

# Compilation

(machine virtuel  
, simulation)

Compilateur ≠ interpréteur



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

Ex: Code source → binaire

| ↴ autre langage  
↳ Assembleur

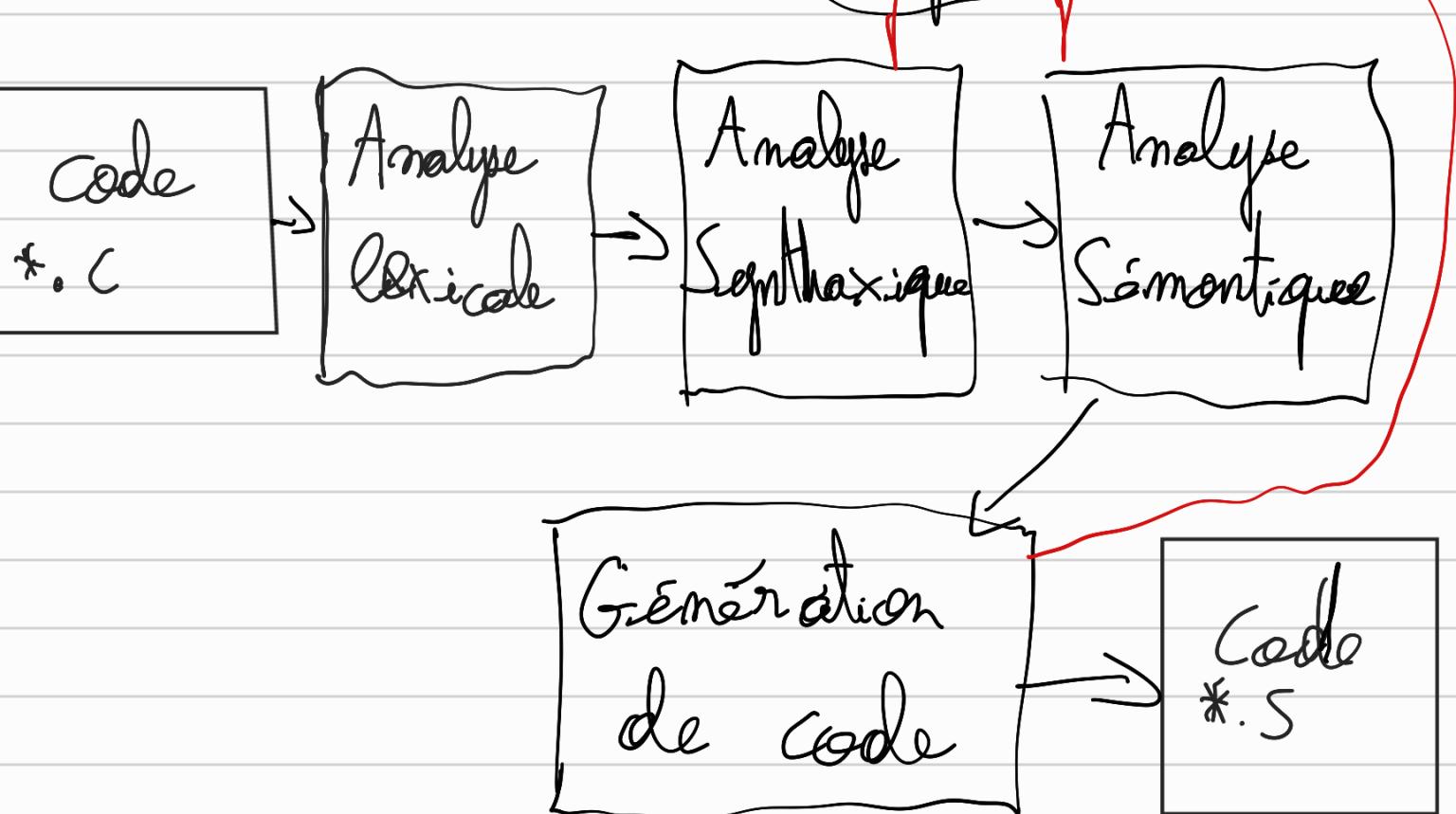
prend le code source (ex: python)  
et exécute sa sémantique

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

Contento entière	Signé étoile
45	O

Identification  
"abc"

↑  
santé nation ou négation ?  
⇒ Analyse Synthétique

En C ces pointeurs sont des structures

```
"java"    ''    ''    "objets"
"python"  ()    ()    "dictionnaire"
```

Valueur prendra 2 types : entier ou chaîne

stuck taken {

int / string values

{

1) en c une énuméra<sup>t</sup> de tt type  
qui permet de savoir le type des  
tokens

(enfer une phrase en mat  
( séquence de token)

## Analyse Syntaxique

Transfomme ces séquences de token en arbre  
syntaxique. Vérifica<sup>t</sup> de la  
syntaxe du code.

Appl de la fd n'est en branche  
générai<sup>t</sup> de code

## Production de code assembleur

## Analyse Sémantique

Vérifica<sup>t</sup> 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

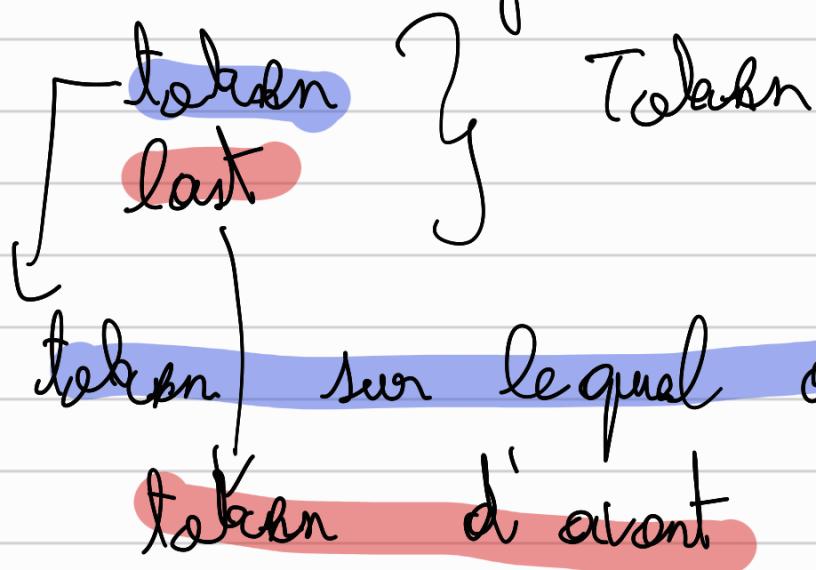
Bande et condition

Fichier à part dans un autre langage

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

# l'analyse Syntaxique

## 2 variables globale dans le code



ex : 1 (+) 2 \* abc  
*last token*

accès à ces 2 variables en lecture

for function next()

Last = token ;

taken  $\leftarrow$  lecture

# Lectura de telcos | serviront

les fonctions :

bad, check( T ) {

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

```
    next();  
    return true; }  
else {  
    return false; }
```

```
}
```

```
accept (T) {
```

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

ErreurFatal();

```
}
```

Possibilité de récupérer l'erreur

liste des tokens :

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

Exemple :  $[a-zA-Z][a-zA-Z_0-9]^*$

- 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:

+ - \* / % ! &

et pour chaque comparaison

< <= > >= == !=

Si prochain caractère constant sur =

& &

||

( )

[ ]

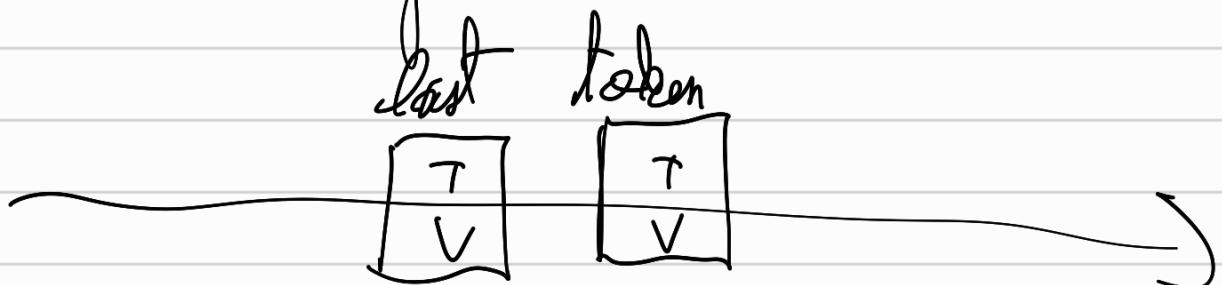
{ }

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

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

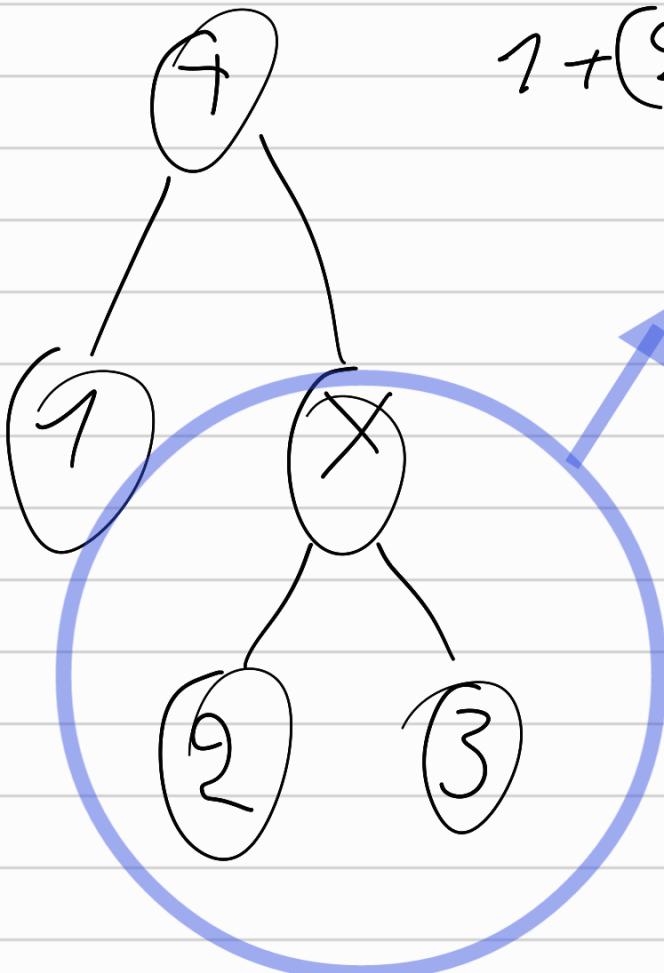
atome

$$1 + 2 \times 3$$

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

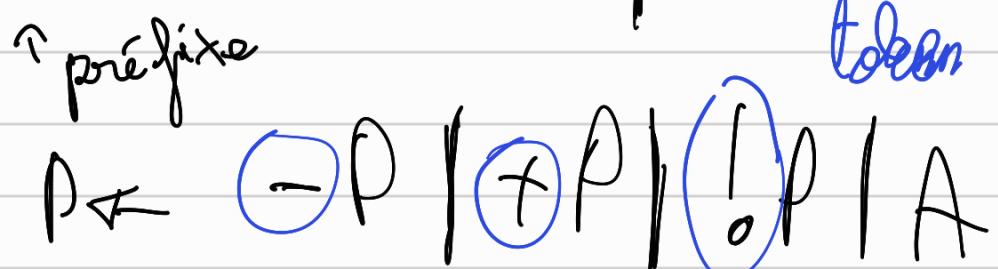
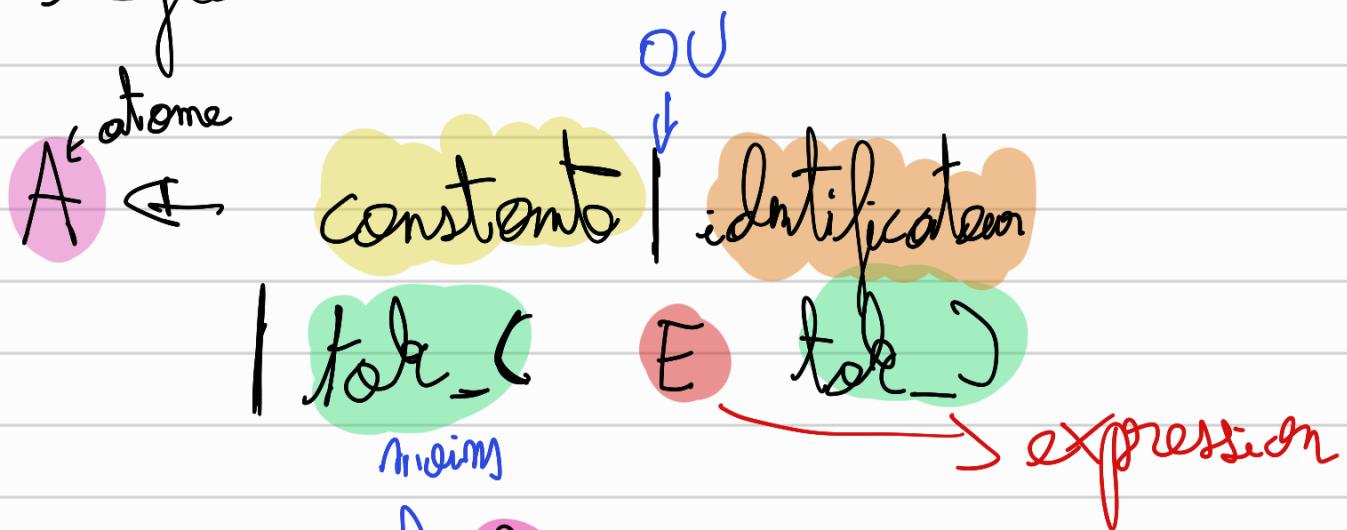
Last ← dernier token de l'atome

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 coenant 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;

    } le plus envoie  
    me change rien  
    + q équivalent à g

} else {

    N = A()

return N.

} return N

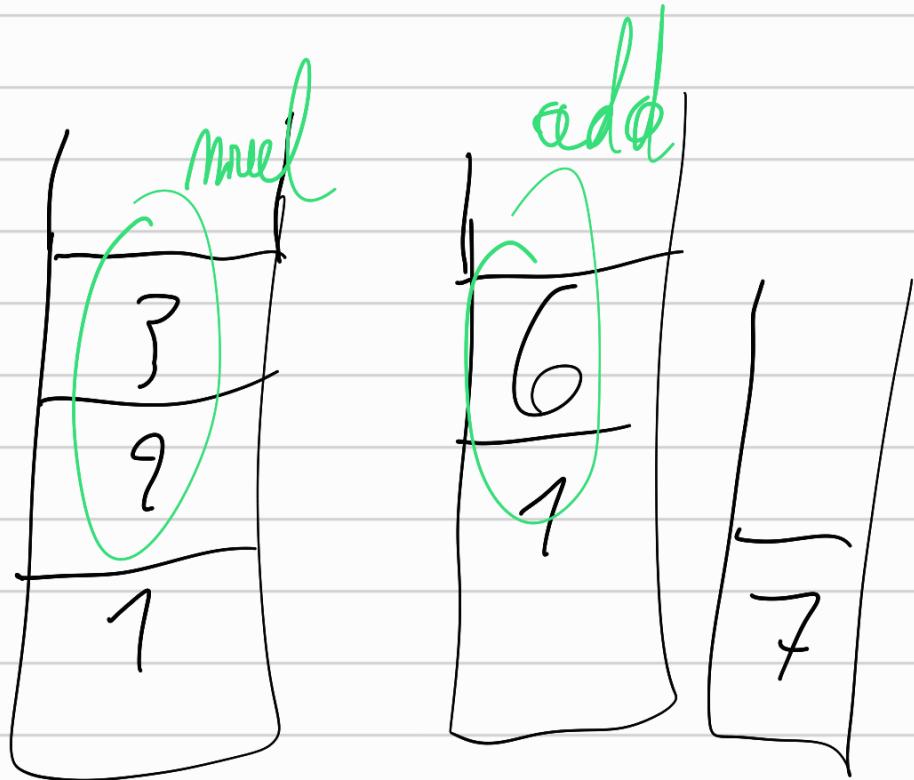
Noeud E()  
return P()

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\_meth:

genecode (No\_enfant[0])

print ("meth")

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 :

$(1 - 2) - 3$

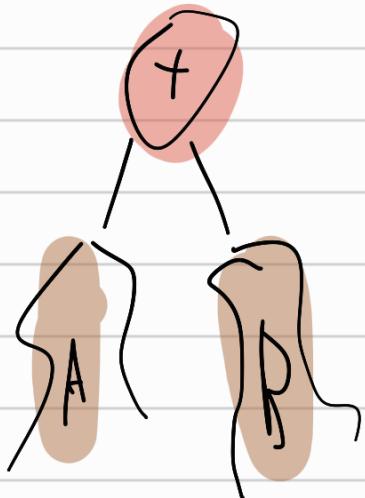
associativité à gauche

$2^1(3^14)$

associativité à droite

Opération d'affectation : associativité à droite

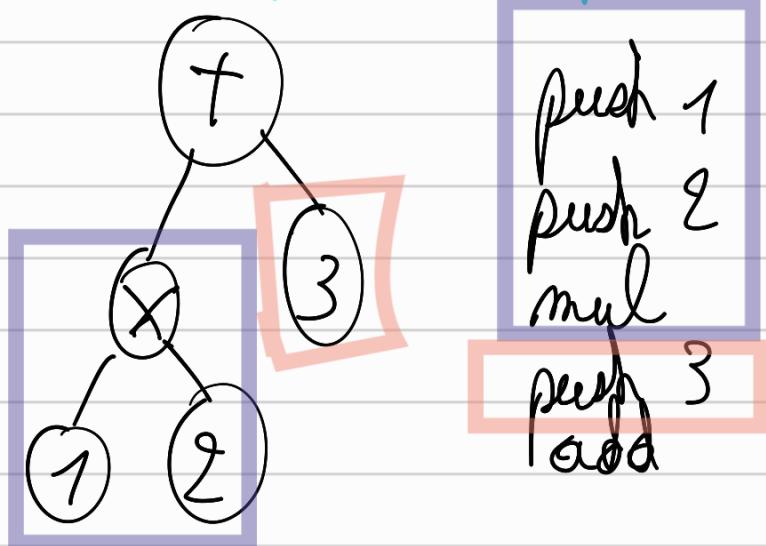
$a = [n = c]$  implique



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]

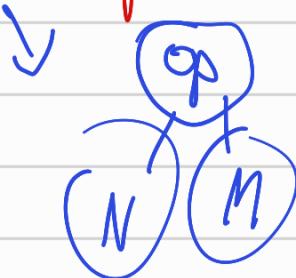
    if ( Op == Prio\_min ) break;

Op. P next()

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

$$N = \text{nodeid} \left( \text{Op\_node}, N_1, N_2 \right)$$

} return N



## Opérations

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

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

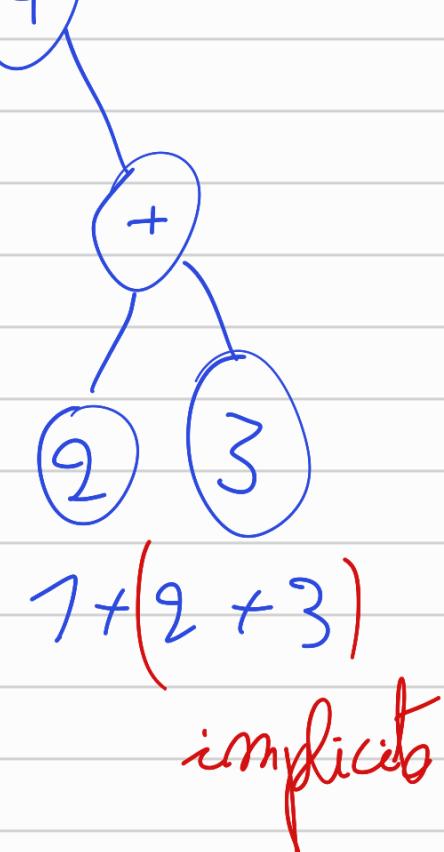
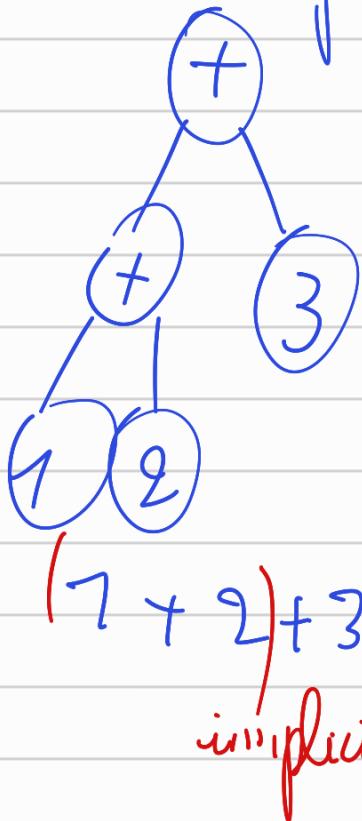
~~node\_stoile~~  $\rightarrow \{ \text{nde} = \text{node\_mult}$   
 $, P=2, \text{Add}=0 \}$

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 : mzm

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

Instruction debug (debug)

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

• start

push 1  
push 2

add

debug

halt

} afficher 3

mzm -d

affichage séquentiel de

toutes les instructions

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

mm . txt : des de tt les instructions  
de la machine

Optimisation à fenêtre (9 lignes à 2 lignes)

[ push □ ] → [ ]  
drop 1 → suppression du 1<sup>er</sup> élément au sommet pile

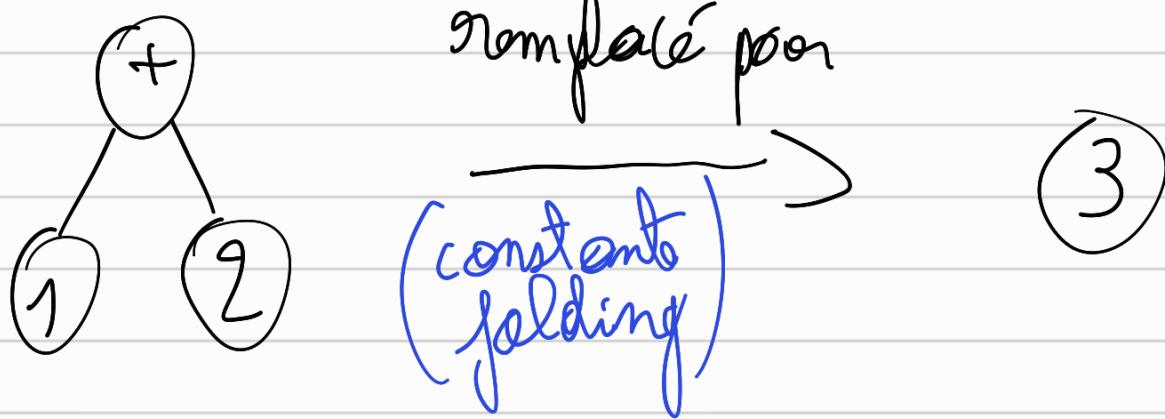
On évite le code inutile créer  
par le compilateur

[ push □ ] → [ drop n - 1 ]  
drop m

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

Optimisation mm . txt :

Optimisation par arbre.

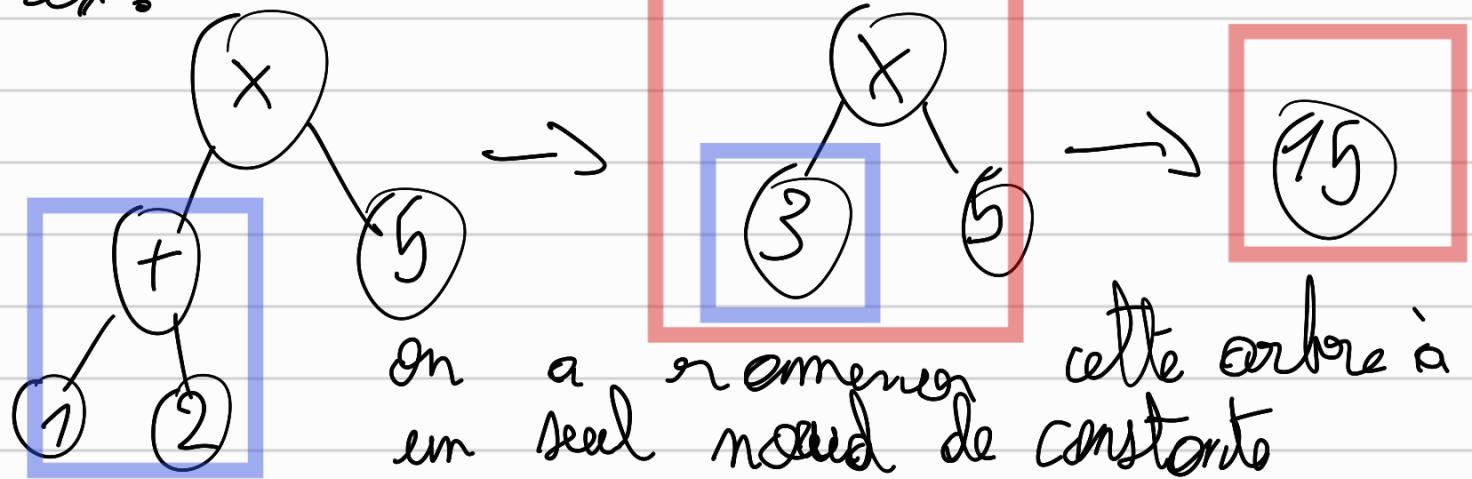


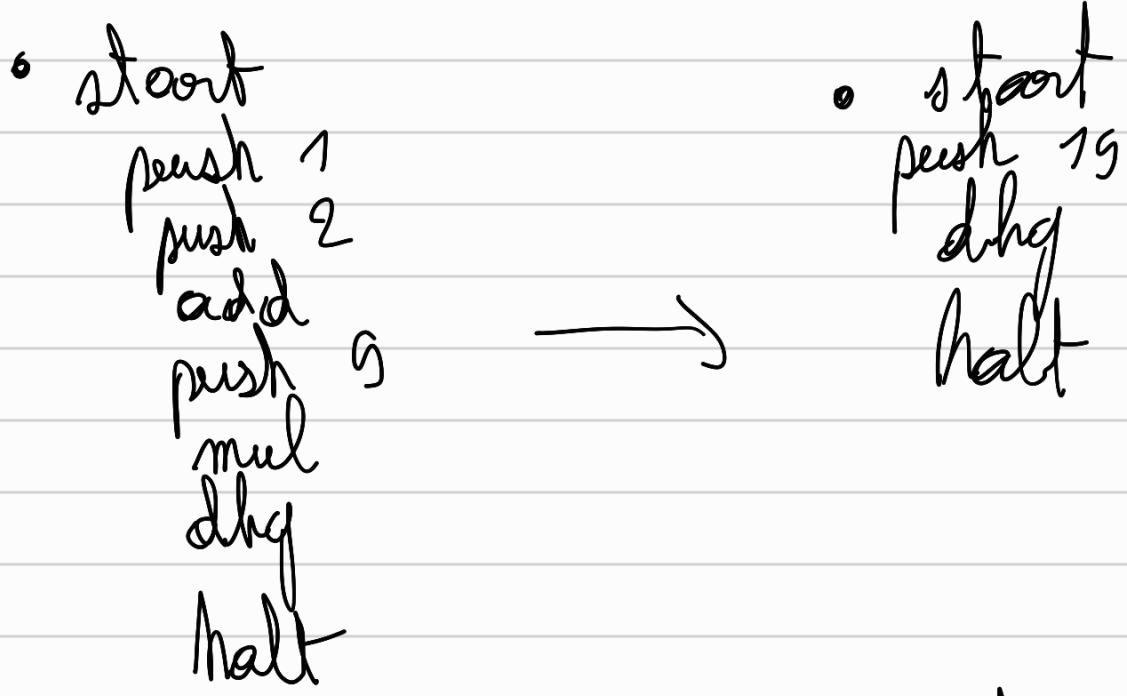
appel d'une fn peut être + car les que d'exécution sont code ex: max

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

- Génor Analyse
- tableau de token
- next recevra le token en cours et le token passer en traversant le tableau

Les variables :

int i, t, s, l, et mons

instruction  $\neq$  expression

ex:

$$\left. \begin{array}{l} 7 + 2 \\ a = 1 + 2 \end{array} \right\} \text{expression}$$

$$a = 1 + 2 \quad \left. \begin{array}{l} \text{Instruction} \\ \text{if} \end{array} \right\}$$

Production  
d'une valeur

L'expression se présente  
en calcul, mais n'est pas  
présente sur la pile

if(  $a < 5$  )  
     $b = 7$  ;

La ④ simple instruction est ;

$I \leftarrow ";"$

| E ;

| I "{" I \* "}"

| de la forme E ;

Tant que ça fait une  
continuité

if( E ) I

pour tester malgré

l'apparition des instructions

Fonction I / | S token

```

if( check(";") ) {
    return Noend / NdeVide
}
} elif( check("{") ) {
    N = Noend( NdBlock )
}

```

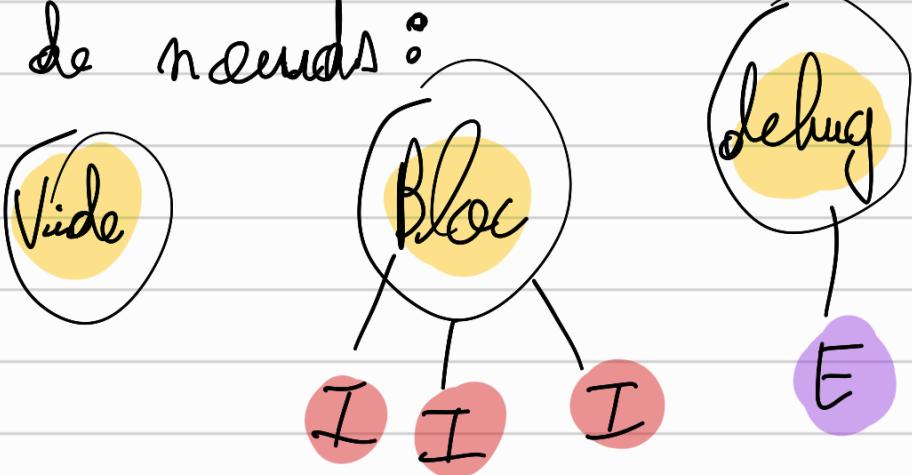
↑ in Vide  
on deit  
gronkogen  
en. noend

```

while( !check("}") )
    N, afdelen Eafant, I()
return N

```

Nieuwean type de noends:



```

elif( check( tok_debug ) )

```

$N = E[0]$

accept( "0" )

return

Noend( NdDebug, N )

Paschien taken  
dit extra ?  
simon correct

$\{ \text{else} \}$      $\leftarrow$  le cas  $E_i^*$

$N = E(\emptyset)^*$ ,

$\text{accept } (";" ) ;$

$\text{return } N \text{seed}(N \text{drop}(N))$

Gene code de l'arbre I ne doit rien laisser sur la pile.

abn expression  $\neq$  abn instruction

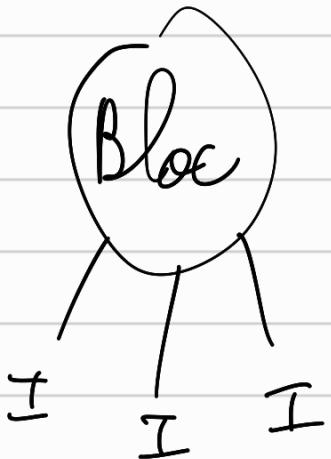


Ce mecanisme permet de differencier les instructions des expressions car on enverra de la pile la valeur mise par l'expression de celle instruction.

Conservier l'arbre mais pas sa valeur de la pile

Gene code des mecanismes

Pas de genicode pour le nœud vide



: Banche :

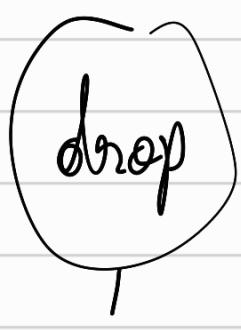
Pour chaque enfant  $N$ :

Genicode( $N$ )



: Genicode(Enfant[0])

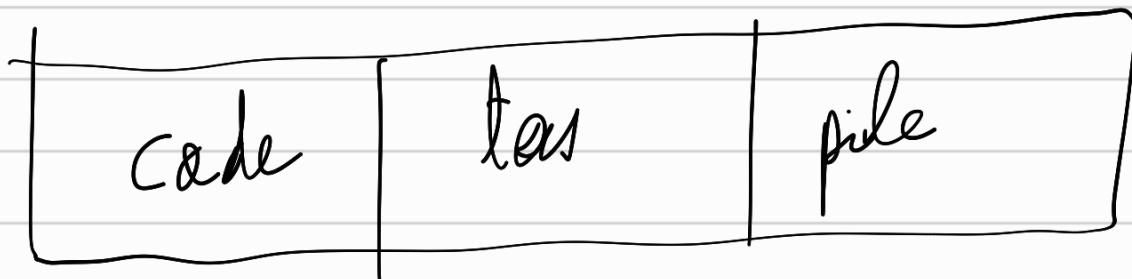
print("dby") → enlève la  
valeur de l'  
expression sur la pile



: Genicode(Enfant[0])

print('drop 1') ← pareil

Schéma mémoire :



Les Blocs délimitent la portée des

# Variables.

$$\left\{ \begin{array}{l} 2+1 \\ 1 \end{array} \right.$$

int a;  
declara la variable

{ int hi;

$$b = 5^{\circ}$$

$$\text{int } \alpha^i$$

a mon trouvée dans le bloc, donc on va dans le bloc supérieur

3

# Table des symboles (4 fonctions)

Sie lösen die Deklaration

Se declarar (non) → se le <sup>u</sup> hace actual

L Erogen

Sr Chorchor (mom)

Simon grenzte im  
September

↳ Renvoie du symbole associé à la

Variable nom

Begin() } est-ce qui est un nouveau bloc  
End() } commence où se termine ?

Association nom : valeur  $\Rightarrow$  Dictionnaire en python

Une variable ne peut être déclarer qu'une fois par bloc.

Déclarer (nom)

si nom exist in

VariTab

Erreur

S = nouveauSymbbole

VariTab[nom] = S

return S

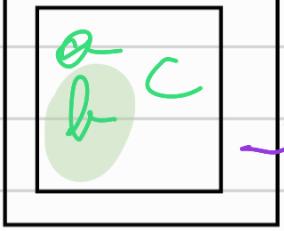
pile de  
table de  
Hachage



Pile

, End()

qd bloc fini on enlève  
ette table de la pile



Une table pour Bloc

Begin()  $\rightarrow$  Voor-push(Ø)

End  $\rightarrow$  Voor = pop()

Chercher( $\alpha$ ): variable qui est actuellement  
 $\alpha$ , donc dans bloc le  $\oplus$  haut

Mais si on cherche( $\beta$ ), alors il faut  
aller dans le bloc suivant

Si Jamais TzV  $\Rightarrow$  ErreurFatal()

Chercher(nom)

Pour chaque Tab T de haut en bas

dans Voor faire {

Si nom exist dans T {

return T[nom]

}

$\hookrightarrow$  retour du symbole

de nom

## Erreur Fatal ("Von non déclaré")

Pile doit accéder à tous les éléments jusqu'en bas, pas qu'en haut sommet

Si langage empêche

- recreer sa propre structure de pile

- 2 pile, une pr sauvegarde, une pr parcours avec pop\_stack du sommet à chaque fois

Ou utilise seulement d'une 2<sup>e</sup> pile

Chercher (nom)

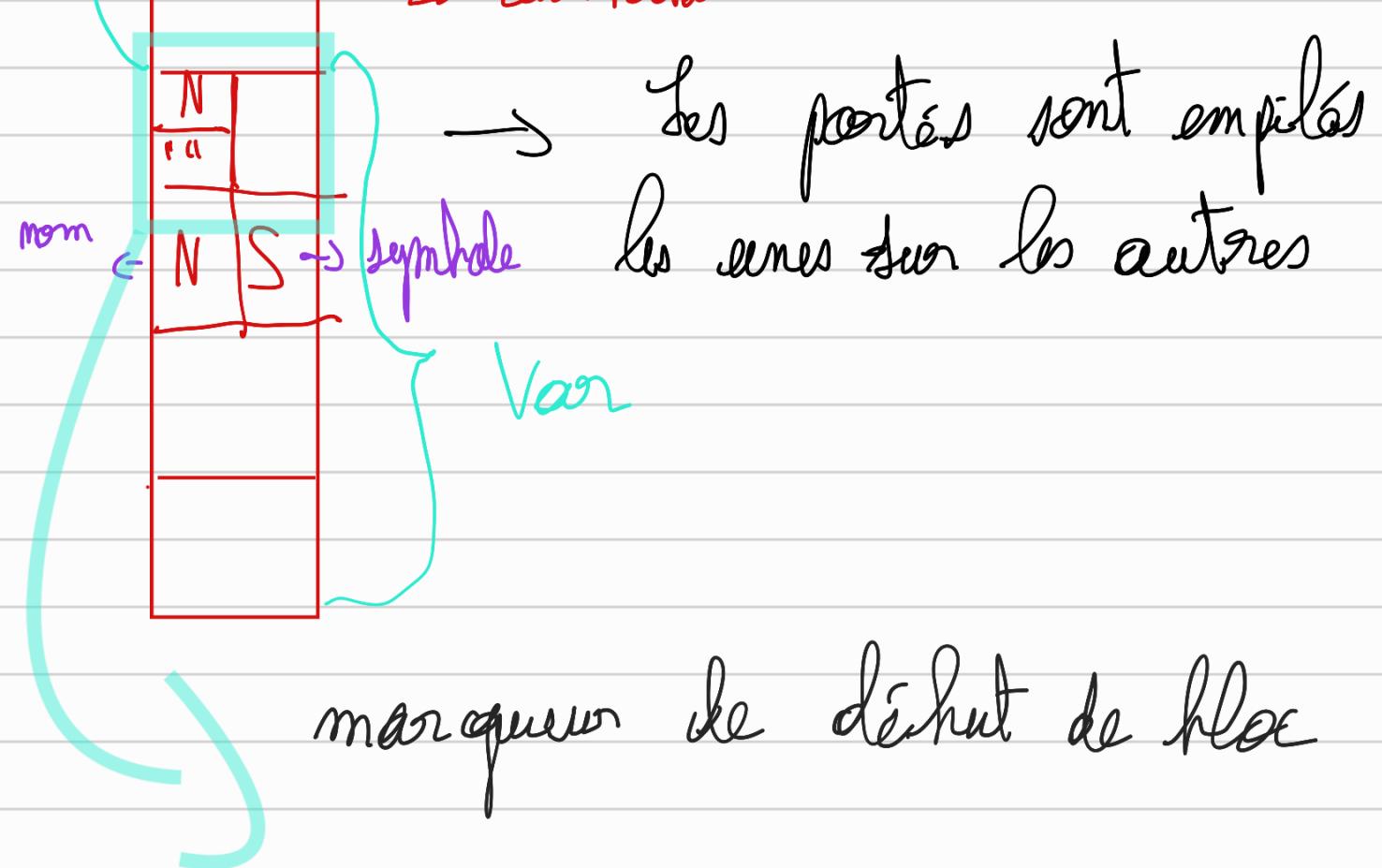
Pour  $i = \text{Sommet jusqu'à } 0$

| Si  $T[i].N = \text{nom}$

| | return  $T[i].S$

Erreur Fatal

Von



DeJearon (mom)

Pour  $i = \text{Sommet}$  jusqu'à 0

Sie  $T[i]_0N = \text{nom}$

# Emerson Fatal

Sie  $T[i], N =$

break ^

$\Sigma$  = Neueren Symbole

T<sub>i</sub>.push-back(Nom, S)

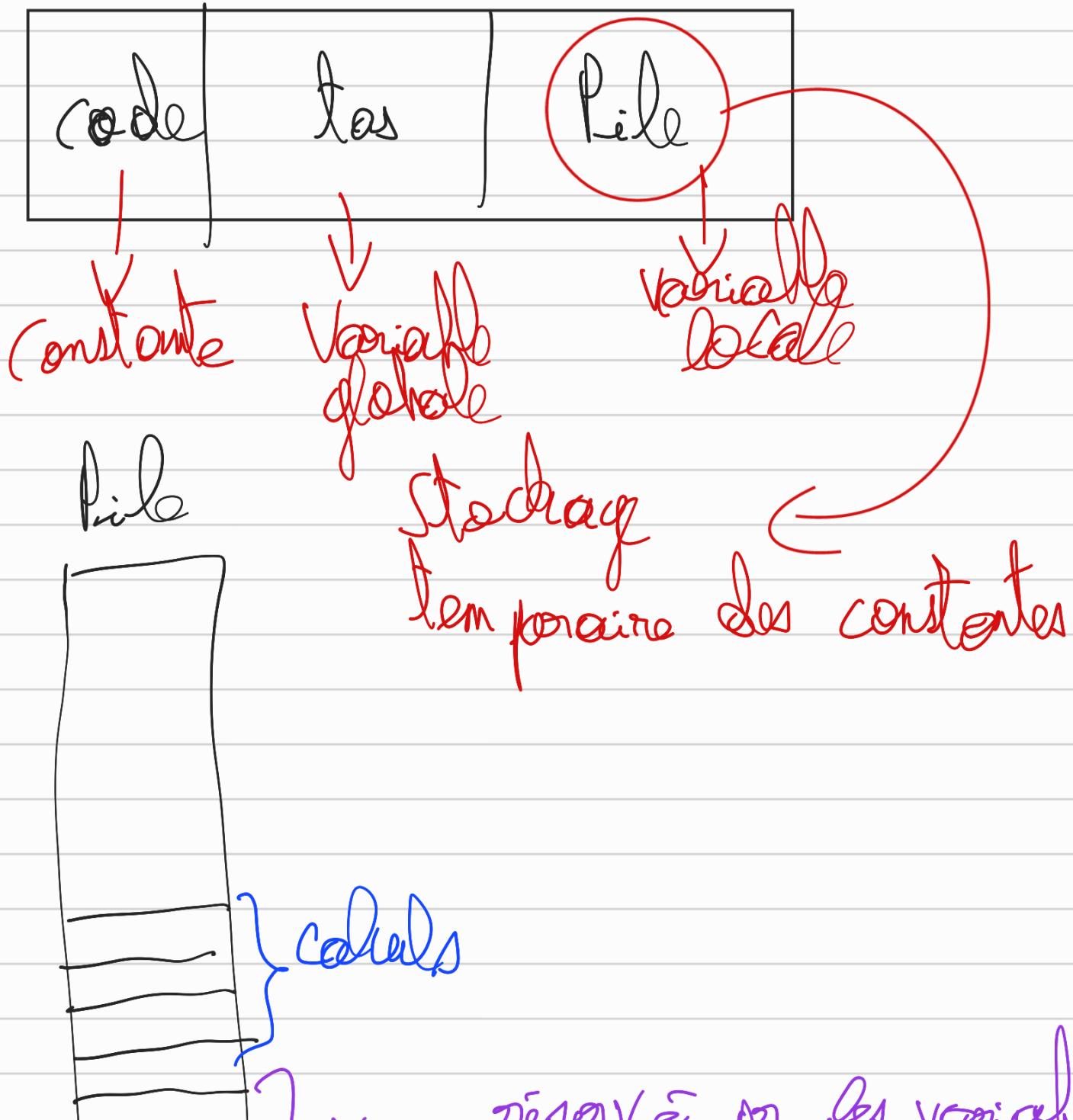
return } .

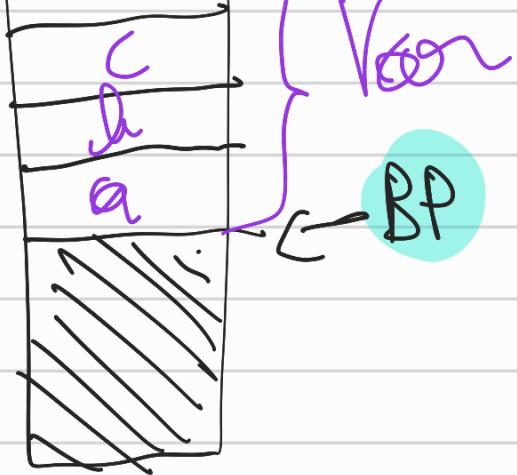
Begin() => T.push( , )

End() => T.pop Jusqu'à "

Cette structure est lors (+) utilisée  
par les compilateurs

En python les symboles seront des noms  
pour le moment.



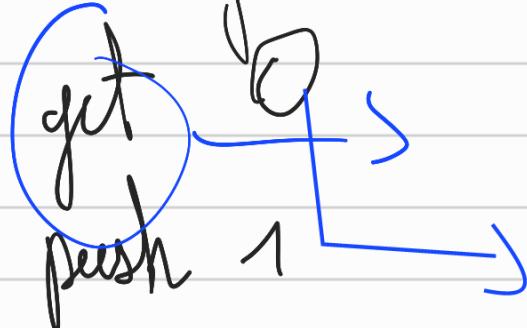


On met les variables sous le calcul,  
pour éviter que elles soient écrasées  
par les calculs

Il faut donc réservé suffisamment de  
place pour toutes les variables local  
du programme.

Instruction `get` van chercher une  
valeur dans la pile et la mettre au  
dessus

Pour faire :  $a + 1$



comment get trouver  $a$ ?

l'indice de  $a$  dans la  
pile à partir de BP

add

Pour faire :  $b = 3$

push 3

Set 1

Pr t<sub>ir</sub> et indice =>

AmaSeq:

A()

Analyse Sémantique

    } else if (check ( tok\_ident )) {  
        return Need ( NodeRef , last , value )

    }



I () {

    int a , b , c ;

    ref  
    "ac"

    } else if (check ( tok\_ident )) {  
        N = Need ( Node\_Seq ) ;  
        do {

accept( too, ident );

No ajout or Enfant (Noed( No\_décl  
, last\_value))

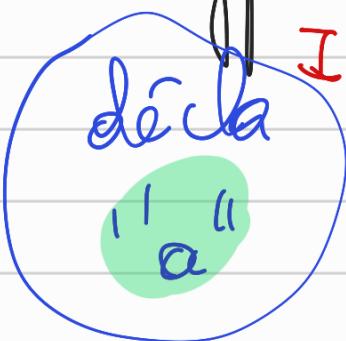
} while( check(' ') );

accept( ";" )

return V;



3 Meilleur type de noeud:  
Seq I declar "a" ref "a" =  
nom de variable expectation



nom de variable

Expression : laisse une valeur sur la pile  
noeud declar => Pas de Générateur

# utile pour analyse démontique

AmaSem :

- compter nb variables
- pr chaque attribut case
- pr chaque ref place des nœuds  
l'info de la place of la cas  
de la var

AmaSem (Nœud N)

switch (N. type) {

départ :

Pour chaque enfant E :

AmaSem (E)

Cose Nœ-Block :

Begin();

Gestion de la  
sortie

Pour chaque enfant E :

A Sem(E)

End();

Case Node\_Declar:

Symbol

$S = \text{declare}(\text{No\_value}) ;$

So position = mb Voor;

mb van ++

nombre de variable

$\Rightarrow$  mb cas & réservé

local

$S = \text{VorLoc};$

globale

fonction

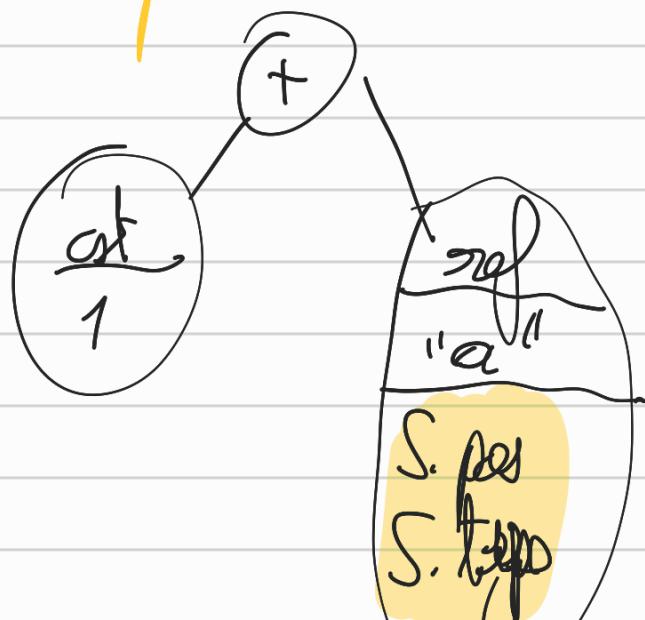
Case Node\_Ref:

$S = \text{declaration}(\text{No\_value}) ;$

No\_symbol = S;

on met le symbole dans le nœud

1+a :



GeneCode()

...  
case NodeDecl:

break;

Case Node\_Block, case Node\_Seq:

GeneCode sur leur enfant

Case Node\_Ref:

if (N. symbol-type == VarLoc){

print ("get", N. symbol-position)

? else {

ErrorFatal(); }

Case Node\_Affection:

GeneCode (N. enfant[1]);

print ("deep");

if (N. enfant[0]. type == Node\_Ref)

ErrorFatal();

if (N. enfant[0]. type == VarLoc)



```
    print ("set", N.E[0].S.position);  
else  
    ErreurFatal();
```

dep: duplique la valeur au-dessus  
de la pile

a = 0<sup>i</sup><sub>1</sub> →

Comme c'est une  
instruction, il  
y a un drop

push Ⓛ  
dup  
set a  
drop 1

on fait dep  
pour garder la  
valeur de la  
variable elle  
même au-dessus de la  
pile

Compile()

while (token.type != EOF)

N = Anal-Sgn()

global ←

nbVar = 0<sup>i</sup><sub>1</sub>  
AnalSem (N)

print ("new", nbVar)

réservation  
de case  
mémoire

Genecode(N)

print("drop", movez)

l'héritage de la Mémoire après  
utilisation.

