



L'ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE ET
D'ANALYSE DES SYSTÈMES

Projet Compilation Compilation du Langage C

Réalisé par :
EL MOUSS Ayman
QATAB Salah-Eddine

Encadré par :
Mr. YOUNESS Tabii

Remerciement

Au terme de ce travail, nous tenons à exprimer notre profonde gratitude à notre chère professeur et encadrant Monsieur Youness Tabii, nous le remercions de nous avoir orientés, aidés et conseillés, nous tenons à vous remercier pour les opportunités du développement intéressant que vous nous avez offertes et qui nous ont fait évoluer professionnellement mais aussi personnellement.

Nous souhaitons remercier également tous les professeurs de l'ENSIAS, bien que les membres de l'équipe pour leurs efforts, leurs engagements, et leurs patiences afin d'accomplir ce projet.

Enfin, nous ne pouvons achever ce projet sans exprimer notre reconnaissance à nos chers parents, nos frères, nos sœurs qui ont toujours été là pour nous, à nos amis pour leur sincère amitié et confiance, et à tous ceux qui ont contribué de près ou de loin à l'élaboration de ce projet.

Résumé

Ce document a pour but de décrire le déroulement de notre projet compila. Il contient l'ensemble des éléments et des étapes du qui nous a permis de compiler un fichier text écrit en langage C par ce langage-même afin d'obtenir les résultats si réussi ou non, dans ce cas elle doit afficher un message erreur selon l'erreur généré lors de l'étape compilation.

Dans un premier lieu, on fait une représentation générale du projet. La deuxième partie a pour objectif de présenter le fonctionnement technique en se persuadant par l'analyseur lexicale puis syntaxique. La dernière partie va être consacré pour la réalisation du projet en ce joignant par des images screenshot.

Table des figures

2.1	Répertoire contenant les fichiers du projet	2
2.2	Les 6 premières lignes du fichier declarations.h	3
2.3	Partie du fichier declarations.h	3
2.4	Partie du fichier functions.h	4
2.5	Partie du fichier main.c	5
2.6	Partie du fichier core.h	5
3.1	Partie du fichier Syntax.h	7
3.2	Partie du fichier Syntax.h	8
3.3	Table des règles syntaxique	9
3.4	Partie du fichier Syntax.h	9
3.5	Partie du fichier Syntax.h	10
3.6	fichier text.c à compiler	10
3.7	Exécution finale sous linux-fedora	11

Table des matières

1	Introduction	1
2	Analyseur Lexical	2
2.1	Fichier declarations.h	3
2.2	Fichier Functions.h	4
2.3	Fichier main.c à l'aide du fichier core.h	5
3	Analyseur Syntaxique	7
3.1	Forme du fichier	7
3.2	Fonctions pour l'Analyseur Syntaxique	8
3.3	La grammaire du langage C	9
3.4	Les Erreurs Syntaxique	10
3.5	Tester l'Analyseur syntaxique	10
4	Conclusion	12
	Conclusion	12

Chapitre 1

Introduction

En informatique, un **compilateur** est un programme qui transforme un code source en un code objet. Généralement, le code source est écrit dans un langage de programmation (*le langage source*), il est de haut niveau d'abstraction, et facilement compréhensible par l'humain. Le code objet est généralement écrit en langage de plus bas niveau (*appelé langage cible*), par exemple un langage d'assemblage ou langage machine, afin de créer un programme exécutable par une machine.

Un compilateur effectue les opérations suivantes : **analyse lexicale**, **pré-traitement** (*préprocesseur*), **analyse syntaxique** (*parsing*), **analyse sémantique**, et **génération de code optimisé**. La compilation est souvent suivie d'une étape d'édition des liens, pour générer un fichier exécutable. Quand le programme compilé (*code objet*) est exécuté sur un ordinateur dont le processeur ou le système d'exploitation est différent de celui du compilateur, on parle de compilation croisée.

Le processus de compilation d'un programme consiste en un certain nombre d'étapes ; en pratique les compilateurs sont écrits pour être capables de les réaliser ensemble, en faisant une seule passe dans sa donnée. Cependant, pour présenter ces étapes séparément permet de mieux comprendre le rôle de chacune de celles-ci.

Les trois grandes étapes :

- L'analyse lexicale
- L'analyse syntaxique
- La production du code objet

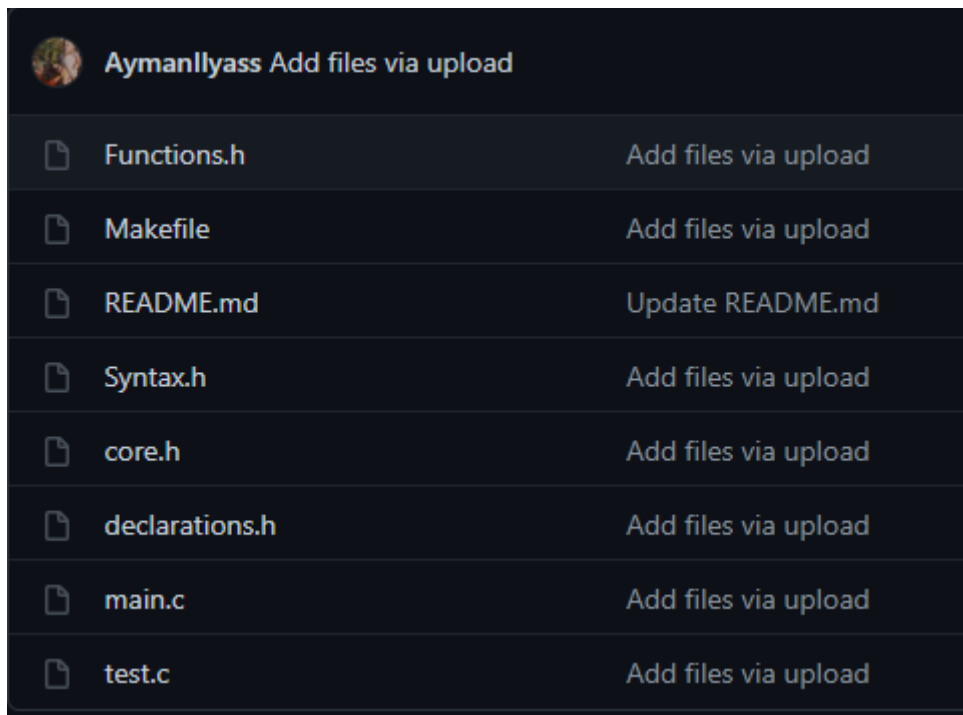
Ici on va présenter ces étapes pour le langage C en langage C même

Chapitre 2

Analyseur Lexical

Dans cette étape, on va présenter la forme générale de notre analyseur, c'est-à-dire, on va mentionner et expliquer le rôle de chacune des fichiers existants dans le dossier de l'analyseur.

En effet : Notre répertoire se compose de **8** fichiers, **2** parmi eux ont l'extension « .c » : **main.c**, **test.c**, 4 autres avec l'extension « .h » : **Functions.h**, **Syntax.h**, **declarations.h** et **core.h**, **1** fichier Makefile et **1** fichier Markdown











Aymanllyass Add files via upload	
 Functions.h	Add files via upload
 Makefile	Add files via upload
 README.md	Update README.md
 Syntax.h	Add files via upload
 core.h	Add files via upload
 declarations.h	Add files via upload
 main.c	Add files via upload
 test.c	Add files via upload

FIGURE 2.1 – Répertoire contenant les fichiers du projet

2.1 Fichier declarations.h

Ce fichier et comme son nom signifie, est le fichier où il existe l'ensemble des énumérations, structures, des listes et des variables qu'on va sûrement utiliser dans notre programme main. Effectivement : on peut décomposer ce fichier en plusieurs parties.

Les premiers 6 lignes sont réservées à l'appel des bibliothèques de langage C ou même des constantes qui vont être utiles par la suite comme SIZE, les bibliothèques appelées sont :

1. `stdbool.h` / `string.h` : il nous permet de manipuler les variables booléennes et chaînes de caractères. (`strcpy` / `strcmp` / ...)
2. `stdio.h` / `stdlib.h` : le minimum des bibliothèques essentielles pour un code C
3. `ctype.h` : un header file qui contient des fonctions pour déterminer le type d'un caractère `isalpha()`, `isdigit()` ...

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <ctype.h>
5  #include <stdbool.h>
6  #define SIZE 10007
```

FIGURE 2.2 – Les 6 premières lignes du fichier declarations.h

La deuxième partie est l'ensemble des listes et enum qui vont définir des variables et des nouveaux serviable types par la suite dans la construction des fonctions :

- **enus** : contient des mots clés définis par le support du projet.
- **Codes_LEX** : un nouveau type de variable crée à partir de l'énumération au-dessous, il contient les tokens des symboles et des mots clés de ce langage, en respectant l'ordre de positionnement de chaque symbole ou mot clés par rapport les autres symboles et mot clés.
- **Codes_LEX_Erreurs** : contient les différents erreurs pouvant s'apparaître lors d'une exécution d'un code du langage C.

```
13  typedef enum
14  {
15      ERROR,
16      NUM,
17      ID,
18      ID_FUNCTION,
19      SPECIAL,
20      STRING,
21      CHAR
22  } enus;
23  typedef enum
24  {
25      // SPECIAL
26      CRO_TOKEN, // [
27      CRF_TOKEN, // ]
28      ACO_TOKEN, // {
```

FIGURE 2.3 – Partie du fichier declarations.h

2.2 Fichier Functions.h

Pour être clair, cet header file n'est pas assez nécessaire pour que notre programme fonctionne, mais pour des raisons d'organisation et d'esthétiques, on a décidé de faire appel à functions.h comme header au lieu de declarations.h

Dans ce fichier on va construire les fonctions nécessaires pour l'extraction des Tokens et la détermination de leurs types selon l'enum CODE_LEX défini dans declarations.h

En Pratique : On va appeler tout d'abord les variables, les bibliothèques, les énumérations, les listes nécessaires pour achever notre objectif :

```
1  #include "declarations.h"
2
3  void read_char()
4  {
5      cour_char = fgetc(file);
6      if (cour_char == '\n')
7          colonne = 1, ligne++;
8      else
9          colonne++;
10 }
11 //-----
12 void read_string()
13 {
14     memset(Cour_Token.value, '\0', 31);
15
16     int nbchar = 0;
17     bool isStringConst = false;
18     bool isCharConst = false;
```

FIGURE 2.4 – Partie du fichier functions.h

Tant que son nom indique, on utilisera dans ce fichier des fonctions :

- read_char() : Cette fonction permet de lire un caractère, elle affecte une nouvelle ligne si elle trouve que ce caractère représente un retour à la ligne
- read_string() : Cette fonction permet de lire une chaîne de caractères
- read_number : Cette fonction permet de lire un caractère nombre à l'aide de boucle do—while et la fonction read_char() pour nous donner en fin un output de type enus

Détails pratiques : l'utilité de cette fonction c'est de faciliter la tâche quand on a le cas des NUM_TOKEN dans la fonction suivante *tokenmap(int type)* qu'on va la diviser en plusieurs parties due à sa grande taille et à la diversité de ses objectifs.

- La déclaration des variables locales, l'élimination des blancs et la lecture du caractère suivant
- La reconnaissance des mots-clés et des identificateurs
- La reconnaissance des Nombres
- Et pour ne pas être dérangés par les commentaires surtout qu'ils n'influencent pas sur le code, on peut les éliminer définitivement
- Reconnaissance des symboles simples
- Reconnaissance des symboles complexes : <=, :=, >=

Enfin, il nous reste d'étudier le cas où on a soit la fin de fichier (EOF) ou un caractère qui n'est pas reconnu dans le langage ceci est formulé à l'aide du fonction Analex()

2.3 Fichier main.c à l'aide du fichier core.h

Ce fichier est une synthèse de tout le travail évoqué avant, car c'est ici où on va utiliser et mettre en ordre tout nos variables et fonctions pour qu'on recoit le résultat souhaité, d'abord pour l'input, on a posé un chemin directif vers le fichier qui contient le code, puis et après le traitement des données on va faire apparaître le résultat dans le terminal, ainsi on va l'enregistrer dans un autre fichier texte pour la conserver et l'enregistrer.

```
1  #include "core.h"
2
3  int main(int *argc, char *argv[])
4  {
5
6      Analex("test.c");
7
8      for (int i = 0; i < curseur - 1; i++)
9          printf("\n %s %d %d ", mes_err[Error_table[i].code_err].message_erreur, Error_table[i].ligneErreur, Error_table[i].colonneErreur);
10     printf("\n \n \n");
11     PROGRAM();
12     if (Cour-Token.token == FIN_TOKEN)
13         printf("\n\nFichier compilé avec success!!!\n");
14     else
15         printf("\n\nFichier erroné\n");
16 }
```

FIGURE 2.5 – Partie du fichier main.c

```
1  #include "Functions.h"
2
3  struct node *ptr = NULL;
4
5  void printErreur(Codes_LEX token)
6  {
7      if (Cour-Token.ligne != prev_Erreur.ligne || Cour-Token.colonne != prev_Erreur.colonne)
8      {
9          printf("\nErreur :%d :%d : %s \n", Cour-Token.ligne, Cour-Token.colonne, maperror[token].message_erreur);
10         prev_Erreur.colonne = Cour-Token.colonne;
11         prev_Erreur.ligne = Cour-Token.ligne;
12         // exit(-1);
13     }
14 }
15
16 void Symbole_Suiv()
17 {
18     if (ptr == NULL)
19         printf("Fichier VIDE ");
20     printf("%s ", Cour-Token.value);
21     ptr = ptr->next;
22     prev_token = Cour-Token;
```

FIGURE 2.6 – Partie du fichier core.h

On a appelé le fichier « Functions.h » car il contient toutes les fonctions qu'on a besoin ainsi que les bibliothèques nécessaires pour l'assurance du bon déroulement de processus de l'analyseur syntaxique.

La liste « tokens » va permettre au programme de faire apparaître ou enregistrer le résultat de l'analyse soit au niveau de terminal ou au niveau de fichier.

On va en premier temps déterminer un chemin pour accéder au fichier qui contenir le code à analyser, après, il faut s'assurer si ce chemin est valide ou non avant commencer pour ne pas avoir des problèmes de compilation (23 <> 26)

On va initialiser le variable `currentToken` de type `token` par la première valeur retournée par la fonction `getToken()` .

Faisons une boucle `do—while` et écrire le résultat soit dans le terminal `print` ou bien dans le fichier `fprintf()` en liant entre le rang de token retourné par `getToken()` et le rang de chaine de caractères dans la liste `tokens` ,

Et voilà : lorsqu'on arrive à `FIN_TOKEN` le programme va sortir de la boucle et enregistrer l'analyse lexicale dans un autre fichier totalement séparé du premier fichier qui contient le code à analyser

Chapitre 3

Analyseur Syntaxique

Un analyseur syntaxique (parser) est basé sur un accepteur, fonction booléenne indiquant si le texte source est conforme aux règles de grammaire définissant le langage.

3.1 Forme du fichier

Définitions et Déclarations :

Où on peut inclure les bibliothèques C et déclarer les variables, les énumérations, les structures et aussi des signatures de fonctions à implémenter dans la suite du fichier :

```
63 void CALL();
64 void ARGS();
65 void ARGLIST();
66 void ARGLIST1();
67 void CONSTANT();
68
69 bool c = 0;
70 struct node *ptr = NULL;
71
72 void Symbole_Suiv()
73 {
74     if (ptr == NULL)
75         printf("^");
```

FIGURE 3.1 – Partie du fichier Syntax.h

3.2 Fonctions pour l'Analyseur Syntaxique

Cette partie est consacré à l'implémentation des fonctions utiles dans la phase de l'analyse syntaxique. La fonction `Symbole_Suiv()` permet de lire les nombres à l'aide des fonctions `isdigit()` et `isalpha()`. La fonction `Symbole_Suiv()` qui vérifie que le symbole courant est un code lexical, sinon une erreur va être détectée ensuite afficher le symbole suivant.

```
72 void Symbole_Suiv()
73 {
74     if (ptr == NULL)
75         printf("A");
76     else
77         // printf("%s ", Cour-Token.value);
78         ptr = ptr->next;
79     prev_token = Cour-Token;
80     Cour-Token = ptr->info;
81 }
82
83 void Test_Symbole(Codes_LEX token)
84 {
85
86     if (Cour-Token.token == token)
87     {
88
89         Symbole_Suiv();
90     }
91     else if (c == 0)
```

FIGURE 3.2 – Partie du fichier Syntax.h

3.3 La grammaire du langage C

Dans cette partie on doit inscrire les différentes règles de production de la grammaire

```
1 //-----Z:, X:= Q:[ T:] R:{ P:}
2 PROGRAM -> DECLLIST
3 DECLLIST -> DECL DECLLIST1
4 DECLLIST1 -> DECL DECLLIST1 | eps
5 DECL -> TYPE FTYPE
6 FTYPE -> VarDeclList ; | idfunc (PARAMS) STMT
7 VARDECLLIST -> VARVAL VARDECLLIST1
8 VARDECLLIST1 -> , VARVAL VARDECLLIST1 | eps
9 VARVAL -> VARID VARVALLIST
10 VARVALLIST -> = SIMPLEEXP | eps
11 VARID -> id VARIDEXTRA
12 VARIDEXTRA -> [ numconst ] | eps
13 TYPE -> int | bool | char
14 PARAMS -> PARAMSLIST | eps
15 PARAMSLIST -> PARAMTYPE PARAMSLIST1
16 PARAMSLIST1 -> , PARAMTYPE PARAMSLIST1 | eps
17 PARAMTYPE -> TYPE PARAMID
18 PARAMID -> id EXTRA
19 EXTRA -> [ ] | eps
20 STMT -> EXPSTMT | SCOPESTMT | CONDSTMT | ITERSTMT | RETURNSTMT | BREAKSTMT | READSTMT | WRITESTMT
21 EXPSTMT -> EXP ; | ;
22 SCOPESTMT -> { LOCALDECLS STMTLIST }
23 LOCALDECLS -> LOCALDECLS1
24 LOCALDECLS1 -> SCOPEDVARDECL LOCALDECLS1 | eps
25 SCOPEDVARDECL -> TYPE VARDECLLIST ;
26 STMTLIST -> STMTLIST1
27 STMTLIST1 -> STMT STMTLIST1 | eps
```

FIGURE 3.3 – Table des règles syntaxique

```
98 void PROGRAM()
99 {
100     DECLLIST();
101 }
102
103 void DECLLIST()
104 {
105     DECL();
106     DECLLIST1();
107 }
108
109 void DECLLIST1()
110 {
111     switch (Cour_Token.token)
112     {
113     case INT_TOKEN:
114         DECL();
```

FIGURE 3.4 – Partie du fichier Syntax.h

3.4 Les Erreurs Syntaxique

Pour chaque symbole on lui associé un code d'erreur et un message d'erreur.
Au début on a déclaré une structure de type `Maptoken_erreur` qu'on a utilisé dans l'analyse syntaxique :

```
253 Maptoken_erreur maperror[100] = {
254     {CRO_TOKEN_ERREUR, " un '['est manquant"}, // [
255     {CRF_TOKEN_ERREUR, " un ']'est manquant"}, // ]
256     {ACO_TOKEN_ERREUR, " un '{'est manquant"}, // {
257     {ACF_TOKEN_ERREUR, " un '}'est manquant"}, // },
258     {COLON_TOKEN_ERREUR, " un ':'est manquant"}, // :
259     {NOT_TOKEN_ERREUR, " un '!'est manquant"}, // !
260     {AFF_TOKEN_ERREUR, " un '='est manquant"}, // =
261     {MOD_TOKEN_ERREUR, " un '%est manquant"}, // %
262     {PV_TOKEN_ERREUR, " un ';'est manquant"},
263     {PLUS_TOKEN_ERREUR, "une operation mathematique ne peut etre effectuer sans operateur"},
264     {MOINS_TOKEN_ERREUR, "une operation mathematique ne peut etre effectuer sans operateur"},
265     {MULT_TOKEN_ERREUR, "une operation mathematique ne peut etre effectuer sans operateur"},
266     {DIV_TOKEN_ERREUR, "une operation mathematique ne peut etre effectuer sans operateur"},
267     {VIR_TOKEN_ERREUR, " un ','est manquant"}, // ,""},
268     {EG_TOKEN_ERREUR, " un '='est manquant"},
269     {INF_TOKEN_ERREUR, "une operation booleenne ne peut etre effectuer sans operateur"},
270     {INFEG_TOKEN_ERREUR, "une operation booleenne ne peut etre effectuer sans operateur"},
271     {SUP_TOKEN_ERREUR, "une operation booleenne ne peut etre effectuer sans operateur"},
272     {SUPEG_TOKEN_ERREUR, "une operation booleenne ne peut etre effectuer sans operateur"},
273     {DIFF_TOKEN_ERREUR, "une operation booleenne ne peut etre effectuer sans operateur"},
274     {PO_TOKEN_ERREUR, " un '('est manquant"},
275     {PF_TOKEN_ERREUR, " un ')'est manquant"},
```

FIGURE 3.5 – Partie du fichier Syntax.h

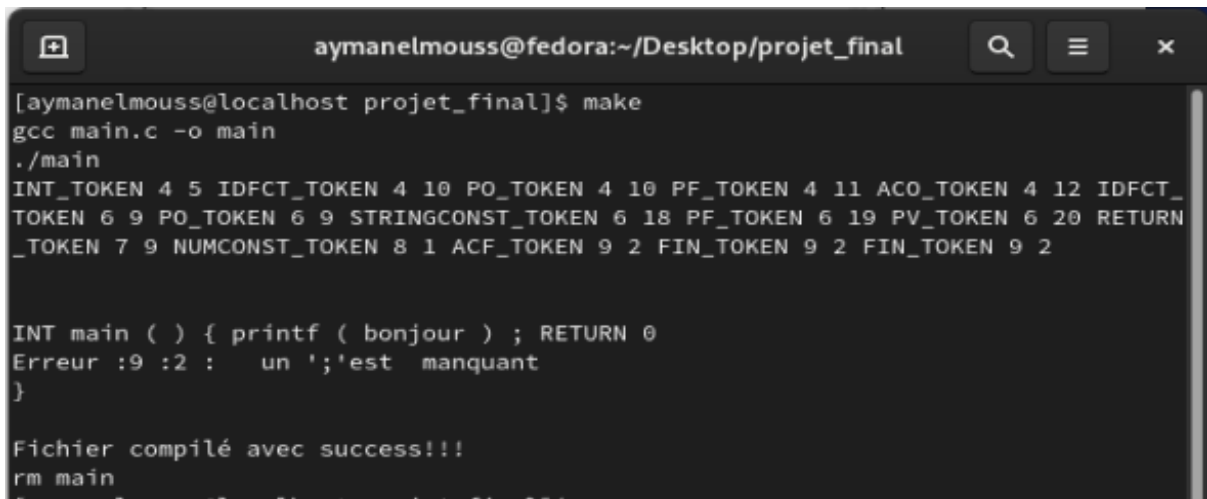
3.5 Tester l'Analyseur syntaxique

Pour essayer notre analyseur syntaxique on va lui fournir le code suivant (fichier test.C) :

```
9 lines (4 sloc) | 50 Bytes
1
2
3
4 int main(){
5
6     printf("bonjour");
7     return 0
8
9 }
```

FIGURE 3.6 – fichier text.c à compiler

Le résultat d'exécution donne : "Fichier compilé avec succès"

A terminal window titled 'aymanelmouss@fedora:~/Desktop/projet_final' with search, menu, and close buttons. The terminal shows the execution of 'make', which runs 'gcc main.c -o main' and './main'. It displays a list of tokens and their counts, followed by a C code snippet for 'main' that prints 'bonjour' and returns 0. An error message 'Erreur :9 :2 : un ';'est manquant' is shown. The compilation is successful, and the file is removed with 'rm main'.

```
[aymanelmouss@localhost projet_final]$ make
gcc main.c -o main
./main
INT_TOKEN 4 5 IDFCT_TOKEN 4 10 PO_TOKEN 4 10 PF_TOKEN 4 11 ACO_TOKEN 4 12 IDFCT_
TOKEN 6 9 PO_TOKEN 6 9 STRINGCONST_TOKEN 6 18 PF_TOKEN 6 19 PV_TOKEN 6 20 RETURN
_TOKEN 7 9 NUMCONST_TOKEN 8 1 ACF_TOKEN 9 2 FIN_TOKEN 9 2 FIN_TOKEN 9 2

INT main ( ) { printf ( bonjour ) ; RETURN 0
Erreur :9 :2 : un ';'est manquant
}

Fichier compilé avec success!!!
rm main
```

FIGURE 3.7 – Exécution finale sous linux-fedora

Chapitre 4

Conclusion

De manière générale, un compilateur a pour objectif de convertir un fichier texte (contenant un code source) en un fichier binaire (par exemple un exécutable). Une fois l'exécutable construit, on le lance comme n'importe quel programme. Ce programme peut se lancer sans que l'on dispose du code source.

Puis on se relance par les phases de compilation : la première étape consiste à écrire le code source en langage C (fichiers .c et .h). Ensuite on lance une compilation, par exemple avec gcc. La compilation (au sens vague du terme) se déroule en trois grandes phases : La précompilation, La compilation et Le linkage.

Bibliographie

- [1] <https://fr.wikipedia.org/wiki/compilateur>.
- [2] https://lagrida.com/informatique/informatique_theorique/analyseur_lexicale.html.
- [3] <https://www.commentcamarche.net/faq/14440-la-compilation-et-les-modules-en-c-et-en-c>.
- [4] <http://www2.ift.ulaval.ca/~dadub100/cours/A19/3101/slides3.pdf>.