

Compilation

(machine virtuel,
simulation)

Compilateur ≠ interpréteur



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

Ex: Code source → binaire

| ↴ autre langage
↳ Assembleur

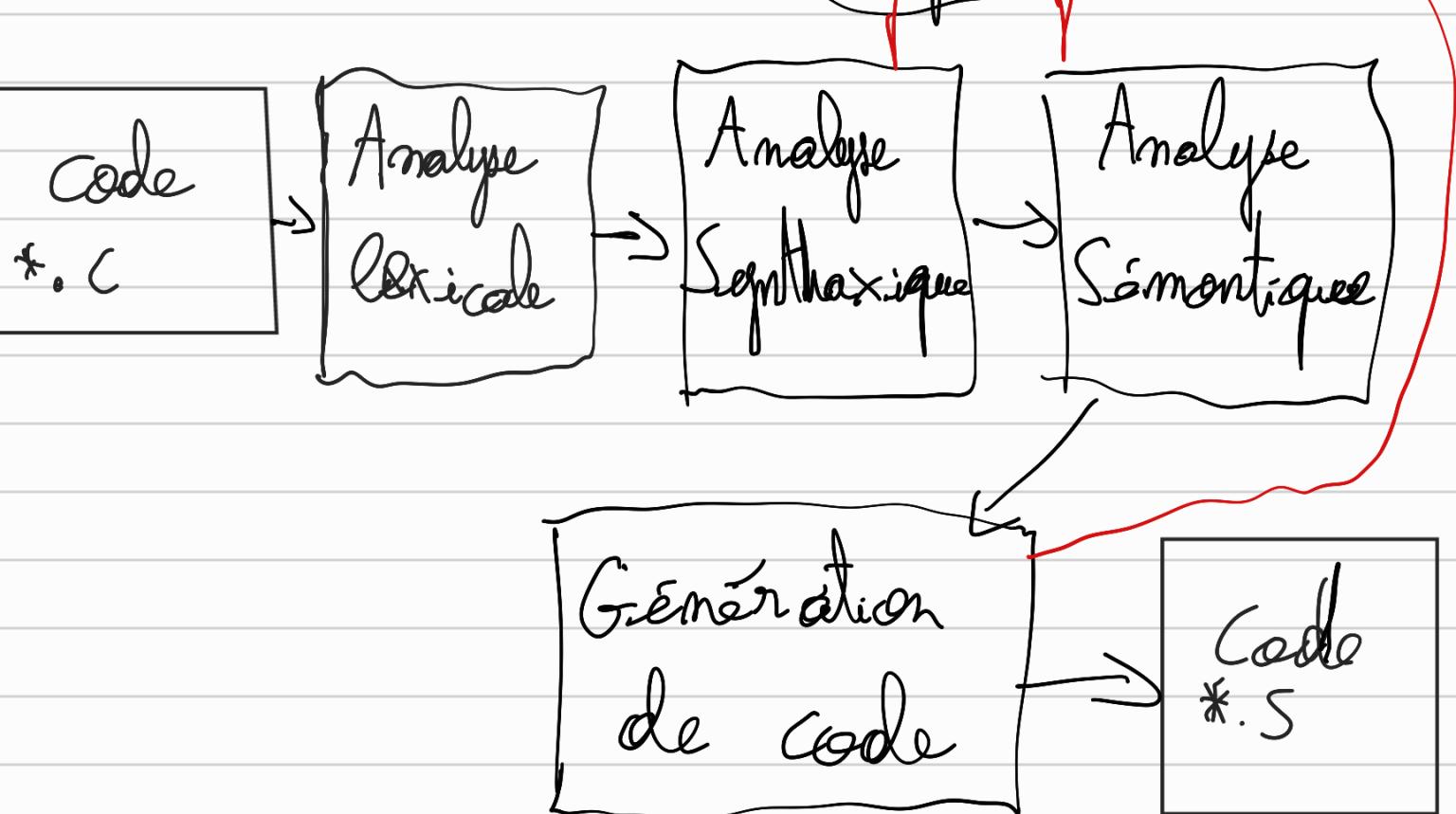
prend le code source (ex: python)
et exécute sa sémantique

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 :

Transforme le programme en token

Type
Value

ext:

Constante
entiere

Signe
moins

<p>Contento entière</p> <p>45</p>	<p>Signe étoile</p> <p>O</p>
---------------------------------------	----------------------------------

Identification
"abc"

↑
santisation ou négation ?
⇒ Analyse Synthétique

En C ces pointeurs sont des structures

Valeur prendra 2 types : entier ou chaîne

stuck taken {

1

int / string value"(

{

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 ajoutés dans les noeuds de l'arbre.

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

est syntaxiquement correct
mais pas sémantiquement

Dans notre schéma il manque 2

- optimisations :
- avant Code *⁵
 - avant Apres Analyse Sémantique

Analyse Lexicale :

2 choix

à la main

ou avec un automate

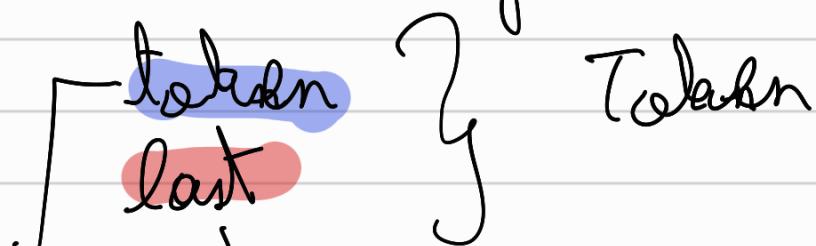
Bande et condition

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 un 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:

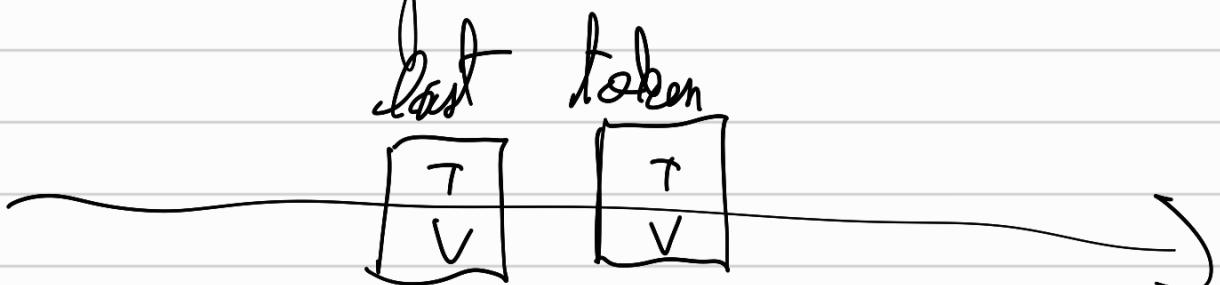
+ - * / % ! &
< = > = = !=

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

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

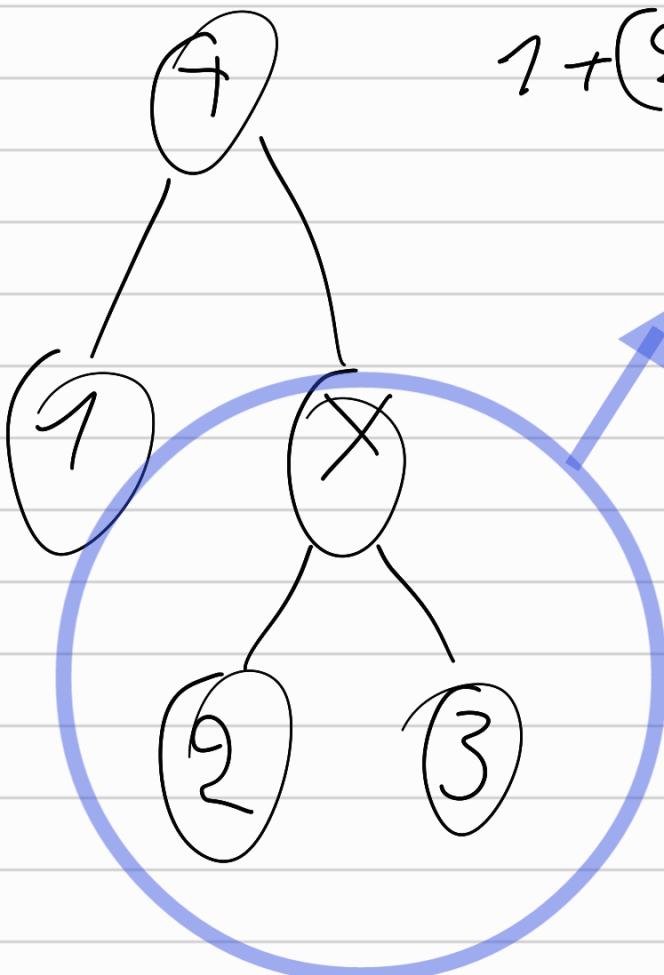
atome

$$1 + 2 \times 3$$

atome

- Constante
- variable ou identificateur
- gpc entre parenthèses

A partir du token courant q à l'if
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

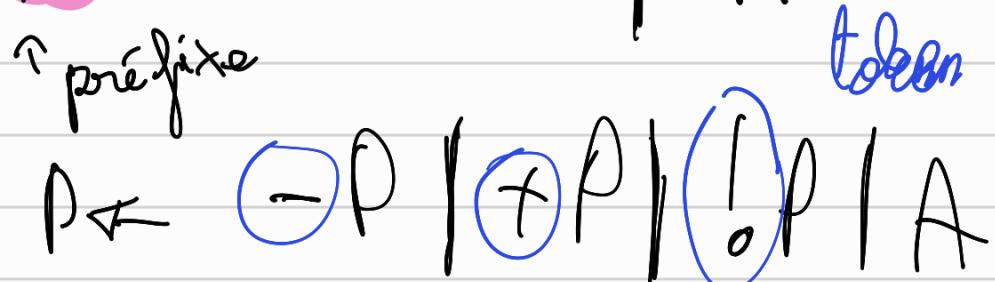
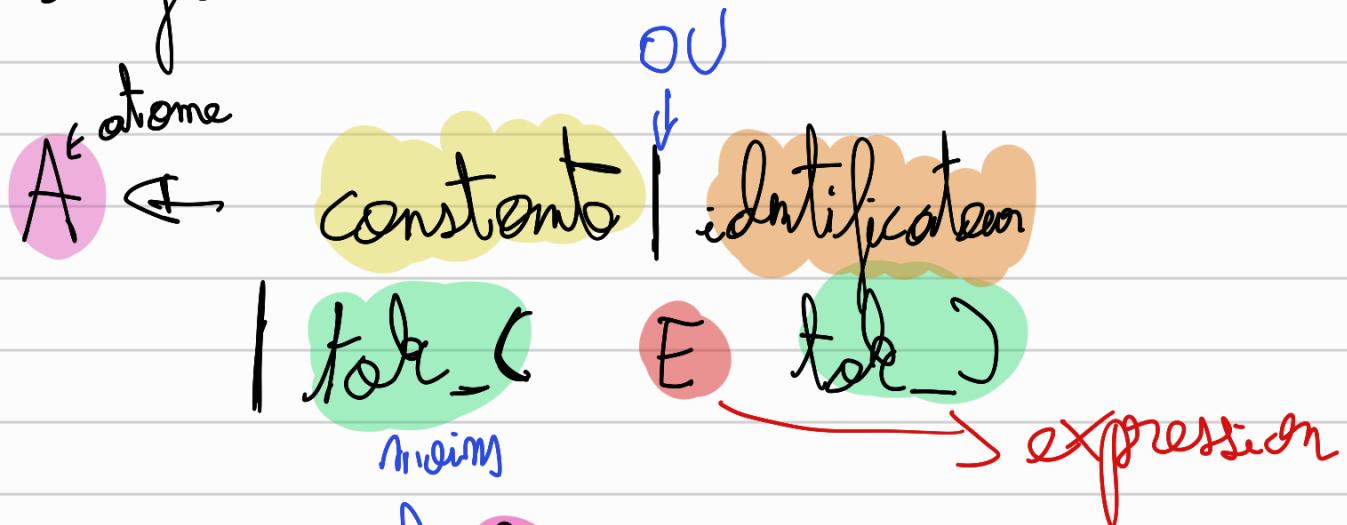
Last ← dernier token de l'atome

et en parallèle construire l'arbre après l'analyse

Atome renvoie un nœud

Qui est-ce que atome doit reconstruire ?

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 courant 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 en arre
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-cade(N)



switch (No_type)

case Node_constant:

print ("push"; No_valour)

case Node_mét:

geneCode (No_enfant[0])

print ("mét")

va produire : ←
push 3

enlève 3 de la pile et la remplace par

la négation logique c à d 0

Gestion des priorités ✓

Gestion de l'associativité :

$(1 - 2) - 3$

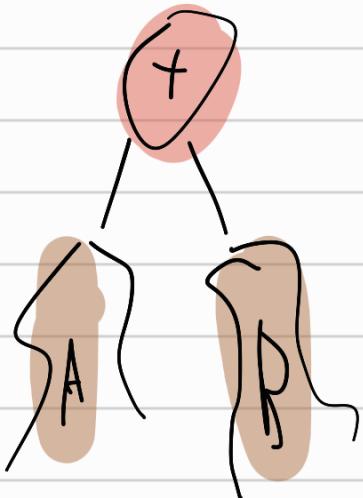
associativité à gauche

$2^1(3^14)$

associativité à droite

Opération d'affectation : associativité à droite

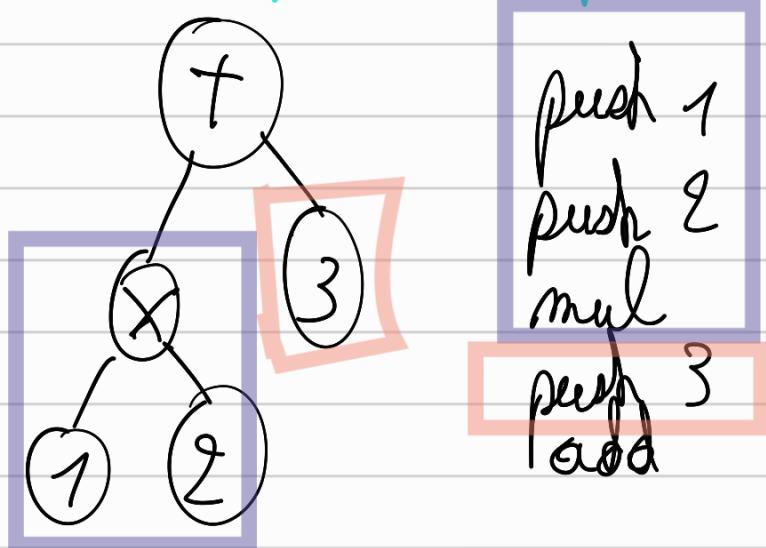
$a = [n = c]$ implique



Case node-plus:

genecode (token.enfant[0])
 genecode (token.enfant[1])
 print ("odd")

On fait un parcours d'arbre en profondeur



Parcours de Brat

Noeud E (Prio_min)

$$N = P()$$

→ priorité minimum des opérations

while (Operateur[token.T] != NULL) {

 Op = Operateur[token.T]

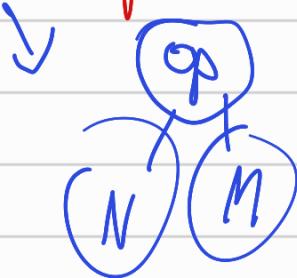
 if (Op == Prio_min) break;

Op. P next()

$$M = E \left(\text{Op}_0.P - \text{Op}_0.\text{Add} \right)$$

$$N = \text{nodeadd} \left(\text{Op_node}, N_1, N_2 \right)$$

} return N



Opérations

~~node_pless~~ $\rightarrow \{ \text{nde} = \text{node_pless}$
priorité de l'opérateur $\rightarrow P=1, \text{Add}=0 \}$

associative à droite
 $\begin{cases} = 1 \text{ si Vrai} \\ = 0 \text{ si Faux} \end{cases}$

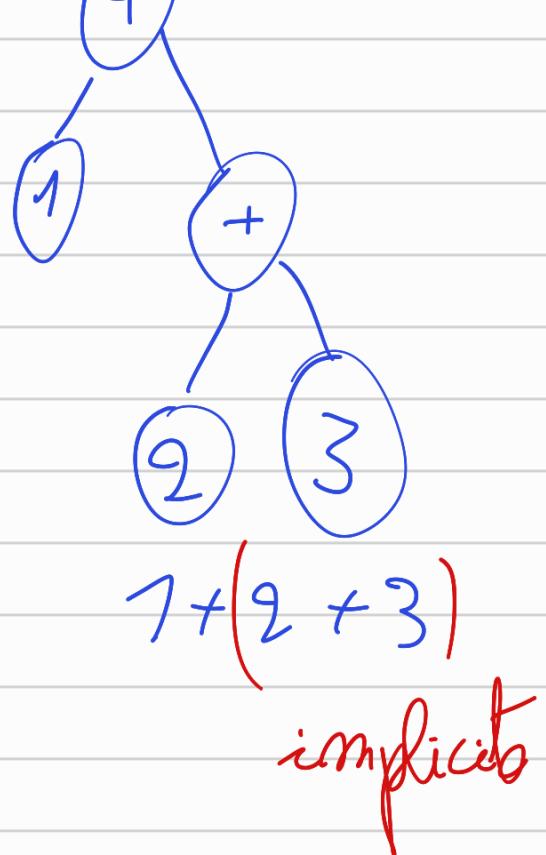
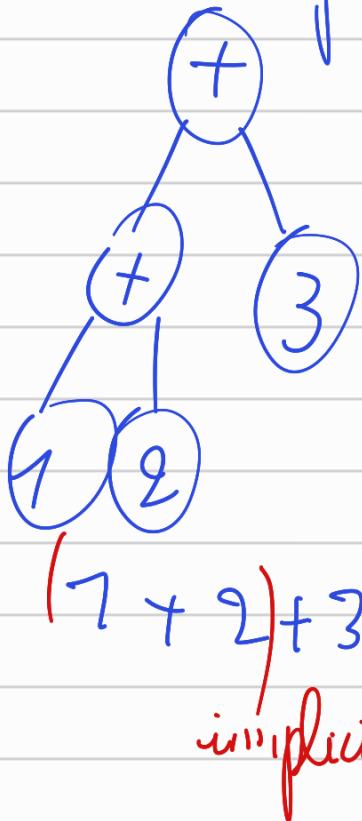
~~node_stoile~~ $\rightarrow \{ \text{nde} = \text{node_mult}$
 $, P=2, \text{Add}=0 \}$

Tableau de valeurs contenant une case pour ex

1 + 2 + 3

associatif à gauche

associatif à droite



Dans table Opérateur on va fixer
les priorités :

token	priorité	Associatif à droite
$=$	1	1
$/ \backslash$	2	0
$\& \wedge$	3	0
$== ! . =$	4	0
$< >$	5	0
$+ -$	6	0
$* / \%$	7	0

Architecture du processeur :

pose.limsi.fr/lavergne : accès processus
trouver en compilateur C

Programme : msn

on donne le code en entrée et exécute
le code à partir de start

Instruction dbg (debug)

enlève la val au sommet de la pile
et l'affiche sur l'écran

• start

push 1

push 2

add

dbg

halt

} afficher 3

msn -d

affichage séquentiel de

toutes les instructions

mm - d - d → affichage de ce qu'il a
à sur la pile à l'env.
exécuter

mm . txt : des de l'assemblage
de la machine

Optimisation à fenêtre (9 lignes à 2 lignes)

[push □] → []
drop 1 → suppression du 1^{er} élément au sommet pile

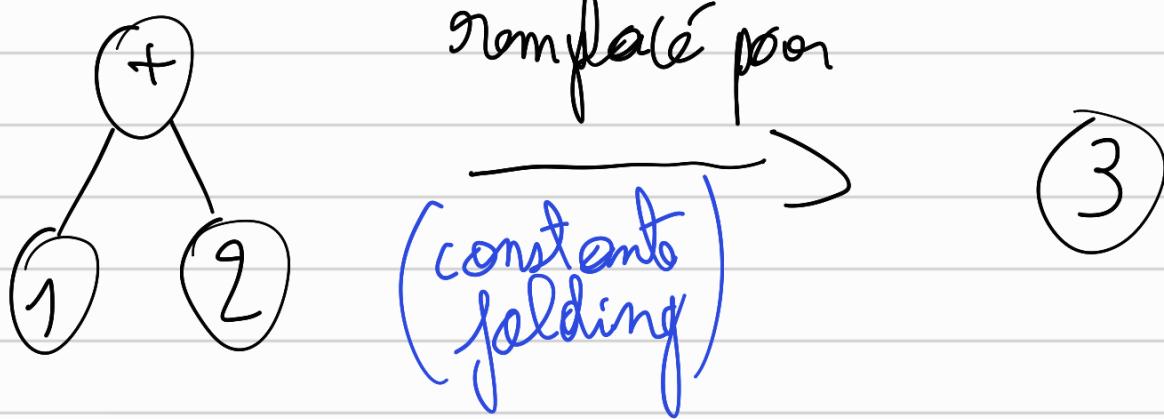
On évite le code inutile créer
par le compilateur

[push □] → [drop n - 1]
drop n

But : créer du code correct peut
nécessiter des choses inefficaces qui
sont corrigable en créant de
patron.

Optimisation mm . txt :

Optimisation par arbre.

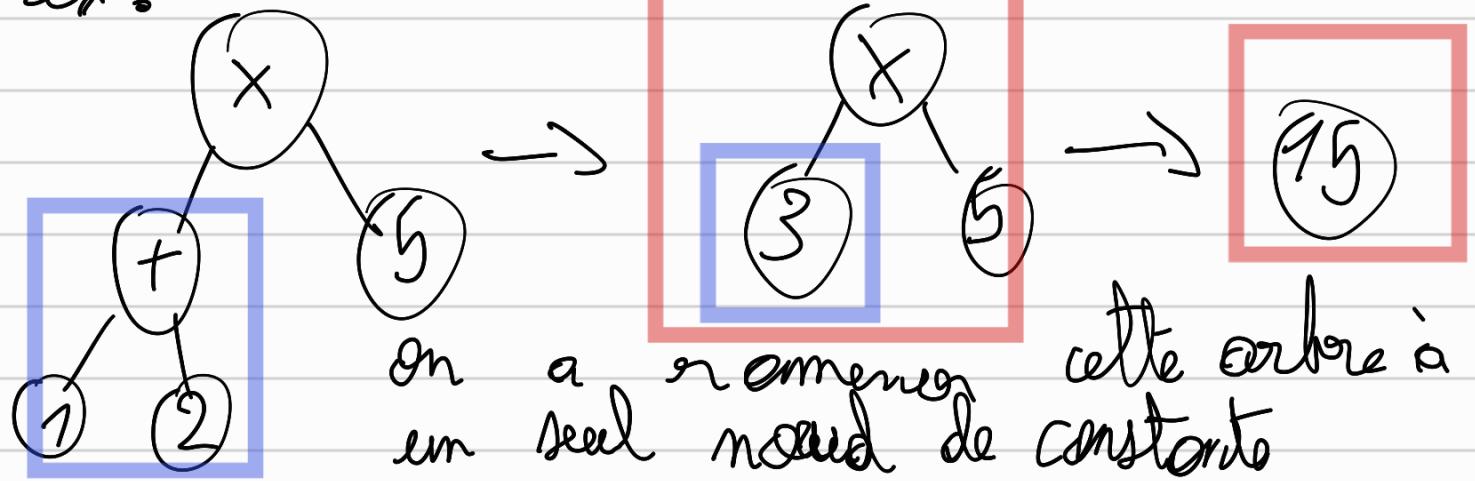


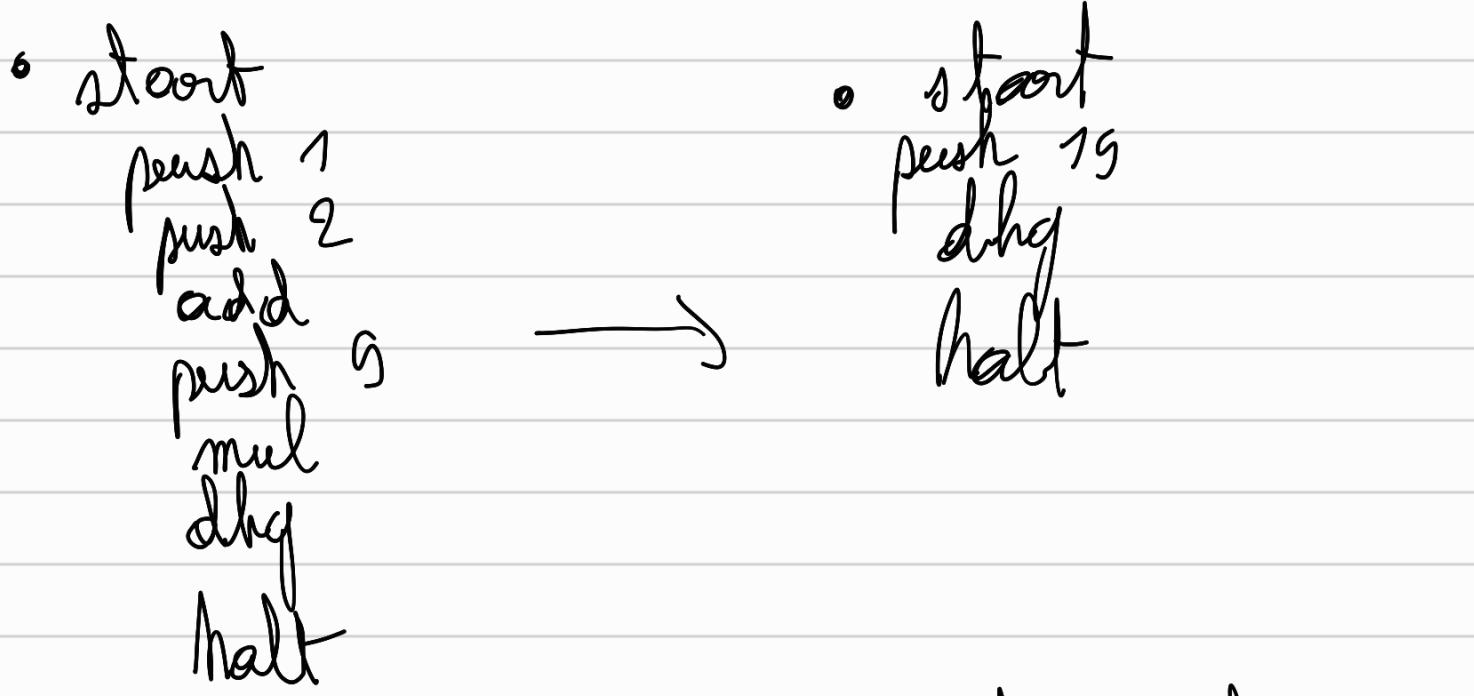
appel d'une fn peut être + car les que d'exécution sont code ex: max

Donc exécu° de sont arbre syntaxique et on remplace ces arguments par les val de l'appel de la fonction

Optimiseur parcours arbre en s'appelant récursivement sur les enfants

ex:





On traite les nœuds enfants avant les nœuds parents.

Optimisation facultative pour le rendu

