

# Compilation

(machine virtuel,  
simulation)

Compilateur ≠ interpréteur



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

Ex: Code source → binaire

| ↴ autre langage  
↳ Assembleur

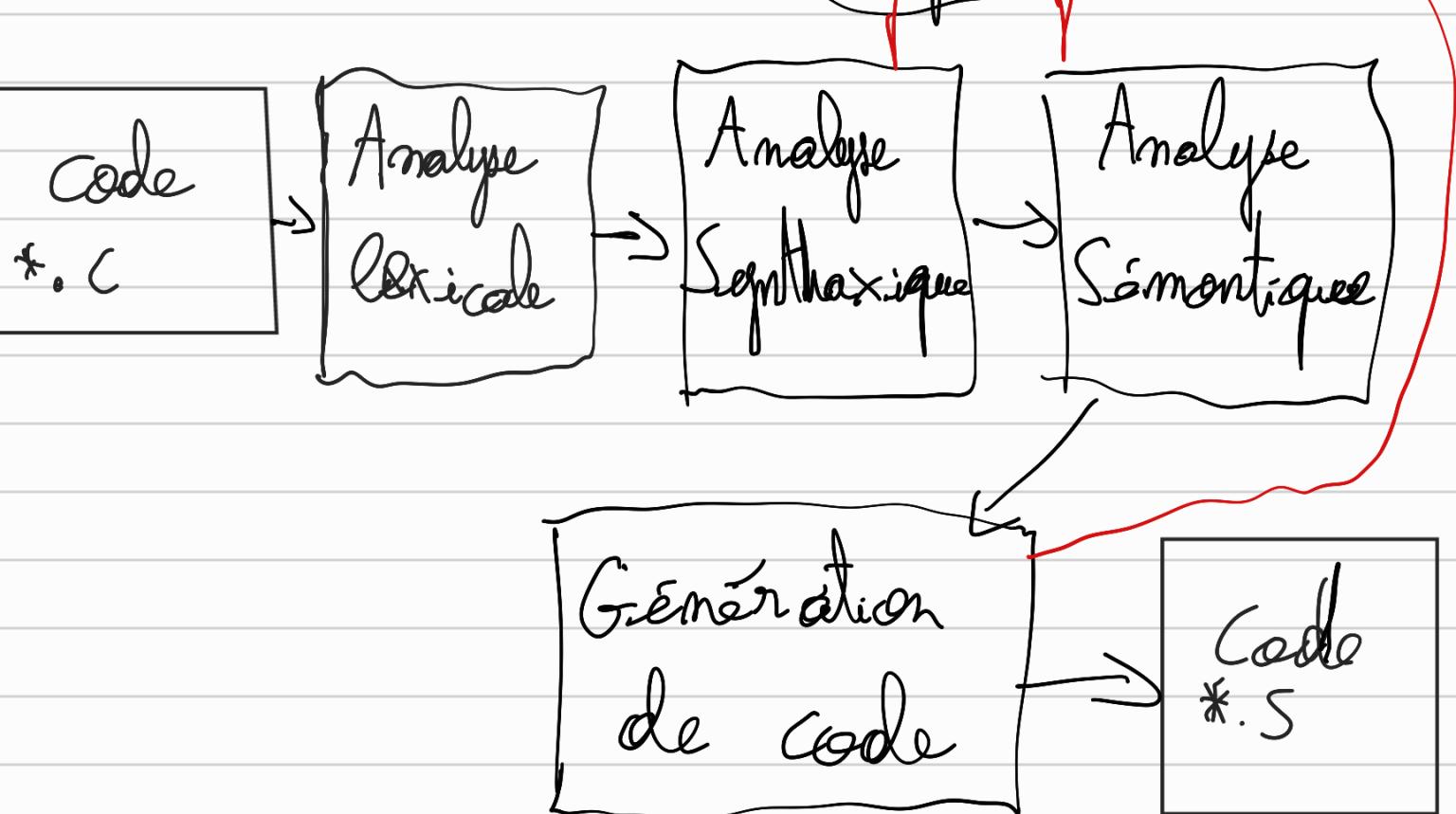
prend le code source (ex: python)  
et exécute sa sémantique

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  
main

Contento entière	Signé étoile
45	O

Identification  
"abc"

↑  
santisation ou négation ?  
⇒ Analyse Synthétique

En C ces pointeurs sont des structures

"java" "python" "c" "c++" "csharp" "object" "dictionnaire"

Valeur prendra 2 types : entier ou chaîne

stuck taken {

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 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

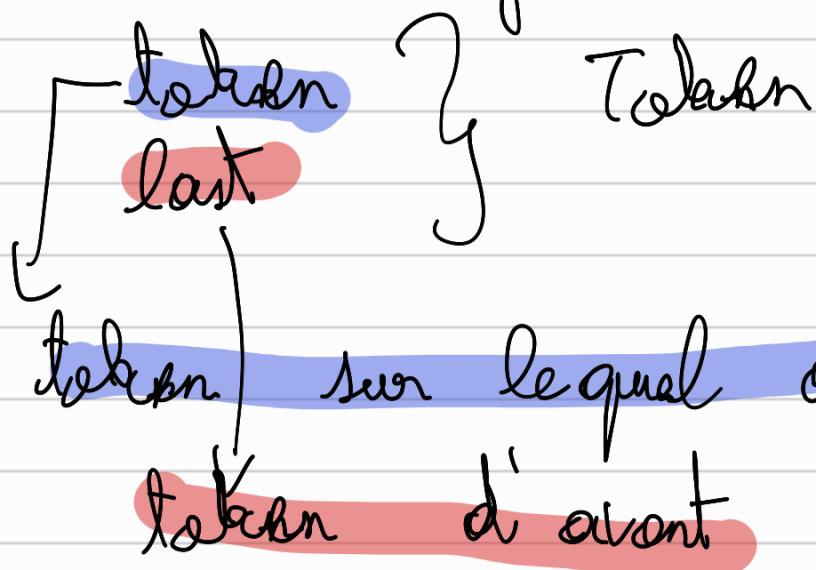
Bande et condition

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 ← lecture

- lecture  
(lecture des télescop  
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.

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:

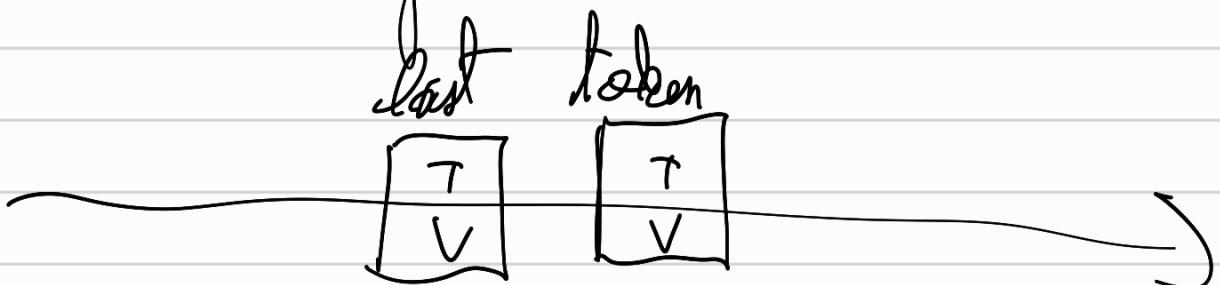
+ - \* / % ! &  
< = > = == !=

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

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

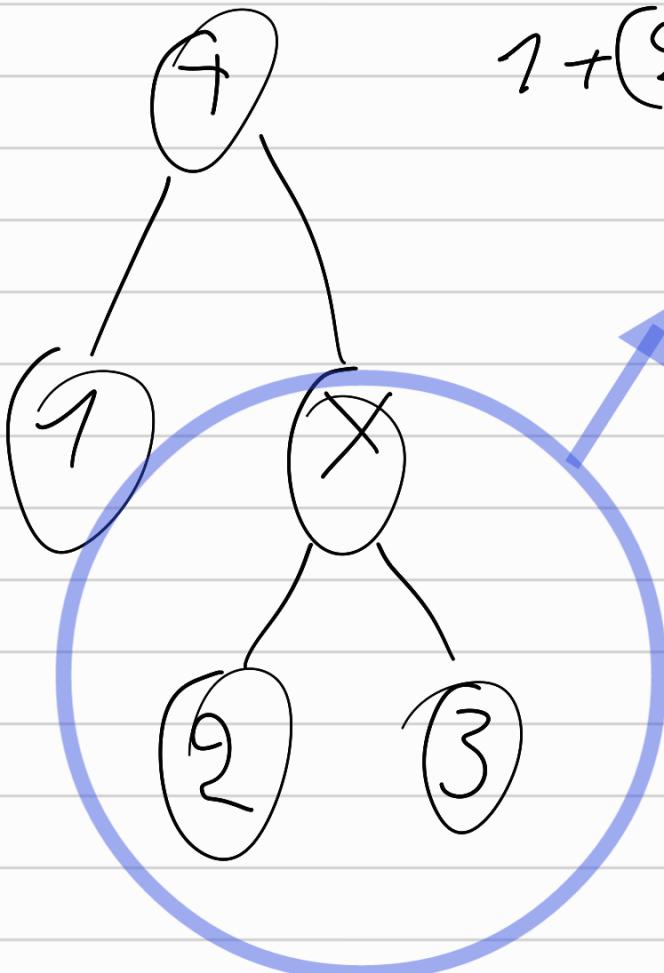
atome

$$1 + 2 \times 3$$

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

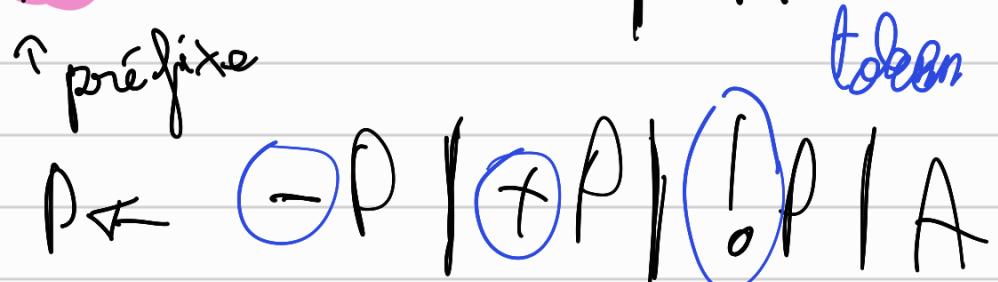
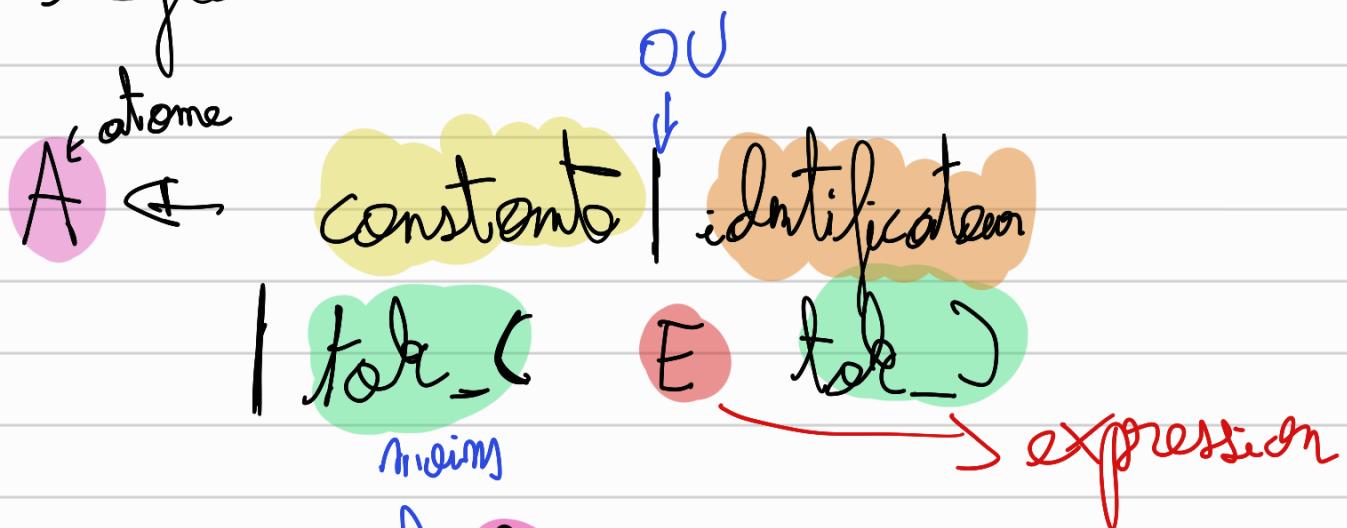
Last ← dernier token de l'atome

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 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 envoie  
ne change rien  
+ q équivalent à g

} return N()

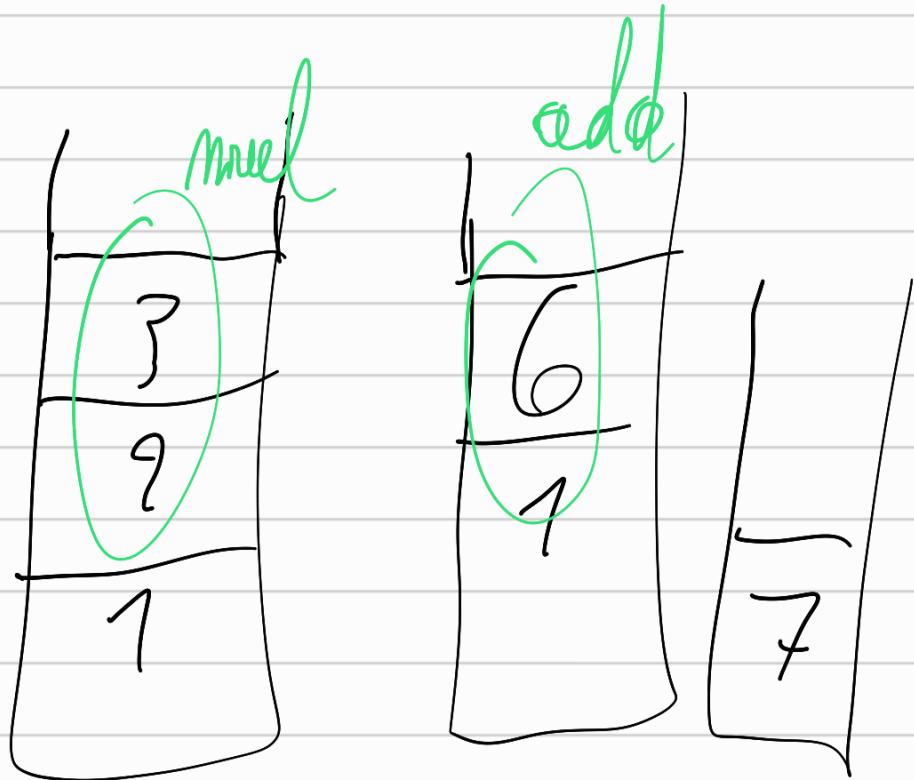
Noeud E()  
return P()

Génération du code

Utilisation d'une pile

Ex:  $1 + 2 \times 3$

push 1  
push 2  
push 3  
mul  
add



①

gener-cade(N)



switch (No\_type)

case Node\_constant:

print ("push"; No\_valour)

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 :

$(1 - 2) - 3$

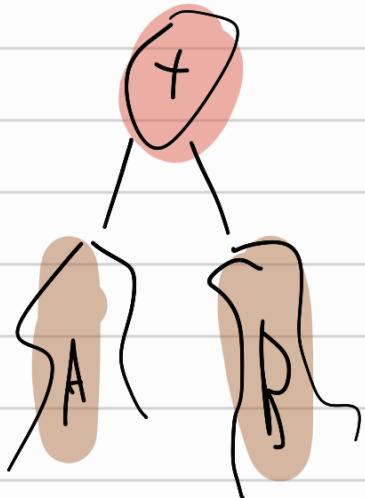
associativité à gauche

$2^1(3^14)$

associativité à droite

Opération d'affectation : associativité à droite

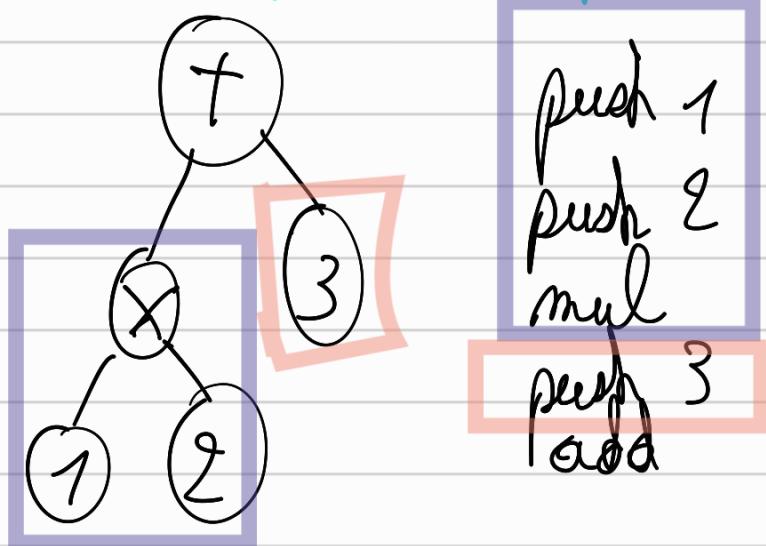
$a = [n = c]$  implique



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

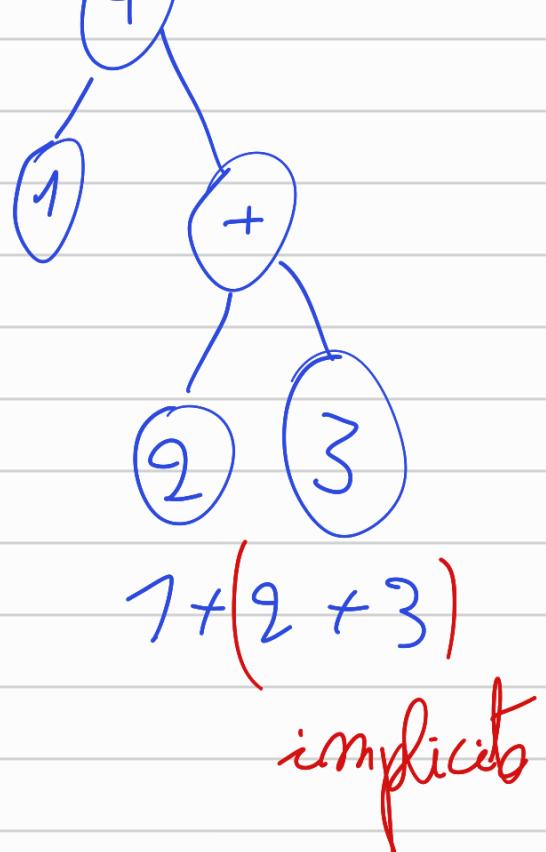
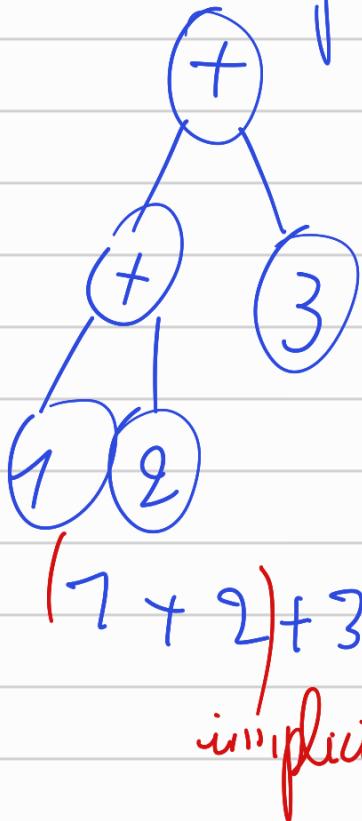
~~node\_stab~~  $\rightarrow \begin{cases} \text{nde} = \text{node\_mult} \\ , P=2, \text{Add}=0 \end{cases}$

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
$  $	2	0
$\& \&$	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<sup>er</sup> élément au sommet pile

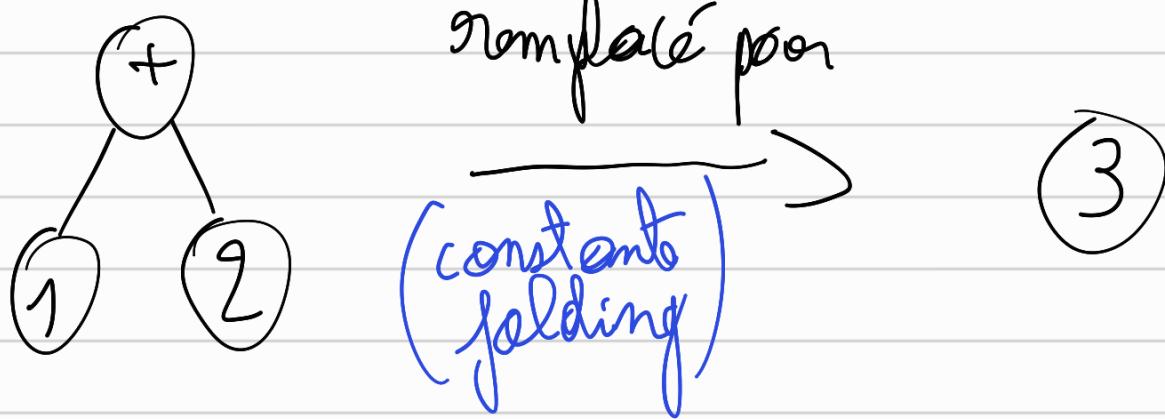
On évite le code inutile créer  
par le compilateur

[ push □ ] → [ drop n - 1 ]  
drop n

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

Optimisation mm . txt :

Optimisation pour arbre.

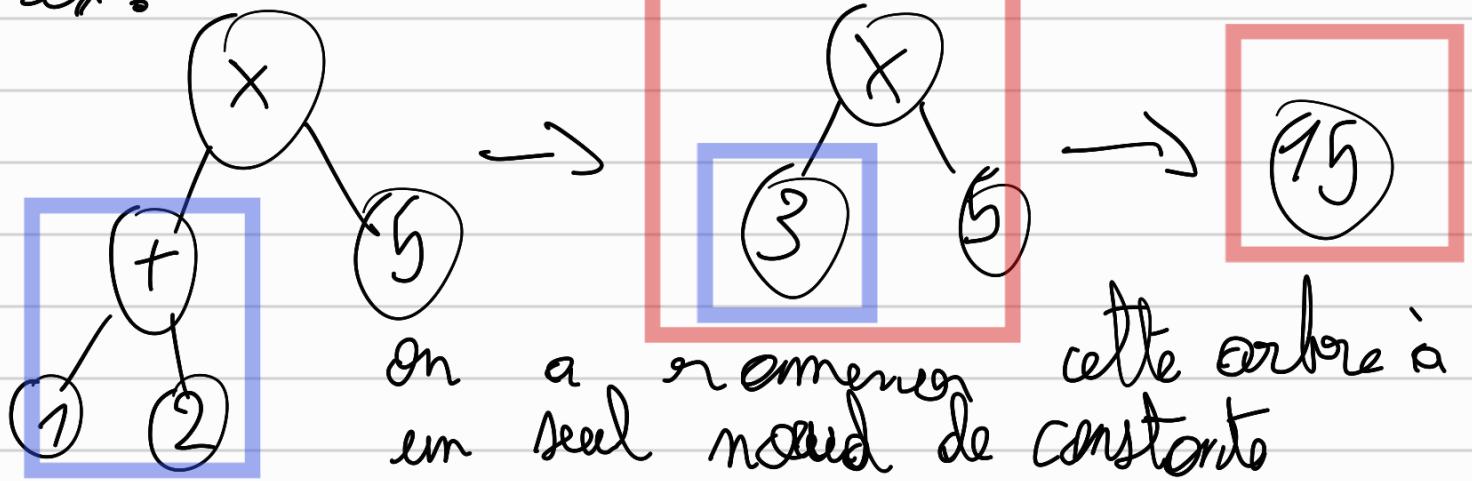


appel d'une fn peut être + car les que d'exécution sont code ex: max

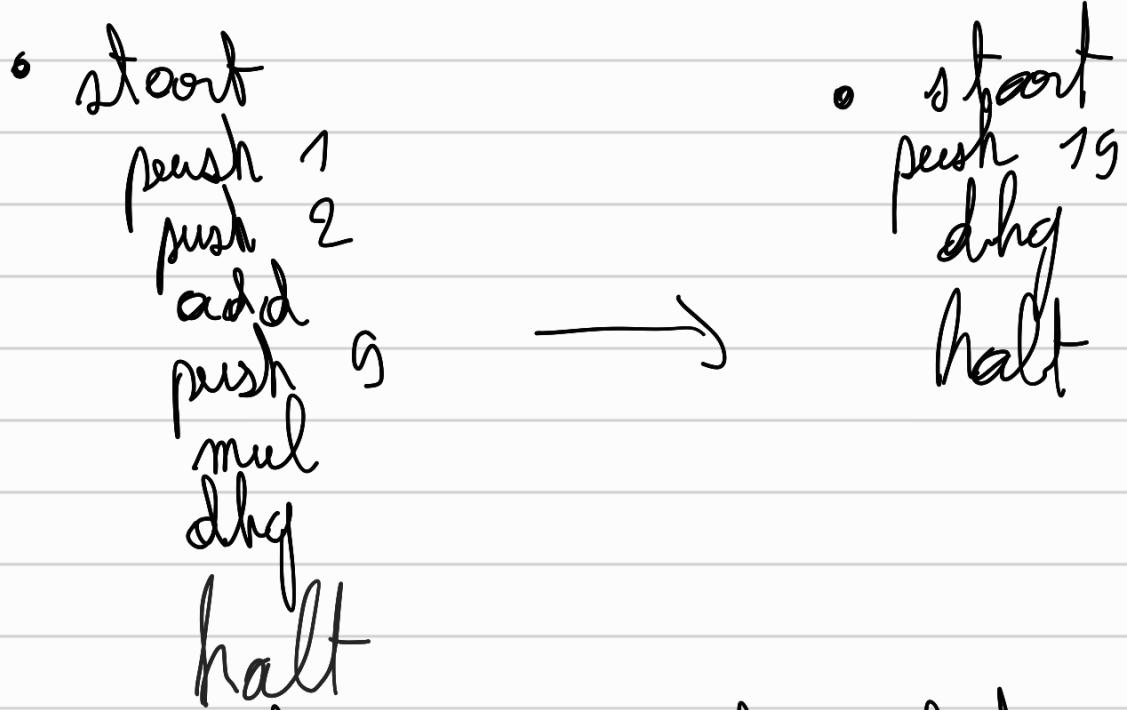
Donc exécu° de sont contre syntaxique et on remplace ces arguments par les val de l'appel de la fonction

Optimiseur parcours contre en s'appelant récursivement sur les enfants

ex:



On a ramené cette arbre à un seul nœud de constant



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, t, ...

instruction  $\neq$  expression

ex:

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

$$a = 7 + 2 \quad \left. \begin{array}{l} \text{Instruction} \\ \text{if} \end{array} \right\}$$

Production  
d'une valeur

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

if ( or < 5 )  
d = 7 ;

La ④ simple instruction est ;

I  $\leftarrow$  ;

| E ;

| I { I \* } ;

| de laif E ;

Tant que ça fonctionne  
on continue

if ( E ) I

pour tester malgré  
l'apparition des instructions

Fonction I / | S  $\rightarrow$  token

```

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

```

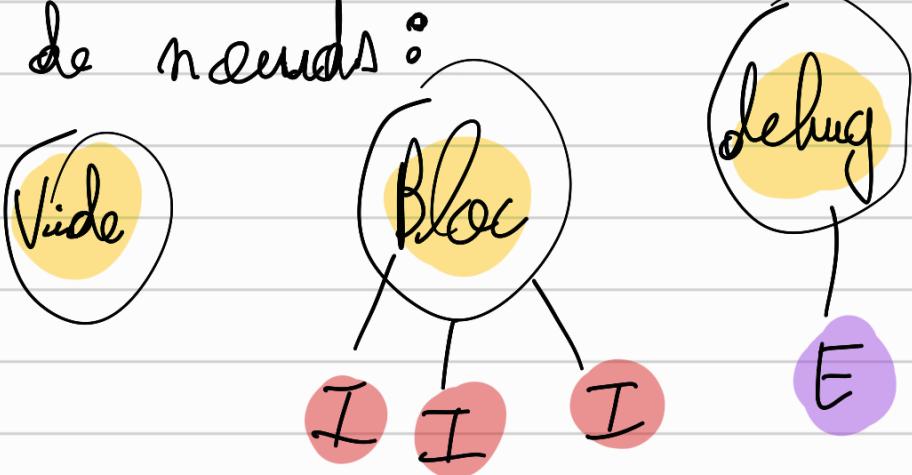
↗ in Vide  
 ↗ on deit  
 ↗ gronkogen  
 ↗ in noend

```

while( !check("}") )
    N. ajout(Efant, I())
return N

```

Nieuwean type de noends:



```

elif( check( tok_debug ) )

```

$N = E[0]$

accept( "0" )

return Noend( NdDebug, N )

Pas achter token  
dit extra op  
simon corriger

$\{ \text{else} \}$      $\leftarrow$  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.

arb expression  $\neq$  arb instruction

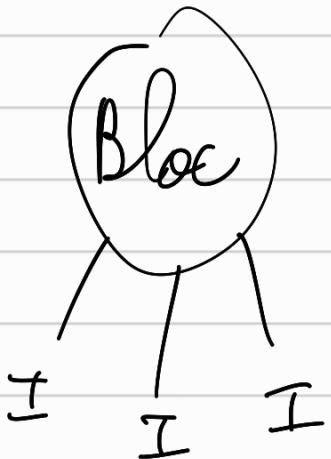


Ce seed nous permet de différencier les instructions des expressions car on enlève de la pile la valeur mise par l'expression de celle instruction.

Conservé l'arbre mais pas sa valeur  
de la pile

Gene code des seeds

Pas de genicode pour le nœud vide



: Banche :

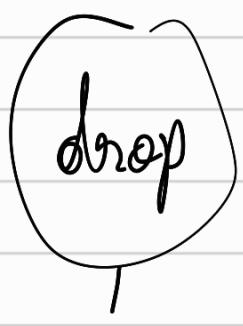
Pour chaque enfant  $N$ :

Genicode( $N$ )



: Genicode(Enfant[0])

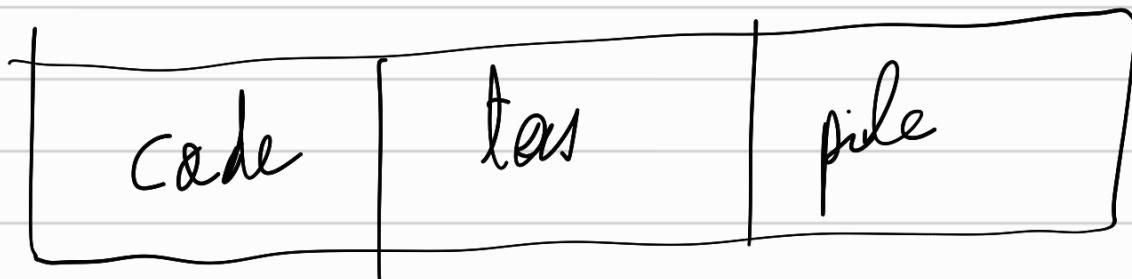
print("dby") → enlève la  
valeur de l'  
expression sur la pile



: 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;  
int a = 3;

{ int hi  
    b = 5;  
    a = -7;  
    int a<sup>ii</sup>;  
    a = 5;

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

3

## Table des symboles (4 fonctions)

Sí, déjá declaro

S+Deklarer (nom) → ist die Umlaut auch aktuell  
↳ Symbole                    ↳ Ergrößen

S\* ChorChor (mom)

Simon grenzige im  
Septuhalle

↳ 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  
que une fois par bloc.

Déclarer (nom)

si nom exist in VarTab

Erreur

S = nouveauSymbbole

VarTab[nom] = S

return S

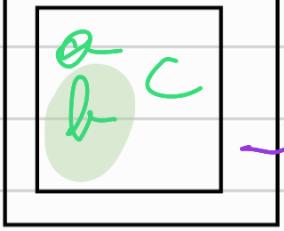
pile de  
table de  
Hachage



Pile

, End()

qd bloc fini on enlève  
ette table de la pile



Une table pour Bloc

Begin() → Voor-push(Ø)  
End → 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 1re pile

Chercher (nom)

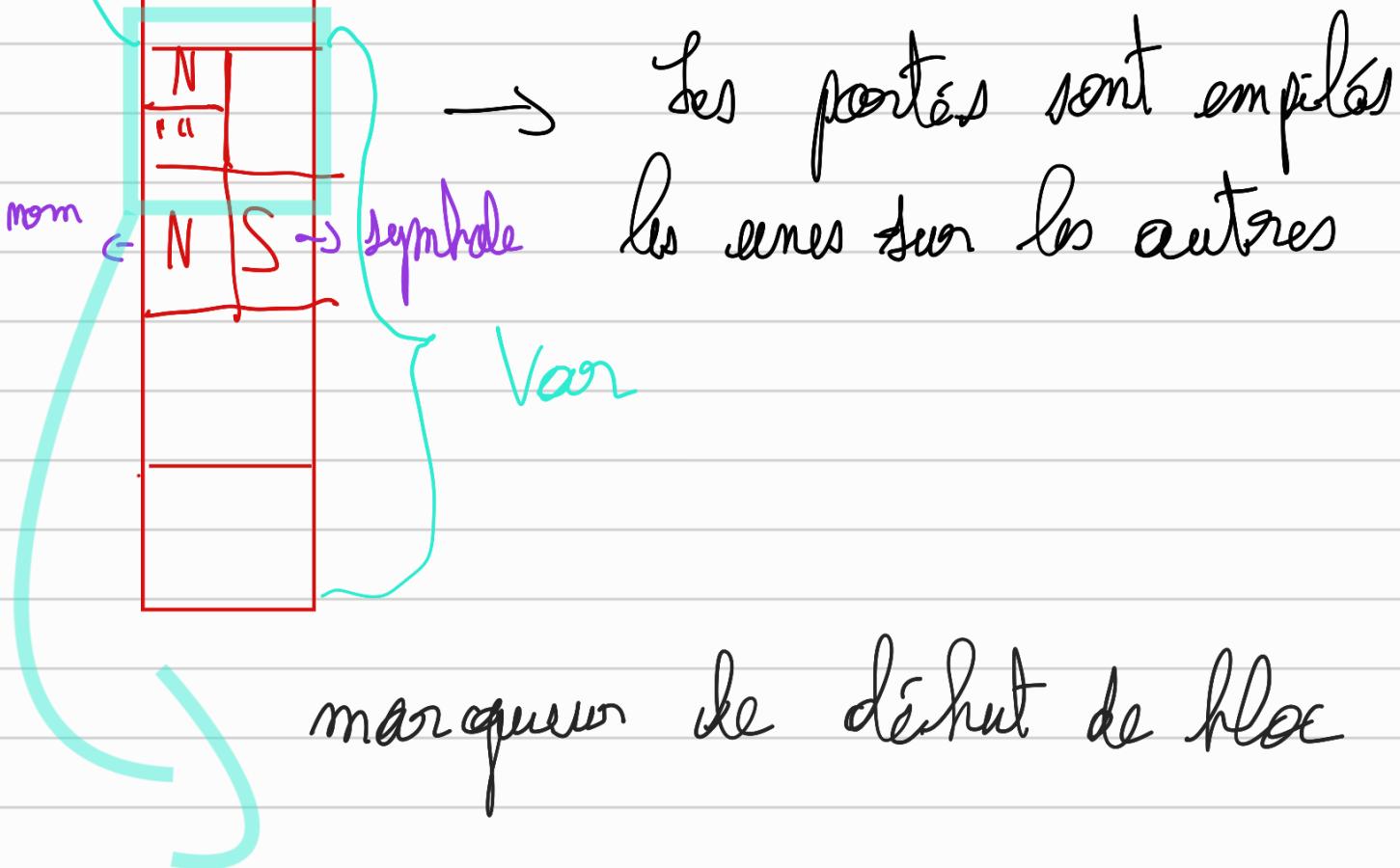
Pour  $i = \text{Sommet jusqu'à } 0$

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

| | return  $T[i].S$

Erreur Fatal

Von



Declarer ( nom )

Pour i = Sommet jusqu' à 0

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

Erreur Fatal ;

Si  $T[:].N = ^$

break ;

S = Nouveau Symbole

$T.push\_back(Nom, S)$

return S :

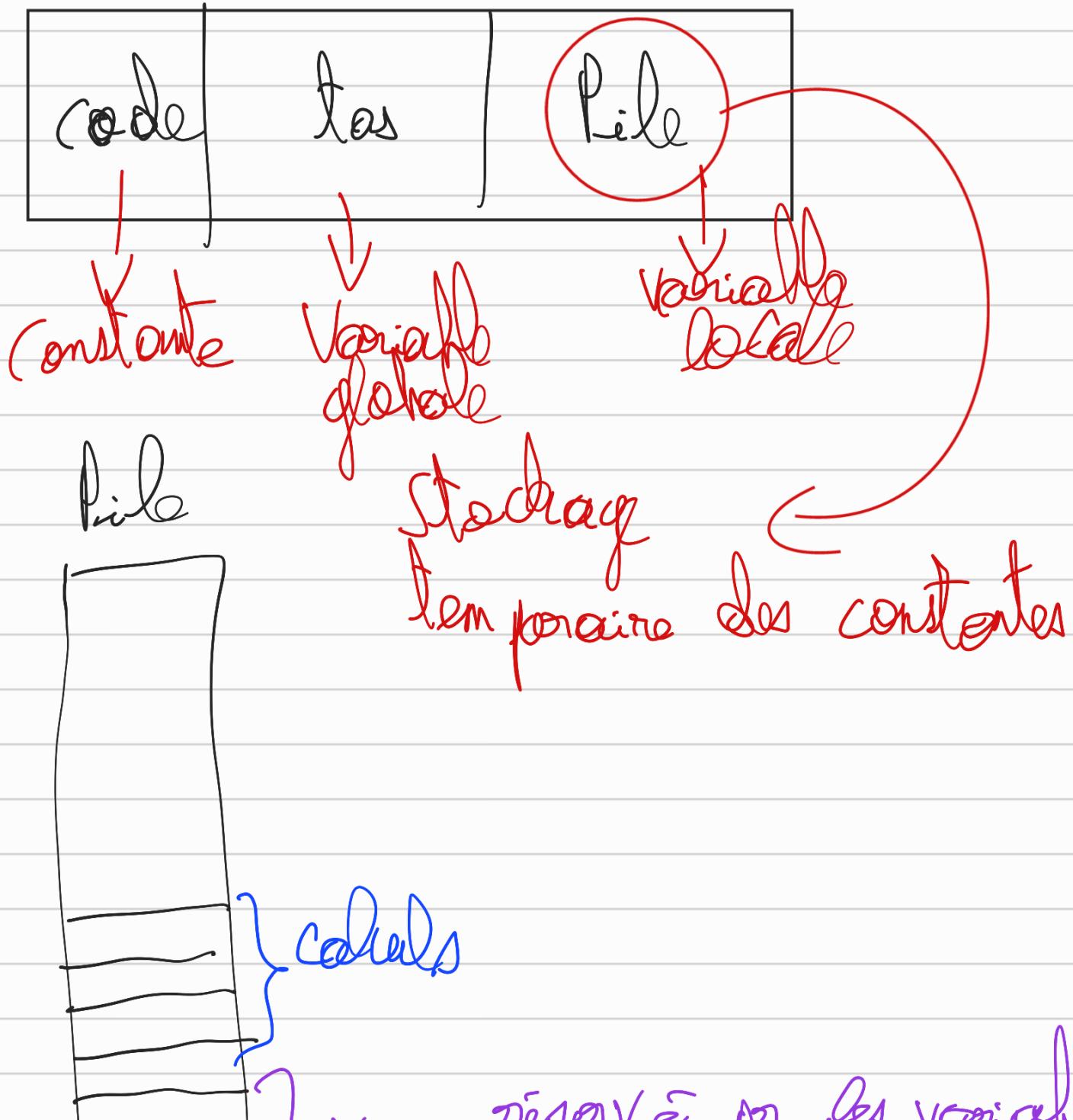
( ) { } ; , , , / \ )

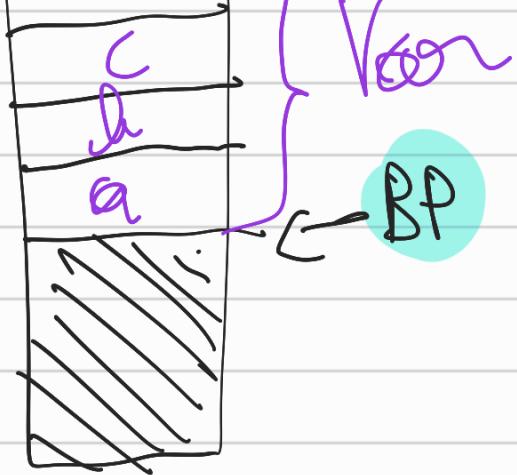
Begin() => T.push( , )

End() => T.pop Jusqu'à "

Cette structure est lors (+) utilisée  
par les compilateurs

En python les symboles seront des noms  
pour le moment.



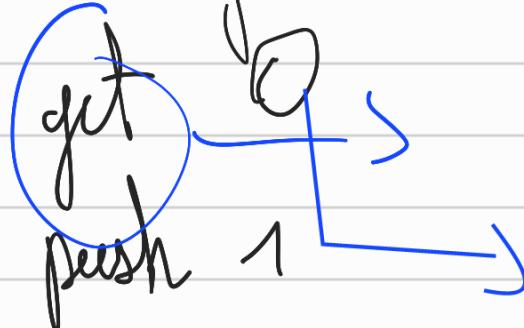


On met les variables sous le calcul, pour éviter que elles soient écrasées par les calculs

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?

indice de a dans b

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  
    " a "

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

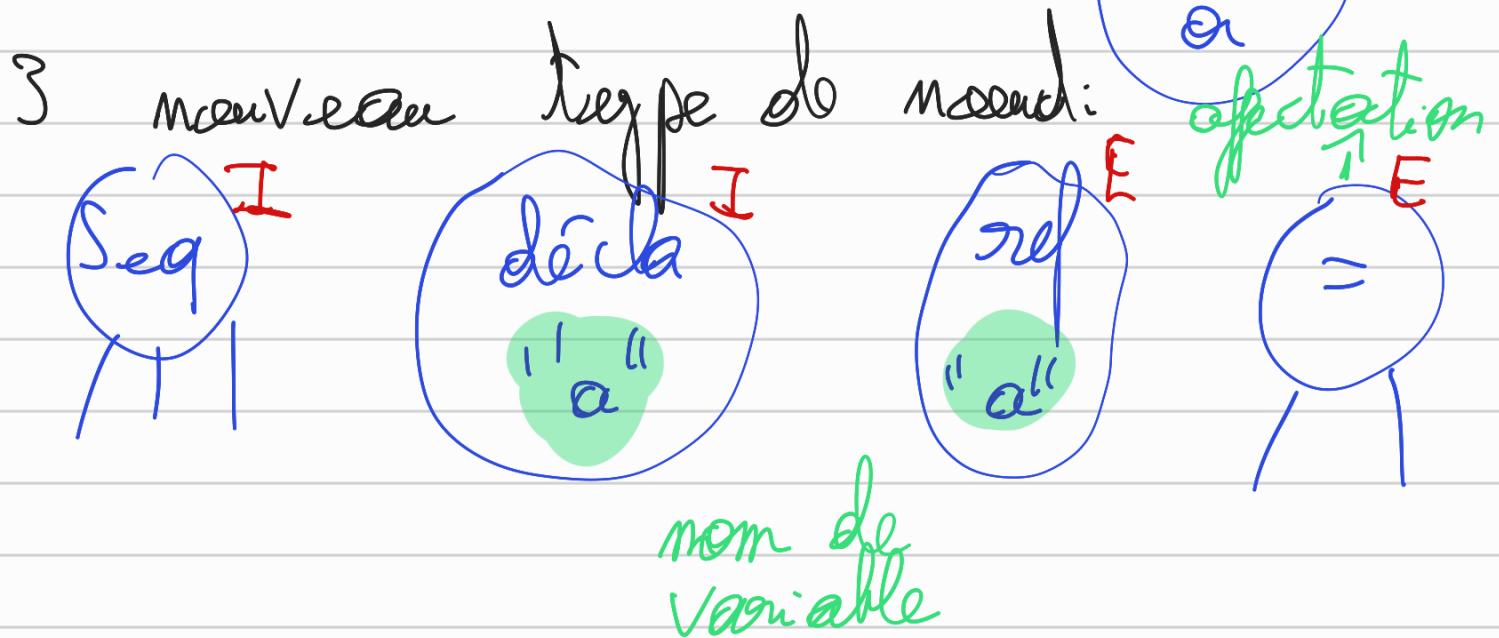
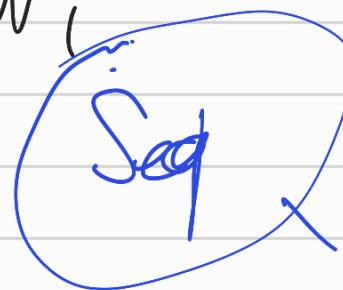
accept( too, ident );

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

} while( check(' ') ) ;

accept( ";" )

return V;



Expression : laisse une valeur sur la pile  
noeud décl => Pas de Générale

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

mb cas  
à réservé

local

$S = \text{VorLoc} ;$

globale

function

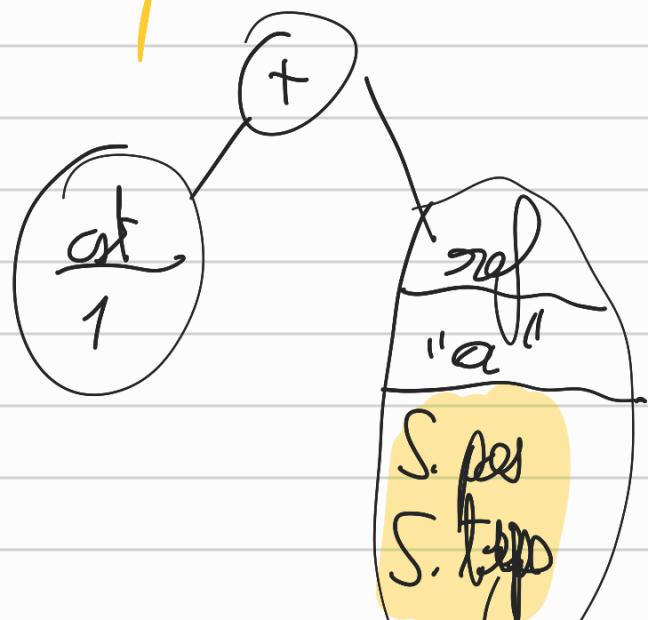
Case Node\_Ref:

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

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 = 0*i*

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

push 0  
dup  
set a  
drop 1

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

Compile()

while (token.type != EOF)

N = Anal-Sgn()

global

nbVar = 0*i*

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;

"marchera  
pas"

Conditionnel et Boucles:

if(E)  $\Rightarrow$  I

... } else if (check(tok\_if)) {  
  accept(tok\_if);  
  e = E();  
}

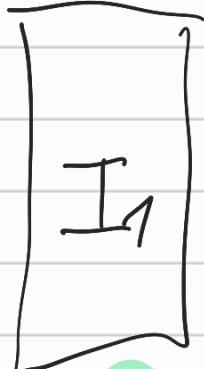
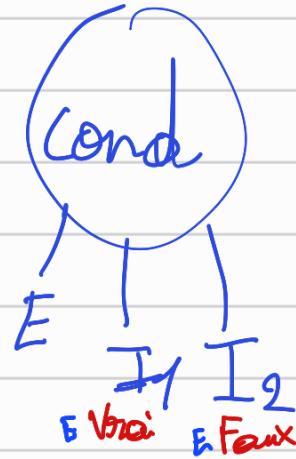
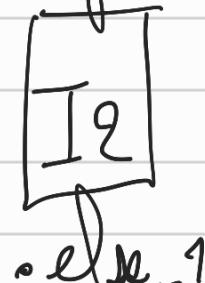
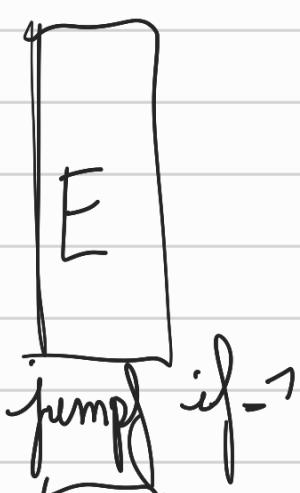
accept(tok\_if);

$i_1 = I(1)^o$   
 $\text{if}(\text{check\_tak\_else})\{$

$N = N_{\text{end}} | N_{\text{Cond}}, e_1, i_1 \}$   
 $\text{if}(i_2)\{$

$\text{AjouterEnfant}\left(N, i_2\right),$

return  $N^o$



Sont des instructions  
jusqu'au label  
spécifié

label

# 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

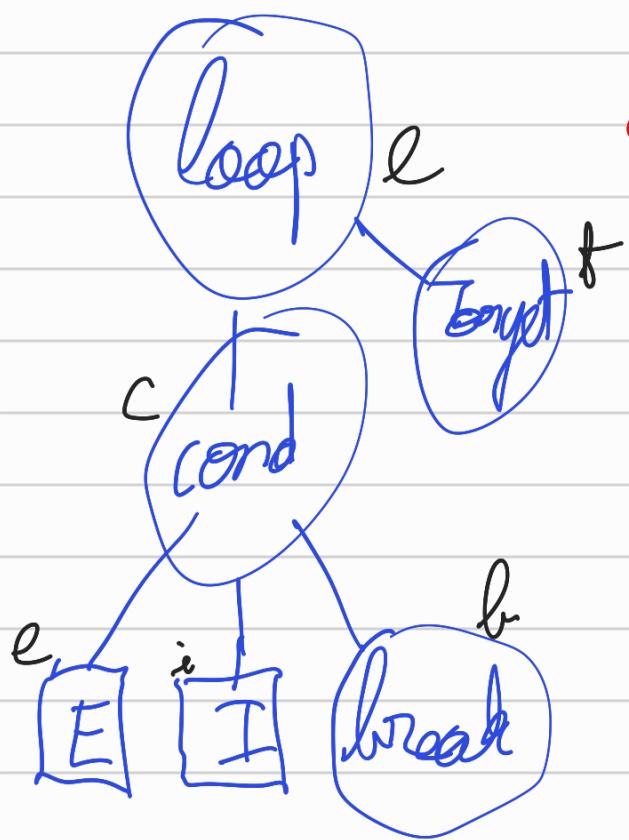
while(E) I

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 )  
    e = E(0)  
    accept( tok\_ )  
    i = I(1)

l = needed( NdLoop )

t = needed( NdTarget )

c = needed( NdCond )

b = needed( NdBreak )

Attaquer Enfant ( l, c )

|| (l, t)

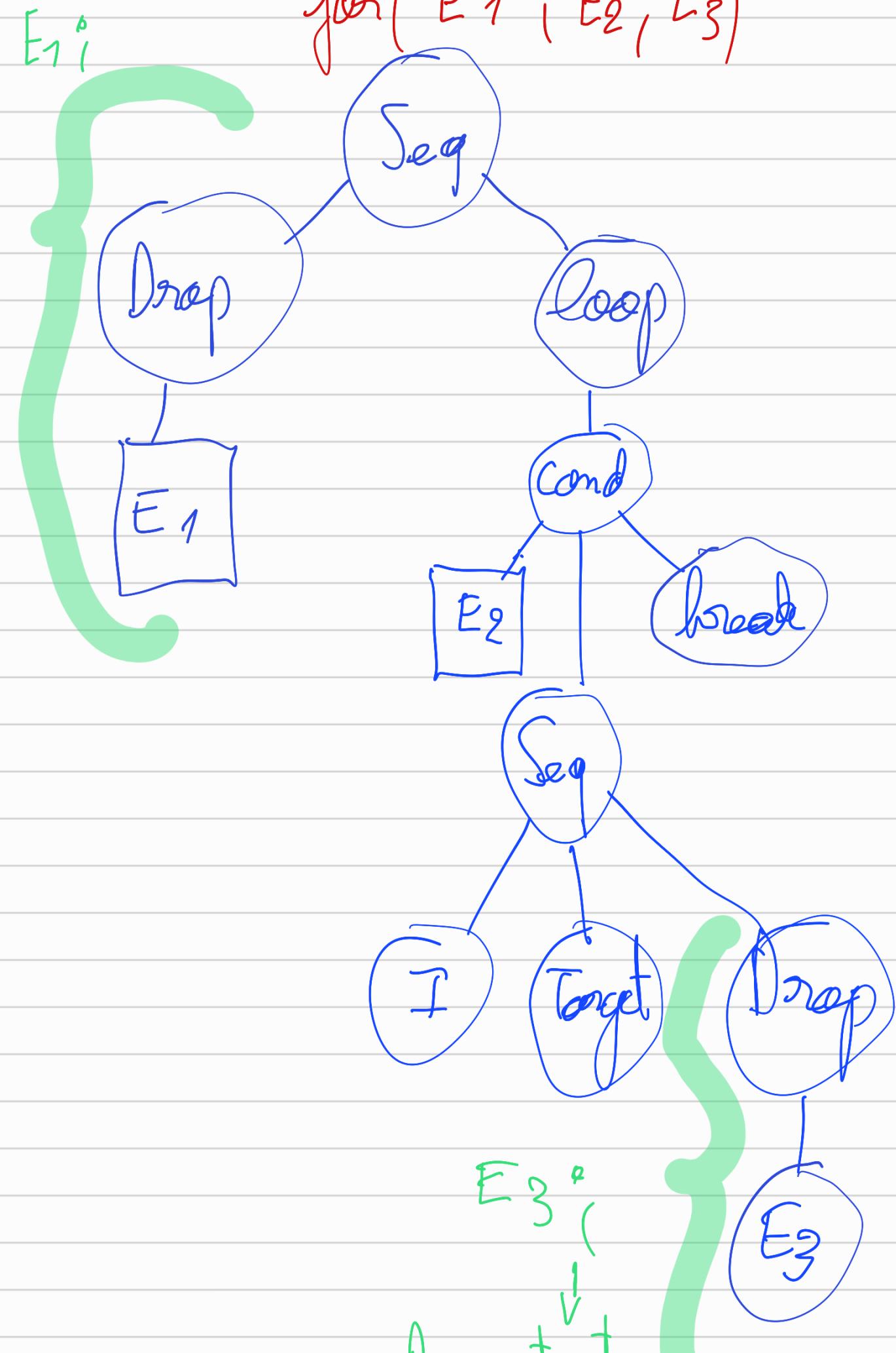
|| (c, e)

|| (c, i)

|| (c, b)

on essaye de compiler:

for( E<sub>1</sub>; E<sub>2</sub>; E<sub>3</sub> )



# Donc instruction

Gene Code  
↓

target →

print("a", lhl-cont)

break → print("jump", lhl-break)

Continue → print("jump", lhl-cont)

I ← break;

I ← continue;

Pour loop:

lhl\_debut = nblabel++

save\_cont = lhl\_cont

save\_break = lhl\_break

Save register  
Sauvegarde  
sécuriser de la  
touche principale  
la touche imbriqué

global lhl-cont = nblabel++

negative

~~lhl\_break = nb\_label++~~ de l'identification  
du label

print ("• l", lhl\_debut)

Pour chaque enfant  
genocode(enfant)

print ("jump l", lhl\_debut)

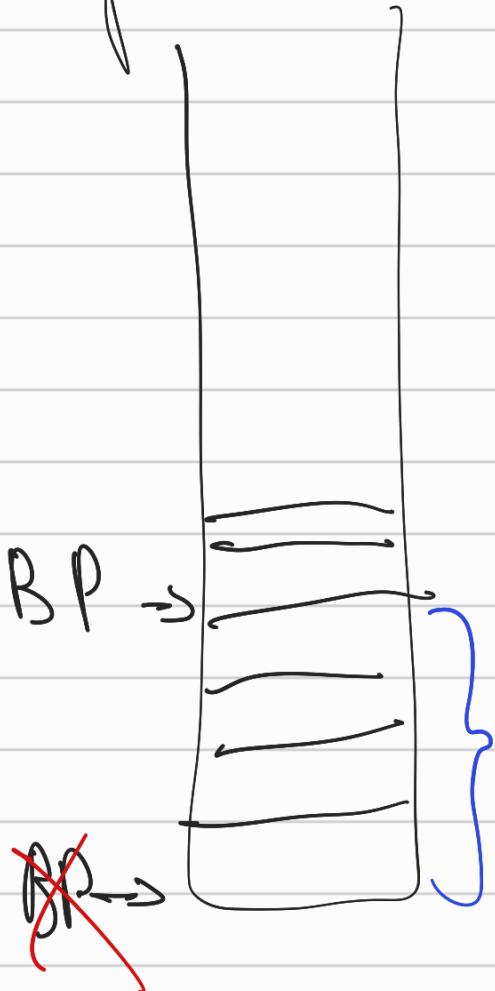
print ("• l", lhl\_break)

lhl\_cont = save\_cont  
lhl\_break = save\_break

\* loop réserve les identifiants des label, c'est pk un break ne peut-être qu'un enfant de loop car sinon son jump ne fera référence à rien.

Les fonctions :

alloca<sup>o</sup> de mémoire pour les variables d'une ft à chaque appel de la fonction.



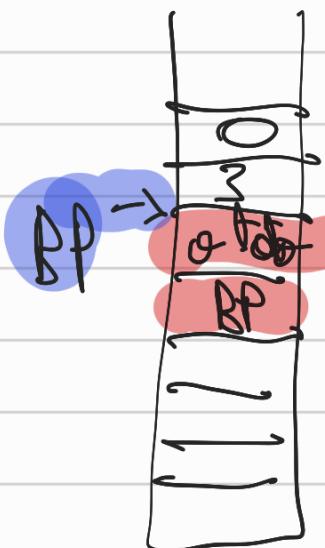
gréservation de mémoire pour les variables, à chaque appel de fonction BP se déplace.

instruction d'appel de fonction

prep tato  
call o

prep va empilé **2 valeurs** sur la pile  
RD et O la valeur de tato

Call 0 appelle la première instruction de toto et va déplacer BP:



→ on remplace cette adresse par l'adresse du return de la ft

Appel avec des paramètres

appel toto

push 5 → paramètre mis sur la pile

call 1 → où paramètre devra faire

Les paramètres seront gérés comme des variables local

Gén. d'un noeud fct pour l'analyse

# Syntaxique



Instruction  
dont décl.  
des variables  
locales

paramètre

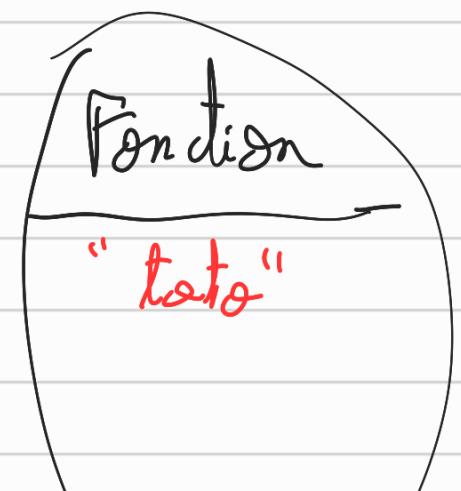
AnaSyn ()      } elle va analyser une  
return I ()      fonction F mtr

F <- int identificateur () I  
      "type"              "nom"              "parenthèse" "corps"

AnaSyn ()

accept ( tok\_int )

accept ( tok\_id )



nom = last. value ;  
 ↗ accept( tok\_.c ) ↘ Sauvegarde  
 accept( tok\_. ) ↘ du nom de la fd  
 i = I() ; ↘ Gestion  
 N = meand( Ndefonc, nom ) ↘ des  
 ↙ Ajouter Enfant (N, i) ↘ paramètres

Ajouter Enfant (N, i)  
 return N<sup>i</sup>

Nos programmes ne seront composé que de fonction.

## Gencode

## Ndefonc

print(".", Novaleur)

print("resm", No.Semval)

I()  $\Rightarrow$  Gencode(N.Dernier Enfant)

print("resm.0") ; son est les autres

Symbole  
 possèdent le nb  
 de var de la  
 fd, leur nom  
 et leur semantique

print("ret") dont des paramètres ce qui me générera pas de code

## Ana Sémantique

### Nde Fnc :

nbvar = 0;

begin () ;

Pour chaque enfant e

AnaSem(e)

→ I()  
et les  
paramètre

end () ;

S = declare (N.valeur)

S.type = Symbole Fonction;

S.nvars = nbvar - NmbEnfant - 1;

NoS = S; on met le Symbole  
dans le noeud fonction

Symbole

correspondent  
à une ft

start

prep main Mémoire pour les var loc et

de cette manière  
on alloue de la

call 0      non pour les paramètres qui  
halt          sont déjà push à l'appel  
                  de la fonction.

Dans la table de traduction des symboles, toutes les fonctions seront au  $\oplus$  bras, et toutes les variables en bout

## Autre Version de AnaSyn()

AnaSyn()

accept ( tok\_int )

accept ( tok\_ident )

$N = \text{Nœud} (\text{NDefc}, \text{last}, \text{value})$

accept ( tok\_C )

while( check( tok\_int ) )

Gestion

accept ( tok\_ident )

AjouterEnfant (  $N$ , Nœud ( NDefc, last, value ) )

Remarque

id / check( tok\_Vocab ) ]

break;  
continue;

accept ( tok - )

AjouterEnfant ( N, I () )

Return N<sup>i</sup>

I ← return E ;



même geneicode et mème que debug  
pareil pr AnaSyn



Gestion des appels de fonction :

I () → E () → P () → S () → A ()

Dans P1 si tous les appels à A() sont des appels à S()

$S() \leftarrow A('')?$



$S()$

$N = A();$

$\text{if}(\text{check}(tak\_C)) \{$

$N = N\text{eend}(N\text{de type}, N)$

$\text{while}(!\text{check}(tak\_)) \{$

$\text{AjouterFinpt}(N, E(0))$

$\text{if}(\text{check}(tak\_)), \{ \text{break}; \}$

$\text{accept}(tak\_virgule); \}$

}

Genocode de NdeAppel

$\Omega(N \rightarrow f \rightarrow (N\text{de}))$

if ( $\text{No. on } [0]$ ). type !=  $\text{NaRef}$ )

ErrorFatal()

if ( $\text{No. on } [0]$ . So.type !=  $\text{NaFunc}$ )

ErrorFatal()

print ("prep", NoValue)

Pour chaque enfant sauf premier

GenCode(e)

print ("call", nbEnfant - 1)

