Documentation Développeur

1 - Introduction

Le répertoire de ce projet est divisé en deux grandes parties, une partie *Compiler* qui contient l'ensemble du code source et une partie *Tests* qui contient la structure de test de ce projet. Nous admettrons par la suite que l'utilisation de l'outil *ifcc* est acquise (cf. documentation utilisateur) ainsi que l'installation, soit du conteneur Docker, soit des librairies Antlr4.

2 - Organisation du code (dossier */compiler*)

2.1 - Fonctionnement du compilateur

Détaillons d'abord le fonctionnement général du compilateur.

Le compilateur utilise la librairie Antlr4 afin d'analyser la grammaire des fichiers à compiler. Cela permet de générer les erreurs de syntaxe dans les fichiers. Si aucune erreur n'est retournée, Antlr génère un arbre lié à la grammaire *ifcc* contenue dans le fichier *ifcc.g4*.

Le compilateur parcourt ensuite cet arbre par le biais d'un Visiteur (*visitor.h* et *visitor.cpp*). Le parcours de cet arbre va créer, dans un premier temps, une représentation intermédiaire du code assembleur et de générer les erreurs sémantiques du code à compiler.

Ensuite, de cette représentation intermédiaire, le compilateur génère le code assembleur final pour l'architecture choisie. Dans notre cas, seule l'architecture x86 est supportée.

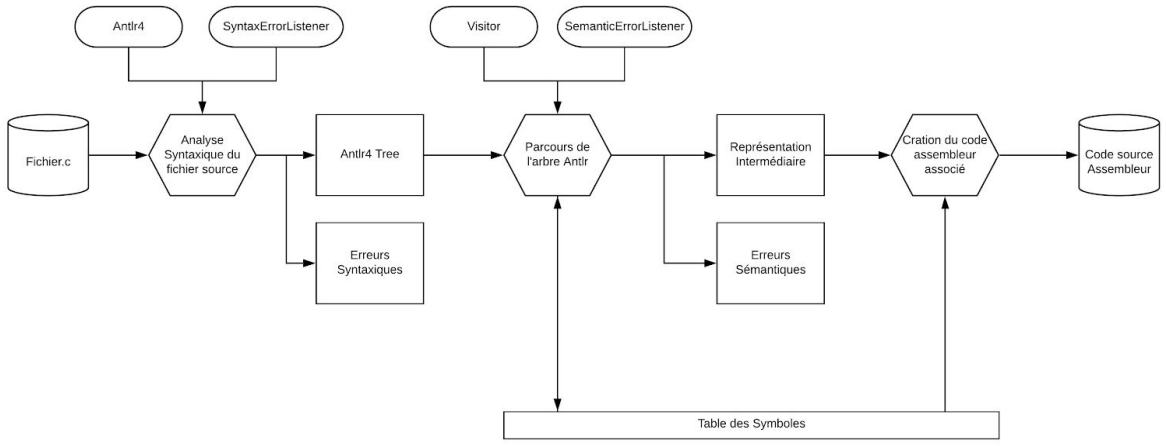

Diagramme d'état transition du compilateur

2.2 - Implémentation du compilateur

Le répertoire *compiler* est le répertoire contenant les fichiers constituant le compilateur *ifcc* en lui-même. Dans ce dossier se trouvent déjà 3 éléments indispensables à l'exécution du compilateur : *ifcc.g4*, *main.cpp* et la classe *Visitor*.

Liste des fichiers du dossier *compiler* :

- **ifcc.g4** : Fichier contenant la grammaire de notre langage (ici C). Cette grammaire est écrite dans le format de Antlr4 qui est l'analyseur que nous utilisons pour notre compilation.
- **main.cpp** : Point d'entrée du programme. Se charge d'appeler les différents analyseurs et parsers et la génération du code assembleur.
- **Visitor** : Classe fille de ifccBaseVisitor composée de surcharges des méthodes faisant la visite des règles de la grammaire. Parcours l'arbre Antlr4 et appelle les bonnes méthodes de visite pour générer la représentation intermédiaire.

Dans le dossier *compiler* se trouvent aussi les dossiers suivants :

.antlr & antlr-generated

Ces dossiers contiennent l'ensemble du code Antlr nécessaire et du code Antlr généré par notre grammaire. Ce code n'est pas à modifier. Seule une classe est utile à notre programme, *antlr-generated/ifccBaseVisitor.h*, qui correspond au template des méthodes de visite de l'arbre antlr à surcharger. La classe *Visitor* hérite de cette classe.

error

Ce dossier contient les classes de gestion d'erreur de notre compilateur. On a deux types d'erreurs à gérer, les erreurs syntaxiques qui sont liées à la grammaire du programme, puis les erreurs de sémantique, par exemple déclarer deux fois une même variable, qui sont des erreurs ne posant pas de problème de grammaire mais qui ne sont pas acceptables.

Liste des classes dans le dossier *error* :

- **Error** : Classe mère des objets Error. Contient le message d'erreur ainsi que des informations sur la position de l'erreur.
- **SyntaxError** : Classe fille de Error représentant une erreur syntaxique.
- **SyntaxErrorListener** : Classe fille de BaseErrorListener pour les erreurs syntaxiques. Se charge de la réception et l'affichage des éventuelles erreurs de syntaxe.
- **SemanticError** : Classe fille de Error représentant une erreur sémantique.
- **SemanticErrorListener** : Classe chargée de la réception et l'affichage des éventuelles erreurs sémantiques.

intermediate-representation

Ce dossier représente la structure de donnée de la représentation intermédiaire. La représentation intermédiaire consiste en un ensemble de CFG (Control Flow Graph), un pour chaque fonction. Chaque CFG gère et offre une interface vers des SymbolTables. Cela permet de gérer les portées (scopes) du code C. Chaque CFG contient des BasicBlocks. Un BasicBlock est un ensemble d'instructions (IRInstr). Ces instructions ne peuvent pas être des sauts (jmp, je, jne, ...). Les sauts pour le passage à un nouveau bloc ou pour les boucles se font lors du passage d'un BasicBlock à un autre. Les instructions (qui ne sont pas des sauts) sont, elles, représentées par des IRInstrs, contenant un type d'instruction et des paramètres. Chaque IRInstrs peut être transformée en différents codes assembleurs selon l'architecture (Rappel : seul x86 est supporté par notre compilateur).

Liste des classes dans le dossier *intermediate-representation* :

- **IRInstr** : classe représentant une instruction assembleur.
- **BasicBlock** : classe représentant un bloc de base. Correspond à une liste d'instructions sans sauts.
- **CFG** : classe représentant le code d'une fonction. Correspond à un graphe de BasicBlocks et gère des SymbolTables.

symbol-table

Ce dossier contient les classes formant la structure de donnée d'une table de symbole. Cette table des symbole a pour le principe de lister les différentes variables d'un CFG avec

leurs propriétes (type, adresse mémoire, définition). Une Symbole correspond à une portée (scope).

Liste des classes dans le dossier *symbol-table* :

- **SymbolTable** : classe représentant la table des symbole contenant les variables d'un BasicBlock
- **Type** : énumération des types disponibles (INT, CHAR, INT_ARRAY)
- **Variable** : Classe mère de tous les objets Variable. Contient un Type et un nom.
- **VariableGlobale** : Classe fille de Variable représentant une variable globale. Contient une adresse sous la forme d'une chaîne de caractères.
- **VariableLocale** : Classe fille de Variable représentant une variable locale. Contient une adresse entière.
- **Array** : Classe fille de VariableLocale représentant un tableau. Contient une taille et une méthode pour récupérer l'adresse d'un élément.

3 - Organisation des tests

Un environnement de tests se trouve dans le sous-répertoire tests et permet d'automatiser les tests fonctionnels.
Les tests sont placés dans le sous répertoire tests/tests, dans des dossiers qui sont nommés d'après les fonctionnalités testées, au format .c . Ce code source C est compilé avec gcc ainsi qu'avec notre compilateur. Les binaires sont exécutés et les résultats sont comparés au niveau du code de sortie (le return du main) ainsi que la sortie standard. Si les deux se comportent de la même manière, le test est validé.
Le script python permettant de faire cela s'appelle pld-test.py et deux scripts bash permettent de l'invoquer : test.sh pour utiliser docker, et test_if.sh pour une utilisation avec antlr sous Ubuntu. Une interface colorée a été implémentée pour mettre en valeur les tests qui passent et ne passent pas, ainsi que des statistiques sur les tests.
Les résultats sont détaillés dans le répertoire pld-test-output : asm-gcc(/pld) pour l'assembleur généré par gcc(/pld), exe-gcc(/pld) pour l'éxécutable, gcc(/pld)-compile pour le retour du compilateur, gcc(/pld)-compile pour les résultats d'éxecution gcc(/pld)-link pour le retour de l'édition des liens, input.c pour le code source.
Ce répertoire peut être effacé avec la commande : $sh test.sh clean