

Compilation

(machine virtuel,
simulation)

Compilateur ≠ interpréteur



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

Ex: Code source → binaire

| ↴ autre langage
↳ Assembleur

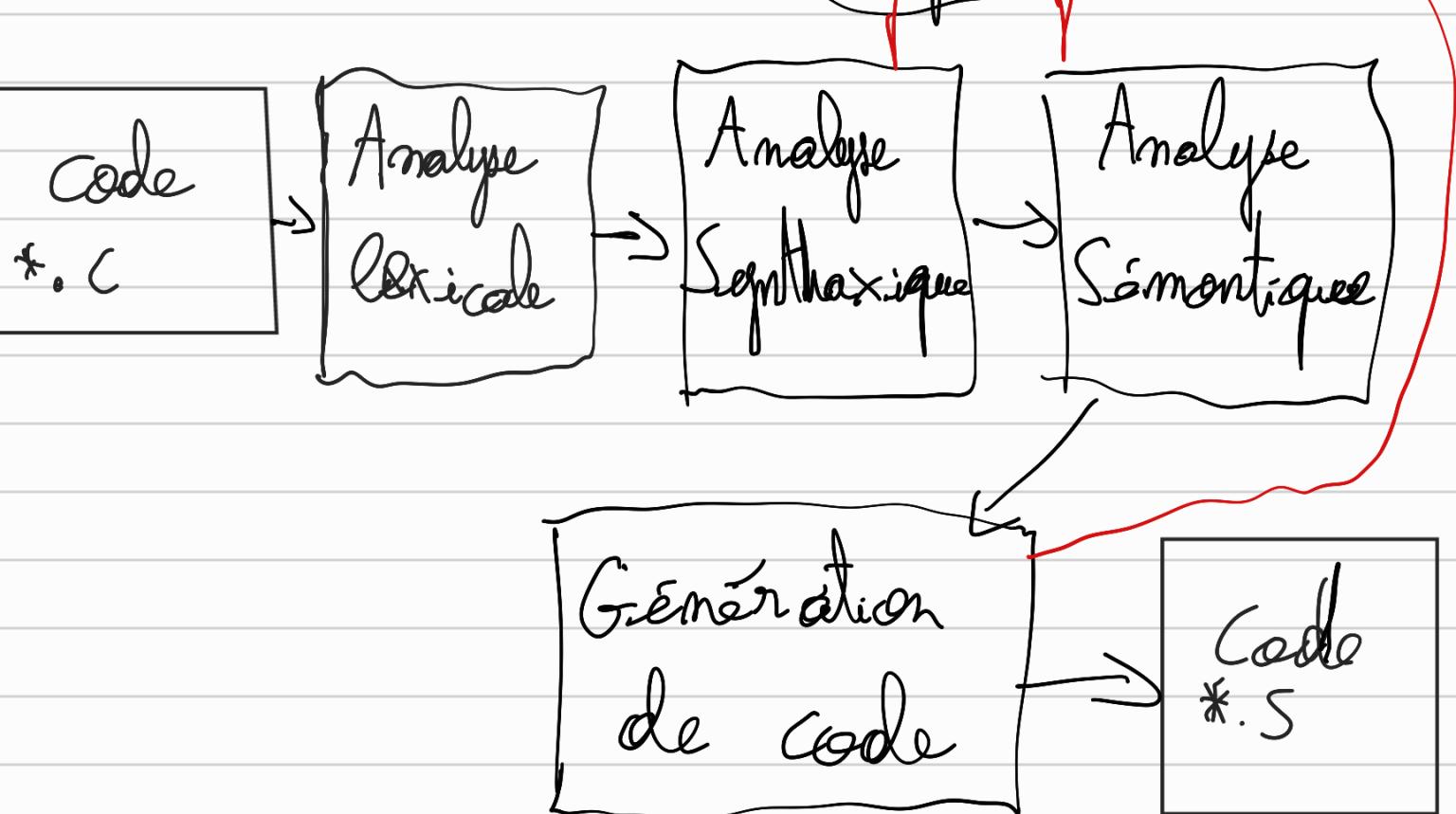
prend le code source (ex: python)
et exécute sa sémantique

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:

Costante
entiere

3

A rectangular box containing handwritten text. The top half contains the word "Siegen" on the first line and "moins" on the second line, both written in cursive. Below this, a horizontal line separates the text from a large, hand-drawn 'X' mark.

Contento entière	Signé et odds
45	O

Identifier
"abc"

↑
santisation ou négation ?
⇒ Analyse Séptomique

En C ces pointeurs sont des structures

La valeur prendra 2 types : entier ou chaîne

stuck taken {

int / string values

{

1) en c une énumération de 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. 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 Syntaxique

2 variables globale dans le code



l'heure sur lequel on est entraîné de l'heure d'avant

Ex : 1 (+) 2 * abc
last token

accès à ces 2 variables en lecture

for function next()

last = token ;

taken ← lecture

lectures des témoins

les fonctions :

bad, check(T) {

```
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:

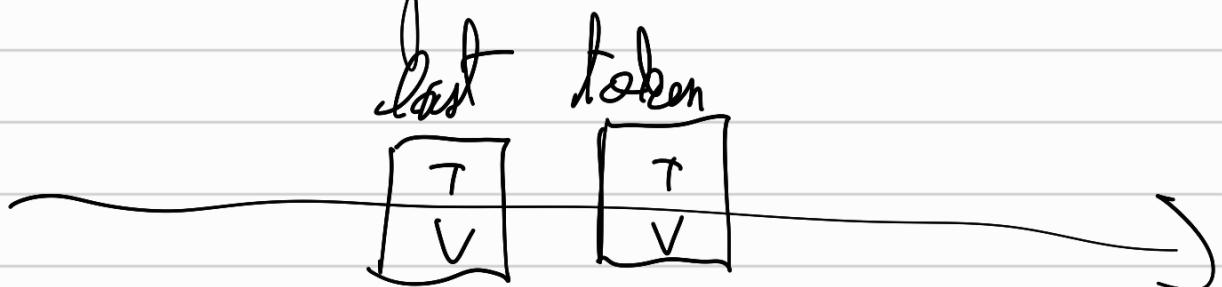
+ - * / % ! &
< = > = == !=

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 (isInfix(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

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

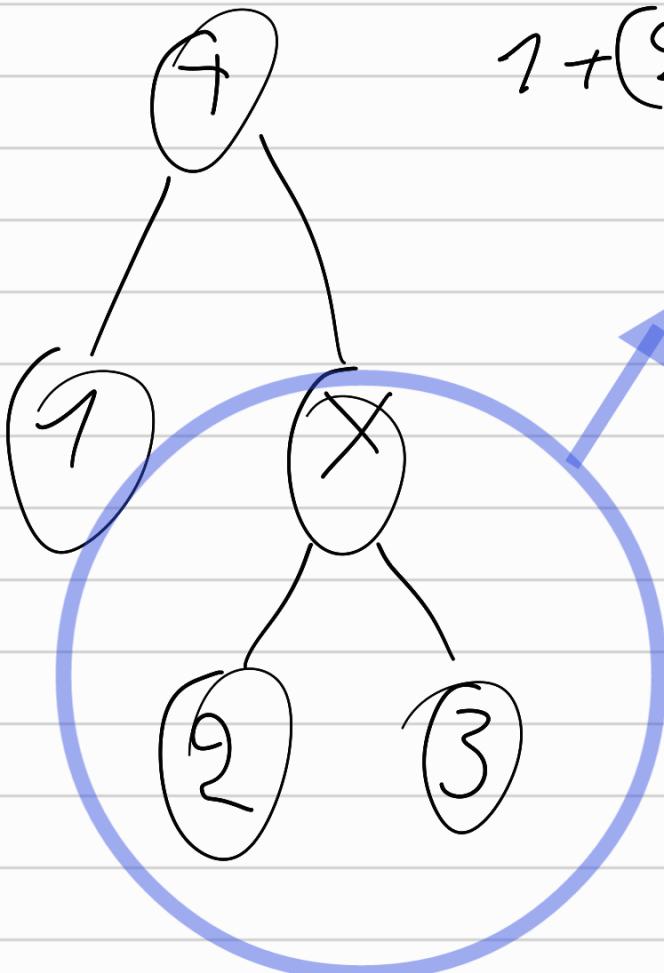
atome

$$1 + 2 \times 3$$

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

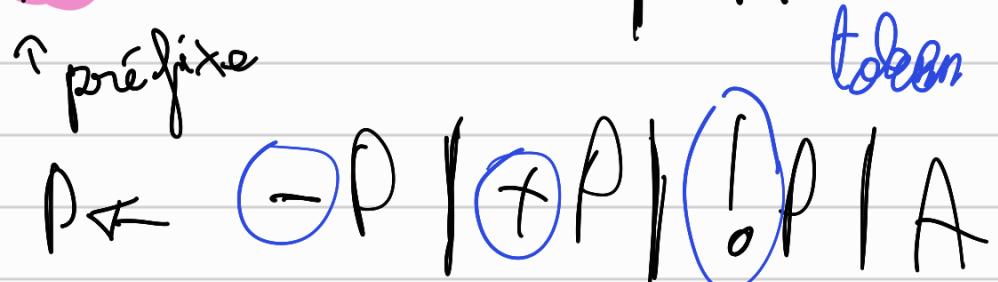
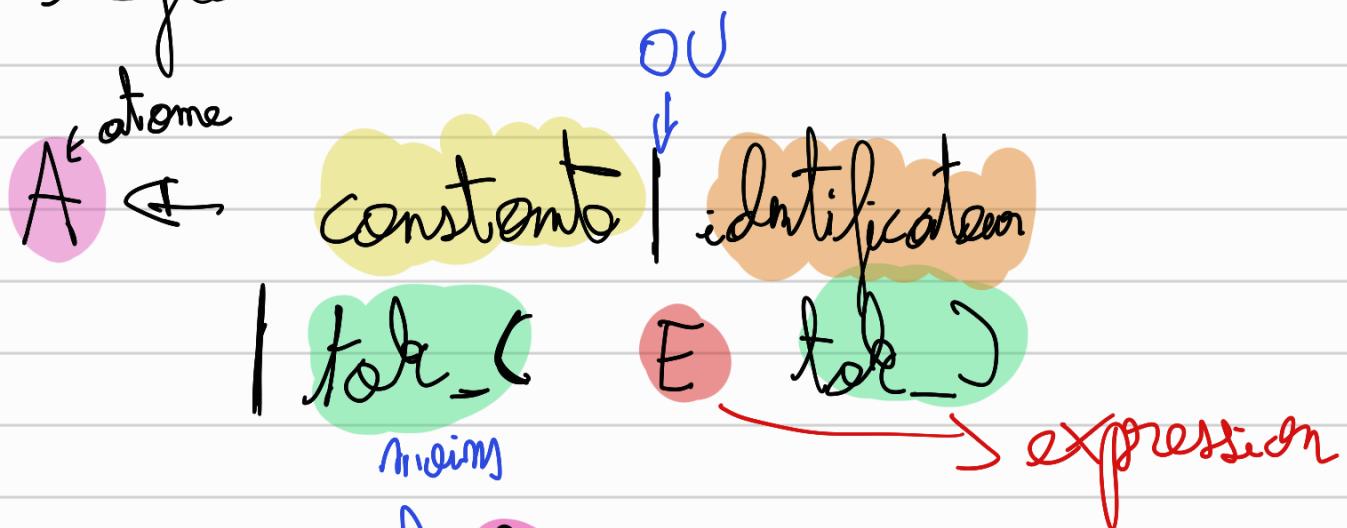
Last ← dernier token de l'atome

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 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;

 } le plus en arre
 me change rien
 + q équivalent à q

} else {

 N = A()

return N.

} return N

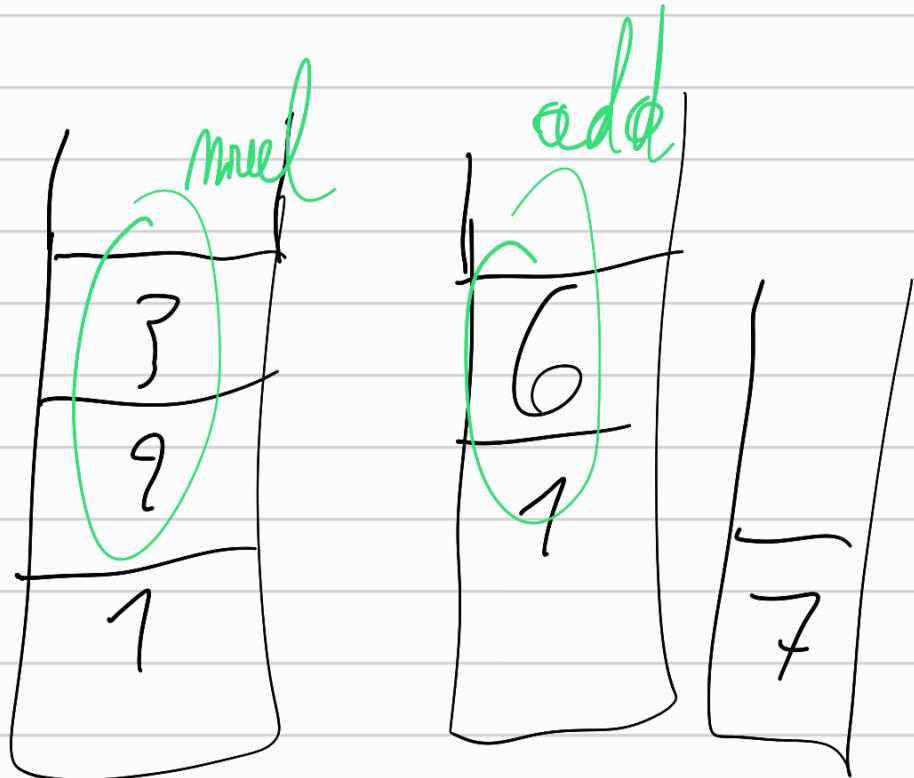
Noeud E()
return P()

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_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 c'est à dire 0

Gestion des priorités ✓

Gestion de l'associativité :

$(1 - 2) - 3$

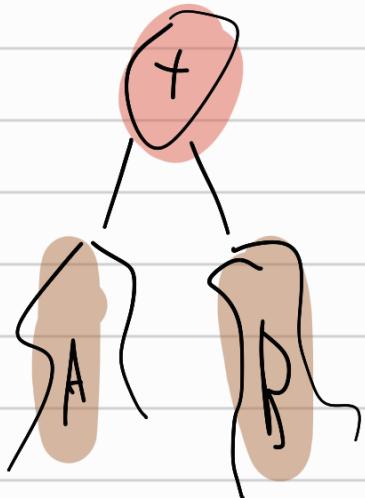
associativité à gauche

$2^1(3^14)$

associativité à droite

Opération d'affectation : associativité à droite

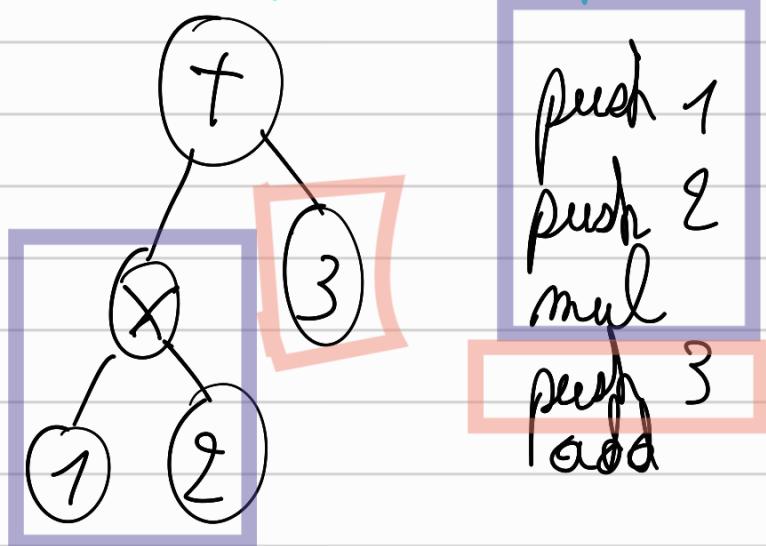
$a = [n = c]$ implique



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_plus~~ $\rightarrow \begin{cases} \text{nde} = \text{node_plus} \\ \rightarrow P=1, \text{Add}=0 \end{cases}$

priorité de l'opérateur

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

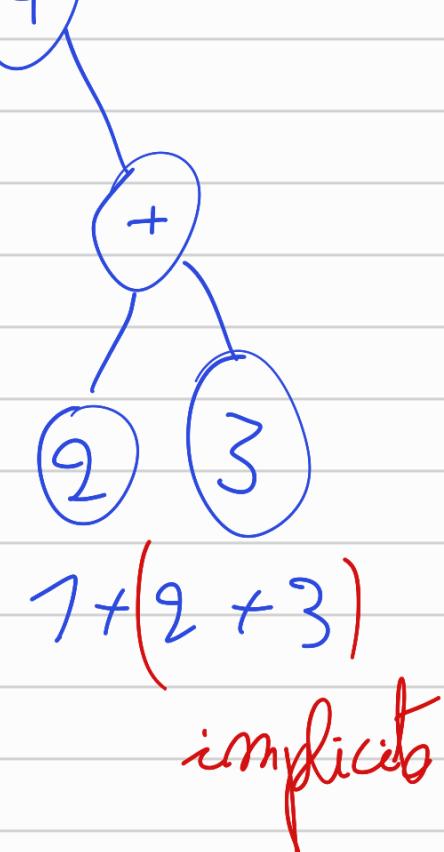
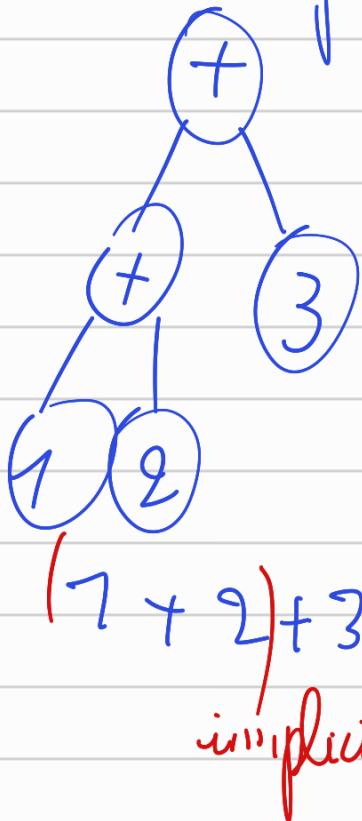
~~node_stade~~ $\rightarrow \begin{cases} \text{nde} = \text{node_mult} \\ , P=2, \text{Add}=0 \end{cases}$

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'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

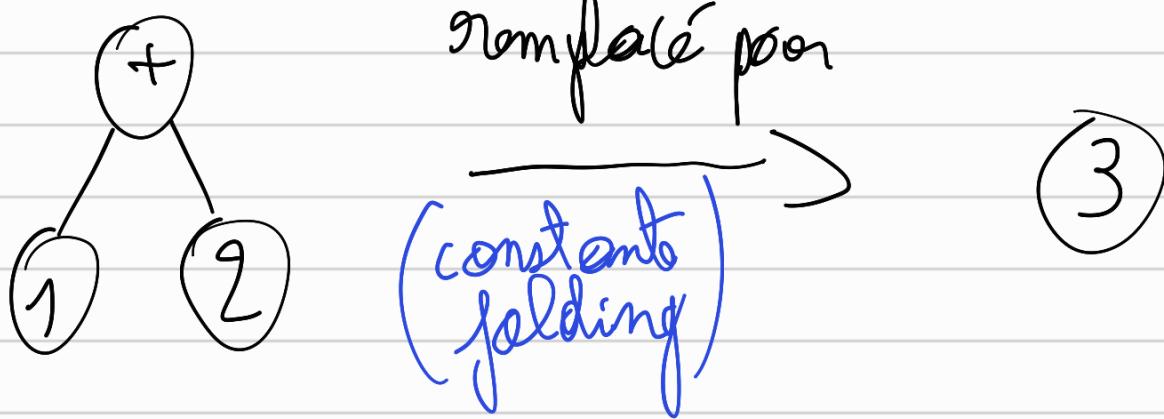
On évite le code inutile créer
par le compilateur

[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 pour arbre.

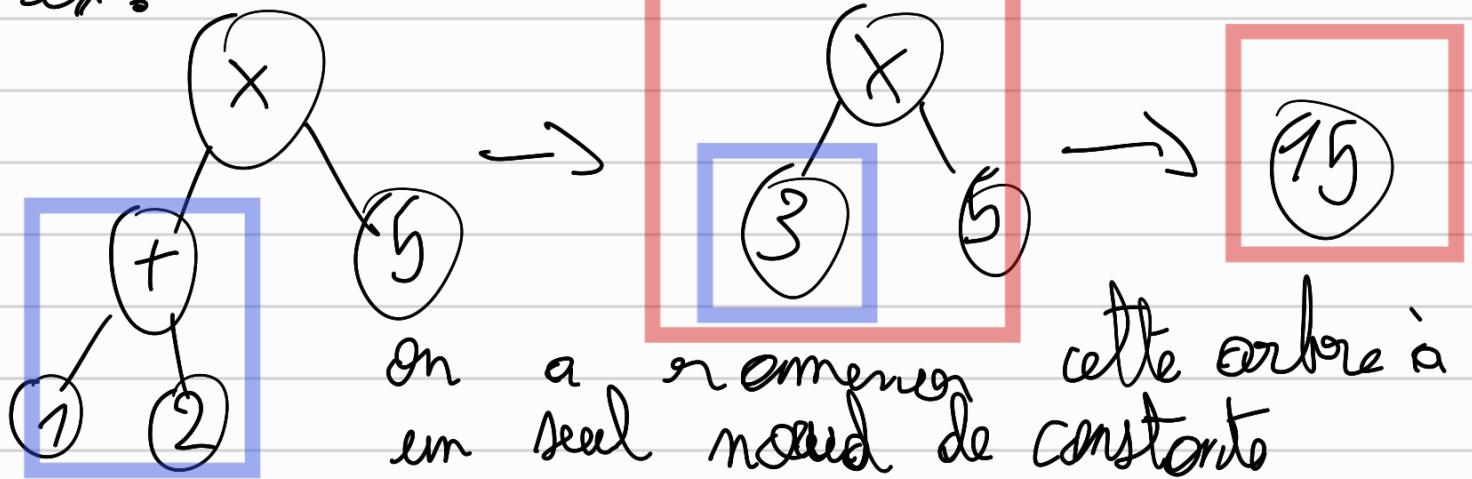


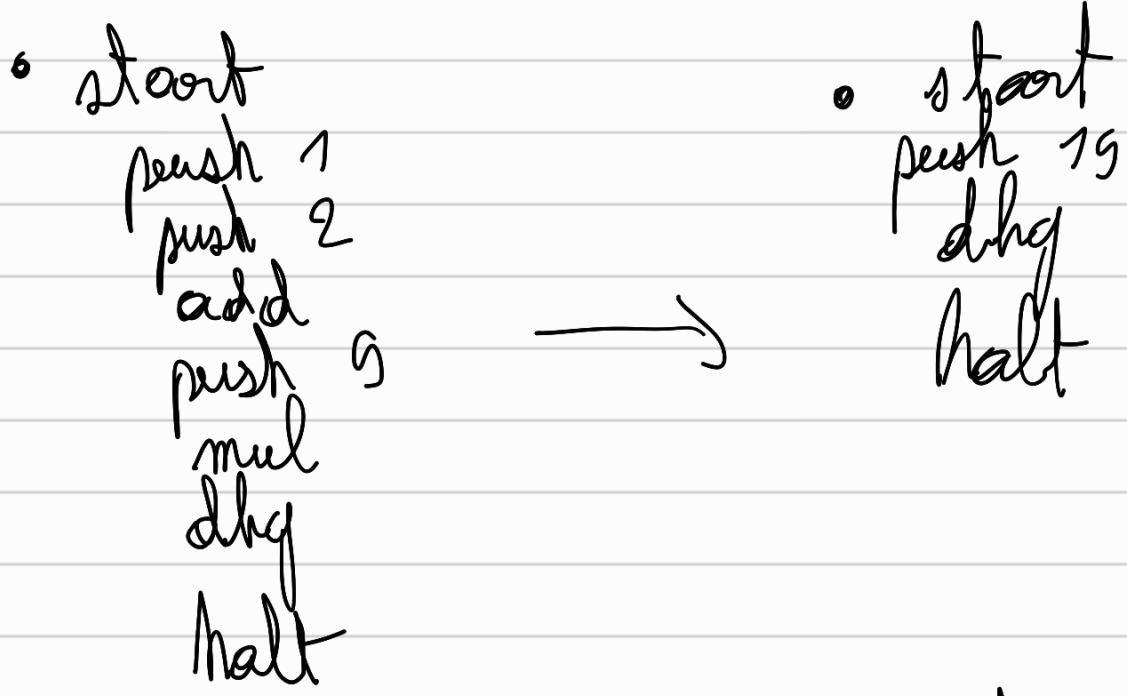
appel d'une fn peut être + car les que d'exécution sont code ex: max

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

Optimiseur parcours contre 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 t1, t2, t3, t4, t5, t6, t7, t8, t9, t10;

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 laq E ;

Tant que ça fait on continue

inf(E) I

pour tester malgré

l'apparition des instructions

Fonction I / | S token

```

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

```

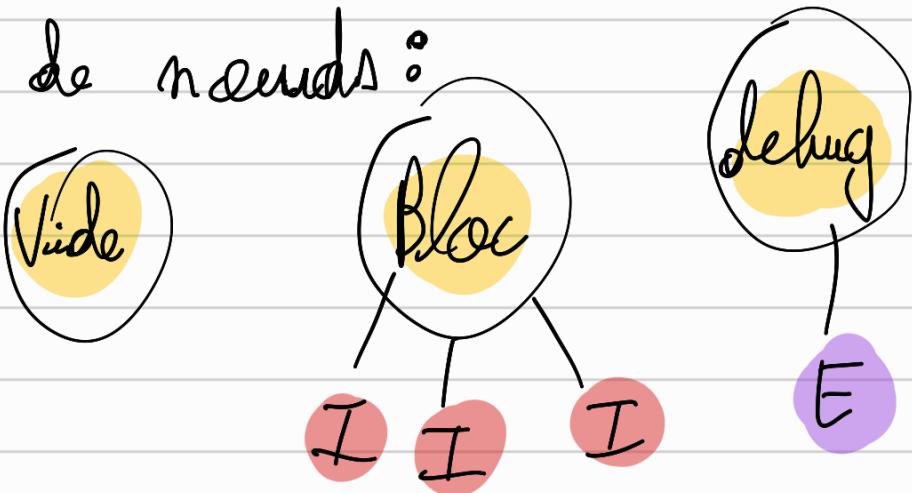
↗ in Vide
 ↗ on deit
 ↗ gronkogen
 ↗ er. noend

```

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)

Prochien taken
 seit extra
 sinnvoller

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

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

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

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

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

abn expression \neq abn instruction



Ce need nous permet de différencier les instructions des expressions car on enlève de la pile la valeur mise par l'expression de celle instruction.

Conservé l'arbre mais pas sa valeur de la pile

GeneCode des needs

Pas de genicode pour le nœud vide



: Banche :

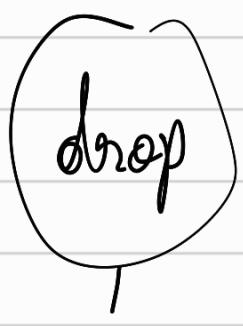
Pour chaque enfant N :

Genicode(N)



: Genicode(Enfant[0])

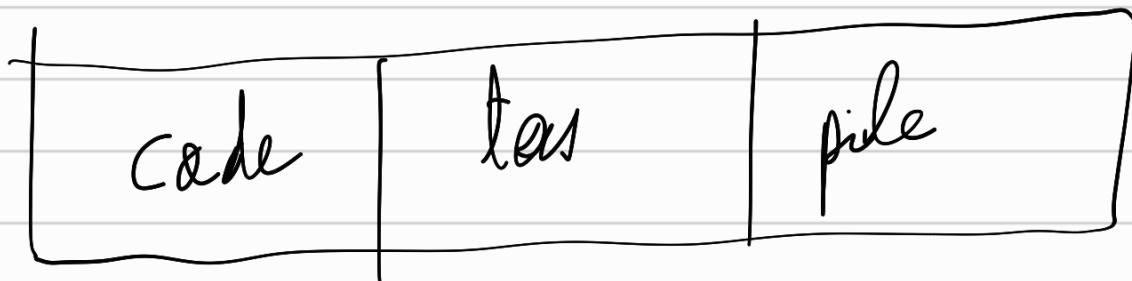
print("dby") → enlève la
valeur de l'
expression sur la pile



: Genicode(Enfant[0])

print('drop 1') ← pareil

Schéma mémoire :



Les Blocs délimitent la portée des

Variables.

```
{ 1 + 1 ;
```

int a;

a = 3

```
int b;
```

b = 5 ;

a = 7 ;

```
int a;
```

a = 5

}

int a;
a non trouvée dans le bloc, donc on va dans le bloc supérieur

Table des symboles (4 fonctions)

Si déjà déclaré

S+Déclarer(nom) → ds de le bloc actuel
- Ecraser

S+ Chercher(nom)

Si non renvoie un symbole

↳ 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

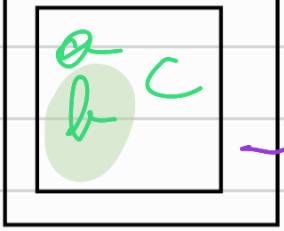


Pile

, End()

qd bloc fini on enlève
cette table de la pile

a b



Une table pour Bloc

Begin() \rightarrow Voor-push(Ø)

End \rightarrow Voor = pop()

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

Mais si on cherche(β), 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 ("Var non déclarée")

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

Si langage empêche

- recreer la 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^e pile

Chercher (nom)

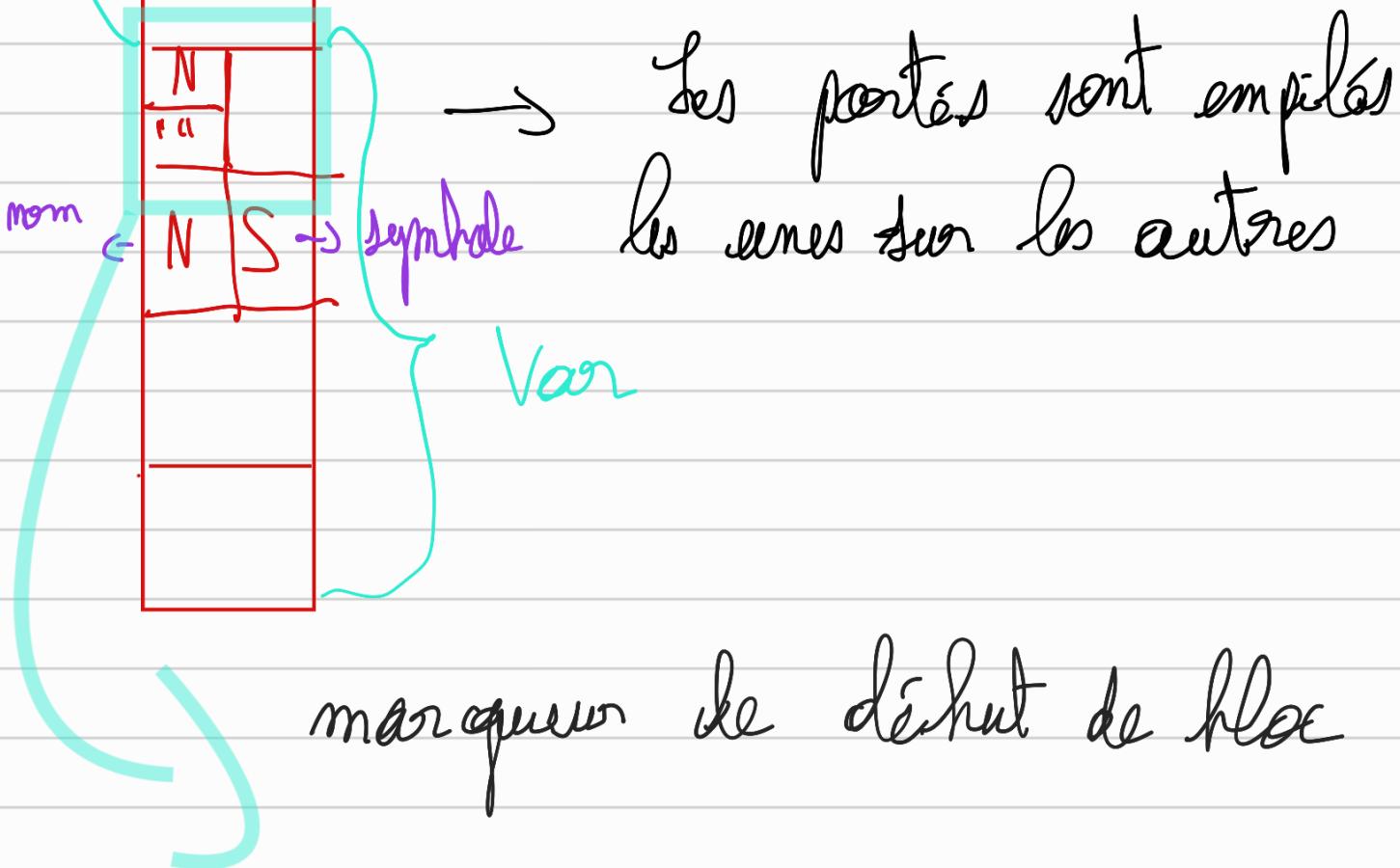
Pour $i = \text{Sommet jusqu'à } 0$

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

| | return $T[i].S$

Erreur Fatal

Van



Declarer (nom)

Pour i = Sommet jusqu'à 0

Si $T[:].oN = \text{nom}$

Erreur Fatal ;

Si $T[:].N = \text{";"}$

break ;

S = Nouveau Symbole

$T.push_back(\text{Nom}, S)$

return S ;

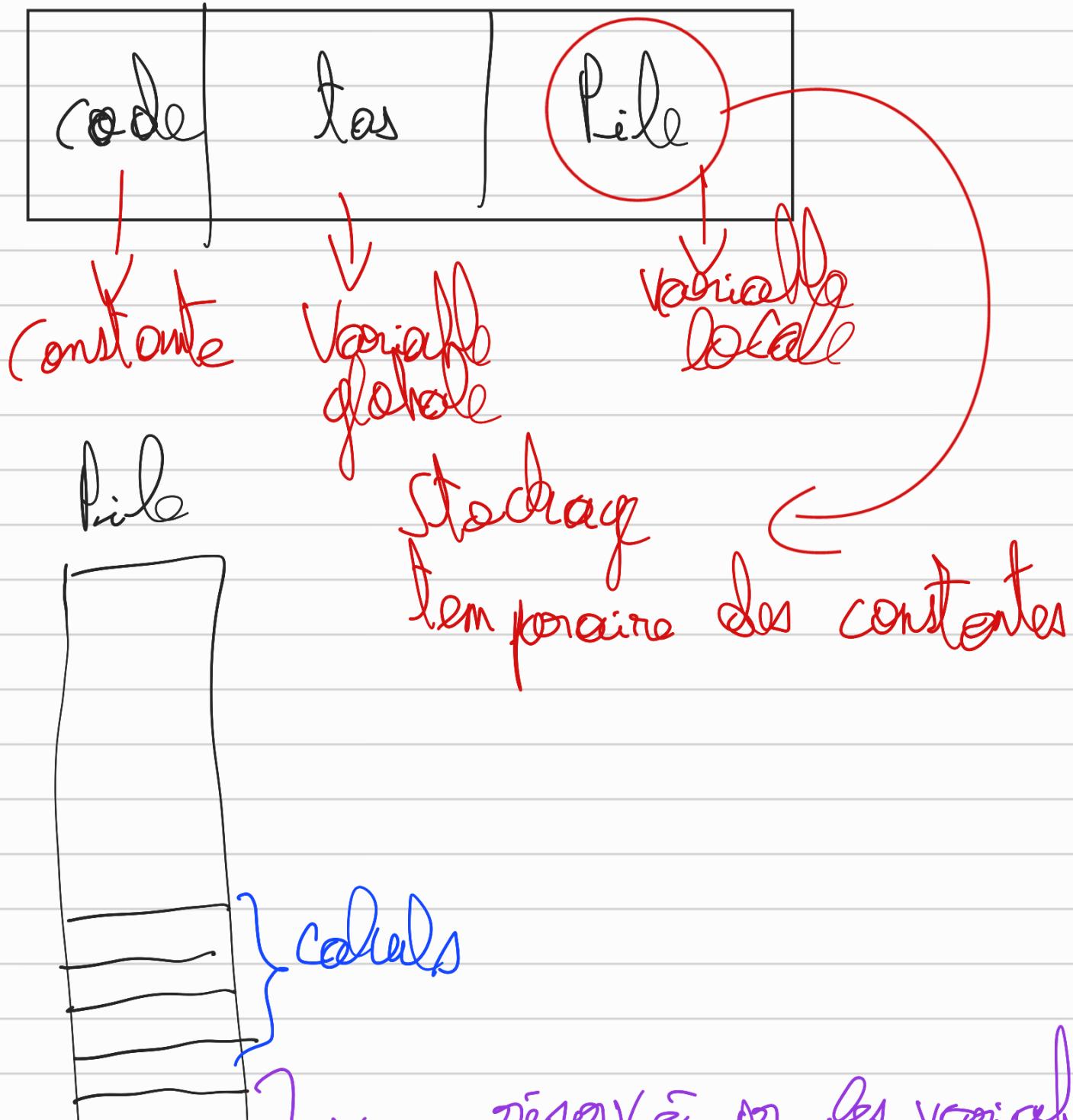
() { } ; , , , / / /)

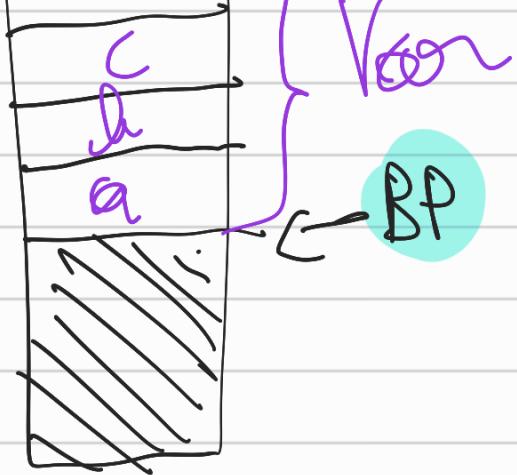
Begin() => T.push(,)

End() => T.pop Jusqu'à "

Cette structure est lors (+) utilisée
par les compilateurs

En python les symboles seront des noms
pour le moment.



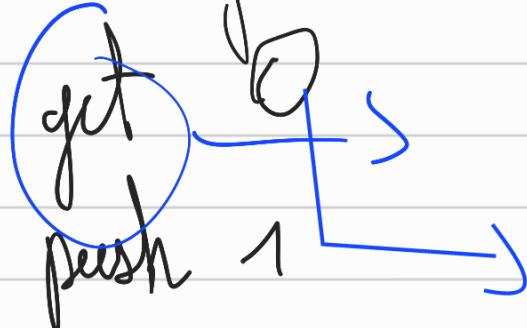


On met les variables sous le calcul, pour éviter que elles soient écrasées par les calculs

Il faut donc réservé suffisamment de case pour toutes les variables locales 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_{ir} 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 {

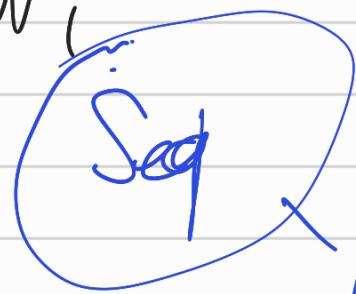
accept(too, ident);

No ajout or Enfant (Noed(No_décl
, last_value))

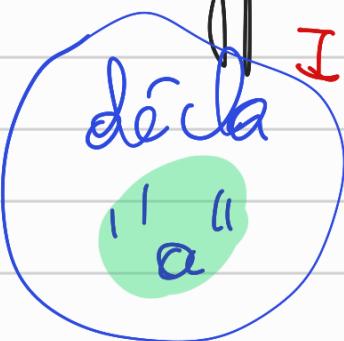
} while(check(' ')) ;

accept(";")

return V;



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



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

utile pour analyse démontique

AnaSem :

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

AnaSem (Nœud N)

switch (N. type) {

départ :

Pour chaque enfant E :

AnaSem (E)

Cose Nœ-Block :

Begin();

Gestion de la
sortie

Pour chaque enfant E :

AnaSem (E)

End();

Case Node_Declar:

Symbol

$S = \text{declare}(\text{No_value}) ;$

So position = mb Voor;

mb van ++

nombre de
variable

\Rightarrow mb cas
à réservé

local

$S = \text{VorLoc};$

globale

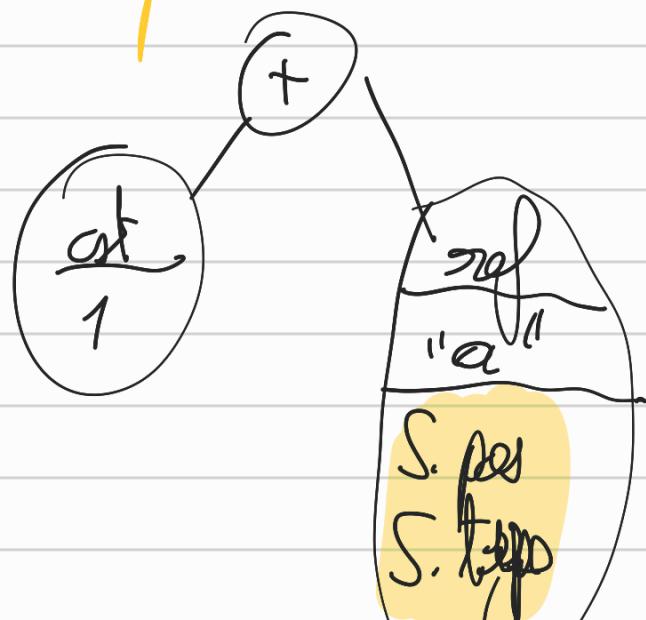
Case Node_Ref:

$S = \text{choose}(\text{No_value}) ;$

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 = 0ⁱ →

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 = 0ⁱ
AnalSem (N)

print ("new", nbVar)

réservation
de case
mémoire

Genecode(N)

print("drop", mVar)

l'héritage de la mémoire après
utilisation.

Pour test:

{ int a;
 a = 3; }

int a;
a = 3;

me
marchera
pas

Conditionnel et Boucles :

if(E) \Rightarrow I

... } else if (check(tok_if)) {
 accept(tok_if);
 e = E();
}

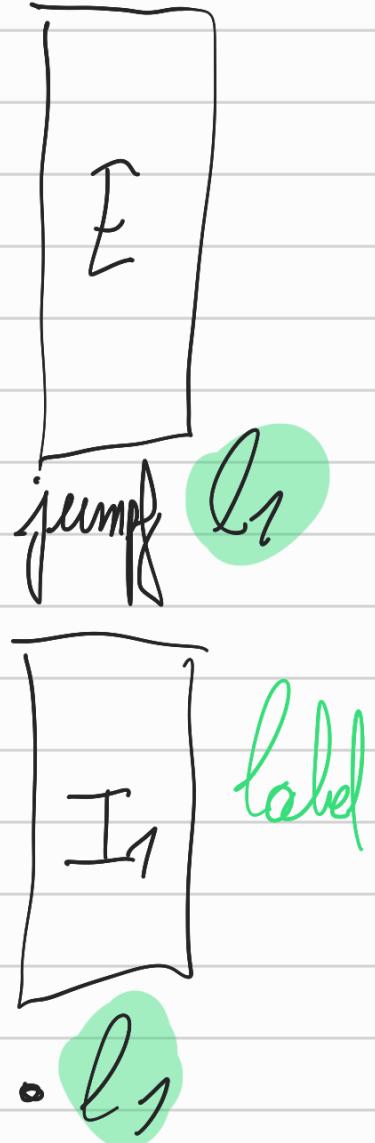
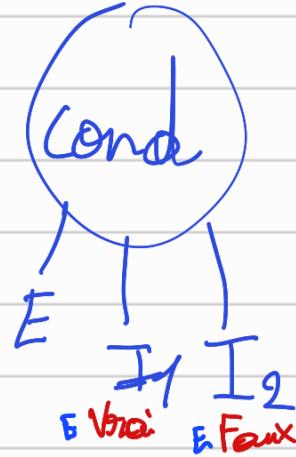
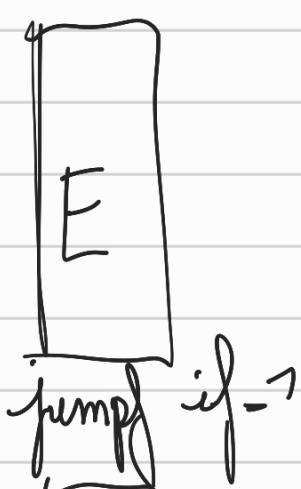
accept(tok_if);

$i_1 = I(1)^o$
 $\text{if}(\text{check_tak_else})\{$

$N = N_{\text{eud}} | N_{\text{Cond}}, e_1, i_1 \}$
 $i_2 = I(1)^o \}$
 $\text{if}(i_2) \{$

Ajouter Enfant $(N, i_2), \}$

return N^o



Sont des instructions
jusqu'au label
spécifié

Genecode



Case N°cond :

if N°bienfaits == 2 {

l1 = nblabel ++;

genecode(N, en[0])

print("jump ", l1)

genecode(N, en[1])

print(" = ", l1)

} else {

l1 = nblabel ++;

l2 = nblabel ++;

2 condition

out

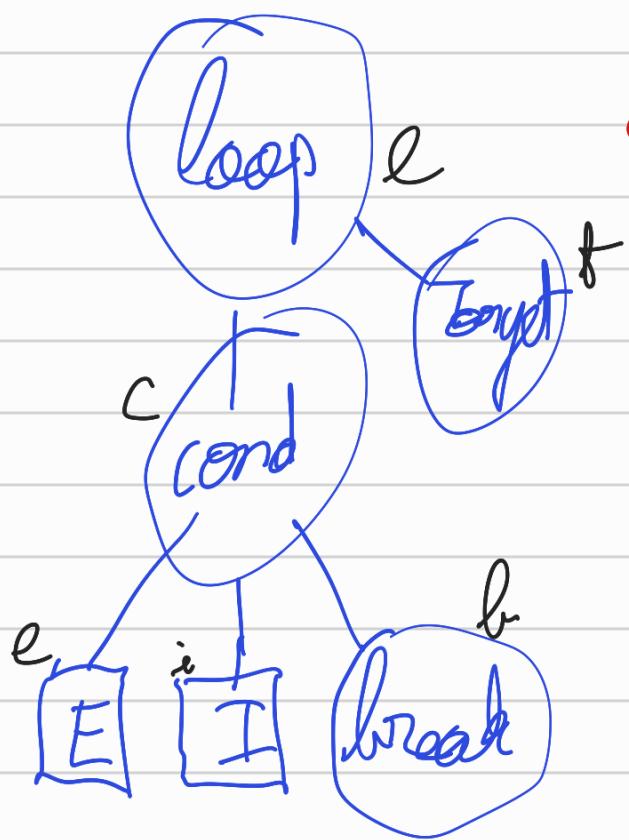
while(E) I

endroit d'edition

loop { To print

report après avoir fait continue

if(E_1)
 else
 {
 E_2
 break;
 }
 }
 for (E_1 ; E_2 ; E_3)
 {
 ↓
 E_1
 loop {
 if (E_2) {
 Target E_3
 }
 else break;
 }
 }



on usage of
 compiler:
 while(E)
 {

```

if (check( tok_while )) {
    accept( tok_E )o;
    e = E(0)o;
    accept( tok_ )o;
    i = I()o;
}

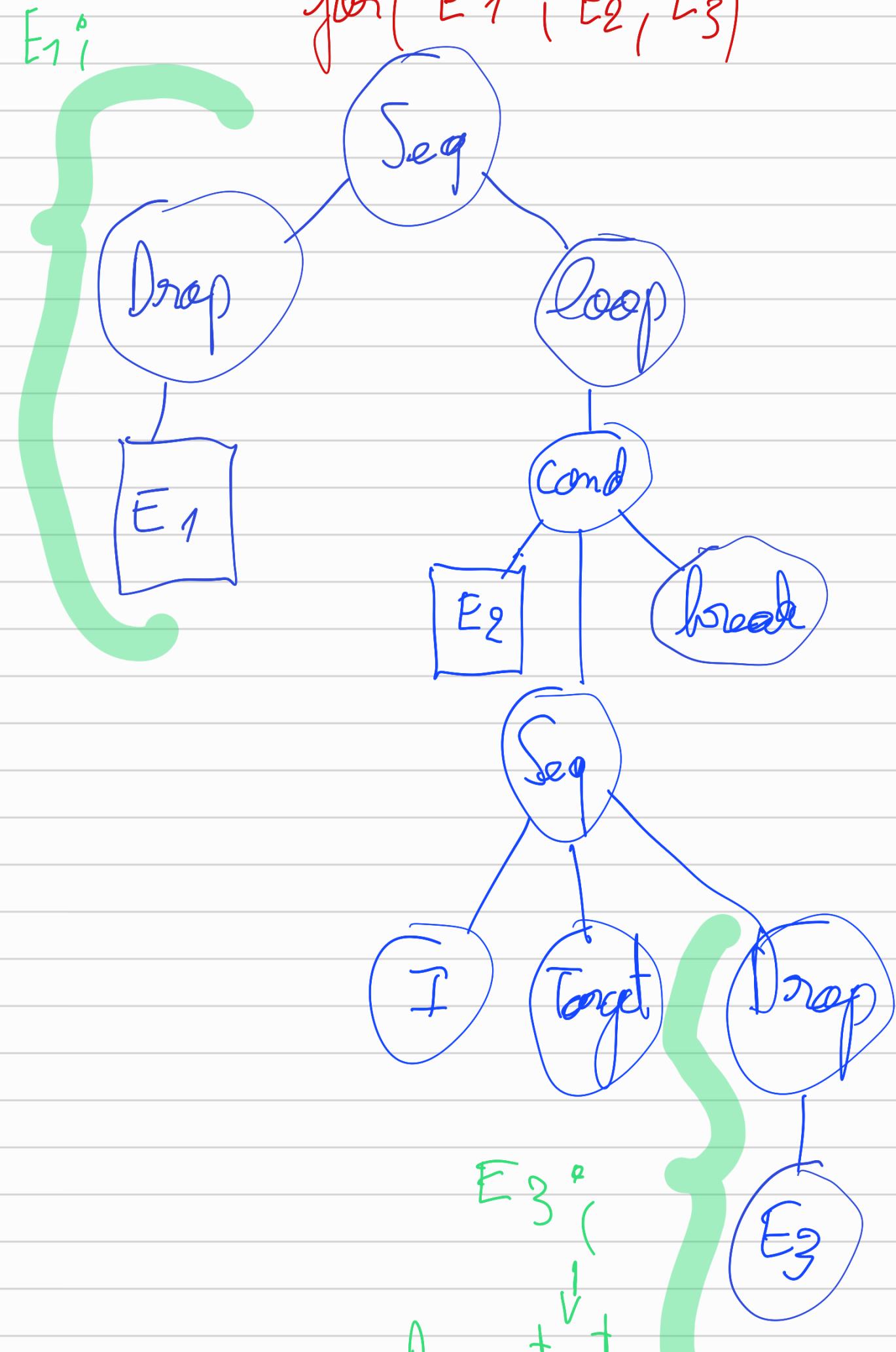
```

$l = \text{need}(\text{NdLoop})$
 $t = \text{need}(\text{NdTarget})$
 $c = \text{need}(\text{NdCond})$
 $b = \text{need}(\text{NdBreak})$
 $\text{AjouterEnfant}(l, c)$

- // (l, t)
- // (c, e)
- // (c, i)
- // (c, b)

on essaye de compiler:

for(E₁; E₂; E₃)



Gene Code
↓

target →

print("a", ll_cont)

Vor global

break → print("jump", ll_break)

Continue → print("jump", ll_cont)

I ← break;

I ← continue;

Pour loop:

ll_debut = nblabel++

save_cont = ll_cont

save_break = ll_break

Save register
alors de la
fauch principe
de la fauch imbriqué

global ll_cont = nblabel++

negative

lhl_break = nb_label++ // de l'identification
print ("• l", lhl_debut) // du label
Pour chaque enfant
genocode(enfant)

print ("jump l", lhl_debut)

print ("• l", lhl_break)

lhl_cont = save_cont
lhl_break = save_break

*loop réservé les identifiants des
label, c'est pk un break ne
peut-être qu'un enfant de loop
cor si on son jump ne fera
référence à rien.

