Cours de TRAD1

Chapitre 1 : Introduction

Définition 01 : Compilateur

C'est un logiciel qui traduit un programme écrit dans un langage source de haut niveau en instructions exécutables.
La compilation est l'ensemble des algorithmes et structures de données nécessaires à la traduction.

Proposition 01 : Schéma général d'un compilateur

Code source -> analyse lexicale (via automates finis), transformations en tokens -> Analyse syntaxique (parser), deux sortes d'analyse : ascendante et descendante -> phrases -> analyse sémantique -> traduction en code intermédiaire -> optimisation (indépendante de la machine cible) -> génération de code assembleur -> optimisation -> programme exécutable

Exemple : 01
Historique des langages :
1954 FORTRAN
1958 LISP
1959 COBOL
1966 SIMULA (classes + objets)
1971 PASCAL
1972 C, PROLOG
1976 ADA (généricité, type abstrait), ML, Caml
1982 C++ (langage C + SIMULA)
1991 JAVA
RUST, PYTHON.....

RAMASSE MIETTE !!!!!! (Culture)

Chapitre 2 : Analyseur lexical

Proposition 02 : Propriétés d'un analyseur lexical

→ lit le texte source caractère par caractère (c'est le seul module de la compilation qui lit le texte).
→ reconnaît les unités lexicales (token) qui sont les mots du langage, et les présente à l'analyseur syntaxique.
→ c'est un automate.

Principales unités lexicales :
- caractères simples : {+, -, /, *, {, }, (,)} etc
- unités de 2 caractères : ++,-=, <=, etc...
- identificateurs : x,y, factorielle
- valeurs numériques : 42,69,etc..
- mots-clés du langage : if, else, etc..

Autres tâches du lexer :

- supprimer les caractères inutiles : espace, tabulations, fin de ligne, fin de fichier etc..
- affiche les erreurs lexicales (caractère inconnu,)
- supprime les commentaires

Le lexer code les unités lexicales reconnues pour augmenter l'efficacité. Il produit ensuite un lexique (une table) de la forme

Identificateurs	Code lexical
"x"	1
"rayon"	2
⋮	⋮

Exemple : Lexique 02
On considère le code source :

```
/*ceci est un commentaire*/
x = y + 2 * x
```

Le codage des caractères simples se fait sur des unités lexicales simples :
- '+' est codé par 1.
- '-' est codé par 2
- etc..

Le codage des caractères doubles se code en général comme ceci :
- '<=' codé par 10
- '>=' est codé par 11
- etc..

Les mots clés du langage :
- '`if`' est codé par 20
- '`then`' est codé par 21
- etc..

Les unité lexicales génériques
- 'idf' est codé par 30
- 'rstes' est codé par 31
- etc..

Chapitre 3 : l'analyseur syntaxique ascendant

But :
- vérifier que la phrase en entrée vérifie la syntaxe du langage, qui est définie par une grammaire.
- vérifier les erreurs syntaxiques (parfois, le parser propose une correction quand il trouve une erreur.)
- ne s'arrête pas à la première erreur rencontrée.

Analyse syntaxique ascendante

- On part du mot du langage à analyser
- **on remplace itérativement des fragments du mot courant qui correspondent exactement à des membres droits de le règle de production par le membre gauche de cette règle.**
- succès si le mot courant final est l'axiome de la grammaire.

Il y a donc deux actions principales lors d'une analyse syntaxique ascendante :

- décalage (ou lecture)
- réduction

Les analyseurs syntaxiques ascendants fonctionnent avec une **pile**.

Exemple : Analyse syntaxique ascendante 03

On considère la grammaire suivante :

$$\begin{cases} S' \to S \\ S \to \mathrm{Ac} \\ A \to \mathrm{AaAb} \\ A \to d \end{cases}$$

Le mot à analyser est “dadbc$” ($ désigne la fin du texte).

On met donc d dans la pile :

d

On passe à la réduction de d par A. On dépile d et on empile A :

A

$\to$ pas de réduction possible donc on lit le prochain caractère du mot :

a
A

$\to$ Aucune réduction possible pour ‘Aa’, donc on lit d et on l’empile :

d
a
A

On a une réduction possible pour d :

A
a
A

$\to$ aucune réduction possible, donc on lit le prochain caractère du mot :

b
A
a
A

On repère la réduction par la règle 3. Notre pile est maintenant :

A

Aucune réduction possible, on lit donc le prochain caractère :

c
A

On repère la réduction avec la deuxième règle :

S

On tombe bien sur l'axiome, donc notre mot est valide.

Dénomination des analyseurs syntaxiques ascendants :

- Que signifie le k dans LR(k) / SLR(k) ? On regarde les k unités lexicales suivantes pour faire la prédiction sur la règle de la grammaire utilisée pour faire une réduction :

$$\begin{cases} \text{Left to right scanning} \\ \text{Right most derivation} \end{cases}$$

1. Analyseur ascendant SLR(1)

Définition 02 : Item

C'est 1 production de G avec un marqueur en partie droite. Par exemple, $[A \to aAbB] \in G$ nous donne 5 items :

- $A \to .aAbB$
- $A \to a.AbB$
- $A \to aA.bB$
- $A \to aAb.B$
- $A \to aAbB.$

Remarque 01

Le mot vide ε **n'est pas une unité lexicale**. On ne lit pas ε. La règle $A \to \varepsilon$ donne l'item

$$A \to .$$

Etat I_i : un ensemble d'items obtenus par fermeture (un ensemble d'items construits à partir de I selon l'algo suivant) :

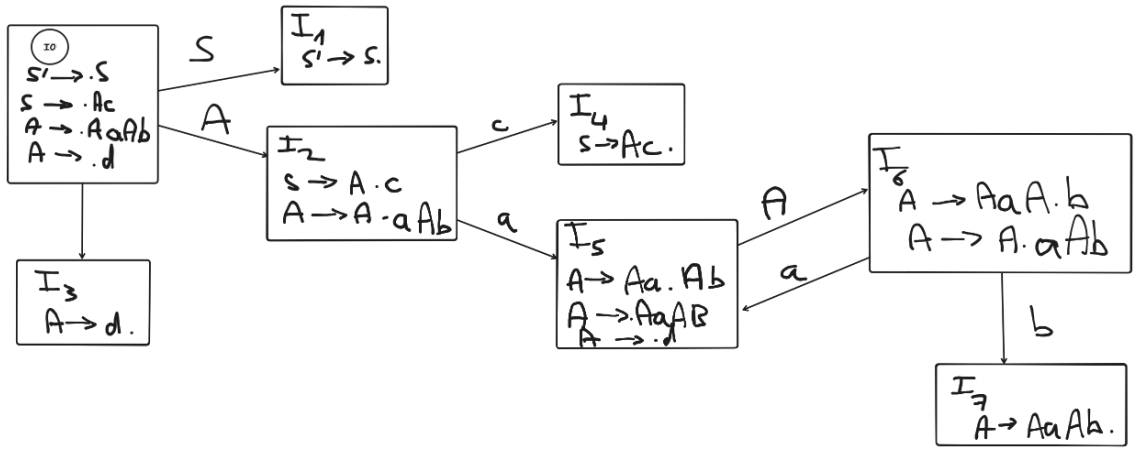
- placer chaque item de I dans fermeture(I).
- si $[A \to \alpha.B\beta] \in$ Fermeture(I) et $[B \to \gamma]$, alors ajouter $[B \to .\gamma]$ à Fermeture(I) sauf si déjà présent.
- Itérer jusqu'à ne plus trouver d'items à ajouter à Fermeture(I)

Exemple : 04

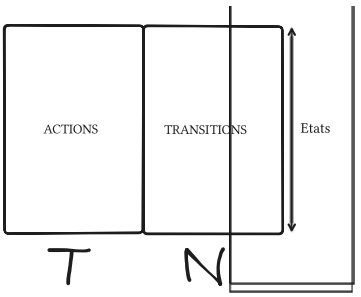
On considère la grammaire suivante :

$$\begin{cases} S' \to S \\ S \to \text{Ac} \\ A \to \text{AaAb} \\ A \to d \end{cases}$$

On va essayer de construire l'automate LR(0).



L'automate LR(0) est en fait "codé" dans une table partagée en :



- table d' "ACTION"
  - si $[S' \longrightarrow S.] \in I_i$, alors $\text{ACTION}(I_i, \$) =$ "accepter"
  - si $[X \longrightarrow \beta.] \in I_i, X \neq S'$ (où S' est l'axiome), alors $\text{ACTION}(I_i, a) =$ "réduire" avec $X \longrightarrow \beta$ où $\forall a \in T \cup \$$
  - Si $[X \longrightarrow \alpha.a\beta] \in I_i$, alors $\text{ACTION}(I_i, a) =$ "aller à état" I_j
  - "erreur" dans toutes les autres cases.
- table de transition :
  - si dans l'automate on a Transition $(I_i, X) = I_j$, alors $\text{TRANSITION}(I_i, X) = j$.

	a	b	c	d	$	S	A	S'
I0				d3		1	2	
I1					OK			
I2	d5		d4					
I3	r3	r3	r3	r3	r3			
I4	r1	r1	r1	r1	r1			
I5							6	
I6	d5		d7					
I7	r2	r2	r2	r2	r2			

d : décaler
r : réduire

Fonctionnement de l'analyseur syntaxique :

Etat	Pile de l'analyseur	Texte source
0	d3 (lecture)	dabcb$
0~~d3~~	r_3, réduire $A \longrightarrow d$ et PAS de lecture	abcb$
0A2	lecture de a	adbc$
0A2a5	lecture de d	dbc$

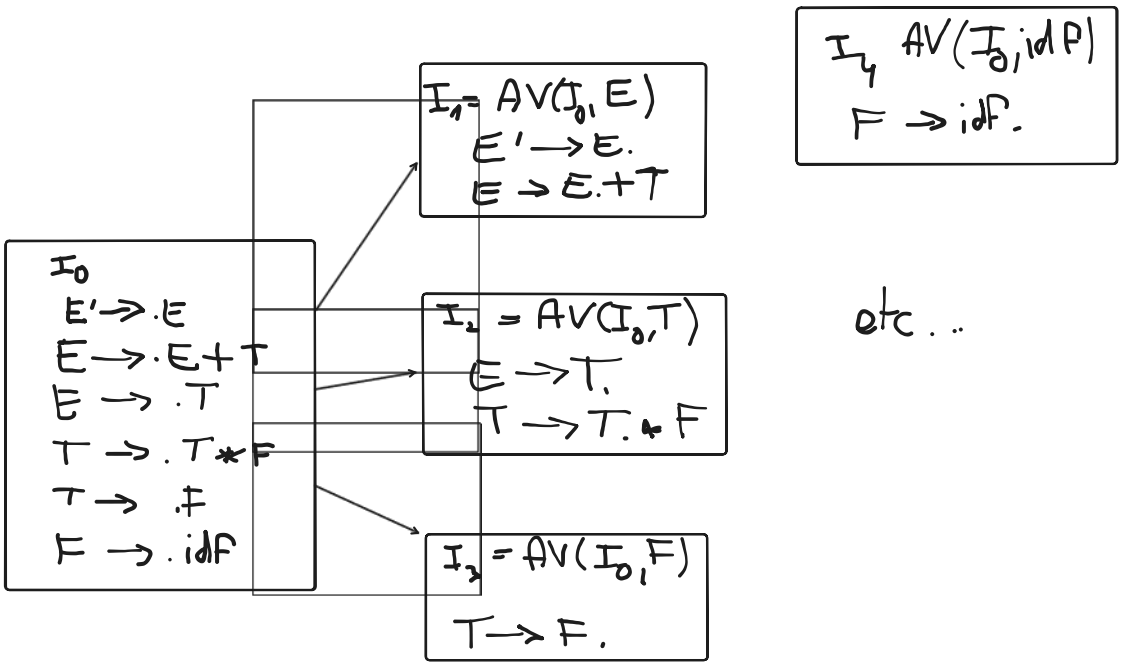
Etat	Pile de l'analyseur	Texte source
0A2a5~~d3~~	réduire $A \longrightarrow d$ et PAS de lecture	bc$
0A2a5A6	d7 (lecture)	bc$
0A2a5A6b7	réduire par $A \longrightarrow AaAb$, et pas de lecture	c$
OA2		c$
OA2c4	d4 + lecture	$
0A2a5A6b7A8	réduction par $S \longrightarrow Ac$	$
0S1	accepter	

Exemple : Analyseur syntaxique SLR(1) 05

On considère la grammaire suivante :

$$\begin{cases} E' \longrightarrow E \\ E \longrightarrow E + T \\ T \longrightarrow T * F \\ T \longrightarrow F \\ F \longrightarrow \text{idf} \end{cases}$$

On obtient l'automate LR(0) suivant :



Etat	Pile de l'analyseur	Texte source
0	d3 (lecture)	a+a*a$
0~~a4~~	r_3, réduire $A \longrightarrow d$ et PAS de lecture	a+a*a$
0F3	réduction par $T \longrightarrow F$	a+a*a$

Etat	Pile de l'analyseur	Texte source
0T2	réduction par $E \longrightarrow T$	a+a*a$
0E1	réduction / lecture	a+a*a$

On a bien un conflit réduction / lecture à la fin de la pile.

Analyse syntaxique SLR(1)
- Basé sur un automate LR(0).
- Table d'analyse syntaxique SLR(1) :
  - partie TRANSITION identique à l'analyseur LR(0)
  - partie ACTION : la seule différence est sur la réduction :
    - **Si** $[A \longrightarrow d.] \in I_i, A \neq S'$ **alors ACTION**$(I_i, a) =$ réduire par $A \longrightarrow d$ pour tout $a \in$ Suiv(A)**.**

Table d'analyse syntaxique ascendant LR(1)

Principe :
Considérons

$$I_i = \begin{cases} A \longrightarrow B.aDa \\ D \longrightarrow B. \\ \dots \end{cases} \text{ et } I_j = \begin{cases} A \longrightarrow Ba.Da \\ \dots \end{cases}$$

$\simeq$ SLR(1) avec la table d'action et de transition. La différence est sur les réductions.
On va ajouter une information supplémentaire dans les items. Un item devient $[A \longrightarrow \alpha.\beta, [a]]$ où $A \longrightarrow \alpha\beta \in G$ et a est un terminal ou $.
Ce symbole a est appelé contexte ou symbole de pré-vision. Ce contexte n'aura un effet que sur les items de la forme $A \longrightarrow > d.[a]$ où l'action dans la table devient : "réduire par $A \longrightarrow d.$ uniquement pour le symbole a "

Proposition 03 : Calcul de fermeture avec contexte

Pour toute règle de la forme $A \longrightarrow \alpha.B\beta[a]$, on transforme les règles de la forme

$$B \longrightarrow \gamma. \text{ en } B \longrightarrow \gamma.[\text{Premier}(\beta a)]$$

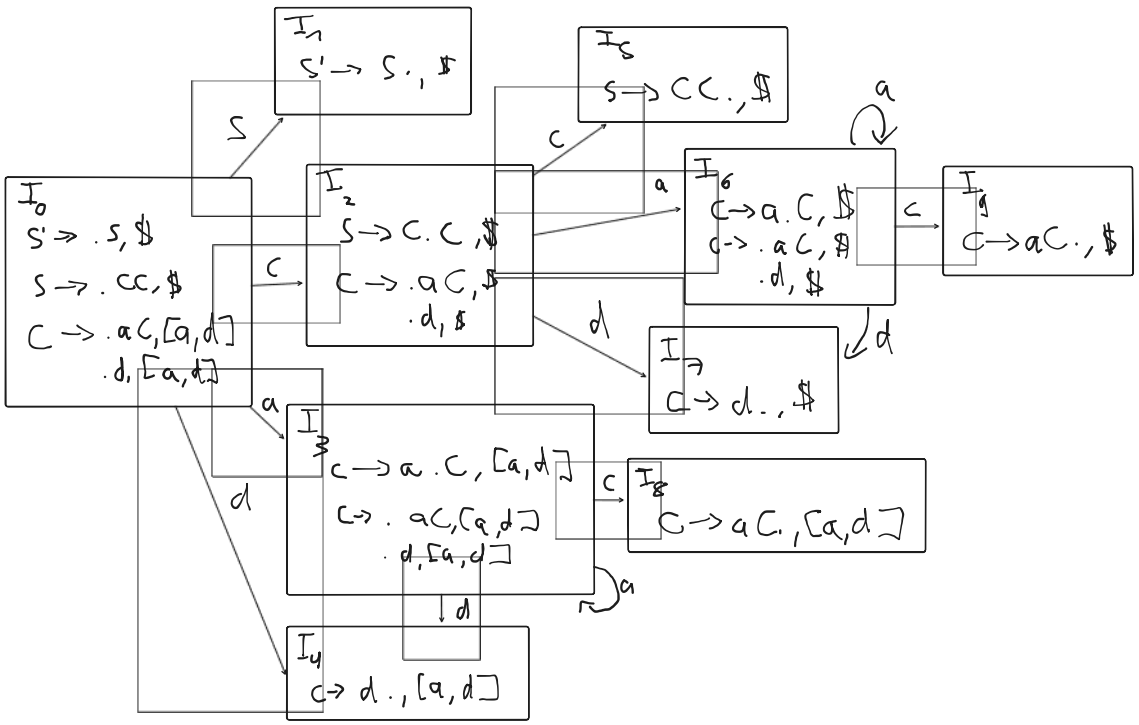
Exemple : Analyseur syntaxique LR(1) 06
On considère la grammaire suivante :

$$\begin{cases} S' \longrightarrow S \\ S \longrightarrow CC \\ C \longrightarrow aC \\ C \longrightarrow d \end{cases}$$

Comment faire une analyse syntaxique ascendante LR(1) ?

- Automate LR(1)
- table
- conclusion

Commençons par l'automate LR(1) :



On obtient donc la table d'analyse suivante LR(1) suivante :

	a	d	$	C	S	S'
I_0	d3	d4		2	1	
I_1			OK			
I_2	d6	d7		5		
I_3	d3	d4		8		
I_4	r3	r3				
I_5			r1			
I_6	d6	d7		9		
I_7			r3			
I_8	r2	r2				
I_9			r2			

Il n'y a pas de conflits lecture / réduction ou réduction / réduction dans la table, donc l'analyseur est LR(1).

Construction d'une table d'analyse LALR(1) = LR(1) avec des fusions d'états, à partir de l'automate LR(1) :
Principe : Fusionner les états ayant le même noyau mais un contexte différent. Par exemple, si on a deux états de la forme $I_i = [C \longrightarrow d.[a, d]]$ et $I_j = [c \rightarrow d.[\$]]$ pour créer l'état $I_{i,j} = [C \longrightarrow d.[a, d, \$]]$
Les analyseurs LALR(1) s'obtiennent par fusion de tous les candidats à la fusion. Si on peut tout fusionner sans conflits, alors l'automate est le plus petit possible.

Analyseur LR(1) et grammaire ambigüe

On considère la grammaire suivante :

$$\begin{cases} S \longrightarrow \text{if } E \text{ then } S \text{ else } S \\ S \longrightarrow \text{if } E \text{ then } S \end{cases}$$

et la phrase "if a then if b then S1 else S2"
Il y a 2 interprétations possibles :

- "if a then (if b then S1 else S2)" (1)
- "if a then (if b then S1) else S2" (2)

Lors d'une analyse ascendante :
On aurait les règles suivantes :

$$\begin{cases} S \longrightarrow \text{if } E \text{ then } S. \\ S \longrightarrow \text{if } E \text{ then } S. \text{ else } S \end{cases}$$

La règle utilisée par (1) devrait être $S \longrightarrow \text{if } E \text{ then } S. \text{ else } S$, tandis que la règle utilisée par (2) devrait être $S \longrightarrow \text{if } E \text{ then } S.$. Mais c'est ambigüe !

Comment lever le problème ?
Il existe deux possibilités :

- On peut modifier la grammaire pour la rendre non ambigüe. Par exemple, on peut ajouter des parenthèses pour lever l'ambigüité.
- On peut ajouter des règles de priorité pour lever l'ambigüité. Par exemple, on peut dire que le "else" est prioritaire sur le "then".

Utiliser les précédences de token.

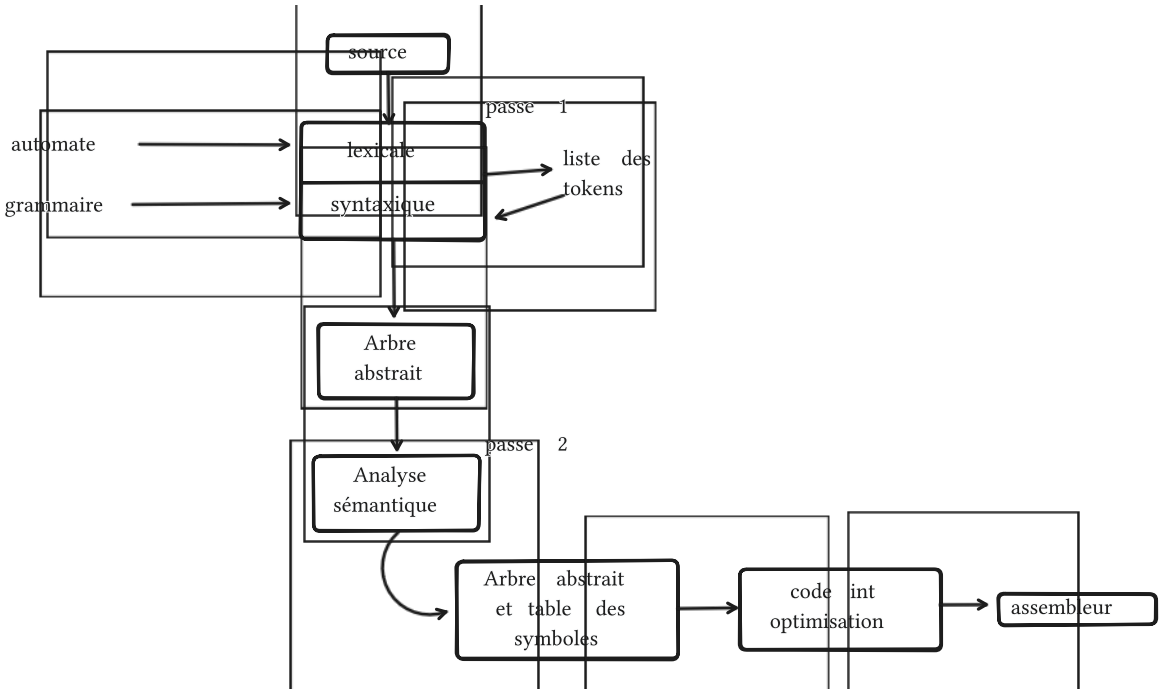
Par exemple, * a une précédence plus haute que +.
Soit

$$\begin{cases} P \longrightarrow \alpha.t\beta[] & \text{décalage sur t} \\ Q --q > \gamma uR.[t] & \text{réduction pour t} \end{cases}$$

Si le symbole de prévision est t, on a 3 possibilités dans cet item, u étant déjà empilé. On va comparer les précédences de u et t.

- u a une précédence supérieure à t : prec(u) > prec(t) $\Longrightarrow$ réduire.
- u a une précédence inférieure à t : prec(u) < prec(t) $\Longrightarrow$ lecture.
- u a une précédence égale à t : prec(u) = prec(t) $\Longrightarrow$ lecture ou réduction selon l'associativité.

Chapitre 4 : Analyse sémantique (Arbre syntaxique, contrôle sémantique)

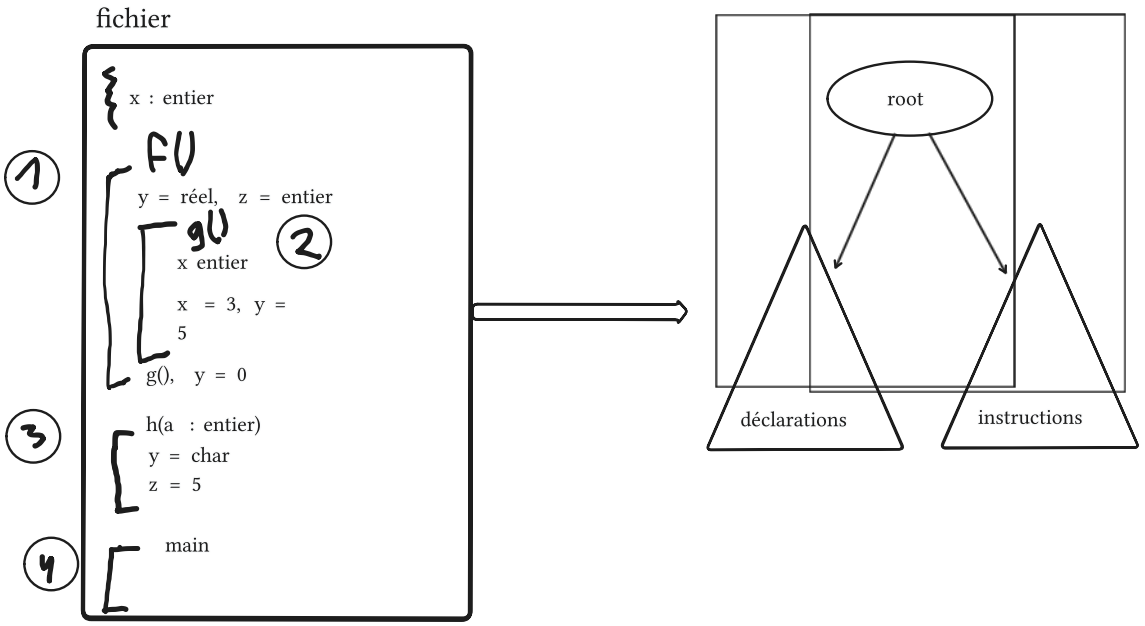


Exemples de règle sémantiques :

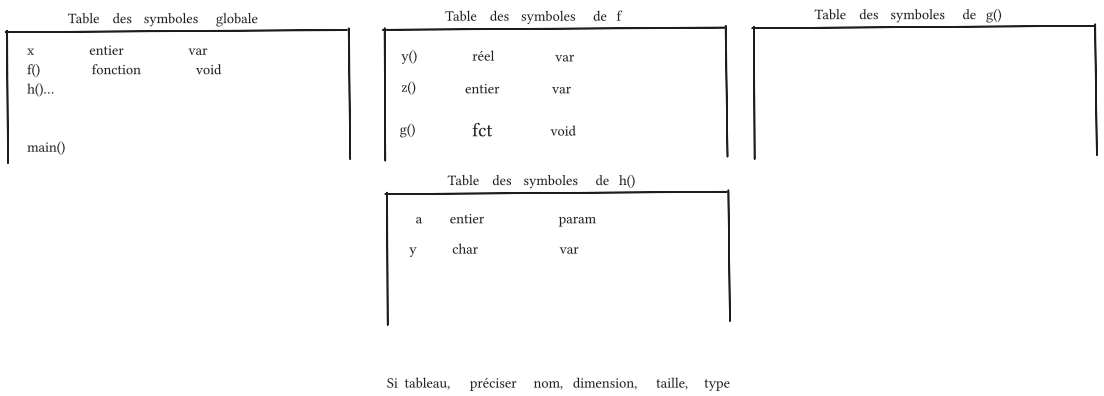
- double déclaration d'une variable ou d'une fonction interdit.
- variables déclarées avant utilisation, et visible dans le scope.
- compatibilité des types
- nombre de paramètres effectifs = nombre de paramètres formels.

Table des symboles :

- Table par bloc
- Contient toutes les déclarations du bloc



Que mettre comme informations pour un symbole ?
On a une table des symboles par bloc :



On utilise une pile ou une table de hash pour gérer les blocs englobants !

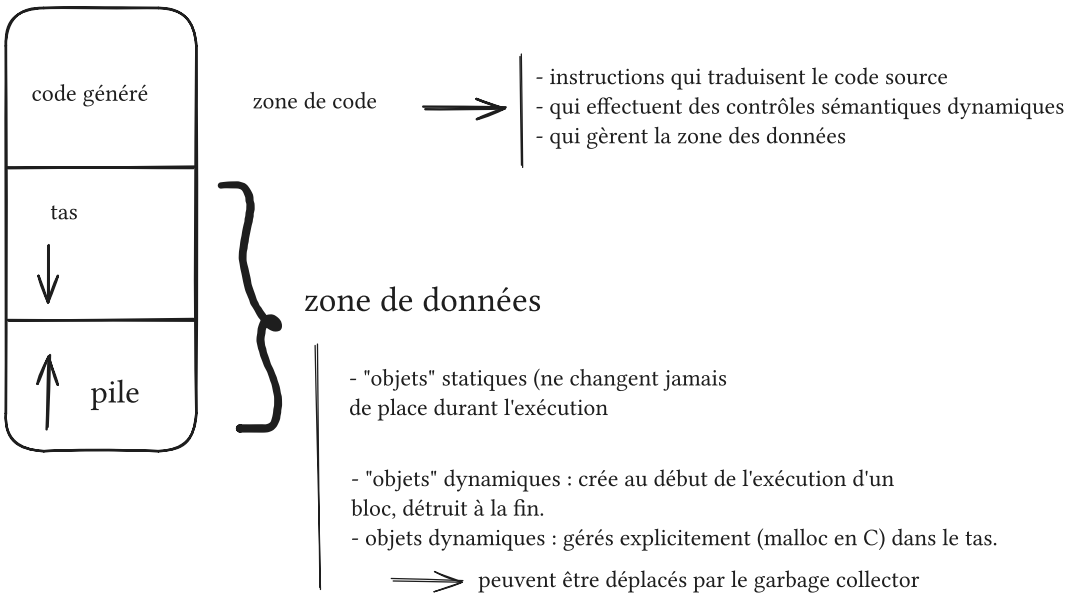
Définition 03 : Portée d'une déclaration (scope)

C'est la portion de code du programme source où l'identificateur est défini.

Définition 04 : Identification

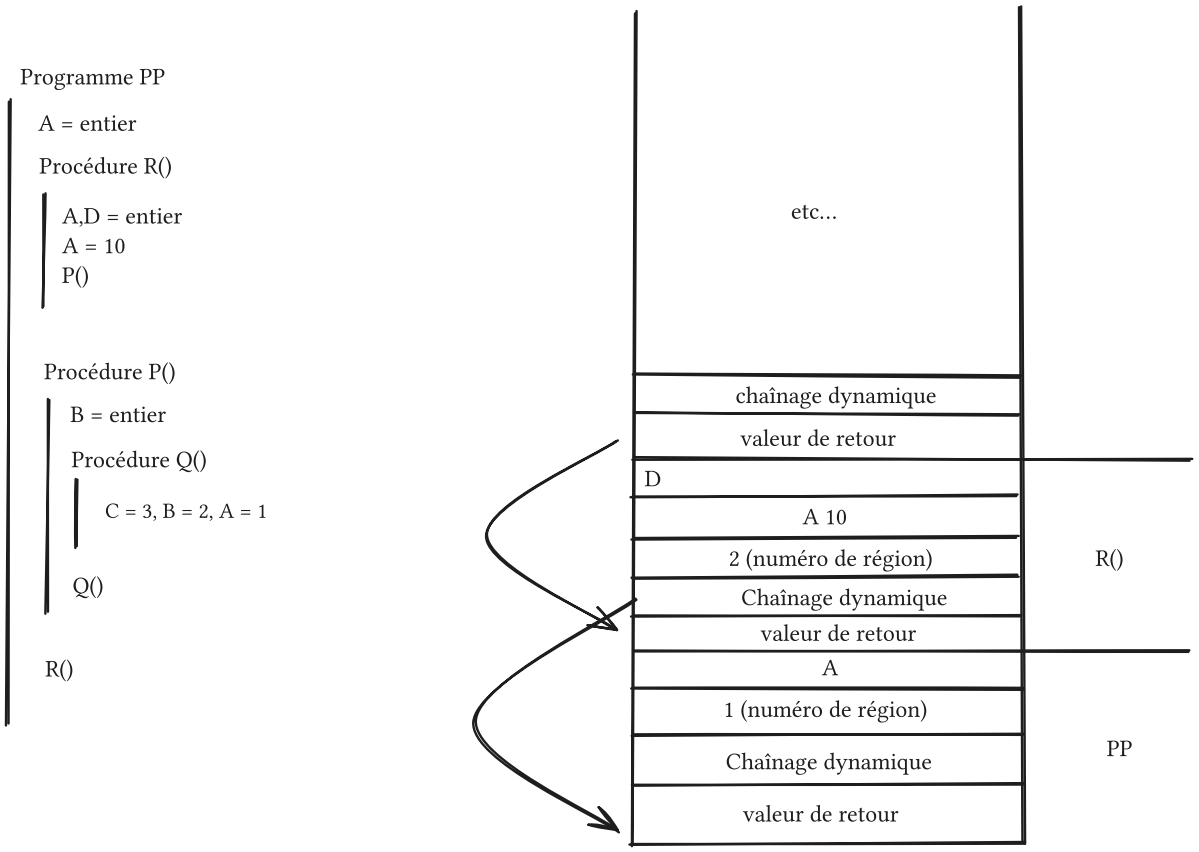
Mécanisme qui associe l'utilisation d'un identificateur à sa déclaration.

Chapitre 5 : Mémoire à l'exécution



On va regarder deux manières de fonctionner :

Utiliser les numéros de région

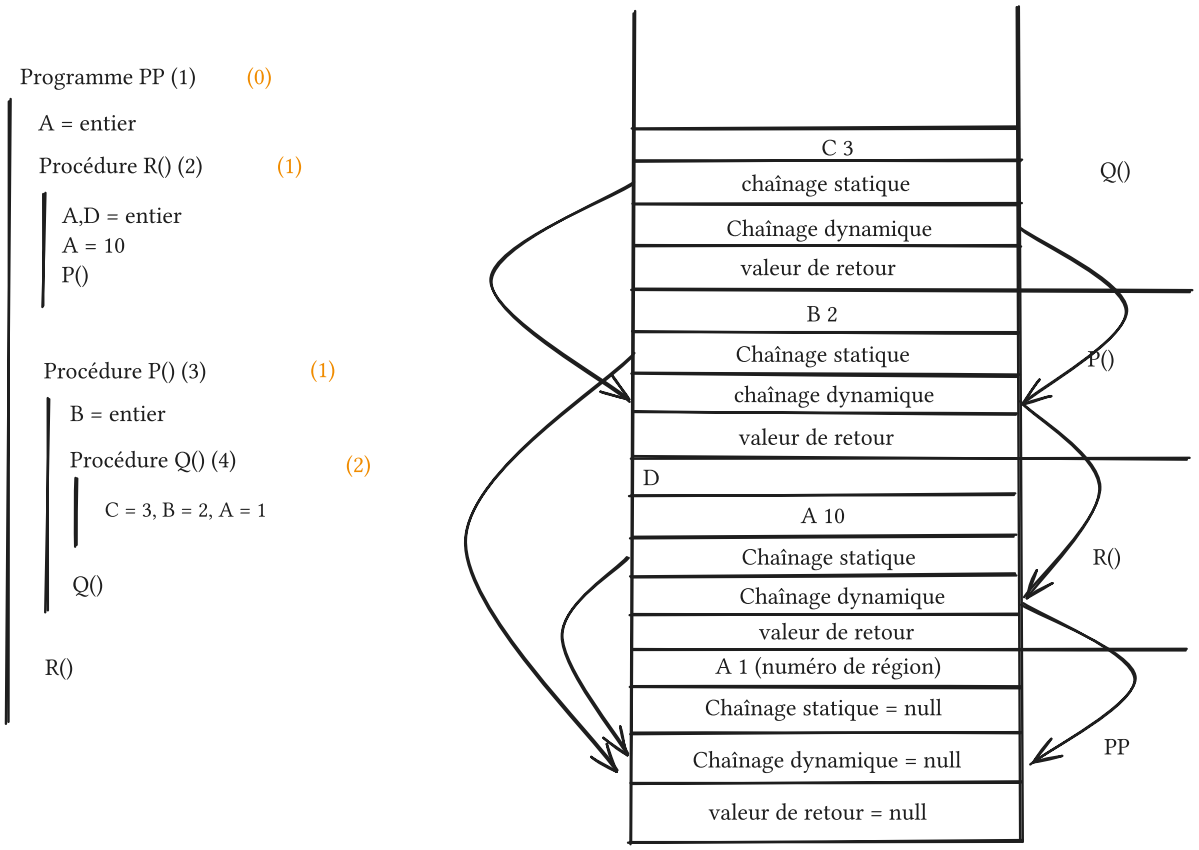


Cette méthode a un inconvénient non négligeable : à chaque fois que l'on cherche une variable, on peut parcourir toute la pile afin de la trouver (même dans des blocs non englobants, et qui ne seraient donc pas valides même si on trouvait une variable).

Le chaînage statique

On va rajouter en orange des numéros de blocs englobants pour chaque région.
Pour retrouver la base du bloc de déclaration d'un objet (non-local évidemment) :

- Les règles de portée des déclarations garantissent que :
  - si une variable I est déclarée dans le bloc X d'imbrication N_X et si I est visible dans le bloc Y d'imbrication N_Y, alors $N_X < N_Y$.
- **Pour accéder à cet objet I non local, il faut remonter de $N_Y - N_X$ châinage statique.**



<u>Mise en place du chaînage statique :</u>
Supposons :

- L'appelant est un bloc d'imbrication N_X.
- L'appelé est un bloc d'imbrication N_Y.

Soit $N_Y - N_X = k$. la différence de niveau.
Le chaînage statique du bloc appelé doit désigner le dernier bloc d'imbrication $N_Y - 1 = N_Z$.
Depuis le bloc appelant, on trouve le bloc d'imbrication N_Z en remontant $N_X - N_Z = N_X - N_Y + 1$. Il faut donc parcourir $N_X - N_Y + 1$ chaînages statiques de l'appelé à mettre en pile.

Il existe une troisième méthode qui permet d'accéder directement à l'adresse, mais elle ne sera pas vue dans ce cours.