Université Gustave Eiffel

License Informatique

Rapport

# Compilation


3ème Année

Juin 2021

# Auteurs


- Lorris CREANTOR
- Jimmy TEILLARD

# Contents

# 1 Introduction

TPC est un très petit sous-ensemble du langage C. Bien que peu pratique et peu utilisable, il existe en tant qu'exercice et projet très intéressant sur l'analyse syntaxique, les grammaires, et la compilation en nasm x86-64 (elf64).

La grammaire du langage est définie dans `./src/parser.y`.

Le langage n'a aucune structure de donnée à l'exception des `struct` (qui ne sont ni récursives, ni imbriquées), et n'a ni boucle `for`, ni pointeurs. Les deux seuls types primitifs sont `int` (entier signé sur 32bits) et `char` (caractère sur 8bits), et les seules instructions de contrôle sont `if` et `while`. Le reste de la syntaxe est similaire à celle du C.

# 2 Mode d'emploi

## 2.1 Pré-requis

- Être sur un système *nix (Linux, MacOS, etc)
- Avoir une version récente de `gcc` installée
- Avoir une version récente de `make` installée

## 2.2 Compiler le Compilateur

Simplement lancer `make` à la racine du projet. L'exécutable sera produit dans le dossier `./bin/`.

## 2.3 Utilisation

tpcc [OPTIONS] [fichier]

- -t: Affiche l'arbre syntaxique du programme donné
- -s: Affiche la table des symboles du programme donné
- -n: Empêche de créer un fichier asm cible (No output)
- -x: Produit un fichier exécutable (compatible avec -n)
- -h: Affiche le message d'aide (ce message)

tpcc peut aussi recevoir son entrée depuis l'entrée standard, en quel cas des fichiers `_anonymous.*` seront créés.

## 2.4 Tests automatiques

Un fichier de tests automatiques est fourni. Il effectue des tests sur tout les fichiers présents dans `./test/`. Soyez cependant sûrs de placer les bons types de tests dans leurs dossiers respectifs (valide, erreur syntaxique, erreur sémantique, et warning).

Pour lancer la suite de tests, lancez le script shell `./buildntest.sh`.

# 3 Difficultés

## 3.1 Structures

La principale difficulté rencontrée lors du projet a été l'implémentation des structures en nasm. L'ajout de ces dernières a requis de nombreuses modifications de notre traitement des variables et valeurs primitives afin d'uniformiser l'ensemble et pouvoir factoriser une partie du code et uniformiser la logique de notre traduction (entre autres lié à notre utilisation de tailles exactes pour représenter les données et non une taille `QWORD` générique, c'est-à-dire qu'un entier est un `DWORD`, et un caractère est un `BYTE`).

Les modifications nécessaires à l'ajout des structures ont été les suivantes :

- utilisation systématique de la pile pour les retours de fonction
- conservation de l'ordre des membres et des données lors d'une copie de ces derniers
- nécessité d'empiler les membres des structures du dernier au premier afin de conserver un sens de lecture logique
- utilisation de boucles pour les affectations de structures

## 3.2 Return et Control Flow

Une autre difficulté intéressante rencontrée à la fin du projet a été de résoudre le problème du "Reached end of control flow in non-void function", c'est-à-dire de pouvoir déterminer qu'une fonction non void est invalide car le programmeur laisse la possibilité d'atteindre la fin de celle-ci sans return (autrement dit : déterminer que chaque branche soit "return completed", et lancer une erreur si ce n'est pas le cas).

La solution à cela a d'abord été de déterminer ce que sont les branches intéressantes à vérifier.

En effet, il est inutile de tester récursivement un simple `if` (sans `else`) : qu'il soit return completed ou non ne change pas le fait que le scope dans lequel il est doit l'être aussi. De la même manière, un `while` n'est pas intéressant à tester. Il est aussi à noter que s'il existe un return dans le scope principal de la fonction (niveau 0), alors toute la fonction est return completed. Cette propriété est issue du fait que si un scope contient un return, alors il est return completed.

Ainsi, les seuls cas à tester récursivement sont les `if` qui ont aussi une branche `else`. Pour cela, on a assimilé le problème à celui du langage de Dyck (parenthèses correspondantes). Dans le cas du langage de Dyck, pour vérifier simplement qu'une suite de parenthèses soit cohérente, on peut utiliser un compteur qui s'incrémente à l'ouverture d'une parenthèse, et qui se décrémente à la fermeture d'une parenthèse, en vérifiant qu'a chaque entrée de scope, ce compteur soit toujours le même à sa sortie (impliquant qu'une suite valide a un compteur à 0 à la fin). Ici, les parenthèses ouvrantes sont les `if` et les `else`, et les parenthèses fermantes sont les return.

Si une branche `if` et sa sœur `else` sont toutes deux récursivement return completed, alors le scope est return completed.