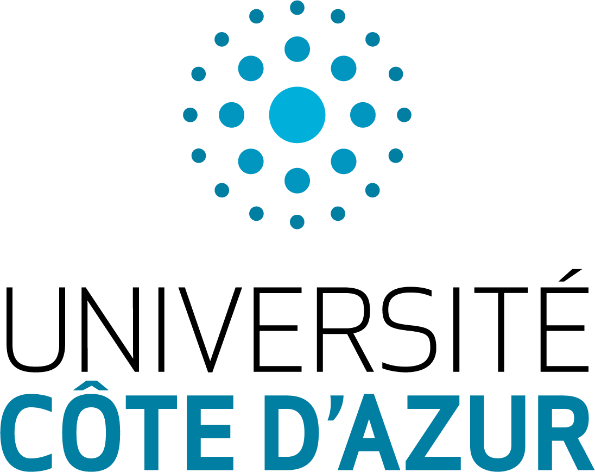
Rapport de Projet de compilation

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 de simplifier l’analyse syntaxique en préparant une représentation plus structurée du code. Dans notre projet, nous avons implémenté l’analyseur lexical à l’aide de l’outil **Lex**, qui utilise des expressions régulières associées à des actions en C pour reconnaître et catégoriser les différents éléments du code. Les tokens ainsi générés sont ensuite transmis à l’analyseur syntaxique construit avec **Yacc**, afin de permettre l’analyse grammaticale du programme.

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

## 4.2. ANALYSE SYNTAXIQUE :

L’analyse syntaxique est la deuxième étape du processus de compilation, juste après l’analyse lexicale. Elle sert à vérifier que la suite de tokens fournie par Lex respecte bien les règles de grammaire du langage, c’est-à-dire qu’elle est correctement structurée. Cette étape permet de repérer les erreurs de syntaxe et de construire une structure en forme d’arbre qui représente l’organisation du programme : c’est ce qu’on appelle l’arbre syntaxique abstrait (AST), que notre compilateur utilise réellement pour la suite du traitement. Pour cette partie, nous avons utilisé **Yacc**, un outil qui permet de décrire les règles du langage et d’associer du code C à chaque règle pour traiter ce qu’elle reconnaît. Yacc travaille main dans la main avec Lex pour analyser le code source et créer l’AST, une structure exploitable pour les étapes suivantes de la compilation.

Nous avons apporté plusieurs modifications sur la grammaire du miniC,y.

Dans un premier temps, nous avons ajouter les règles push, pop, pushf et popf a chaque début et fin de bloc, elles sont utilisé seulement pour gérer les **piles de tables de symboles** pour l’analyse sémantique. Cette modification n’impacte pas la syntaxe.

Nous avons également séparer la règle fonction en deux partie mais cela ne modifie pas la grammaire, cette modifications a pour but de pouvoir déclarer la fonction dans une table de symbole avant de traiter le corps de la fonction (utile pour les fonctions récursives).

Nous avons ajuster la règle liste\_parms car il n’était possible d’avoir un seul paramètre que dans la forme fonction(,param) nous avons donc ajouté la règle l\_parm qui permet d’avoir un unique paramètre sans avoir la virgule avant. Cette modification permet également de simplifier génération de l’AST.

La règle liste\_instructions a également été modifier, elle renvoie vers le non terminal l\_instrucions qui peux etre dérivé en une unique instruction ou en plusieurs instructions.

Dans sélection, nous avons regrouper les règles CASE et DEFAULT comme suis :

SWITCH ‘(‘ expression ‘)’ ‘{‘ push liste\_switch\_case pop ‘}’  
Cela permet dans la syntaxe d’avoir des case et default seulement si il y a un switch.

Pour la règle variable : nous avons séparer les variables et les tableaux pour faciliter la génération de l’AST.

Nous avons traiter les binary\_op directement dans la règle expression pour faciliter la gestion des priorité dans la construction de l’AST.

Liste\_expression, n’était pas adapter pour avoir une seule expression de part la , obligatoire devant (liste\_expressions ‘,’ expression). Nous avons donc également ajouté l\_expr qui prend en compte une expression sans virgule ou plusieurs expressions.

## 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