



Theorie des langages

Note de T.Hafsaoui
cours de Nicolas Meloni 2021

Table des matières

1	Chapitre 0 : Une premiere approche du compilateur	2
1.1	Introduction	2
1.2	Historique	2
1.3	Une rapide presentation	2
1.3.1	Bootstrapping	3
1.4	Exemple de compilation	4
1.4.1	Python	4
1.4.2	C	4
1.5	L'outils Make	5
1.6	Structure d'un compilateur	6
2	Chapitre 1 : Traduction elementaire	7

2.1	Analyse lexicale	7
2.2	Analyse syntaxique	8
3	Chapitre 2 : Automate	9
3.1	Langage	9

1 Chapitre 0 : Une premiere approche du compilateur

1.1 Introduction

En informatique en generale, et tout particulierement dans la sciences de l'informatique on suit une logique similaire celle ci ce decompose de cette maniere pour notre tres chere compilateur.

1. Theorie mathématiques : theorie des langage
2. Modelisation ou traduction du probleme en struture de donnees informatique : arbre, graphe
3. Toruver un algorithme efficace afin de resoudre nos probleme
4. Programmation

Le cours est baser sur la bible de la compilation " *Le Dragon*" (Compilation :Principes, technique et outils).

1.2 Historique

L'Histoire des compilateur commence en 1950 precedament les programmes etait ecrit en langage machine avec les inconenients que cela accompagne. Le premier compilateur est ecrit par Grace Hopper pour le langage A, divers linguiste et Knuth aurons aussi un role important dans nos compilateur moderne avec par exemple *GCC*(GNU compiler collection) comme aboutissement, qui aujaud'hui compile pas moins de 7 langage dont le C et fait meme le cafe.

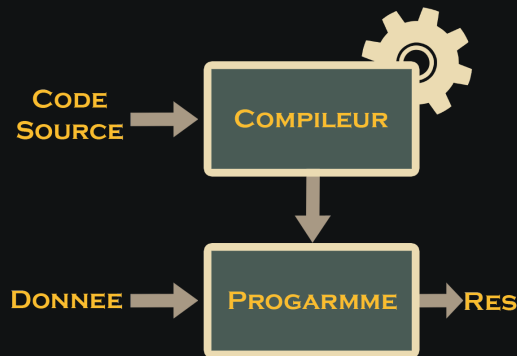
1.3 Une rapide presentation

Compilateur : Un compilateur est esentiellement un traducteur, il prend en entree un programme ecrit dans un langage $L1$ et produit un programme equivalent dans un langage $L2$.

L'usage veut que le programme en sortie soit en langage machine, c'est a dire executable par la machine.

On peut distinguer deux familles de compilateur :

1. Compilateur(les **vrai**)
Produit du code machine dont l'exécution se fait ulterieurement avec des donnees
2. Interpreteur
Lit le programme et effectue lui meme les opration sur les donnees.



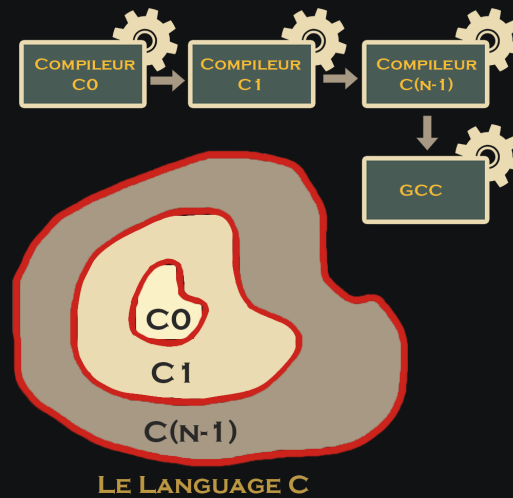
En generale le compilateur produit du code de maniere plus efficace(vitesse,taille,consomation memoire/electrique). En effet au contraire du compilateur il peut prendre la totalite du code source comme base de travail et faire des optimisation de maniere globale, ce qu'evidament un interpreteur ne peut faire.
remarque : Un programme C est, en generale, 10x plus rapide que sont equivalent en python.

On parle souvent de langage compiler ou interpreter mais il s'agit d'un abus de langage. Un langage de programmation est avant tout une norme tout langage peut donc etre compiler ou interpreter.

1.3.1 Bootstrapping

Le compilateur C *gcc* est ecrit de maniere assers peu intuitive en...C. On se retrouve donc face a un probleme d'oeuf et de poule, on parle dans ce cas la de probleme de *Bootstrapping* ou amorcage dans la langue de Moliere. Pour resoudre ce probleme on ecrit un premier compilateur en langage machine pour un sous-ensemble du C, puis on etend succesivement le compilateur a tout le langage a l'aide des compilateurs intermediaires.

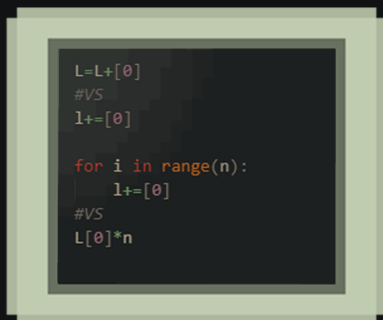
Appelons $C_1 C_2 \dots C_n$ les sous ensembles de C :



A la fin lorsque l'on se retrouve avec un compilateur C écrit en langage C_{n-1} on peut enfin écrire un compilateur C .

1.4 Exemple de compilation

1.4.1 Python



Lors de l'exécution d'un script, l'interpréteur compile les instructions en *Bytecode* (langage de la machine virtuelle de Python), la machine virtuelle exécute ensuite le bytecode. En général, le processus produit un fichier en .Pyc (Python compiler), ce fichier permettra une exécution plus rapide en cas de 2nd utilisation. Le module *dis* (pour disassembleur) permet d'obtenir le *Bytecode* d'un script. On peut ainsi de cette manière étudier la différence d'efficacité entre deux méthodes.

1.4.2 C

La compilation avec *GCC* se fait en 4 phases :

- Préprocesseur : Le fichier source est analysé et des transformations textuelles sont effectuées (`# define`, `# include`, `# ifdef`).

- Compilation : Traduction en assembleur (verification de la norme du langage).
- Assemblage : Traduction en langage machine (fichier en .O)
- Édition de lien : Les fichiers sont liés entre eux pour produire l'exécutable.

Les options -E, -S, -c permettent l'arrêt de la compilation à chacune de ces phases.

1.5 L'outil Make

C'est un outil de génération de fichiers, en général l'outil se présente sous forme de règles dans un fichier nommé *Makefile* ou *makefile*. Ces règles sont constituées d'une série d'instructions sous la forme :

Cible : dépendance 1...dépendance n
Commande

Lors de l'exécution de la commande *make* l'outil analyse les commandes successives, les règles et surtout les dates de dernière modification des dépendances. Si l'une de ces dates est plus récente que celle de la cible *make* exécute les différentes instructions afin de mettre à jour la cible.

Petite exemple :

```
Myprog : matrix.o vecteur.o
gcc -Wall matrix.o
main.o vecteur.o -o Myprog
main.o: main.c
gcc -c main.c

vecteur.o: vecteur.c
gcc -c vecteur.c

matrix.o: matrix.c vecteur.o
gcc -c matrix.c
```

- Cas 1 : Lors du premier appel à *make* l'outil crée tous les fichiers .o puis le programme *Myprog*.
- Cas 2 : Tout est compilé mais *main.c* est modifié, seule la commande qui concerne la cible *main.o* et *Myprog* seront de nouveau exécutées.
- Cas 3 : On modifie *vecteur.c* avec tout déjà compilé. De manière logique *vecteur.o*, *matrix.o* et *Myprog* seront recompilés.

Tips : Il est possible de simplifier l'écriture des makefile on autorise l'utilisation de variable et de raccourcis. Ici une variable permet simplement de donner un nom a une chaîne de caractère :

```
Non_de_chaine=$(chaîne de char)
```

Il existe aussi des variables prédefinies :

- \$@ : le nom de la cible
- \$^ : ensemble de dépendance
- \$< : première dépendance

Cible Phony : Il est possible de simplifier l'écriture des makefile on autorise l'utilisation de variable et de raccourcis. Ici une variable permet simplement de donner un nom a une chaîne de caractère :

```
Clean :  
rm -f *.o *
```

1.6 Structure d'un compilateur

On peut décomposer le processus de compilation en 2 grandes phases :

1. L'analyse (Partie frontale) :

qui consiste à construire une représentation abstraite du programme afin d'en dégager la structure grammaticale. Il s'agit d'extraire du flot de données en entrée les différents composants et de les relier entre eux en fonction de la grammaire.

2. La synthèse (Partie finale) :

se sert des données de l'analyse pour produire le programme cible.

On découpe en générale ces deux phases en différentes parties :

1. L'analyse lexicale :

Le but de cette phase est de transformer le flot de données en une séquence d'unités lexicales, une unité valide étant définie par le langage.
ex :

- une suite de chiffres représentant un nombre
- une suite de lettres représentant un mot-clé
- certains caractères spéciaux sont vus comme des opérateurs (*, +, -)

2. L'analyse syntaxique :

prend les données fournies par le lexer et produit une représentation hiérarchique suivant la grammaire.

ex : $y = x + 10$ devient $\langle ID, y \rangle, \langle EGAL, = \rangle, \langle ID, x \rangle, \langle OP, + \rangle, \langle NB, 10 \rangle$

3. L'analyse sémantique :

Ici on vérifie que les instructions ont un sens

ex : $x = 10 + [1]$

4. Production du code intermediaire :

On se sert des données précédentes pour créer du code machine mais uniquement pour une machine abstraite.

5. Production du code :

Optimisation finale du code à partir du code intermédiaire avec par exemple des options spécifiques (*gcc -O1 -O2 -O3*)

2 Chapitre 1 : Traduction élémentaire

Le but de ce chapitre est d'écrire un programme capable d'évaluer une expression arithmétique, c'est à dire les expressions de la forme $((2+3)7*9)$, ce programme nous permettra de mettre en œuvre les grandes étapes de la compilation d'un programme, c'est à dire l'analyseur lexical/syntaxique ainsi que la génération de code et peut-être l'optimisation.

2.1 Analyse lexicale

La première étape consiste à définir l'alphabet du langage étudié ici on considère :

$$\Sigma : \{0, 1, 2 \dots 9, +, -, *, /, (,)\}$$

On définit un mot du langage comme étant une suite de caractères/lettres/symboles. Pour définir proprement la norme d'un langage en terme de lexème on utilise la notation suivante :

|NB → 0|1|2|...|9

|OP → +|-|*|/

|PO → (

|PF →)

Exemple de lexème

Le but de l'analyseur lexical est de transformer un flot de symboles en suite d'unités lexicales de la forme : $\langle \text{lexème}, \text{valeur} \rangle$

ex : $9 * (5 - 2)$ L'analyseur lexical doit produire $\langle NB, 9 \rangle \langle OP, * \rangle \langle PO, (\rangle \langle NB, 5 \rangle \langle OP, - \rangle \langle PF,) \rangle$

Algo Lexer :

```

1  def Lexer(s):
2      #Donner: une chaîne de caractere
3      #Var: L une liste
4      list_ul = []
5      i = 0
6      OP = ['+', '*', '-', '/']
7      for chr in s:
8
9          if(chr.isnumeric()):
10             list_ul.append(("NB",int(chr)))
11
12             elif chr in OP:
13                 list_ul.append(("OP",chr))
14
15             elif chr == '(':
16                 list_ul.append(("PO",chr))
17
18             elif chr == ')':
19                 list_ul.append(("PF",chr))
20
21             elif chr != ' ':
22                 print("Erreur")
23                 return(1)
24             i+=1
25     return list_ul
26
27
28
29
30

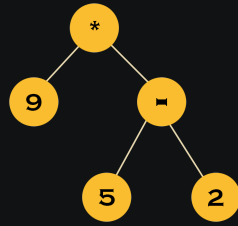
```

2.2 Analyse syntaxique

Le but est de transformer notre liste d'unités lexicales en arbre abstrait/syntaxique. Cette arbre devra refléter la structure grammaticale du langage, sachant qu'une grammaire se définit par un ensemble de règles appelées productions. On distingue deux types de syntaxe :

1. Terminaux, correspondant au lexème du langage
2. Non-Terminaux, qui sont des variables

Une fois la grammaire G définie, on sait que G dérive une expression e si il existe une suite finie de productions partant du symbole de départ et produisant e .



Prenon l'exemple de :
 $9*(5-2)$
 Expr- Expr Op Expr

3 Chapitre 2 : Automate

3.1 Langage

Pour résoudre les nombreux problèmes que pose l'analyse lexicale des langages de programmation nous allons étudier une partie importante de la science de l'informatique : la théorie des langages et des automates.

Alphabet : un alphabet est un ensemble fini, dont les éléments sont appelés symboles ou lettres.

Mot : soit σ un alphabet, un mot est une suite de symboles. par exemple :
 $w : w_1, w_2, \dots, w_n$ un mot sur σ .