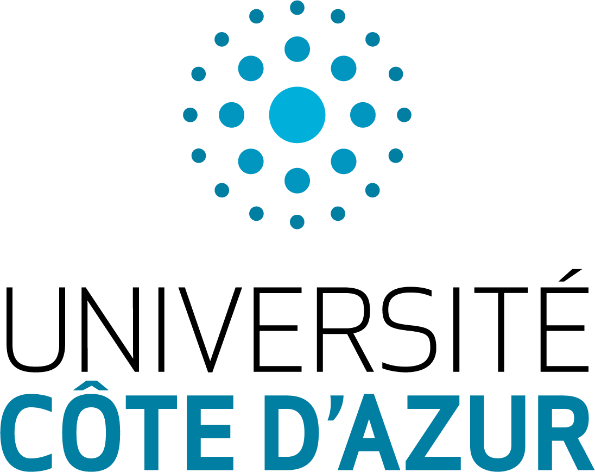
Rapport de Projet de compilation

Compilateur miniC

Langage miniC vers DOT

**2025**

****

**Réalisé par :**

**CARRIBA NOSRATI Daniel et DI PLACIDO Bence**

**SOMMAIRE**

Table des matières

[1. INTRODUCTION 3](#_Toc198754652)

[2. MÉTHODE DE TRAVAIL 4](#_Toc198754653)

[3. ARBORESCENCE DU PROJET 4](#_Toc198754654)

[4. DÉVELOPPEMENT 5](#_Toc198754655)

[4.1. ANALYSE LEXICALE 5](#_Toc198754656)

[4.2. ANALYSE SYNTAXIQUE: 5](#_Toc198754657)

[4.3. ANALYSE SÉMANTIQUE 6](#_Toc198754658)

[4.4. GÉNÉRATION DE CODE 7](#_Toc198754659)

[5. CE QUE LE PROGRAMME PERMET 8](#_Toc198754660)

[6. ERREURS DÉTECTÉS PAR L’ANALYSE SÉMANTIQUE 8](#_Toc198754661)

[6.1. ERREURS 8](#_Toc198754662)

[6.2. WARNINGS 8](#_Toc198754663)

[7. CONCLUSION 9](#_Toc198754664)

# 

# 1. INTRODUCTION

Dans le cadre de l’UE Compilation, nous avons dû réaliser un compilateur du langage C capable de traduire ce dernier en langage intermédiaire DOT, sous formes de graphes.

En réalisant ce projet, nous avons pu mettre en pratique les connaissances acquises durant le cours et les TD.

Nous avons essayé d’avancer dans le rythme du cours afin de pouvoir mettre en œuvre les connaissances acquises.

Dans un premier temps nous avons complété l’analyse lexicale (ANSI-C.l) qui génère les tokens. Par la suite, nous avons réalisé l’analyse syntaxique a l’aide des définitions donner dans le fichier miniC., durant l’évolution du projet, nous avons modifier les règles de la grammaire (voir [**analyse syntaxique**](#_4.2._ANALYSE_SYNTAXIQUE)). De plus, nous avons implémenter l’analyse sémantique pour respecter aux maximum les attentes du langage miniC et généré le code DOT conformément aux consignes.

Pour les différentes parties du projet, nous avons utilisé les technologies respectives :

* LEX , pour l’analyse lexicale
* YACC, pour l’analyse syntaxique
* C, pour la sémantique et la génération de code

Pour ce projet, nous avons dû réaliser un compilateur a deux parses. Lors de la première parse, le compilateur détecte les erreurs sémantiques et remplie une structure de donnés qui pour la seconde parse va être utiliser pour faire la traduction en DOT.

# 2. MÉTHODE DE TRAVAIL

Afin d’avancer à un bon rythme et de ne pas nous égarer dans le projet, nous avons établi différents objectifs à réaliser chaque semaine. Nous nous sommes régulièrement concertés, sur place à Valrose ou à distance via Discord, afin de discuter de nos avancées ou des problèmes rencontrés.

Nous avons utilisé Git et la plateforme GitHub pour la gestion de versions, afin de mener à bien le projet.

# 3. ARBORESCENCE DU PROJET

├── ANSI-C.l //fichier contenant l’analyse lexicale

├── exempleminiC.c

├── Grammaire-miniC.pdf

├── Grille-Evaluation-Projet-Compilation.pdf

├── Makefile //fichier Makefile pour compiler le code, exécuter le code ou encore supprimer les exécutables et fichiers dot

├── miniC.y //fichier contenant l’analyse syntaxique, analyse sémantique et génération de code.

├── Projet-Compilation-2025.pdf

├── Rapport

│ └── Rapport-CARRIBA\_NOSRATI-DI\_PLACIDO.docx //rapport du projet format word

├── README.md //readme au format md pour github

├── README.txt //readme au format txt

├── Tests

│ ├── fichiers tests //fichiers tests fournie par le professeur

│ ├── ...

├── TestsError //fichiers tests retournant différentes erreurs

│ ├── lexical\_error.c

│ ├── syntax\_error.c

│ ├── variable\_deja\_declaree\_bloc.c

│ └── variable\_void.c

└── utils //répertoire contenant les fichiers c et h utiliser par le compilateur

├── arbres.c

├── arbres.h //struct arbre et fonctions pour les arbres

├── couleurs\_terminal.h

├── dot.c

├── dot.h //fonctions pour générer du code au format dot

├── extras.c

├── extras.h //fonctions utiles

├── noeuds.c

├── noeuds.h //struct nœuds et fonctions pour les nœuds

├── tables\_symboles.c

└── tables\_symboles.h //struc pile de tables de symboles et fonctions utiles

# 4. DÉVELOPPEMENT

## 4.1. ANALYSE LEXICALE

L’analyse lexicale est la première étape de la compilation, consistant à lire le code source pour le découper en unités significatives appelées lexèmes (ou tokens), comme les mots-clés, les identificateurs, les opérateurs, etc. Elle permet ainsi de simplifier l’analyse syntaxique en éliminant les détails liés à la structure du texte brut. Pour notre projet, nous avons implémenté l’analyseur lexical à l’aide de l’outil **Lex**, qui permet de définir des expressions régulières associées à des actions en C. À chaque correspondance avec une expression, Lex exécute l’action correspondante, ce qui permet d’identifier et de classer les différents éléments du code source de manière automatique et efficace.

Nous avons supprimé les tokens inutiles pour le langage miniC et nous avons ajouté les commentaires qui sont ignoré.

## 4.2. ANALYSE SYNTAXIQUE :

TODO

## 4.3. ANALYSE SÉMANTIQUE

TODO

## 4.4. GÉNÉRATION DE CODE

# 5. CE QUE LE PROGRAMME PERMET

# 6. ERREURS DÉTECTÉS PAR L’ANALYSE SÉMANTIQUE

## 6.1. ERREURS

TODO

## 6.2. WARNINGS

TODO

# 7. CONCLUSION

TODO