

Compilation

(machine virtuel,
simulation)

Compilateur ≠ interpréteur



programme qui transforme un autre d'une forme à une autre.

Ex: Code source → binaire

| ↴ autre langage
↳ Assembleur

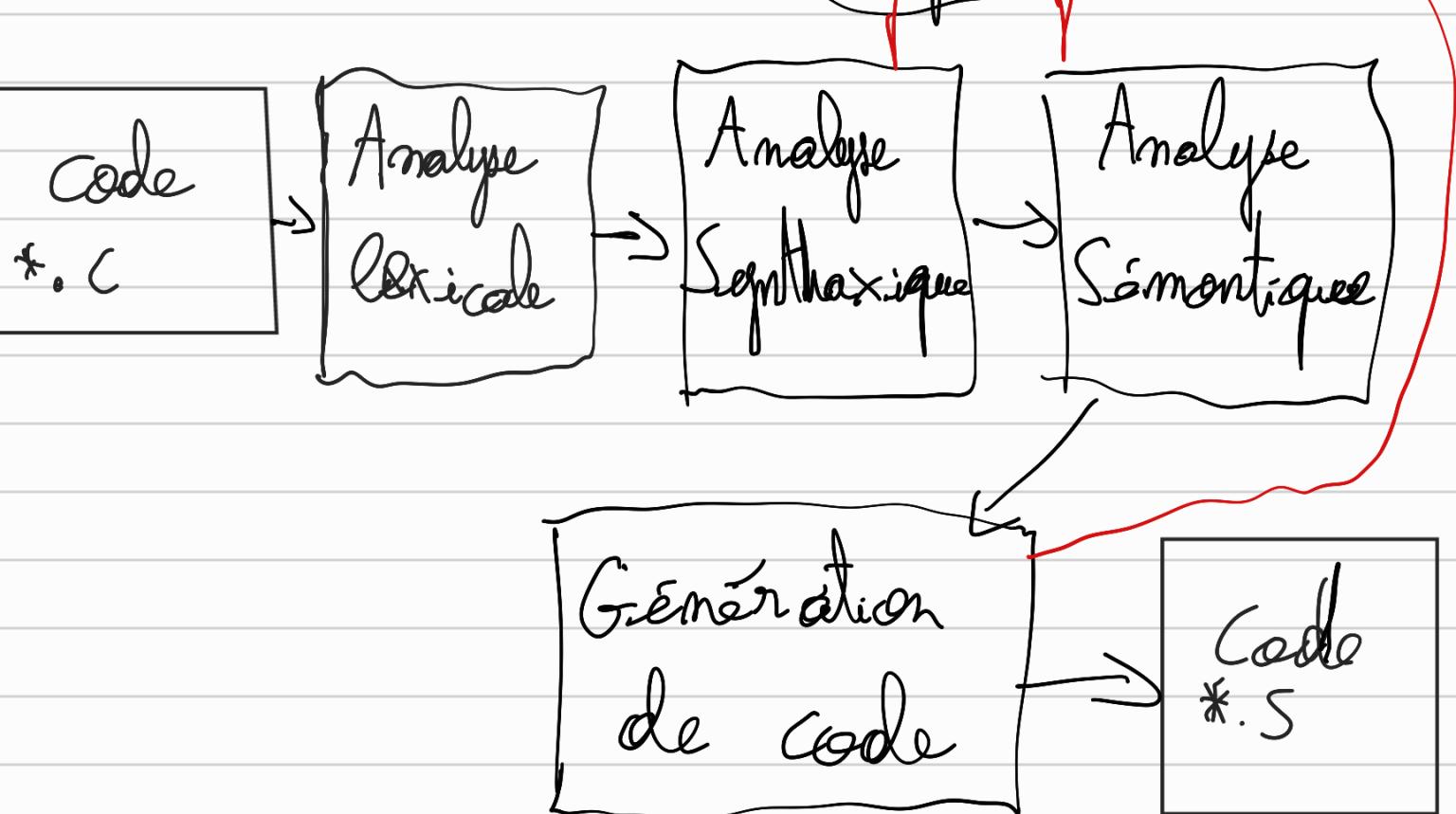
prend le code source (ex: python)
et exécute sa sémantique

Notre compilateur:

*S: Assembleur

- minimiser les chaînes de caractères





Analyse lexicale:

ex:

3 - 45 * doc
on ASCII

32

52 53

Savoir que le 4 et 5 forme un nombre
en faire une constante à stocké
tout en sachant qu'il y a des espaces
avant et après. Pour ça :

Transformer le programme en token

Type
Value

ext:

Constante
entiere

3

Signe
moins

Contento entière	Signé étoile
45	O

Identification
"abc"

↑
Soustraction ou négation ?
⇒ Analyse Synthétique

En C ces pointeurs sont des structures

Valueur prendra 2 types : entier ou chaîne

stück teken {

1

int / string values

{}

1) en C une énumération de type
qui permet de savoir le type des
tokens

(enver une phrase en mot
(séquence de token)

Analyse Syntaxique

Transfomme ces séquences de token en arbre
syntaxique. Vérification de la
syntaxe du code.

Appl de la fd n'est en branche
générale du code

Production de code assembleur

Analyse Sémantique

Vérification des types (pour la
cohérence)
Déterminer la portée des variables

qui seront ajouté dans les noeuds de l'arbre.

$$\underline{\text{ex. }} 3 = 5 + 1^*$$

est syntaxiquement correct
mais pas sémantiquement

Dans notre schéma il manque 2

- optimisation : - avant Code *
- avant Apres Analyse Sémantique

Analyse Lexicale :

2 choix

à la main

ou avec un automate

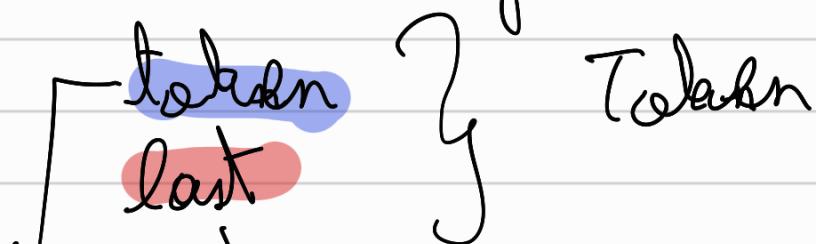
Bande et condition

Fichier à part dans un autre langage

Stockage des tokens sous forme de flux, il corrigé un par un vers

l'analyse Sémantique

2 variables globale dans le code



token) sur lequel on a accès de
basseur
token d'avant

ex : 1 (1) 2 * abc
last token

accès à ces 2 variables en lecture

la fonction next()

last = token ;

token ← lecture

(lecture du token suivant)

les fonctions :

bool check(+) {

```
if (token.type == T){
```

```
    next();  
    return True;  
} else {  
    return False;  
}
```

```
}
```

```
accept (T) {
```

```
    if (!check(T))
```

```
        ErreurFatal();
```

```
}
```

Possibilité de récupérer la erreur

liste des tokens :

• identificateur, dans notre cas ce seront des lettres en minuscule.

simon : [a-zA-Z][a-zA-Z0-9]*

• Un token pour chaque mot-clé :

int, for, while, if, else, do, break

, continue, return

On reçoit l'identification et on la
compose avec les mots-clés pour savoir
quel token utiliser.

- constante
- EOF (end of file) à la fin du fichier
on met ce token
- Un token pour opération:

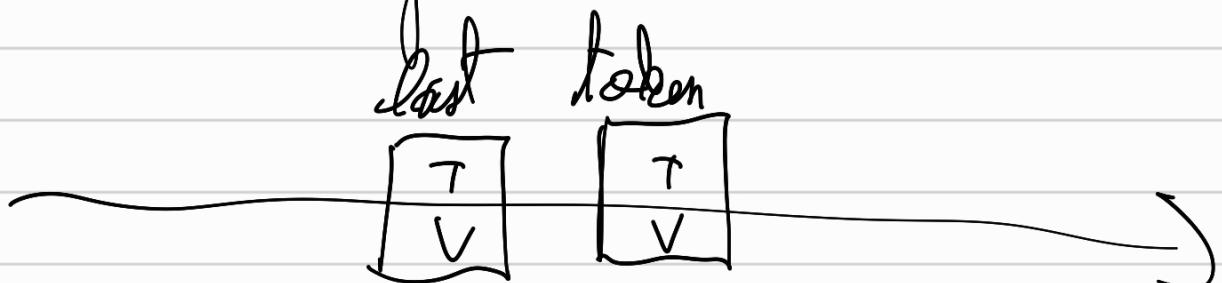
+ - * / % ! &
< = > = == !=

Si prochain caractère constant ou =

& & || () [] { }
et les séparateurs
2 tokens différents

() ^ =

Dans le code on a une vision sur 2 tokens (last, token) mais pas sur ceux passés ou futurs.



next()

last.T = token.T
last.V = token.V

While (isSpace (code [pos]))

pos ++

C = code [pos ++]

if (isChiffre (c)) {

} else if (isAlpha (c)) {

} else {

switch(c){

case '+' : token T = tok_plus ;

main ()

init Analex

← position à 0

next()

→ While (token.T != tok_eof) {

A = AnaSem()

AnaSem(A)

GenCode(A)

}

Gré
corbe

pas utile
sans variable

Analyse Syntaxique : Gré corbe
des tokens

Chaque noeud est typé

Combinaison de noeud pour représenter
les noeuds.

fonction de noeud → fonction token

Negle des nœuds → nœuds vides

struct nœud {

 int type ;

 int valeur ;

 Vector<nœud> enfant ;

}

N = nœud ("Node_constant", 3)

N = nœud ("Node_negation", false)

valeur



fils

AnaSyn()

token doit être le premier
token d'un atome valide
Simon Erreun Fataal

return E()

Nœud Atome()

atom complexe

$$7 + (2 \times 3) \leftarrow \text{évaluation}$$

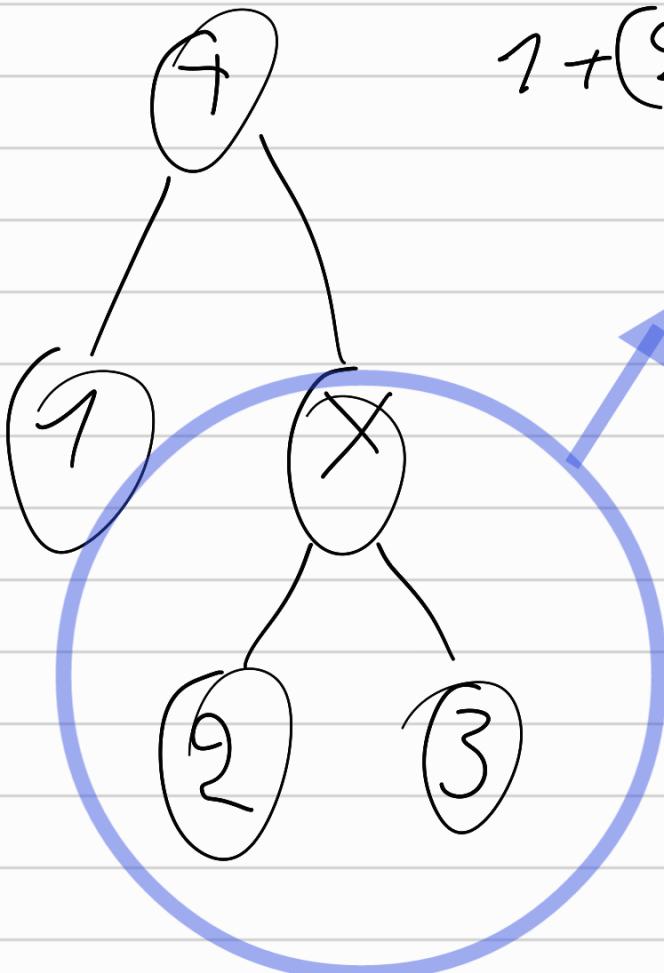
atome

$$1 + 2 \times 3$$

atome

- Constante
- variable ou identificateur
- gpc entre parenthèses

A partir du token courant q'a l'id
une constante (en identificateur
gpc entre parenthèse)



$$7 + (2 \times 3)$$

Parenthèse implicite

Monter jusqu'à avoir consommé tous les token qui compose l'atome

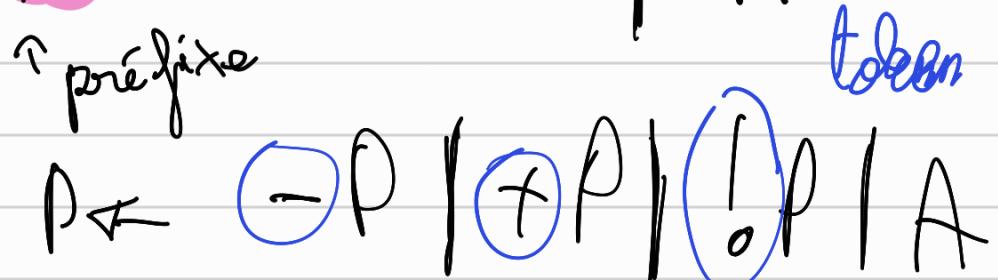
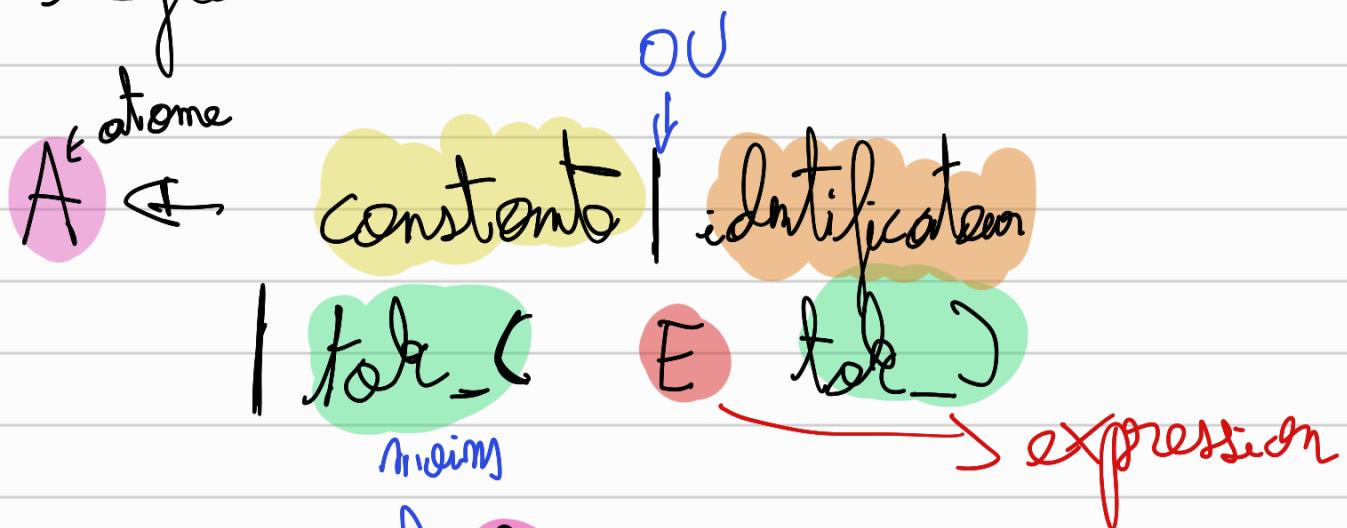
Last ← dernier token de l'atome

et en parallèle construire l'arbre après l'analyse

Atome renvoie un nœud

Qui est-ce que atome doit reconnaître ?

rigole :



E ← P

(Analyze recursive descendants)

Noeud A() → applies to check on a avoncon

if (check (tok_constant)) {

return Noeud ("Node_constant", last_value)

} else if (check (tok_id)) {

ErrorFatal ("Not gd") provisieer coor pas
de variable

} else if (check (tok_parentheseOpen)) {

N = E()

accept (tok_parentheseClose)
return N

} else {

ErrorFatal()

}

\ token corespond pas
élement d'un atome

Nœud P()

if (check(tok_moins)) {
 N = P()

} else if (check(tok_Exlam)) {
 N = P()

} else if (check(tok_Plus)) {
 N = P()

return N;

} else {

N = A()

return N.

le plus envoie
ne change rien
+ q équivalent à g

} return N()

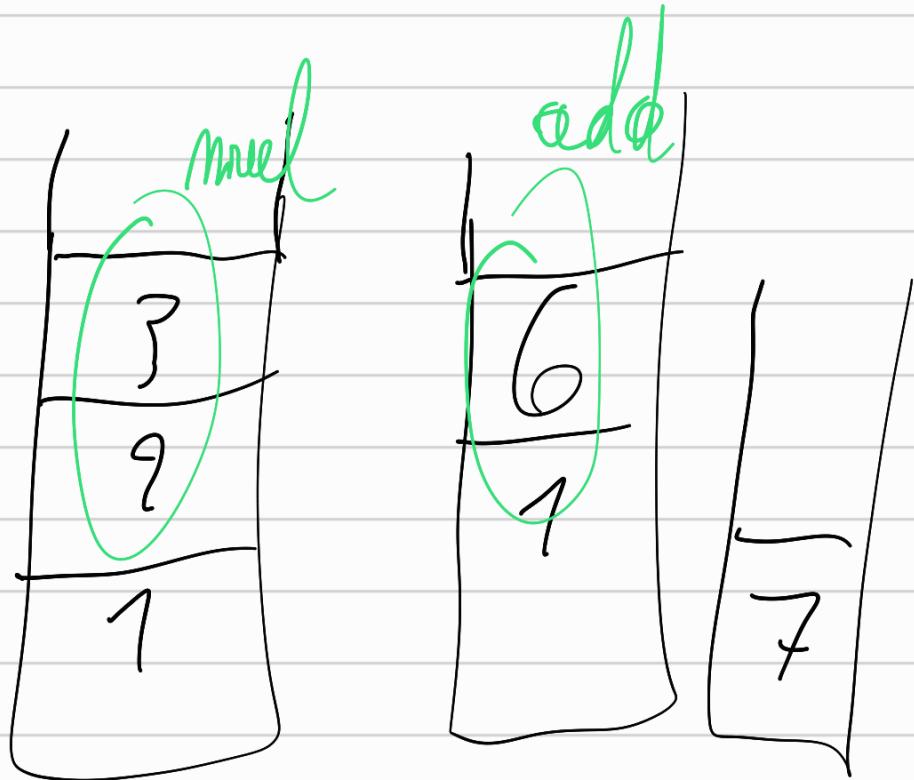
Noeud E()
return P()

Génération du code

Utilisation d'une pile

Ex: $1 + 2 \times 3$

push 1
push 2
push 3
mul
add



①

gener_code(N)



switch (No_type)

case Node_constant:

print ("push"; No_value)

case Node_not:

genecode (No_enfant[0])

print ("not")

va produire : ←
push 3

enlève 3 de la
pile et la
remplace par

la négation
logique càd 0

