

# Documentation développeur

---

*ANDRIANARISOLO Elie*







*FARHAT Widad*

*SARR Seynabou*

*TONG An Jun*

---

# Sommaire

	<b>1. Introduction</b>	<b>3</b>
	<b>2. Structure du projet</b>	<b>3</b>
	<b>3. Organisation du répertoire compiler</b>	<b>4</b>
	3.1. Fonctionnement du compilateur ifcc	4
	3.2. Implémentation du compilateur ifcc	4
	<b>4. Organisation du répertoire test</b>	<b>6</b>



# 1. Introduction

Ce projet est organisé en deux parties principales :

- Compiler qui contient le code source complet.
- Tests qui abrite la structure de test du projet.

Nous supposons par la suite que l'utilisation de l'outil `ifcc` est bien maîtrisée (cf. documentation utilisateur) et que tous les prérequis, cités dans la documentation utilisateur, sont validés.



# 2. Structure du projet

Vous trouverez ci-dessous la structure des principaux éléments du projet :

```
pld-comp/  
├── compiler/  
│   ├── build/  
│   ├── generated/  
│   ├── src/  
│   │   └── IR/  
│   ├── config.mk  
│   ├── ifcc.g4  
│   └── Makefile  
├── tests/  
│   ├── testfiles/  
│   ├── ifcc-test-output/  
│   ├── ifcc-test.py  
│   └── ifcc-wrapper.sh  
└── documentation/
```



## 3. Organisation du répertoire compiler



### 3.1. Fonctionnement du compilateur ifcc

Commençons par explorer le fonctionnement global du compilateur. Le compilateur utilise la bibliothèque ANTLR4 pour analyser la grammaire des fichiers à compiler. Cela permet de détecter les erreurs de syntaxe dans les fichiers. Si aucune erreur n'est détectée, ANTLR4 génère un arbre syntaxique basé sur la grammaire ifcc contenue dans le fichier `ifcc.g4`.

Par la suite, le compilateur parcourt cet arbre en utilisant un visiteur (`CodeGenVisitor.h` et `CodeGenVisitor.cpp`). Ce parcours crée initialement une représentation intermédiaire du code assembleur et identifie les erreurs sémantiques dans le code à compiler.

Pour finir, à partir de cette représentation intermédiaire, le compilateur génère le code assembleur final pour l'architecture assembleur x86.

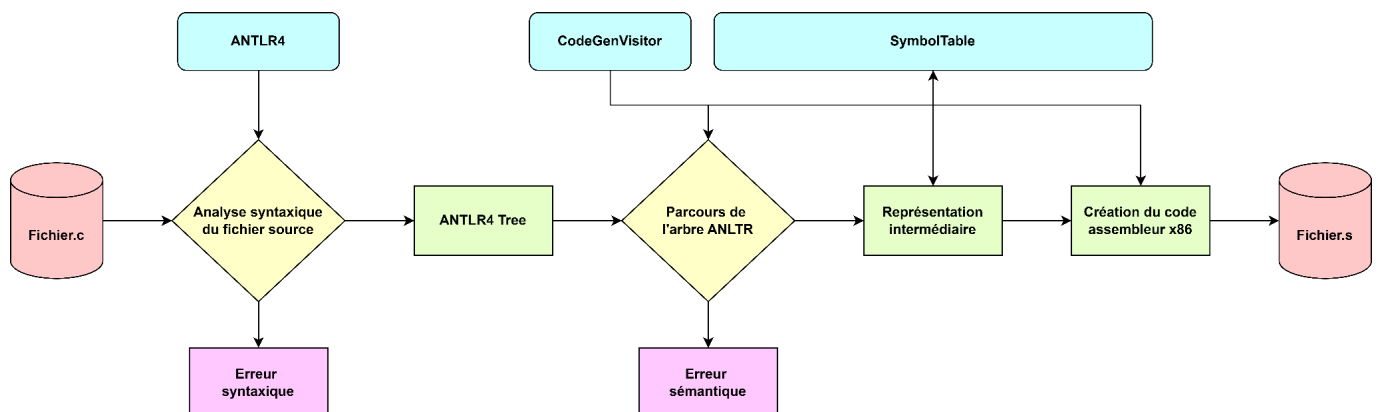


Figure 1. Diagramme d'état transition du compilateur



### 3.2. Implémentation du compilateur ifcc

Le répertoire compiler est le répertoire contenant les fichiers constituant le compilateur ifcc en lui-même. Vous y trouverez notre code source en C++, notre grammaire, ainsi que notre Makefile.

Les fichiers dans le répertoire compiler sont les suivants :

- `ifcc.g4` : La grammaire du compilateur ifcc, qui sera analysée par la bibliothèque ANTLR4.
- `Makefile` : Un fichier makefile pour compiler le code source. Il permet aussi d'exécuter d'autres options explicitées dans la documentation utilisateur.

- `config.mk` : Ce fichier est utilisé en tant qu'include dans le Makefile pour isoler du makefile la partie qui va différer d'un OS à l'autre.
- Le sous-répertoire `generated` contenant le code ANTLR4 généré par la grammaire `ifcc.g4`. Ce code n'est pas à modifier. Seule une classe est utile à notre programme, à savoir `generated/ifccBaseVisitor.h`, qui correspond au template des méthodes de visite de l'arbre ANTLR4 à surcharger. La classe `CodeGenVisitor` hérite de cette classe.
- Le sous-répertoire `build` contenant les fichiers générés lors de la compilation du projet.
- Le sous-répertoire `src` contenant le code source qui suit :
  - `main.cpp` : Ce fichier contient la fonction principale, qui lance les fonctions d'ANTLR4 pour analyser la syntaxe du programme d'entrée, puis nos visiteurs de syntaxe pour générer le code IR, et enfin écrit le code assembleur en utilisant l'IR.
  - `CodeGenVisitor.*` : Ce fichier contient tous nos visiteurs de syntaxe, un pour chaque expression définie dans notre grammaire.
  - `SymbolTable.*` : Ce fichier correspond à la classe de table des symboles, où toutes les variables et fonctions du programme d'entrée sont stockées.
  - `ErrorHandler.*` : Ce fichier correspond à une classe simple de journalisation des erreurs.
  - Un sous-répertoire `IR` représentant la structure de données de la représentation intermédiaire. Il contient le code source suivant :
    - `IRInstr.*` : Ce fichier correspond à la classe représentant toutes les instructions (`copy`, `ldconst`, `op_add`, ...) assembleurs qui peuvent être placées dans la représentation intermédiaire.
    - `BasicBlock.*` : Ce fichier correspond à la classe représentant les blocs de base, qui contiennent des instructions IR et les exécutent les uns après les autres pour générer le code assembleur x86.
    - `CFG.*` : Ce fichier correspond à la classe représentant le graphe de flux de contrôle, qui gère tous les blocs de base du programme similaire à du C.



## 4. Organisation du répertoire test

Un environnement de tests se trouve dans le sous-répertoire `tests` et permet d'automatiser les tests fonctionnels. Tous nos tests (i.e. programmes semblables à du C à tester) sont placés dans le sous-répertoire `testfiles`. Le fichier Python `ifcc-test.py` permet d'exécuter un ou plusieurs tests :

```
$ python3 ifcc-test.py testfiles # Pour exécuter tous les tests fournis
$ python3 ifcc-test.py [CHEMIN]/mon_fichier_test.c # Pour exécuter un seul test
```

Le répertoire `testfiles` contient des tests concernant toutes les fonctionnalités prises en charge. Les tests sont placés dans des dossiers qui sont nommés d'après les fonctionnalités testées.

NB : Le sous-répertoire `13_test_special` de `testfiles` contient des tests qui présentent des boucles infinies inhérentes ou ne sont pas pris en charge par notre compilateur.

Après l'exécution des tests un répertoire `ifcc-test-output` est généré. À l'intérieur, d'autres sous-répertoires sont créés pour chaque test exécuté. Par exemple, dans le cas où l'on effectue tous les tests du répertoire `testfiles`, `ifcc-test-output` aura cette structure :

```
└─ tests
   └─ ifcc-test-output
      └─ testfiles-01_01_00_return42
         └─ testfiles-01_01_01_empty
            └─ testfiles-01_01_02_semi_colon
               ...
```

Dans chaque sous-répertoire `test_files-*`, vous trouverez :

- `asm-gcc.s` : le code assembleur généré par `gcc`.
- `asm-ifcc.s` : le code assembleur généré par notre compilateur `ifcc`.
- `exe-gcc` : le binaire généré par `gcc`.
- `gcc-compile.txt` : le message de compilation écrit par `gcc`.
- `gcc-execute.txt` : le message d'exécution écrit par `gcc`.
- `gcc-link.txt` : le message de liaison écrit par `gcc`.
- `ifcc-compile.txt` : le message de compilation écrit par notre compilateur `ifcc`.
- `ifcc-execute.txt` : le message d'exécution écrit par notre compilateur `ifcc`.
- `ifcc-link.txt` : le message de liaison écrit par notre compilateur `ifcc`.
- `input.c` : le programme semblable à du C testé.

Le script Python `ifcc-test.py` compare si les résultats (`*-compile.txt`, `*-link.txt` et `*-execute.txt`) sont identiques avec `gcc` et `ifcc` (i.e., notre compilateur).