Rapport

Compilation

FABREGOULE ANTOINE
ETIENNE CYRIL

Introduction

Le projet de compilation a pour objectif la conception d'un compilateur pour un langage de programmation simplifié, appelé tpc, qui est un sous-ensemble du langage C. L'objectif principal est de créer un compilateur capable de détecter les erreurs lexicales, syntaxiques et sémantiques, tout en générant un code cible correct. Le programme source écrit en tpc sera donc traduit en un fichier assembleur qui pourra être assemblé et exécuté.

Organisation du rendue

- bin
- obj
- rep
- src
  - ├── codegen.c
  - ├── codegen.h
  - ├── lexer.l
  - ├── parser.y
  - ├── semantics.c
  - ├── semantics.h
  - ├── symbol_table.c
  - ├── symbol_table.h
  - ├── tree.c
  - └── tree.h
- test
  - ├── good
  - ├── sem-err
  - ├── syn-err
  - └── warn
- _anonymous.asm
- makefile
- run_tests.sh

Analyse Sémantique

Les fichiers *semantics.c* et *semantics.h* sont responsables de l'analyse sémantique du programme source. Cette étape vérifie la cohérence des déclarations et des types dans le programme, en utilisant une **table des symboles** et **l'AST** pour stocker des informations sur les variables et les fonctions. Le module détecte des erreurs sémantiques telles que des redéclarations de variables, des variables non définies, et des incohérences de types.

Table des Symboles

Les fichiers symbol_table.c et symbol_table.h contiennent la gestion de la table des symboles.
La table des symboles stocke les informations sur les variables, les fonctions et leurs types respectifs.
Elle est utilisée pour effectuer des vérifications sémantiques pendant la compilation, telles que la détection de variables non définies ou de conflits de types.

Nous avons fait le choix de structurer cette table en séparant les variables globales des fonctions, en créant deux sous-tables distinctes.
Concernant la sous-table des fonctions, chaque fonction est représentée par un identifiant, ainsi que deux listes : une pour les variables locales et une pour les arguments.

Toutes ces tables sont des listes chaînées, à l'exception de symbol_table, qui est une structure regroupant les listes chaînées des variables globales et des fonctions.

Génération de Code

La génération de code consiste à transformer l'arbre syntaxique annoté et la table des symboles en un fichier assembleur conforme à la sémantique du programme source.
Cette étape a été mise en œuvre dans les fichiers codegen.c et codegen.h, en suivant les règles du langage TPC.

Elle s'appuie sur des fonctions spécifiques à chaque type d'instruction (assignation, appel de fonction, conditions, boucles, etc.) afin de produire un code correct.

Par défaut, nous avons implémenté les fonctions getint(), putint(), putchar() et getchar().

L'utilisateur ne peut pas modifier ces fonctions, car elles sont prédéfinies. Chaque fonction parcours des endroits précis de l'AST pour générer le code voulu.

Difficultés rencontrées

Deux difficultés principales ont été rencontrées : la gestion des static et le typage dans certaines expressions arithmétiques. Ces problèmes ont été corrigés, et la majorité des tests fonctionnent correctement. Pourtant, le score des tests automatiques reste bas.

Nous avons aussi dû modifier le parseur pour forcer la création de nœuds dans certains cas, par exemple avec | '!' F { $$ = makeNode(NOT_ef); }, qui ne construisait pas les enfants attendus.

Amélioration effectuées

L'une des améliorations majeures apportées au compilateur concerne la gestion fine des instructions return. Nous avons mis en place une vérification stricte du type de retour des fonctions, en comparant le type attendu avec le type réellement retourné. Lorsque les types sont compatibles mais différents (par exemple, retour d'un int dans une fonction censée retourner un char), le compilateur n'interrompt pas l'exécution mais émet un avertissement explicite.

Pour executer notre programme, il faut faire un make puis ./bin/tpcc < *test.tpc,* les options -t pour afficher l'arbre et -s pour afficher la table des symboles sont disponibles.

Tests

Nous avons organisé les tests en quatre catégories distinctes, afin de valider chaque aspect du compilateur. Voici la description des différents jeux d'essais :

1. **Programmes tpc corrects sans avertissements** : Ce jeu d'essais contient des programmes tpc valides, sans erreurs ni avertissements.

2. **Programmes avec des erreurs lexicales ou syntaxiques** : Ce jeu d'essais contient des programmes avec des erreurs de syntaxe ou de lexique. L'objectif est de vérifier que le compilateur détecte correctement ces erreurs et renvoie un code d'erreur approprié.
3. **Programmes avec des erreurs sémantiques** : Ce jeu d'essais contient des programmes avec des erreurs sémantiques, comme des variables non déclarées ou des conflits de types. L'objectif est de s'assurer que le compilateur détecte ces erreurs et affiche un message d'erreur pertinent.
4. **Programmes tpc corrects avec des avertissements** : Ce jeu d'essais contient des programmes valides mais générant des avertissements (par exemple, affectation d'un `int` à un `char`). L'objectif est de vérifier que le compilateur émet correctement des avertissements tout en générant le code cible.

Script de Déploiement des Tests

Un script a été créé pour automatiser l'exécution des tests. Ce script exécute chaque programme de test, vérifie le code de sortie de la compilation et génère un rapport unique indiquant les résultats pour chaque test. Le rapport inclut également quatre scores globaux, représentant la performance du compilateur dans les différents types de tests.

Conclusion

Ce projet a été une expérience enrichissante, permettant de consolider nos compétences en programmation assembleur et en gestion de projets.

L'exécution des tests a permis de valider que le compilateur est capable de détecter correctement les erreurs lexicales, syntaxiques et sémantiques, et de produire un code cible fonctionnel. Le script de déploiement a automatisé le processus de tests, générant un rapport détaillé des résultats et des scores globaux, ce qui facilite l'évaluation des performances du compilateur.