Theorie des langages

Note de T.Hafsaoui
rours de Nicolas Meloni 2021

Table des matières

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

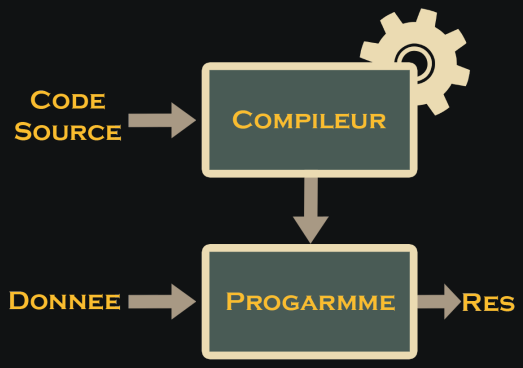
L'Histoire des compilateur commence en 1950 precedament les programmes etait ecrit en langage machine avec les inconenients que cela accompagne.
Le premier compilateur est ecrit par Grace Hopper pour le langage A, divers linguiste et Knuth aurons aussi un role important dans nos compilateur moderne avec par exemple *GCC*(GNU compiler collection) comme aboutisment, 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'execution se fait ulterieurement avec des donnees
2. Interpreteur
   Lit le programme et effectue lui meme les opration sur les donnees.



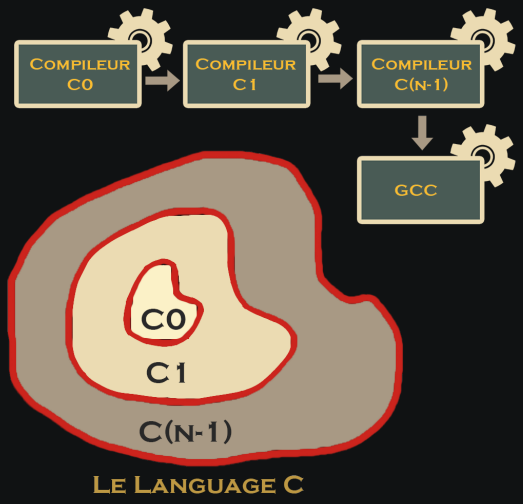
En generale le compilateur produit du code de maniere plus efficace(vitesse,taille,consomation memoire/electrique). En effet au contraire du compilateur il peut prendre la totalite du code source comme base de travaille et faire des optimisation de maniere globale, ce qu'evidament un interpreteur ne 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 language peut donc etre compiler ou interpreter.

1.3.1 Bootstrapping

Le compilateur C *gcc* est ecrit de maniere asser peut intuitive en...C. On se retouve donc fasse a un probleme d'oeuf et de poul, 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 succesivment le compilateur a tout le language a l'aide des compilateur intermediaire.

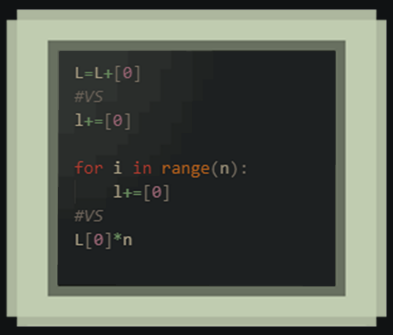
Appelons C_1 $C_2 \dots C_n$ les sous ensemble de C :



A la fin lorsque l'ont se retouve avec un compilateur C ecrit en langage C_{n-1} on peut enfin ecrire un compilateur C.

1.4 Exemple de compilation

1.4.1 Python



```
L=L+[0]
#VS
l+=[0]

for i in range(n):
    l+=[0]
#VS
L[0]*n
```

Lors de l'execution d'un scripte, l'interprete compile les instruction en *Bytecode*(langage de la machine virtuelle de python), la machine virtuelle execute ensuite le bytecode. En generale le processus produit un fichier en .Pyc(python compiler), ce fichier permettra une execution plus rapide en cas de 2^{nd} utulisation. Le module *dis*(pour disambleur) peremet d'obtenire le *Bytecode* d'un scripte. On peut ainssi de cette maniere etudier la difference d'efficacite entre deux methode.

1.4.2 C

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

— Prepoceseur :Le fichier source est analyse et des transformation textuelle sont effectue (# define, # include, # ifdef).

— Compilation :Traduction en assembleur (verification de la norme du langages).
— Assemblagetraduction en langages machine(fichier en .O)
— Edition de lienLes fichier sont lier entre eux pour produire l'executable.

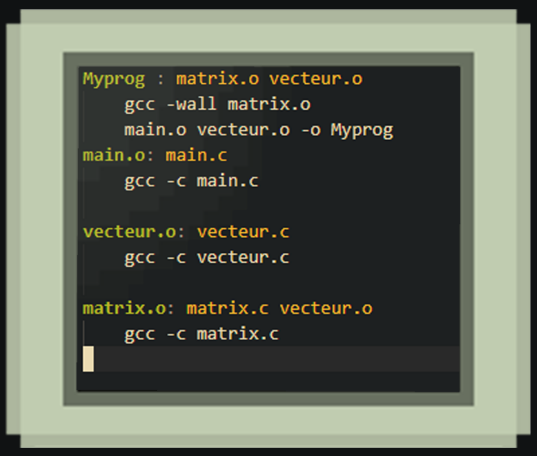
Les option -E, -S, -c permete l'arret de la compilation a chacune de ces phases.

1.5 L'outils Make

C'est un outils de generation de fichier, en generale l'outils se presente sous forme de regle dans un fichier nommer $Makefile$ ou $makefile$. C'est regle sont constitueer d'une serie d'instruction sous la forme :

Cible : dependance 1... dependance n
Commande

Lors de l'execution de la commande *make* l'outils analyse les commande succesive, les regle et surtout les date de derniere modification des dependance. Si l'une de ces date est plus recente que celle de ca cible *make* execute les differente instruction afin de mettre a 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 a *make* l'outils cree tous les fichier .o puis le programme Myprog.
— Cas 2 : Tout est compiler mais main.c, et modifier seule la commande qui concerne la cible *main.o* et $Myprog$ seront de nouveaux executer.
— Cas 3 :On modifie vecteur.c avec tout deja compiler. De maniere logique vecteur.o matrix. et $Myprog$ seront recompiler.

Tips : Il est possible de simplifier l'ecriture des makefile on autorise l'utulisation de variable et de racourcis. Ici une variable permet simplement de donner un nom a une chaine de chractere :

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

Il existe aussi des variable predefinie :

- $@ : le nom de la cible
- $^ : ensemble de dependance
- $< : premiere dependance

Cible Phony : Il est possible de simplifier l'ecriture des makefile on autorise l'utulisation de variable et de racourcis. Ici une variable permet simplement de donner un nom a une chaine de chractere :

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

1.6 Structure d'un compilateur

On peut decompose le processus de de compilaction en 2 grande phase :

1. **L'analyse**(Partie frontal) :
   qui consiste a construire une representation abstraite du programme afin d'en degager la structure gramatical. Il s'agit d'extraire du flot de donnees en entre les different composant et de les relier entre eux en fonction de la grammaire.
2. **La synthese**(Partie final) :
   se serre des donnees de l'analyse pour produire le programme cible.

On decoupe en generale ces deux phases en differente partie :

1. **L'analyse lexicale** :
   Le but de cette phase et de transforme le flot de donnees en une sequence d'unite lexicale, une unite valide etant definie par le langage.
   ex :
   - une suite de chiffre representant un nombre
   - une suite de lettre representan un mot clee
   - certain caractere speciaux sont vue comme des operateur(*,+,-)
2. **L'analyse syntaxique** :
   prend les donner forunit par le lexer et produit representation hierachique suivant la grammaire.
   ex : $y = x+10$ devient $< ID, y >, < EGAL, =>, < ID, x >, < OP, + >, < NB, 10 >$
3. **L'analyse sementique** :
   Ici on verifie que les instruction est un sens
   ex : x= 10+[1]

4. **Production du code intermediaire** :
   On se serre des donnee precdente pour cree du code machine mais uniquement pour une machine abstraite.
5. **Production du code** :
   Optimisation final du code a partir du code intermediaire avec par exemple des option specifique(*gcc* -O1 -O2 -O3)

2 Chapitre 1 : Traduction elementaire

Le but de ce chapitre est d'ecrire un programme capable d'evaluer une expression arithmetique, c'est a dire les expression de la forme ((2+3)7*9), ce programme nous permettra de mettre en oeuvre les grande etape de la compilation des programme, c'est a dire l'analyseur lexicale/syntaxique ainsi que la generation de code et peut etre l'optimisation.

2.1 Analyse lexicale

Le premier etape consiste a definir le l'alaphabet du langage etudier ici on considere :

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

On definit un mot du langage comme etant une suite de caractere/lettre/symbole. Pour definir proprement la norme d'un langage en terme de lexeme on utulise la notation suivante :

|NB→ 0|1|2|...|9
|OP→+|-|*|/
|PO→(
|PF→)
Exemple de lexeme

Le but l'anlyseur lexicale et de transformer un flot de symbole en suite d'unite lexicale de la forme : $< lexeme, valeur >$
ex :$9 * (5 - 2)$ L'analyseur lexicale doit poduire $< NB, 9 >< OP, * >< PO, (>< NB, 5 >< OP, 2 >< PF,) >$
Algo Lexer :

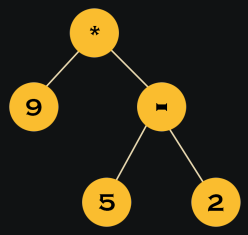
```python
def Lexer(s):
    #Donner: une chaîne de charactere
    #Var: L une liste
    list_ul = []
    i = 0
    OP = ['+','*','-','/']
    for chr in s:

        if(chr.isnumeric()):
            list_ul.append(("NB",int(chr)))

        elif chr in OP:
            list_ul.append(("OP",chr))

        elif chr == '(':
            list_ul.append(("PO",chr))

        elif chr == ')':
            list_ul.append(("PF",chr))

        elif chr != ' ':
            print("Erreur")
            return(1)
        i+=1
    return list_ul
```

2.2 Analyse syntaxique

Le but et de tranformer notre list d'uniter lexicale en arbre abstrait/syntaxique. Cette arbre devras refletais la stuture gramatical du langage, sachant qu'une grammaire se definit par un ensemble de regle appele production. On distingue deux type de syntaxe :

1. Terminaux, corespondant au lexeme du langage
2. Non-Terminaux, qui sont des variable

Une fois la grammaire G definit, on sait que G derive une expression e si il exitse une suite fini de produit partant des symbole de depart et produisant e.



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

3 Chapitre 2 : Automate

3.1 Langage

Pour resoudre les nombreux probleme que pose l'analyse lexicale des lanagge de programation nous allons etudier une partie important de la sciences de l'informatique la theorie des langages et des automate.
Alphabet : un alaphabet est un ensemble de fini, dont les element sont appele symbole ou lettre.
Mot : soit σ un alaphabet, un mot est une suite de simbole. par exemple : w **:** $w_1, w_2 \ldots, w_n$ un mot sur σ.