

Compilation

(machine virtuel,
simulation)

Compilateur ≠ interpréteur



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

Ex: Code source → binaire

| ↴ autre langage
↳ Assembleur

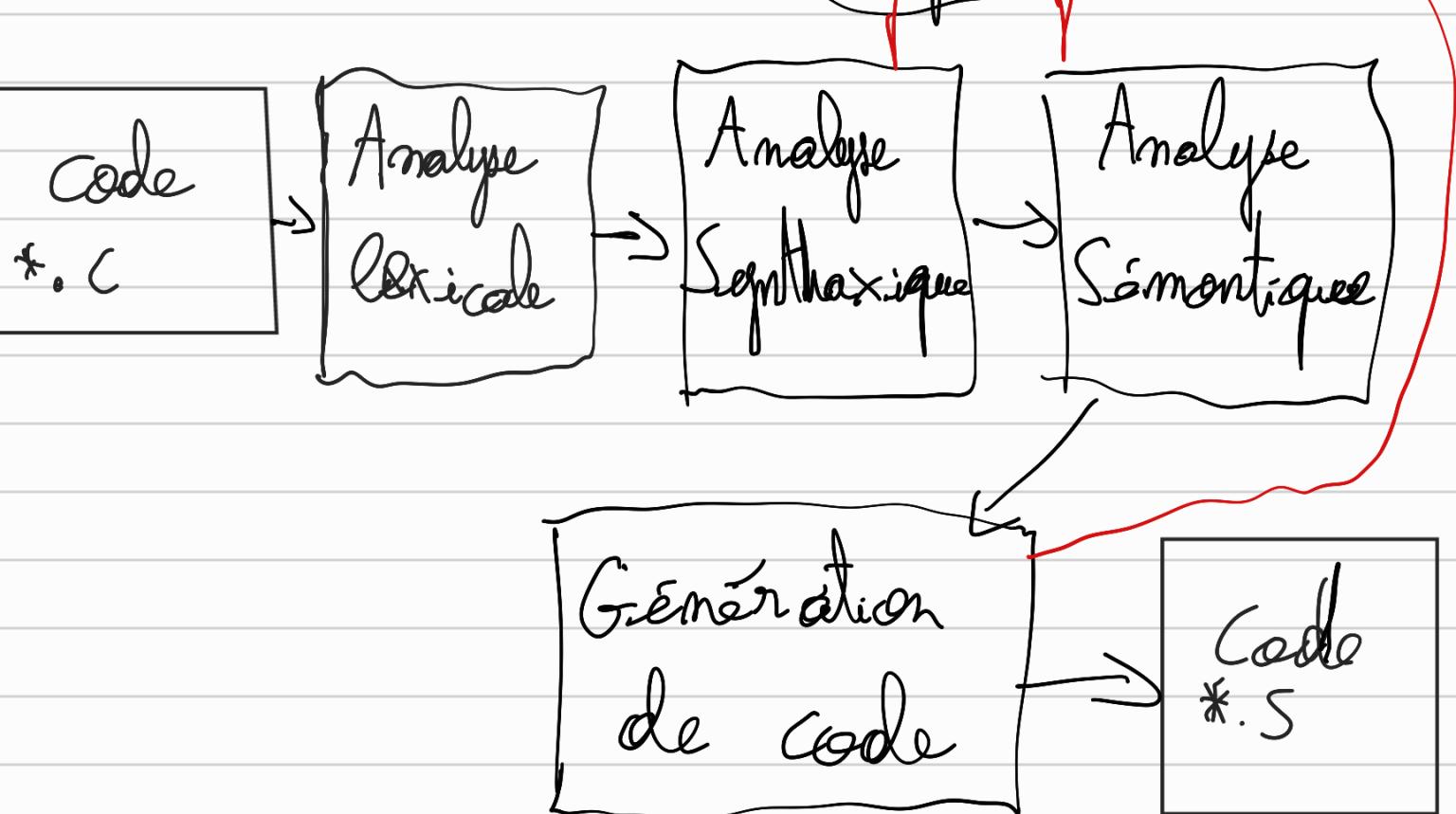
prend le code source (ex: python)
et exécute sa sémantique

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

Signe
moins

Contento entière	Signé étoile
45	O

Identifier
"abc"

↑
santé nation ou négation ?
⇒ Analyse Septique

En C ces pointeurs sont des structures

Valueur prendra 2 types : entier ou chaîne

steck taken {

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

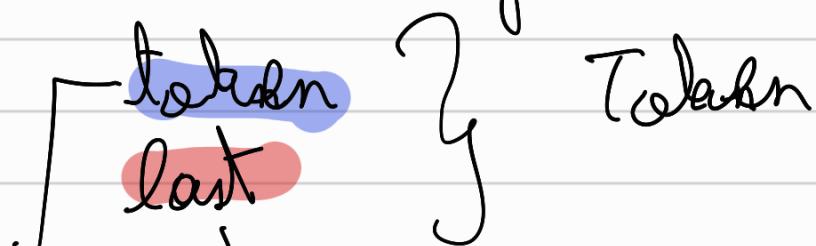
Bande et condition

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 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:

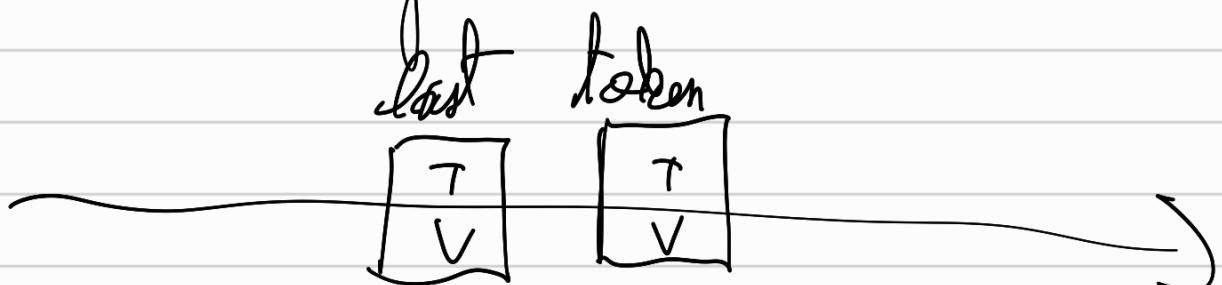
+ - * / % ! &
< = > = == !=

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



fil

AnaSyn()

token doit être le premier
token d'un atome valide

Simon Erreun Fataal

return E()

Nœud Atome()

atom complexe

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

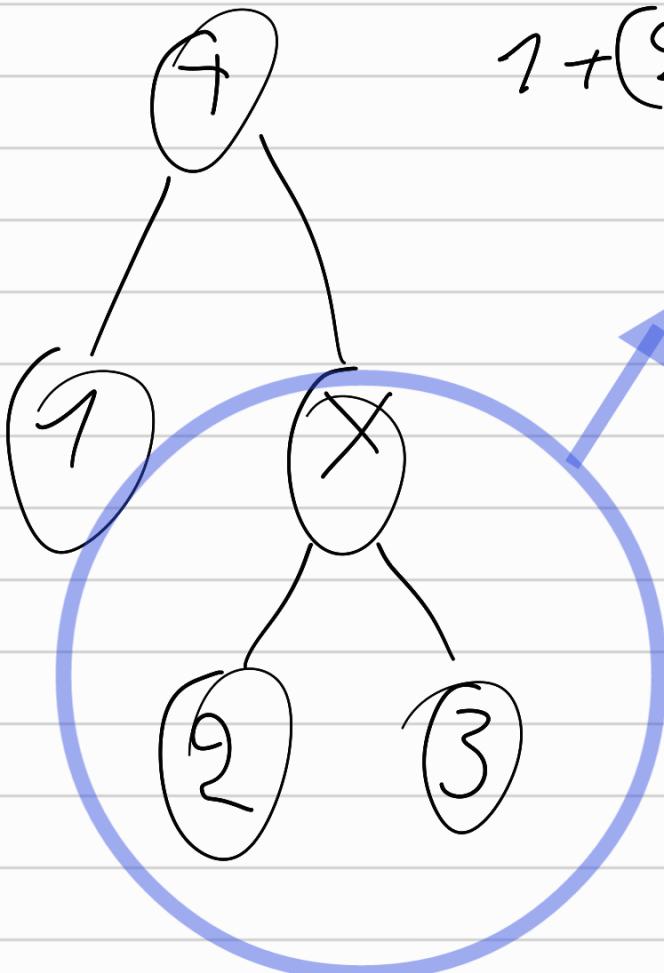
atome

$$1 + 2 \times 3$$

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

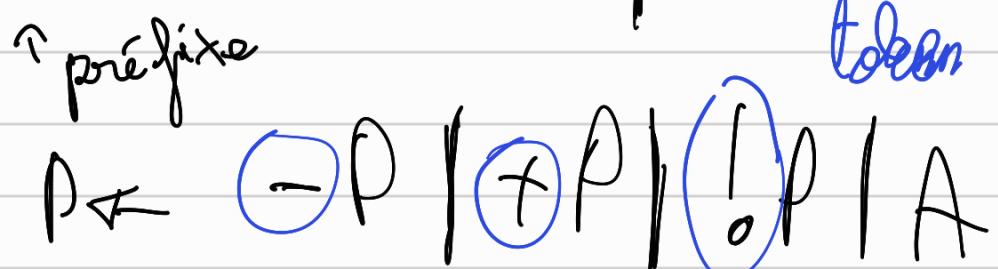
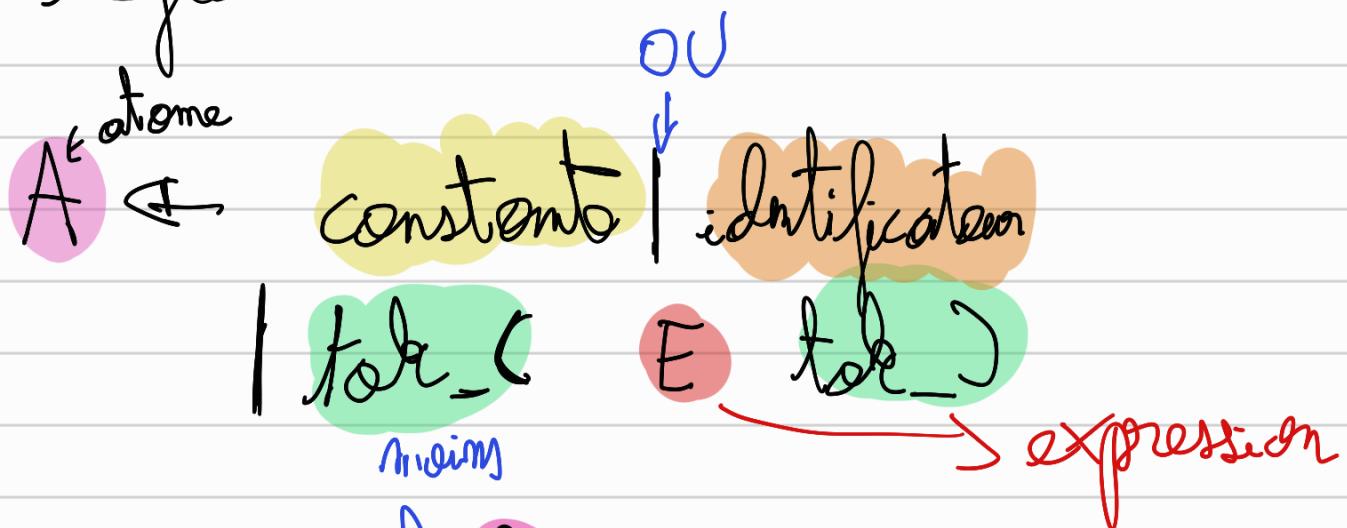
Last ← dernier token de l'atome

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 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 :

$(1 - 2) - 3$

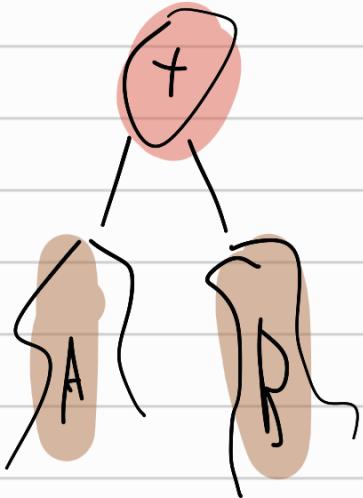
associativité à gauche

$2^1(3^14)$

associativité à droite

Opération d'affectation : associativité à droite

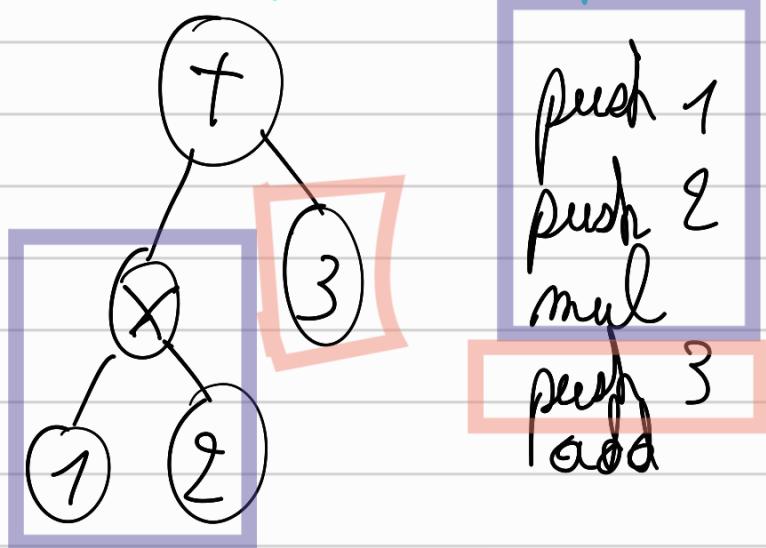
$a = [n = c]$ implique



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()

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

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

} return N



Opérations

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

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

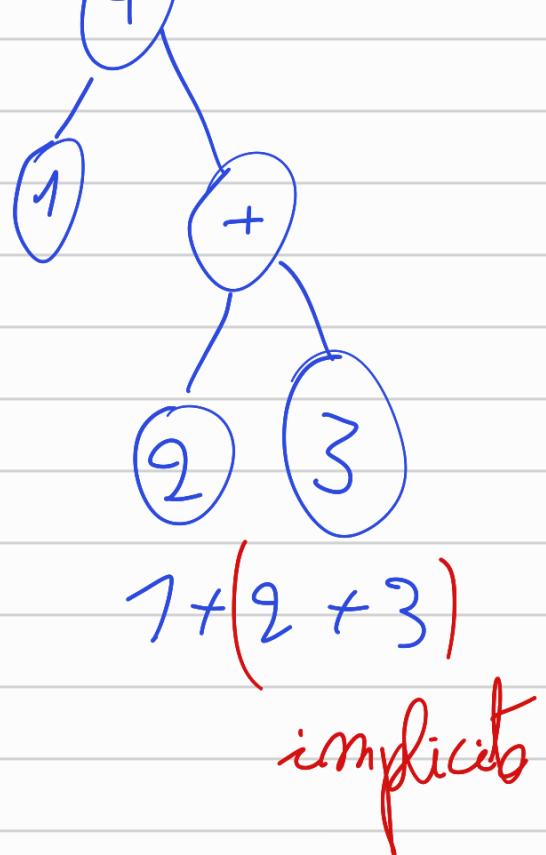
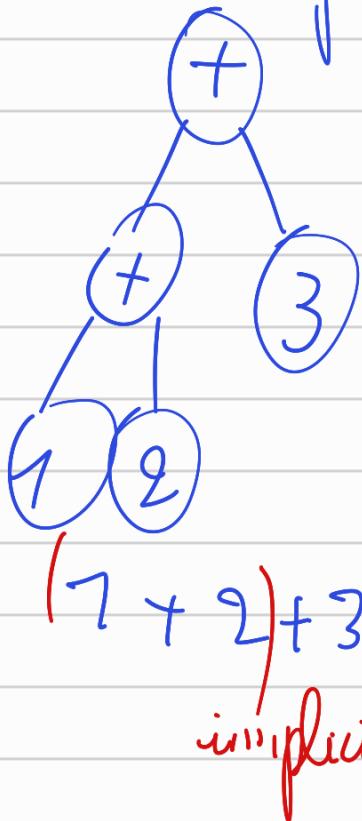
~~node_stoile~~ $\rightarrow \{ \text{nde} = \text{node_mult}$
 $, P=2, \text{Add}=0 \}$

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'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

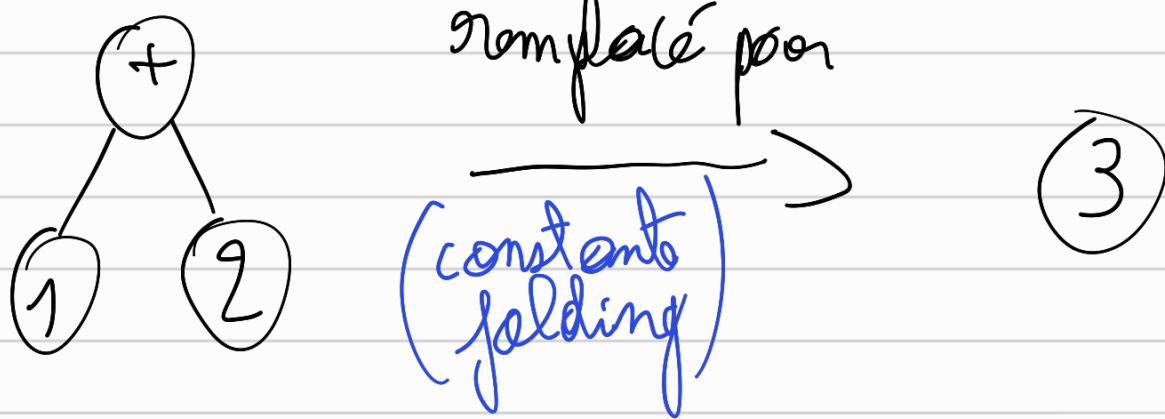
On évite le code inutile créer
par le compilateur

[push □] → [drop n - 1]
drop n

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

Optimisation mm . txt :

Optimisation par arbre.

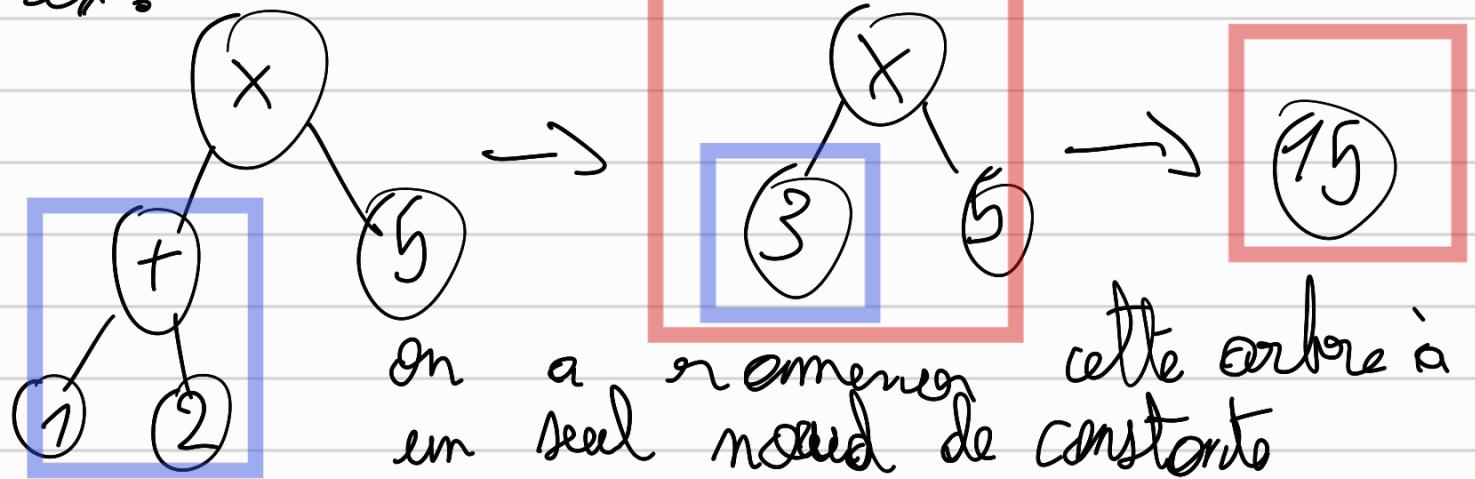


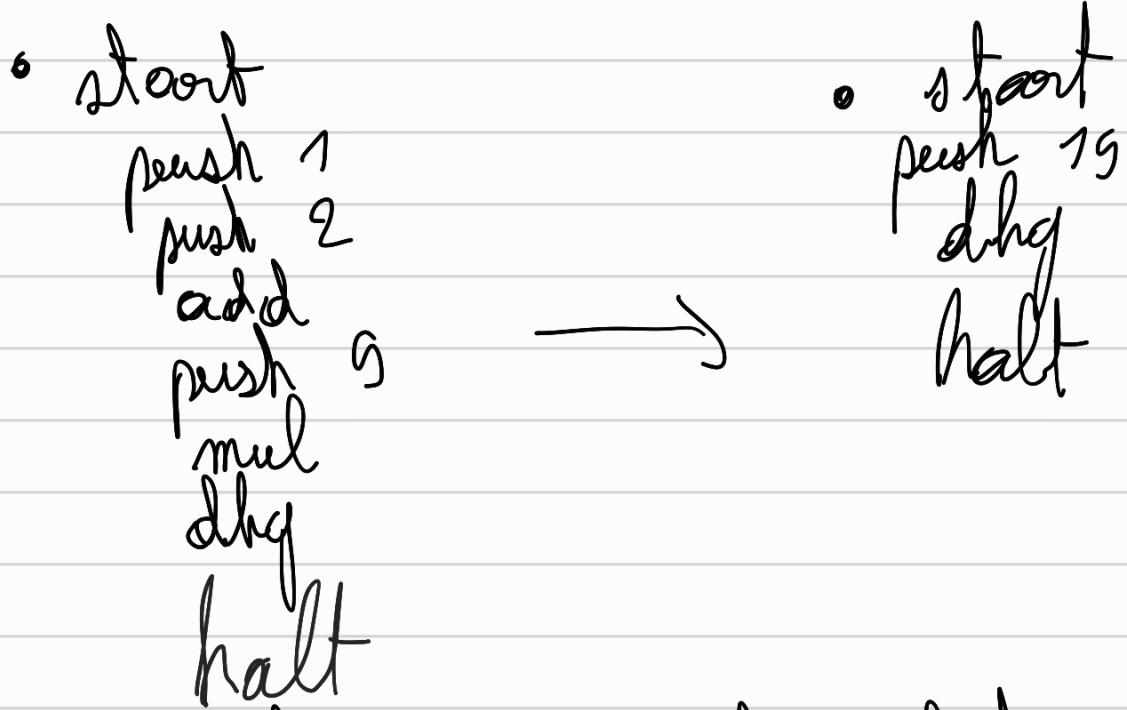
appel d'une fn peut être + car les que d'exécution sont code ex: max

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

Optimiser parcours contre en s'appelant récursivement sur les enfants

ex:





On traite les nœuds enfants avant les nœuds parents.

Optimisatio 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, tipe, et mons

instruction \neq expression

ex:

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

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 )
}

```

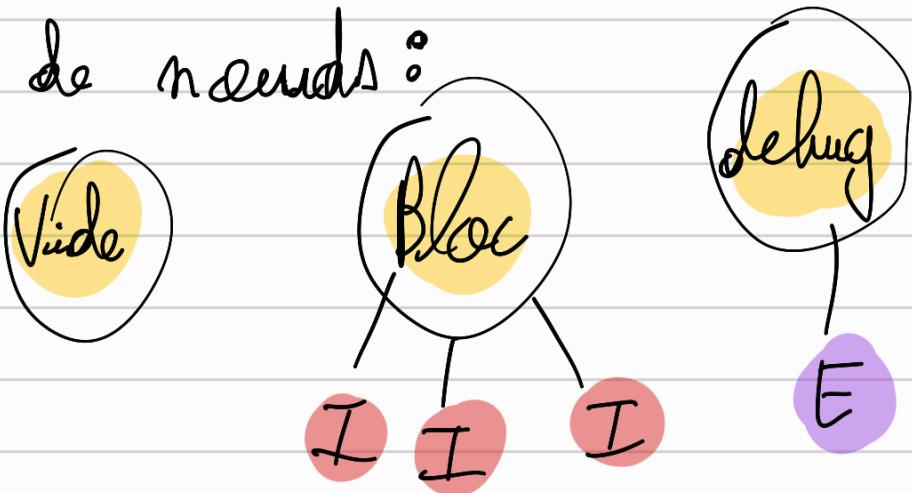
↗ in Vide
 ↗ on deit
 ↗ gronkogen
 ↗ in noend

```

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

```

Nieuwean type de noends:



```

elif( check(tot_debug) ) {

```

$N = E[0]$

accept("0")

return Noend(NdDebug, N)

Probeer een totale
sint-totra?

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 instruci° des expressions car on enverra de la pile la valeur mise par l'expression de cette 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])

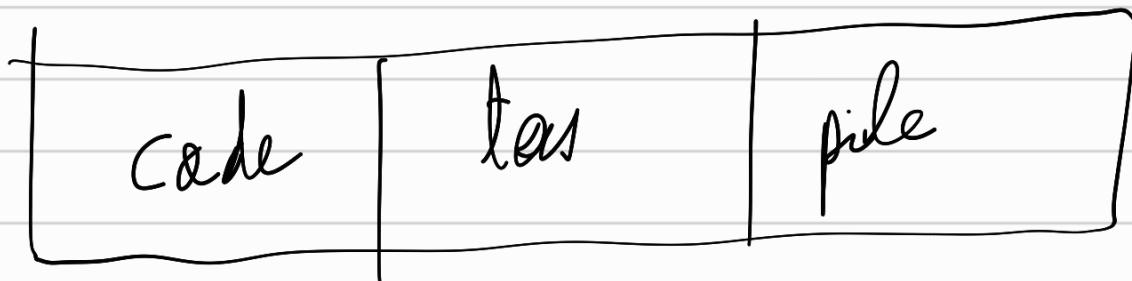
print("dby") → enlève la
valeur de l'
expression sur la pile



: Genicode(Enfant[0])

print('drop 1') ← pareil

Schéma mémoire :



Les Blocs délimitent la portée des

Variables.

{ $1 + 1 =$

int a;

int a;

a = 3

int b;

b = 5;

a = 7;

int a;

a = 5

int a;
déclaré le + proche

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 bloc actuel

↳ Symbole

- Ecrire

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
que une fois par bloc.

Déclarer (nom)

si nom exist in VarTab

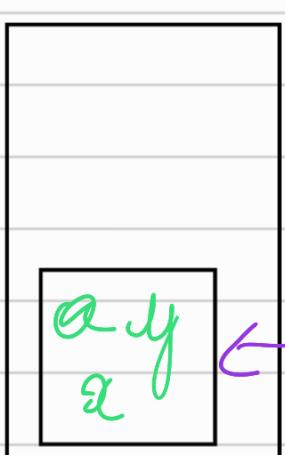
Erreur

S = nouveauSymbbole

VarTab[nom] = S

return S

pile de
table de
Hachage

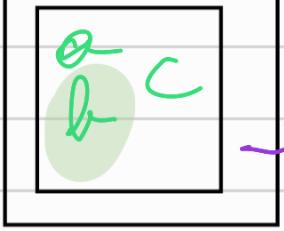


Pile

, End()

qd bloc fini on enlève
cette table de la pile

a y
x



Une table pour Bloc

Begin() → Voor-push(Ø)
End → Voor = pop()

Chercher(α) : variable qui est actuellement
 α , 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)

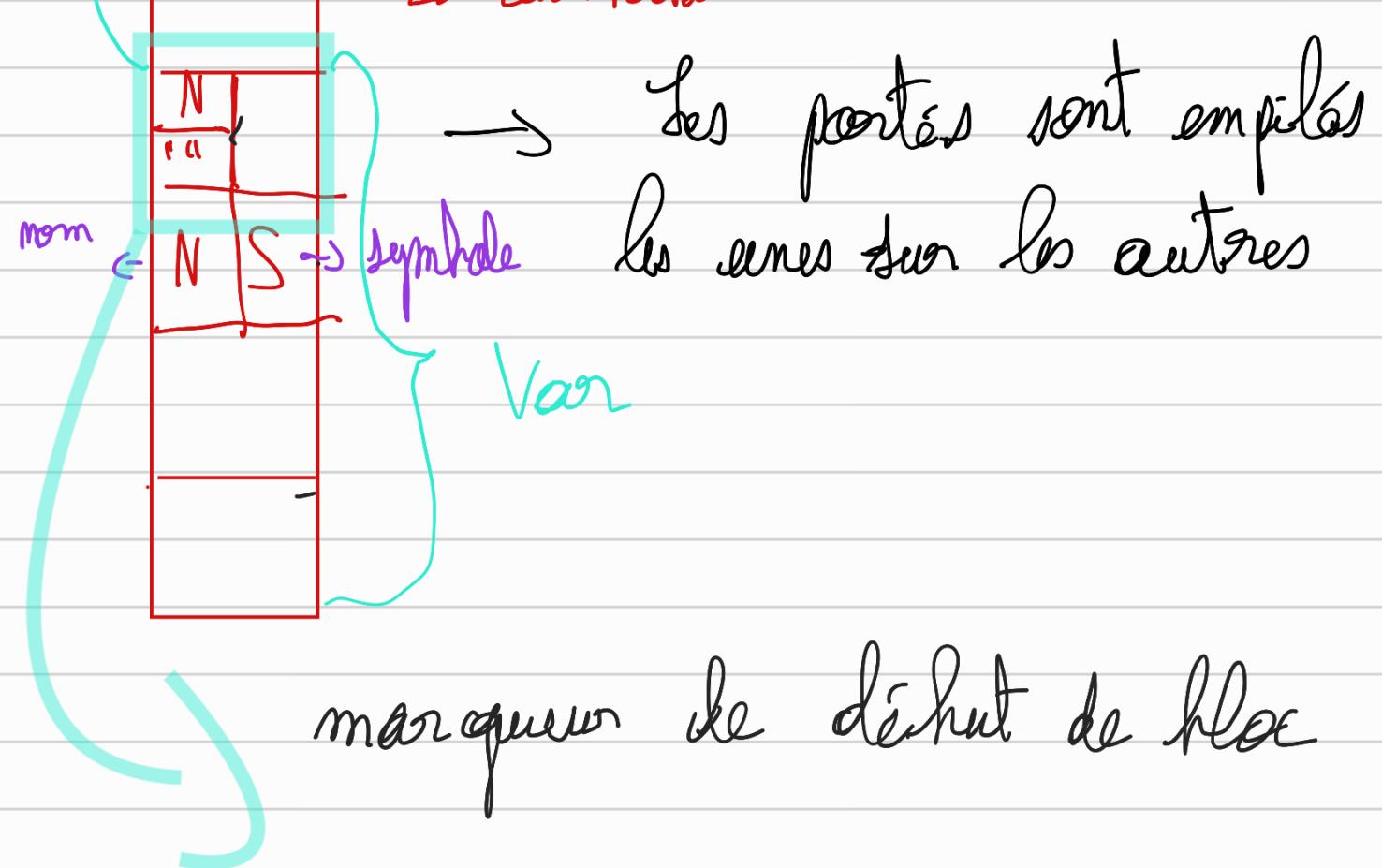
Pour $i = \text{Sommet jusqu'à } 0$

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

| | return $T[i].S$

Erreur Fatal

Van



DeJearon (mom)

Pour $i = \text{Sommet}$ jusqu'à 0

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

Emerson Fatal

Sie $T[i], N =$

break ^/

Σ = Newean Symbole

T_i.push-back(Nom, S)

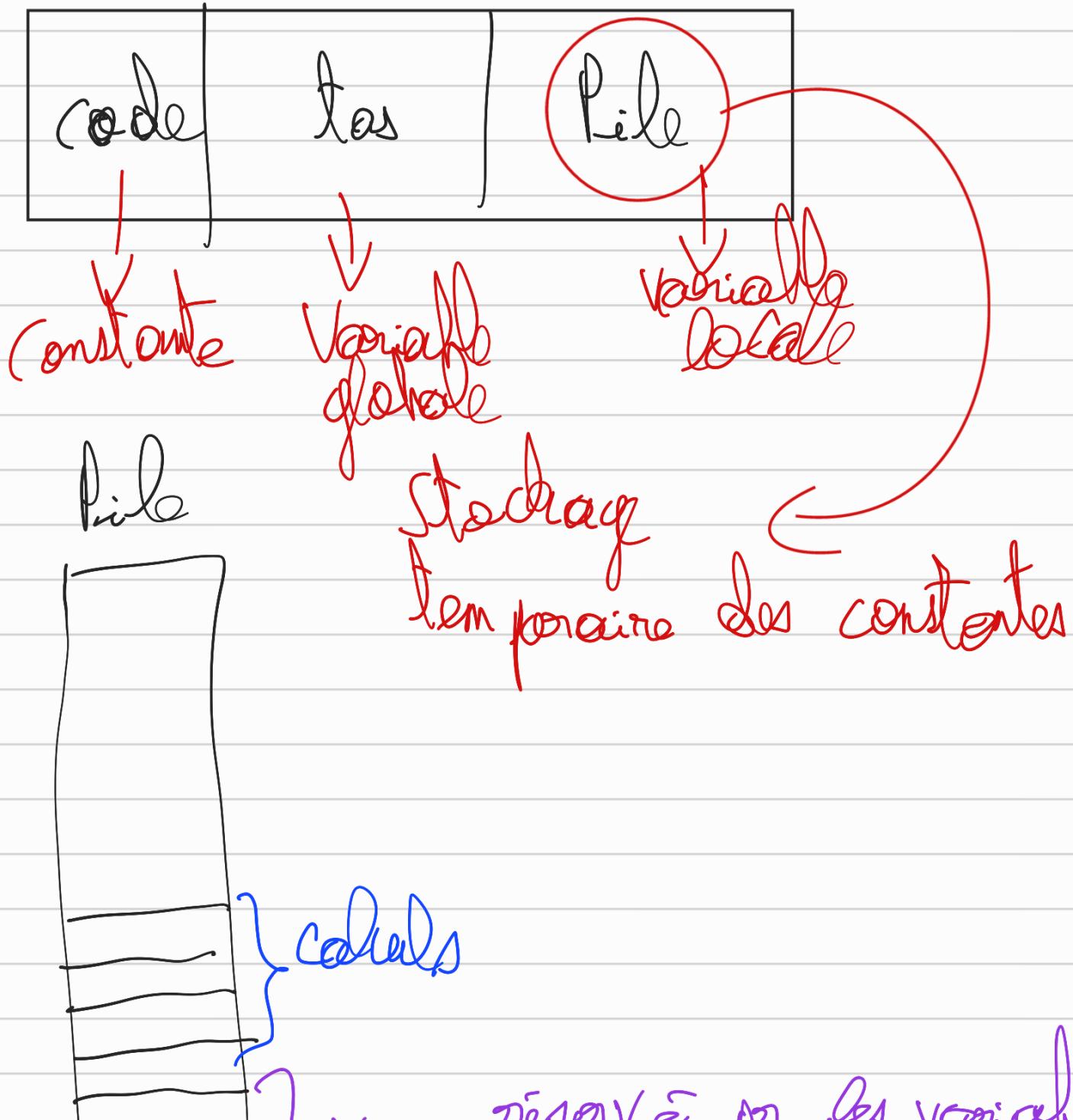
return } .

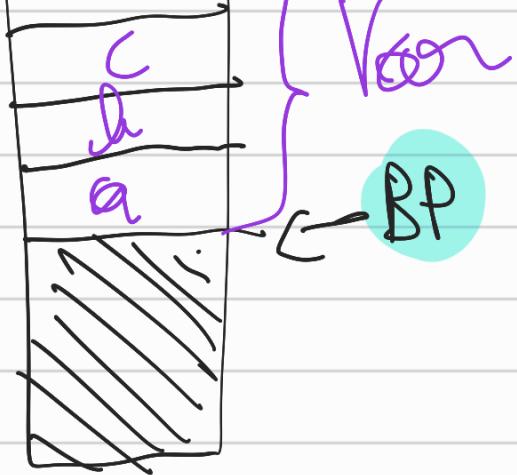
Begin() => T.push(,)

End() => T.pop Jusqu'à "

Cette structure est lors (+) utilisée
par les compilateurs

En python les symboles seront des noms
pour le moment.



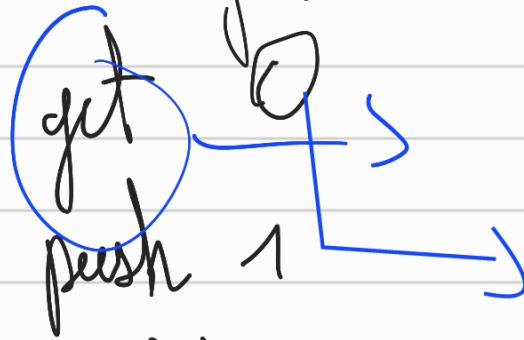


On met les variables sous le calcul, pour éviter que elles soient écrasées par les calculs

Il faut donc réservé suffisamment de case pour toutes les variables locales du programme.

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

Pour faire : $a + 1$



comment get trouver a?

l'indice de a dans b

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
 " a "

 } else if (check (tok_ident)) {
 N = Need (Node_Seq) ;
 do {

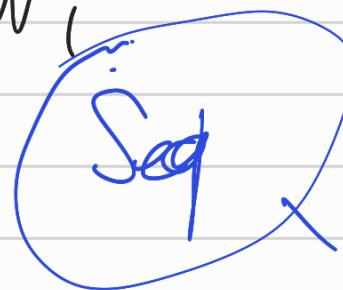
accept(too, ident);

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

} while(check(' ')) ;

accept(";")

return V;



3 Noeud valeur type de noeud: expectation

Seq I declar E ref E = E

nom de variable

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 & réservé

local

$S_{\text{steps}} = \text{VorLoc};$

global

function

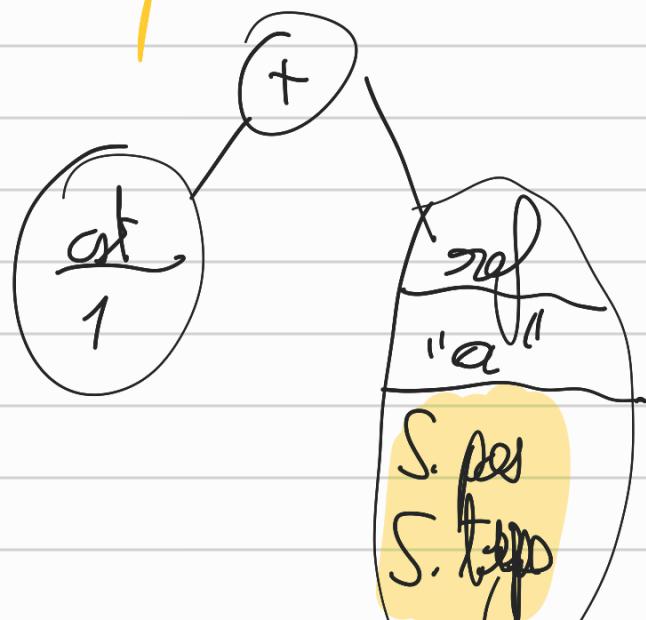
Case Node_Ref:

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

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*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
même 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();
}

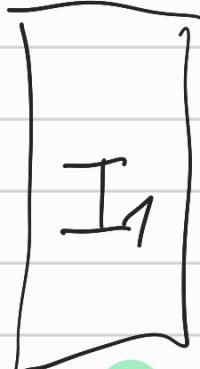
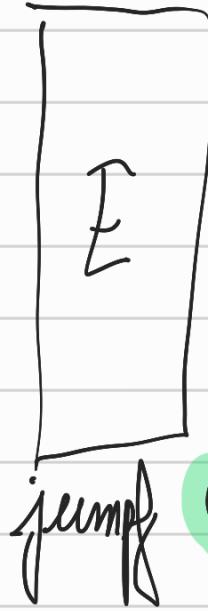
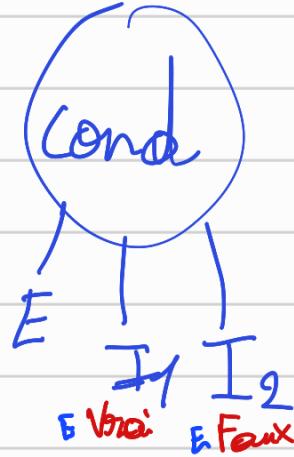
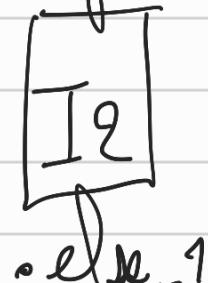
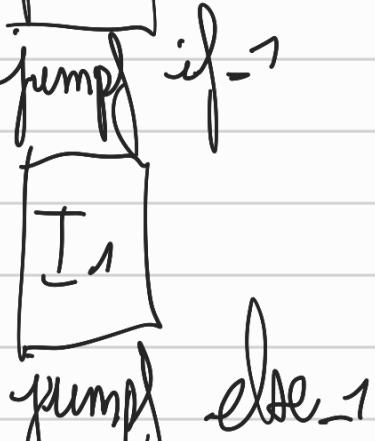
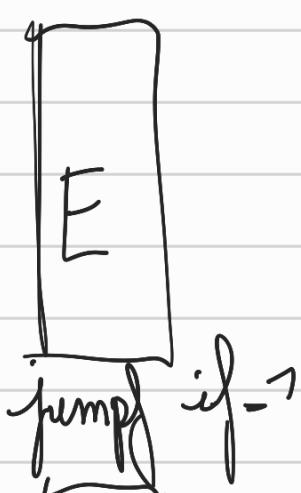
accept(tok_if);

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

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

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

return N^o



Sont des instructions
jusqu'au label
spécifié

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

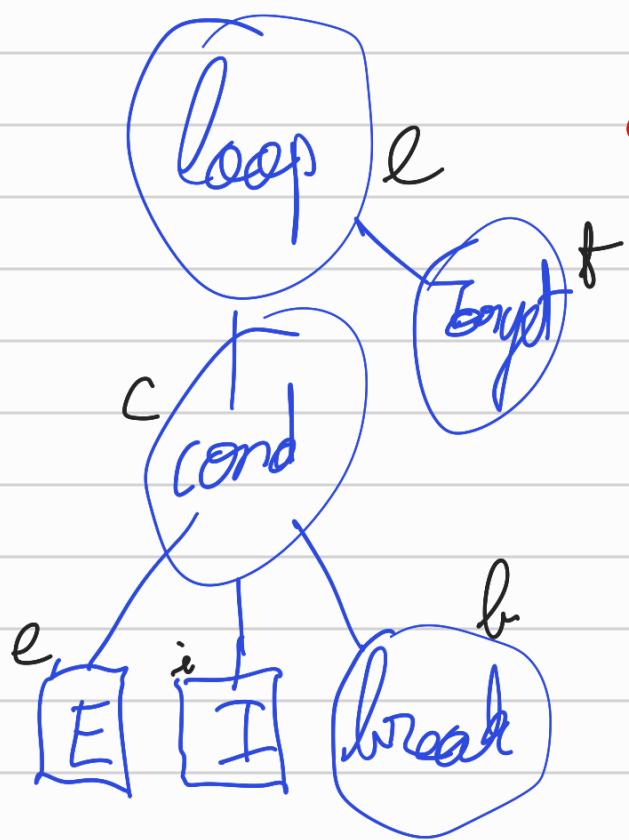
while(E) I

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;
}

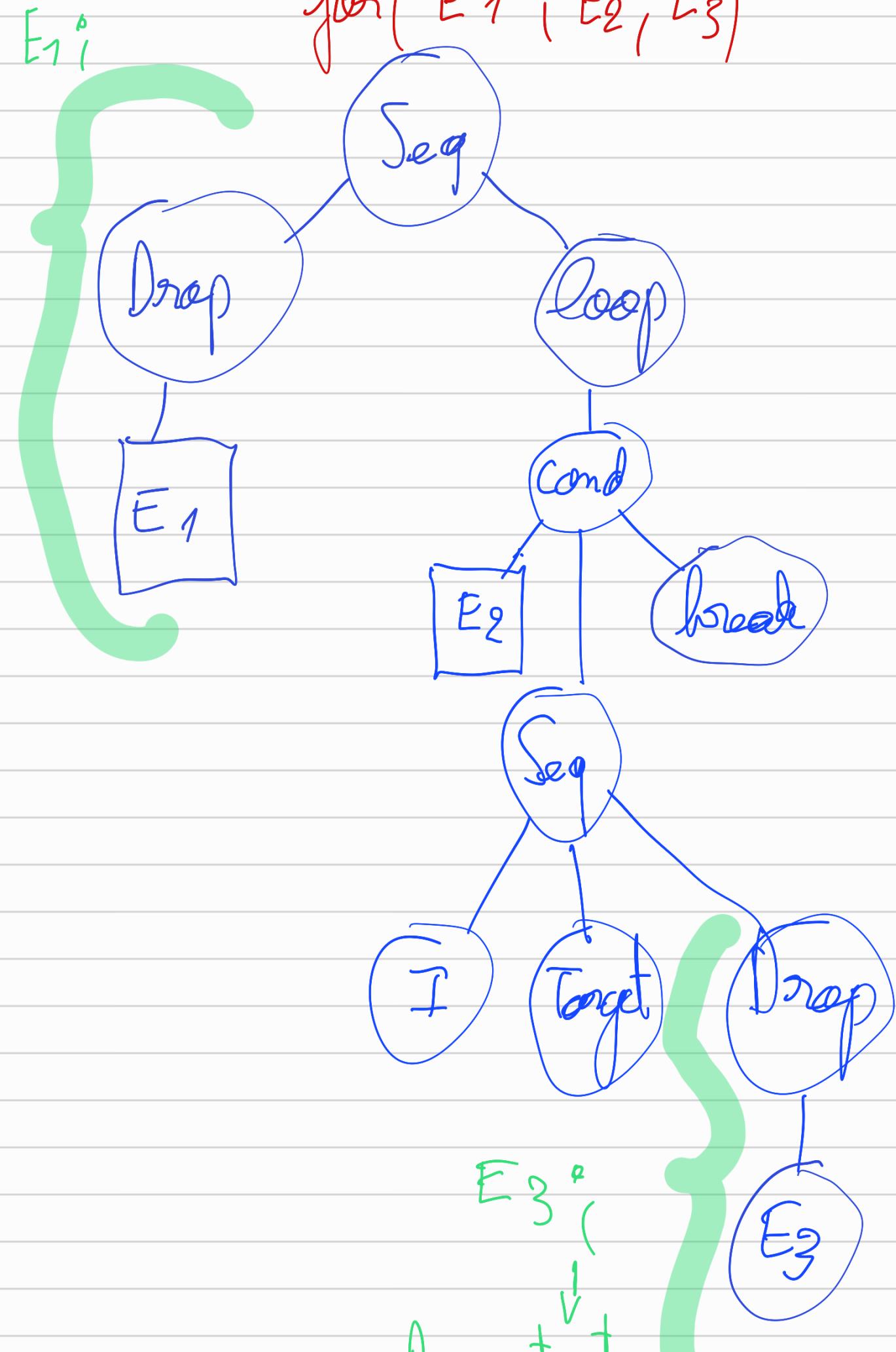
```

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

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

on essaye de compiler:

for(E₁; E₂; E₃)



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)

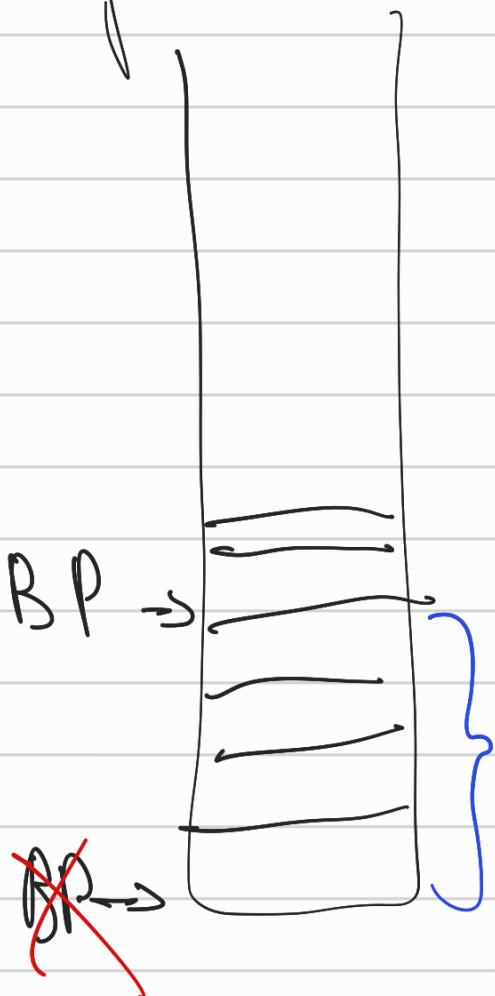
print ("• l", lhl_break)

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^o de mémoire pour les variables d'une ft à chaque appel de la fonction.



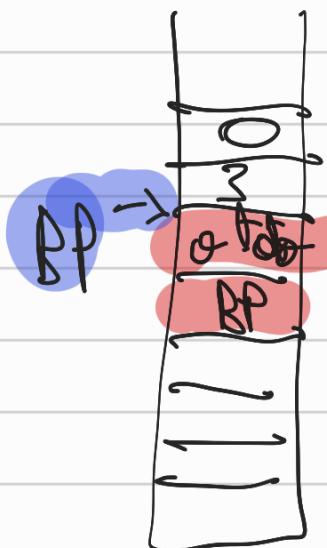
gréservation de mémoire pour les variables , à chaque appel de fonction BP se déplace .

instruction d'appel de fonction

prep tato
call o

prep va empilé 2 valeurs sur la pile

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 → 1er paramètre sur la pile

Les paramètres seront gérés comme des variables local

Gén. d'un noeud fct pour l'analyse

Syntaxique



Instruction
dont décl.
des variables
locales

paramètre

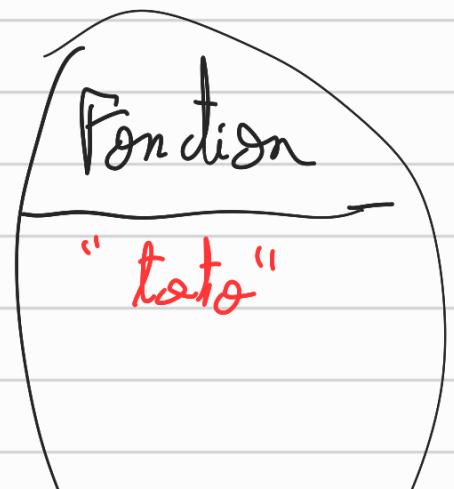
AnaSyn () } elle va analyser une
return I () fonction F mtr

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 Nⁱ

Nos programmes ne seront composé que de fonction.

Gencode

Ndefonc

print(".", Novaleur) ↗ Symbole
 print("resm", No.S.mvar) ↗ perséant le nb
 ↙ de var de la
 fd, trou pour
 une Sémantique
 $I() \Rightarrow \text{Gencode}(N, \text{Dernier Enfant})$
 print("resto") ↗ : son tt les autres

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

start

De cette manière
on alloue de la

prep main Mémoire pour les var loc et

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 Nⁱ

I ← return E ;



même geneicode et noménd que dehug
pareil pr AnaSyn

A diagram illustrating a stack frame. It shows a rectangle divided into two horizontal sections. The top section is labeled "E" and the bottom section is labeled "ret". A red bracket on the right side of the rectangle is labeled "GeneCode".

Gestion des appels de fonction :

I () → E () → P () → S () → A ()

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

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



$S()$

$N = A();$

$\text{if}(\text{check}(tak_C)) \{$

$N = N\text{eend}(N\text{deType}, N)$

$\text{while}(!\text{check}(tak_)) \{$

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

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

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

}

Genocode de NdeAppel

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

if (N_0 .on[0].type != NodRef)

ErrorFatal()

if (N_0 .on[0].S0.type != NodFnc)

ErrorFatal()

print ("prep", N_0 .valeur)

Pour chaque enfant sauf premier

GeneCode(e)

print ("call", nbEnfant - 1)

Symbole n'a que 2 types
fonction et valeur

Rendre le

22/10/2023

93h 59

On peut faire un
rendu

- 95 pts par jour de retard

Envoyer par mail

Code :

- archive zip ou tar.gz
- README :
 - Savoir comment lancer le compilateur
 - les bibliothèques utilisées

Code à compiler, donné par le nom d'un fichier .c dans l'entrée standard

"Il ne doit pas devoir aller dans le code pour tester"

Code compiler, dans un fichier ou dans la sortie standard

Pas de plantage ou de code faux

↳ Utiliser des messages d'erreur personnalisé expliquant l'erreur

Expliquer à quel point on est dans la réalisation du compilateur :

- instructions

la syntaxe

- fonctions

...

Test de reconnaissance des Expressions

, Variables ("utilisés de debug")

Conditionnelle, Béndes, Fonction

Pointeur et Tableau

Rapport:

format: • .txt • .pdf

chaque des éléments de test
avec un paragraphe :

• est-ce que s'est fait ?

• est-ce qu'on l'a testé ?

• est-ce que ça marche ?

Pr M les valeurs :

- pourquoi ça ne marche pas ?

tous les infos pr montrer que
on a réfléchi au pb.

- Qu'est-ce qui a été le \oplus dur
et pourquoi ?

Il manque des notes

Compil()

compil-fichier ("lib.c")

compil-fichier ("test.c")

```
print("start")  
print("prep init")  
print("call O")  
print("prep main")  
print("call P")  
print("halt")
```

Compile-fichier(f) {
 Init & make(f)
 While token.type != tok_eof {
 N = Ana_Sym()
 Ana_Sem(N)
 Généralise(N)
 }
}

?

Dans link.c

int init() {

}

int print_num (int V) {

 • • •

}

test.c contiendra le main

des pointers :

int pⁱ

p = 345ⁱ

p = &a <sup>on met
5 dans a</sup>

*p = 5ⁱ

~~a = *p;~~

adresse de 345

$a = *p;$

$*p = 777;$

$**p = 72;$

à l'adresse de 197 on met 12

int T;

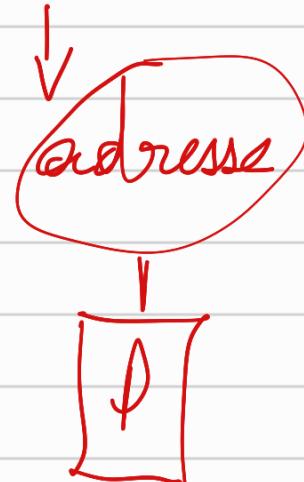
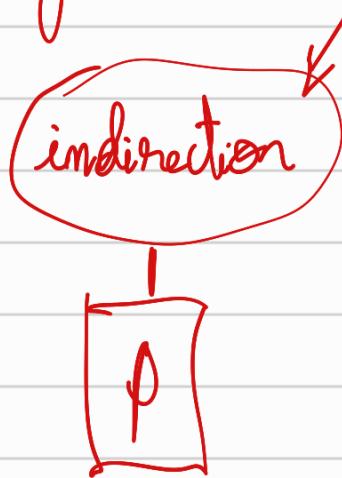
T = malloc(4);

T[3] = 5;

P ← +P | -P | !P | S

on va ajouter $*P | \&P$

2 nouveaux noeuds

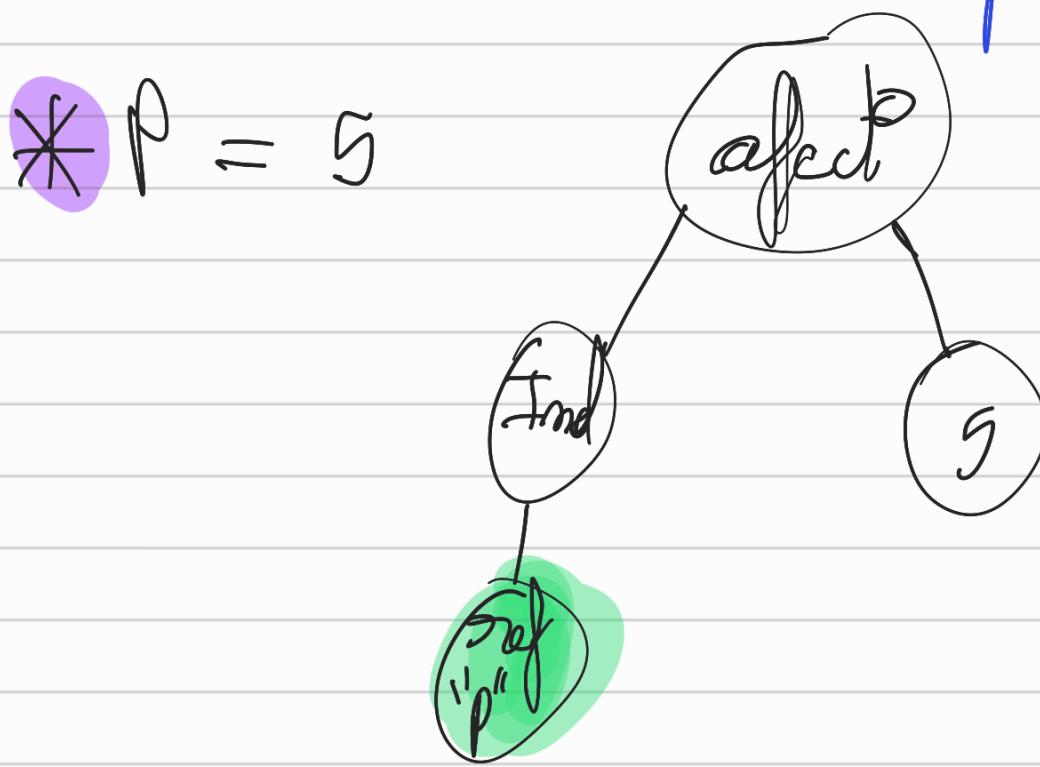


Gencode de l'indirection :

No Ind:

genCode(N° enfant[0])
pr("read")

prend l'élément en haut de la pile va lire la valeur à cette adresse et la met au dessus de la pile.



Note Affectation:

...
} else if (N° enfant[0].type == NodeInd)

GenCode(N° enfant[0].enfant[0])

Print ("Write")

prend l'adresse et la valeur au-dessus
de la pile et met la valeur à cette adresse

e-value r-value

* P] = [5

Nde Adr :

on doit trouver l'adresse d'une variable
locale dans le code à ce moment

$$\& b \Rightarrow BP - (pos(b) + 1)$$

Pour obtenir BP on va faire
un **prep** d'une fonction et on
fait ensuite un **drop** à de celle

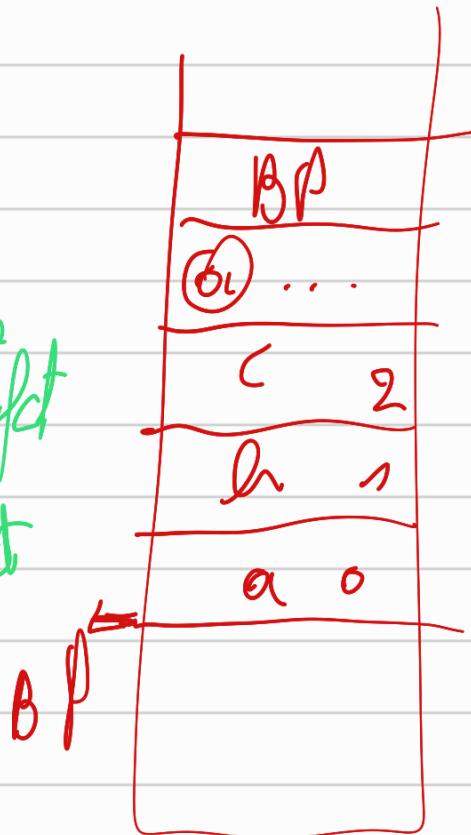
manière on garde en dessous de
la pile l'adresse de BP.

prep .start

swap → on inverse les cases de BP et de l'adresse de la fct
drop 1 → drop de l'adr de la fct

push pos(B) + 1

sub



Genocode Nd_Adr:

```
if( No_enfom[6].type != NdRef){
```

```
ErrorFatal;
```

Vérifier aussi si

```
}
```

```
pr( " prep .start" )
```

Voorlopig dags van
Symbolen

```
pr( " swap" )
```

pr("drop 1")

pr("push", N.enf[0].S. pos + 1)

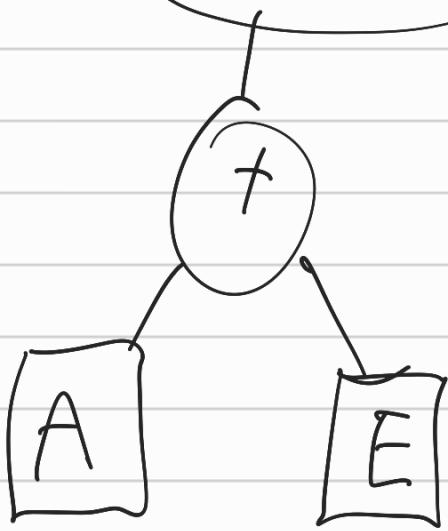
pr("seek")

Les Tableaux :

$a[b] \leftarrow *(\alpha + b)$

$S \leftarrow A(i \dots i) [E]$

indirection



$A[E]$
↓
atome expression

Dans la fonction S() :

} $\text{disc}(\text{check}([\text{'[']})) \{$

$e = E(0);$

$\text{accept}([\text{'J'}]);$

fabriquer l'arbre

}

Plus on monte dans la pile (l'adr est petite).

Les fonctions malloc et free :

taille du code, dans $*o$ on a l'adresse de la 1ère case libre après le code



Bon pour m'aide et que
But: allouer de la mémoire

on va mettre malloc et free dans
lib. C pour qu'ils puissent être utilisés
partout.

Versions Basic

• int malloc (int n) {
 int r = *Oⁱ // Sauvegarde de l'adr de la 1^{ère} case libre

 *O = *O + n^o // incrément
 return r^o } // return n^o]

Renvoie le bloc de n cases
libres à partir du code dans la
mémoire

• int free (int p) { }

*60 000




Bloc taille libre

- Si un Bloc est occupé on va utiliser la taille pour déplacer vers un autre Bloc

- Si Bloc libre trop grand, on prend ce dont on a besoin et le reste devient un Bloc de mémoire libre

Chaque Bloc prend taille + 3 cases mémoires.

↑ But de free

Qd on libère un Bloc on regarde si les Blocs voisins sont libres et si c'est le cas, alors on les fusionnent.

C'est pq après chaque bloc on met sa taille pr aller en courries et en avant.

malloc : parcours listes des blocs
pour voir si ils sont libres, si blocs
vraiment trop gros, on le découpe
et on le marque occupé.

free : parcours listes des blocs, si
blocs libres et côte à côte, ils les
fusionnent.

Correction :

GeneCode :

NdFunc

nbVar = 0

S = de dare(..)

begin

end

S. mb voor

gcc m3m -o m3m

./m3m frustxt

