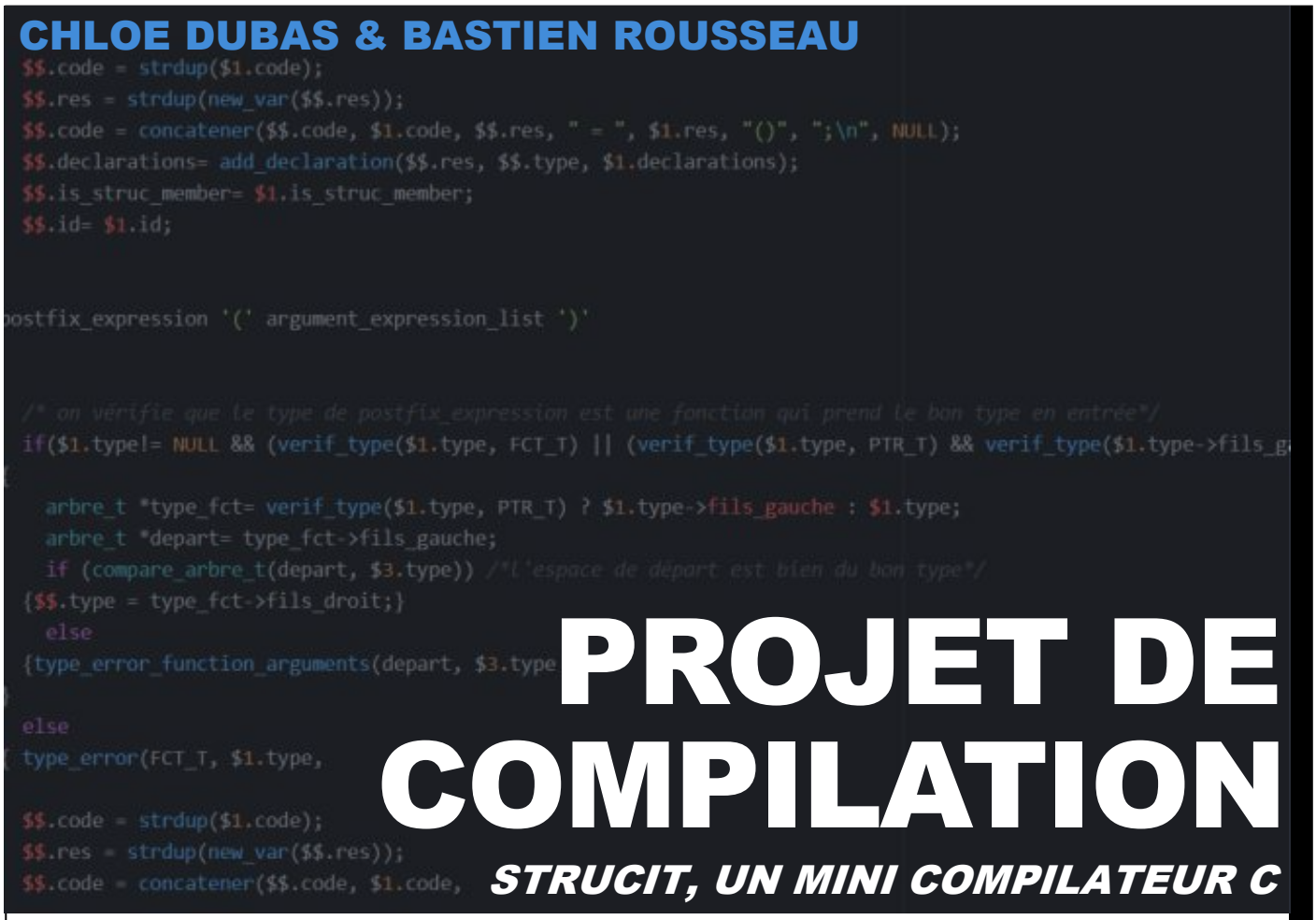
CHLOE DUBAS & BASTIEN ROUSSEAU



PROJET DE COMPILATION

STRUCIT, UN MINI COMPILATEUR C

Ce rapport remplace la soutenance initialement prévue, mais qui n'a pu avoir lieu en raison des circonstances dues au COVID-19.

Strucit est un compilateur qui traduit un dialecte du C vers un autre dialecte du C, plus proche de la machine, en code 3 adresses. Ce compilateur a été réalisé grâce aux outils Lex et Yacc dans le cadre du projet de l'UE Compilation de la licence 3 Informatique.

À travers ce rapport, nous allons présenter notre démarche, les diverses fonctionnalités et spécificités du compilateur ainsi que ses limites.

FRONTEND

Nous allons détailler notre approche de l'analyse lexicale, syntaxique et sémantique de la partie frontend.

1. MODIFICATIONS APPORTÉES AUX FICHIERS FOURNIS

Tout d'abord, des fichiers de base étaient fournis avec le sujet, notamment les grammaires des parties frontend et backend (dans un descripteur Yacc), ainsi que les spécifications Lex du langage C (ANSI).

Dans un premier temps, nous avons enlevé les éléments de ces fichiers non concernés par les dialectes, et ajouté la reconnaissance des commentaires pour la partie Lex.

Nous avons aussi modifié quelques éléments de la grammaire :

- Les entrées et sorties de nouveau bloc se font avec des accolades. Nous avons introduit de nouvelles productions `entree` et `sortie` dans le but d'ajouter une action sémantique, afin de gérer la portée des variables ;
- Initialement, le `sizeof` ne permettait pas des expressions comme `sizeof(type)` (par exemple : `sizeof(int)`). Ainsi, nous avons ajouté une production qui accepte ces expressions ;
- Dans le dialecte frontend, les structures ne peuvent être manipulées qu'avec des pointeurs. Nous avons donc supprimé la production qui acceptait des expressions telles que : `name.champs` ;
- Nous avons géré l'ambiguïté du `IF…ELSE` grâce à la règle de priorité Yacc %prec ;
- Comme indiqué sur le forum, nous avons décidé d'accepter les pointeurs sur type de base pour les retours de fonctions ;
- Nous avons également décidé d'accepter les commentaires sur une ligne avec la syntaxe : `//commentaire`.

Par ailleurs, nous avons modifié quelques fichiers de tests afin qu'ils soient corrects avec le langage d'entrée :

- cond.c : ajout de conditions pour tester les && et les `||` ;
- expr.c : changement des opérateurs >> et << (non accepté par le frontend) par des `*` ;
- pointeur.c : suppression de l'opérateur ++ (non accepté par le frontend) et ajout de la déclaration `extern int *malloc( int size );`.

2. STRUCTURES DE DONNÉES EMPLOYÉES

L'implémentation de notre compilateur a nécessité diverses structures de données.

L'un des éléments essentiels au sein d'un compilateur est la table des symboles (permettant de garder en mémoire les identifiants utilisés dans le fichier source). Pour cela, nous avons implémenté la table des symboles au moyen d'une table de hachage, et défini la portée grâce à une pile. La table sur le sommet de la pile correspondant au bloc le plus imbriqué.

Les symboles sont une représentation des variables du programme source, et sont constitués d'un nom, et d'un type. Dans la partie frontend, les types acceptés sont : int, void, pointeur, fonction, et structure. De plus, afin de différencier les variables initialisées dans le programme et les paramètres de définition de fonctions, nous avons ajouté aux symboles un champ `is_arg`.

Un autre élément important pour un compilateur est son système de typage, qui associe une expression de type à chaque symbole et permet de détecter des erreurs sémantiques. Afin de pouvoir comparer les expressions de type, nous les avons implémentées grâce à des arbres binaires dont la racine constitue le type principal. De plus, la comparaison de structures nécessite également la comparaison du nom des champs.

Enfin, dans le but de gérer les informations sémantiques, nous avons introduit une structure de données contenant les attributs sémantiques suivants :

- `code` : contient le code généré ;
- `declarations` : contient les déclarations des nouvelles variables temporaires ;
- `res` : contient le nom de la variable temporaire dans laquelle est stockée le résultat du code précédent ;
- `type` : expression de type associé à une expression ;
- `id` : utilisé dans les déclarations pour garder en mémoire le symbole déclaré et modifier son type une fois qu'il est calculé ;
- `is_struc_member` : permet de savoir si l'expression est l'accès à un champ d'une structure (`name→champ`). Utile pour le déréférencement des pointeurs sur structure ;
- `is_ptr_fct` : utilisé dans les déclarations pour savoir si le symbole déclaré est un pointeur sur fonction.

BACKEND

Nous allons maintenant détailler notre approche de la génération du code backend et son analyse.

Avant tout, il convient de rappeler que le sujet exigeait uniquement l'analyse lexicale et syntaxique du backend, l'analyse sémantique étant réalisée à la main. Nous avons donc adapté les spécifications Lex et Yacc afin de répondre aux critères imposés.

Notre travail a principalement reposé sur la traduction du code d'origine en code 3 adresses dans les actions sémantiques de la partie frontend.

Nous avons dû prendre en compte quelques spécificités du dialecte généré :

- Les structures n'existent pas dans le backend, il a donc été nécessaire de traduire les accès aux champs des structures par un calcul d'adresse (l'alignement n'est pas pris en compte) ;
- L'appel à une fonction `sizeof` est directement calculé dans les routines sémantiques. Cette fonction réservée n'existe donc plus dans le backend et est remplacée par sa valeur ;
- Les seuls pointeurs existants dans le backend sont de type `void*`. Tous les types de pointeurs sont donc ainsi traduits ;
- Les variables temporaires (resp. les labels), utilisés dans le code 3 adresses, sont de la forme `_tempX` (resp. `_labelX`), où X est un nombre associé à un compteur. Si ce nom de variable est déjà utilisé dans le code source (ce qui normalement n'est pas accepté), alors le compteur est incrémenté ;
- Puisqu'il s'agit de code 3 adresses, les expressions booléennes avec AND et OR, les instructions conditionnelles (IF et IF...ELSE), et les boucles (FOR, WHILE), sont donc traduites avec des instructions de branchements (IF...GOTO…, GOTO).

Nous avons décidé de réaliser le parseur backend avant de démarrer la génération du code afin de vérifier que notre code produit était lexicalement et syntaxiquement correct.

LIMITES ET DIFFICULTÉS

Les tests fournis avec le sujet sont compilés avec succès par strucit. Afin de compléter les tests et de chercher les limites de notre compilateur, nous avons élaboré des tests supplémentaires (non inclus dans l'archive rendue).

Cependant, même si le compilateur fonctionne sur les tests fournis, nous avons rencontré des difficultés sur certains points. Afin de résoudre ces problèmes, nous avons adopté des solutions dont nous ne sommes pas pleinement satisfaits. Elles ont tout de même été implémentés, par manque de temps et à défaut d'en trouver d'autres.

En effet :

- Nous avons introduit une variable globale `type_retour`. Ainsi, nous pouvons vérifier que le type de l'instruction `return` et celui attendu par la définition de la fonction sont bien identiques ;
- Enfin, nous avons rencontré des difficultés avec la manipulation des attributs hérités (typage pour la déclaration des variables), et plus spécifiquement pour la gestion des pointeurs sur fonctions. Pour pouvoir reconnaître ce type, nous avons procédé de la façon suivante : à la production `direct_declarator → (declarator)`, si le type de `declarator` est un pointeur, alors on présuppose que `direct_declarator` est une fonction. Cette méthode provoque cependant des problèmes avec `int (*var) ;`

- Il y a des erreurs avec les structures définies récursivement. En effet, les instructions telles que `liste→suivant→suivant` provoquent une erreur sémantique ;
- L'espace de nom des variables et l'espace de nom des fonctions ne sont pas séparés. Cependant, le corps d'une fonction se trouve dans une portée différente de sa définition. Ainsi, il devient possible de nommer une variable avec le nom d'une fonction (ce qui risque de provoquer une erreur sémantique).

CONCLUSION

Ce projet réalisé en binôme dans le cadre de notre licence d'informatique constitue un projet important dans notre formation.

Nous l'avons démarré la semaine où nous avons reçu le sujet. Cela nous a permis de travailler régulièrement jusqu'à la date du rendu, et de pouvoir rendre un projet fini et complet sans être pris par le temps. Chaque semaine, nous nous sommes organisés des séances de travail, pendant lesquelles nous avons réfléchis et avancé ensemble.

Notre principale difficulté sur ce projet a finalement concerné les spécificités techniques du langage C (langage non pratiqué depuis environ 1 an), ainsi que celles de Lex/Yacc.

Finalement, ce projet fût relativement difficile, mais très formateur. Nous avons pu mettre en application nos connaissances théoriques vues en cours, et cela nous a permis de mieux comprendre le fonctionnement d'un compilateur.

Nous avons trouvé intéressant et plaisant de réaliser un compilateur au moins une fois dans notre vie d'informaticien.

MERCI !

Notre binôme remercie M. Sid Touati et Mme. Cinzia Di Giusto, nos enseignants pour l'UE Compilation, pour les réponses qu'ils ont apporté à nos questions tout au long de ce projet et tout au long du semestre.