Analyse lexicale : La phase d'analyse lexicale lit le programme source à compiler, caractère par caractère et tente de reconnaître une suite d'éléments lexicaux (entité lexicale). Elle se base sur les grammaires de type 3, des expressions régulières et les automates d'états finis déterministes. Cette phase peut échouer lorsque des erreurs lexicales existent dans le programme source. Elle remplit la table des symboles au fur et à mesure qu'elle reconnaît une entité lexicale (var ou identificateur, étiquette, …).

Les erreurs qu'on peut détecter dans cette analyse lexicale sont:
- détecter les caractères n'appartenant pas au langage
- Vérifier la longueur des idf et constantes si limite de la longueur existe
- etc...

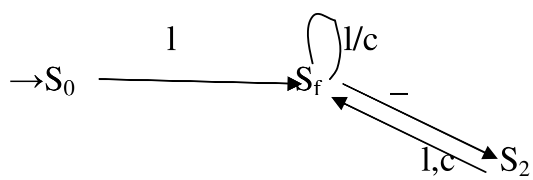
L'entrée de cette phase est le programme source à compiler. La sortie, s'il n'y a pas d'erreur, est une suite d'entités comme par exemple :

MR1 IDF1 SEP1 MR2 IDF2 SEP2 IDF3 SEP2 IDF4 SEP1 ……. MR10 SEP 20

Où MR_i correspondent aux mots réservés du langage, IDF_i correspondent aux noms de variables utilisés dans le programme, SEP sont des séparateurs, …. On trouve aussi les opérateurs arithmétiques, opérateurs logiques, etc….

Exemple : Donner un automate reconnaissant un identificateur. On définit par exemple un identificateur comme une suite de lettre et de chiffre, commençant par une lettre, pouvant contenir un tiret « _ » mais pas 2 « _ » consécutifs et ne doit pas se terminer par un «tiret « _ ».

S0 est l'état initial, Sf est l'état final



Cet automate est déterministe, sinon il faut le rendre déterministe. Sa table d'analyse est : (S_0 est l'état initial, S_f est l'état final)

	l	c	_
S_0	S_1		
S_f	S_1	S_1	S_2
S_2	S_1	S_1	

Reconnaître les chaines suivantes :

a) ab_c   sort en S_f et fin de chaine donc identificateur correcte.
b) ab__c   blocage en (S_2,_) donc erreur « identificateur non reconnu ».
c) ab_   fin de la chaine en S2 qui n'est pas final donc erreur.

Analyse syntaxique : Le but de cette analyse est de reconnaître la suite des entités reconnues lors de la phase lexicale. Elle se base sur les grammaires de type 2, des grammaires algébriques et des automates à pile. Le résultat de cette phase peut etre est un arbre syntaxique. C'est l'une des phases les mieux formalisées de toute la compilation.

On vérifie dans cette phase que le texte du programme est conforme à la syntaxe du langage définie par la grammaire du langage. Il est important de bien voir que seule la **forme** est prise en compte dans l'analyse syntaxique : le **fond**, à savoir la signification du code source, n'est quant à lui vérifié que dans l'analyse sémantique.

Il existe 2 méthodes d'analyse syntaxique :

- Les méthodes descendantes
- Les méthodes ascendantes

A) Méthodes descendantes : elles consistent à partir de l'axiome et par une série de dérivation aboutir au programme à analyser syntaxiquement. Il existe 2 types de méthode d'analyse descendante :

- Les méthodes non déterministes
- Les méthodes déterministes

A1) Méthodes descendantes non déterministes :

- **Parallèle** : elle consiste à construire plusieurs arbres syntaxiques en même temps, jusqu'à aboutir à un succès (suite d'entités cherchée). Si tous les arbres (toutes les possibilités) n'aboutissent pas alors le programme est erroné syntaxiquement. Le problème de cette méthode est qu'elle consomme beaucoup de temps d'exécution et de place mémoire.

- **Retour-arrière** : un arbre syntaxique est démarré, lorsqu'il y a une règle qui a plusieurs possibilité, par exemple $A \rightarrow \alpha/\beta/\theta$ qu'on peut appliquer, on choisit une règle et on mémorise les autres choix. Si un chemin ne marche pas, on revient au dernier point de choix pour essayer une autre règle. Si on ne trouve pas de chemin donnant le programme à analyser, le programme est erroné syntaxiquement. On a le même problème du temps d'exécution et de l'espace que la précédente.

A2) Méthodes descendantes déterministes : Elles consistent à emprunter un chemin et aller jusqu'au but (pas de retour arrière). Il n'y pas de non déterminisme dans ces méthodes. Avant de faire une analyse descendante déterministe, il faut s'assurer que la grammaire n'est pas récursive à gauche (directe et indirecte). Si elle est récursive à gauche il faut enlever cette récursivité.

Récursivité à gauche : Une grammaire est non récursive à gauche, si elle ne contient pas de règle $A \to A\alpha$ (récursivité directe) et de règles $A \to^* A\alpha$ (récursivité indirecte) où $\alpha \in (T \cup N)^*$ et A un NT.

Comment enlever une récursivité directe ? : Soit une règle générale
$A \to A\alpha_1/A\alpha_2/\ldots/A\alpha_n/\beta_1/\beta_2/\ldots/\beta_m$, on la transforme en les règles :

$A \to \beta_1/\beta_2/\ldots/\beta_m/\beta_1A'/\beta_2A'/\ldots/\beta_mA'$
$A' \to \alpha_1/\alpha_2/\ldots/\alpha_n/\alpha_1A'/\alpha_2A'/\ldots/\alpha_nA'$

Comment enlever la récursivité indirecte ?: Soit une récursivité $A \to^* A\alpha$,

- Enlever tout d'abord les récursivités directes éventuelles
- Ordonner les NT par $A_i < A_j$ si on $A_i \to A_j\psi$
- On obtient $A < A_1 < A_2 < \ldots < A_n < A$, on substitue A_i dans A_{i-1} de n à 1.
- On obtient une récursivité directe qu'on enlève comme ci-dessus.

Il existe plusieurs méthodes pour faire cette analyse descendante déterministes :

Descente récursive : On associe une procédure à chaque non terminal A de la grammaire. Le corps de la procédure exprime le MDP de la règle. On parcourt le MDP de la règle de gauche à droite, lorsque le terme courant est un terminal on le teste avec le terme courant de la chaine et lorsque ce terme courant est un non terminal, on fait appel à la procédure correspondante. Dans une règle $A \to \alpha B\psi$ et avant d'appeler une procédure correspondant au nom terminal B et par soucis d'optimisation, on vérifie si les k terme courant appartient au $deb_k(B\psi.suiv_k(A))$. (voir plus loin pour la définition deb et suiv).

Pour pouvoir utiliser cette méthode, il faut que la grammaire soit non récursive à gauche (voir plus haut) et factorisée. Une grammaire est non factorisée si elle contient une règle de la forme $A \to \alpha\psi / \alpha\beta$. Pour factoriser une grammaire :

La règle $A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3/\ldots/\alpha\beta_n / \psi$ est transformée en:

$A \to \alpha A' / \psi$
$A' \to \beta_1 / \beta_2 / \beta_3/\ldots/\beta_n$

Exemple: Soit la grammaire récursive suivante:

S → b(T) / a
T → T,S / S   on a une récursivité directe à ce niveau, on l'enlève:

S → b(T) / a
T → **ST' / S**
T' → **,ST' / ,S**

Elle n'est plus récursive, mais elle n'est pas factorisée dans les 2 dernières règles.

$S \rightarrow b(T) / a$	qui nous donne	$S \rightarrow b(T) / a$	c-à-d	$S \rightarrow b(SB) / a$
$T \rightarrow SB$		$T \rightarrow SB$		$B \rightarrow ,SB / €$
$B \rightarrow T' / €$		$B \rightarrow ,SB / €$		
$T' \rightarrow ,SB$				

qui est non récursive et factorisée.

<u>Méthode LL(k)</u> : (k>=1) Vérifier tout d'abord que la grammaire est non récursive à gauche (une grammaire récursive gauche est non LL(k)). Une analyse LL(k) est une méthode d'analyse dans laquelle on regarde k caractères de la chaine à analyser pour décider de l'action à appliquer. Pour cette méthode, on doit calculer des ensembles « début » et « suivant » définis comme suit :

$Deb(\alpha) = \{ t \in T / \alpha \rightarrow^{*} t\, \alpha_1 \} \cup \{ € \text{ si } \alpha \rightarrow^{*} € \}$ c.-à-d. tous les terminaux par lequel α peut commencer plus € si ce α donne €.

$Suiv(A) = \{t \in T \cup \{\#\} / S\# \rightarrow^{*} \psi A \beta \text{ et } \beta \rightarrow^{*} t\sigma \}$ c-à-d tous les terminaux qui peuvent apparaitre après A

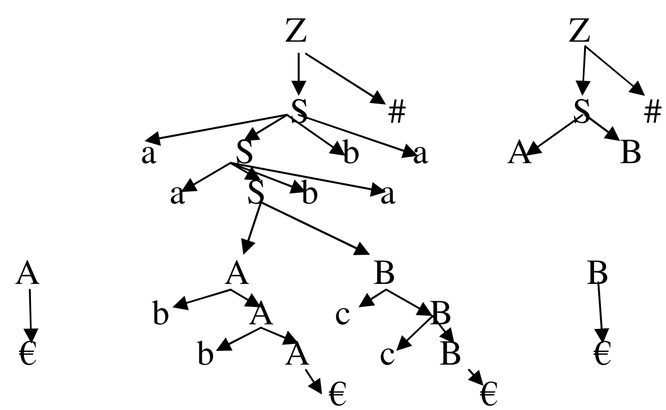
avec $\sigma, \sigma_1, \beta, \beta_1 \in (T \cup N)^{*}$ $\psi \in T^{*}$ et $A \in N$.

<u>Exemple:</u> Déterminer les ensembles début_1 et suiv_1 pour la grammaire:

$S \rightarrow aSba / AB$
$A \rightarrow bA / €$
$B \rightarrow cB / €$

deb(S) contient "a" de aSba,
contient $deb_1(A) - €$ puisque $S \rightarrow AB$, c-à-d {b}
contient $deb_1(B) - €$ puisque $S \rightarrow AB$ et $A \rightarrow €$ c-à-d {c}
contient € puisque $S \rightarrow AB$ et $A \rightarrow €$ et $B \rightarrow €$
donc $deb_1(S) = \{a, b, c, €\}$
$deb_1(A) = \{b, €\}$
$deb_1(B) = \{c, €\}$

$suiv_1(S) = \{b, \#\}$
$suiv_1(A) = \{c, b, \#\}$
$suiv_1(B) = \{b, \#\}$



Définition: On dit qu'une grammaire G est LL(K) ssi $\forall$ A $\rightarrow\alpha/\beta$ on a

$$\mathbf{deb_k(\alpha.suiv_k(A)) \cap deb_k(\beta.suiv_k(A)) = \emptyset}$$

On rajoute les suivants de A pour prendre en compte le cas où α ou β donne €, ou le cas où k est supérieur à ce que peut dériver α ou β.

On peut vérifier si une grammaire G est LL(k) en donnant sa table d'analyse et on vérifie que cette table est mono-définie. On construit cette table d'analyse T comme suit: C'est une table n x m où n est le nombre de non terminaux de la grammaire et m = (nombre de terminaux + 1)^k (on rajoute +1 pour le #). On met à la puissance k pour prendre toutes les combinaisons possibles de k terminaux. *Dans la case $T(A, deb_k(\alpha.suiv_k(A)))$ on met la règle $A \rightarrow\alpha$*

Pour l'analyse d'une chaine, on procède comme suit:

Pile	**chaine**	**Action** à faire (voir la table)
#S	.... #	....
......A	on regarde k car	*si T(A,k car) = Ø* *alors erreur* *sinon empiler dans pile α^R de $A\rightarrow\alpha$*
.......a	a......	*avancer*
....	....	....

A la fin, si on a dans la pile #S et il ne reste que # dans la chaine alors la chaine est accepté sinon c'est une erreur (chaine n'appartient pas au langage et donc erreur syntaxique).

Remarque: LL(k) $\Rightarrow$ LL(k+1) k=1,2... mais LL(k) =/$\Rightarrow$ LL(k-1) k=2,3....
Lorsqu'on veut vérifier si G est LL(k) on commence avec k=1, puis 2 etc...

Exemple: On reprend la grammaire de l'exemple précédent:

G: $S \rightarrow b(SB) / a$
$B \rightarrow ,SB / €$

Pour que G soit LL(1), il faut que:

$deb_1(b(SB)) \cap deb_1(a) = \emptyset$ et
$deb_1(,SB) \cap deb_1(€.suiv_1(B)) = \emptyset$

Calcul des ensembles deb et suiv:

$deb_1(S)=\{a\ b\}$ $suiv_1(S)=\{\#\ ,\ )\}$ $deb_1(B) = \{,\ €\}$ et $suiv_1(B)=\{\ )\ \}$
$deb_1(b(SB)) \cap deb_1(a) = \{ b \} \cap \{ a \} = \emptyset$ donc ok pour cette règle
$deb_1(,SB) \cap deb_1(€.suiv_1(B)) = \{ , \} \cap \{) \} = \emptyset$ donc ok pour cette règle
Cette grammaire est LL(1).

Procédures de la descente récursive:

Donner les procédures pour la descente récursive, de la grammaire précédente:

On rajoute la règle: Z → S #   S → b(SB) / a   B → ,SB / €

```
procedure Z(...)  /*Z → S #*/
debut
        si tc ∈ deb1(S)
          alors S(....)
                si tc=#
                    alors succès
                     sinon erreur
               fsi
          sinon erreur
        fsi
fin


Procedure B(..)  /*B → ,SB / €  */
debut
    si tc=','
      alors tc:=ts;
            si tc ∈deb1(S)
              alors S(...)
                    si tc ∈ deb1(B.suiv1(B))
                       alors B(....)
                      sinon erreur
              sinon erreur
     sinon si tc ∈ deb1(€.suiv(B))
               alors debut fin /*rien*/
               sinon erreur
  fin
```

```
procedure S(...)   /* S → b(SB) / a*/
debut
  case tc of
  'b': debut tc:=ts;
         si tc='('
          alors tc:=ts
                 si tc ∈ deb1(S)
                     alors S(.....)
                          si tc ∈ deb1( B) )
                           alors B(...)
                                si tc=')'
                                 alors tc:=ts
                                 sinon erreur
                           sinon erreur
                     sinon erreur
          sinon erreur
    fin
 'a': tc:=ts;
    else erreur; /*inutile*/
fin
```

B) Méthodes d'Analyse ascendante. De son nom, c'est une méthode d'analyse ascendante, c-à-d qu'elle part de la chaine à analyser, on fait une série de réduction (on réduit le α par A lorsqu'on utilise la règle $A \rightarrow \alpha$) jusqu'à aboutir à l'axiome. Il existe plusieurs méthodes d'analyse ascendante:

- LR(k) pour k=0,1,2,.... développée par Knuth.
- SLR(k) .... développée par DeRehmer & Pennello
- LALR(k)...... " " "

Le nombre k correspond au nombre de terminaux, de la chaine à analyser, à regarder pour décider de l'action à prendre. En pratique et en général on travaille avec des grammaires pour k=1 (des fois k=0).

On a une hiérarchie contenant l'ordre d'inclusion des différentes méthodes:
• les grammaires **SLR(1)** où "S" signifie "simple", est le cas le plus contraint, mais couvre le moins de langages ;
• les grammaires **LALR(1)** où "LA" signifie "lookahead" couvre beaucoup plus de langages que SLR, tout en gardant à la table d'analyse la même taille que dans le cas SLR(1) ;
• les grammaires **LR(1)** proprement dites sont les plus générales, mais au prix de tables d'analyse beaucoup plus volumineuses.

Exemple: La taille des tables pour un langage de la complexité grammaticale de Pascal est de l'ordre de la *centaine* dans les cas SLR(1) et LALR(1), et du *millier* dans le cas LR(1). La méthode LALR(1) est la plus utilisée dans les compilateurs actuels.

I) Méthode d'analyse LR: Il existe 2 manières pour vérifier si une grammaire est LR ou non: par les contextes ou par les items.

I1) Par les Contextes: Pour étudier cette méthode, on définit les notions de contexte suivantes: Pour chaque règle $A \rightarrow \alpha$:
- **Contexte gauche**: qu'on note CTXG = le contenu de la pile = ce qui a été reconnu jusqu'à présent lorsqu'on veut utiliser cette règle, ce CTXG € $\#.(T \cup N)^*$
- **Contexte droit**: qu'on note CTXD = ce qui peut rester à analyser, il € $T^*\#$.

- On calcule le **Contexte LR(k)** d'une règle, comme étant la concaténation du contexte gauche avec k caractères du contexte droit, il appartient à $\#.(T \cup N)^*$. (Une règle peut avoir plusieurs contextes).

LR(k): on regarde k car du contexte droit "c-à-d ce qui reste à analyser" pour décider de l'action à entreprendre.
LR(0): le contexte gauche suffit pour décider de l'action à entreprendre.

Définition: On dit qu'une grammaire est LR(k) par la méthode des contextes s'il n'a y pas un CTX LR(k) qui est sous mot gauche d'un autre CTX LR(K) c-à-d on n'a pas 2 CTXLR(k), α et $\alpha\varphi$ avec α € $(T \cup N)^*$ et φ € $(T \cup \#)^*$.

En ayant cette propriété sur les contextes, on peut dessiner un automate déterministe des contextes qu'on peut utiliser pour vérifier si une chaine appartient au langage ou non.

Implémentation: Pour implémenter cette méthode LR(k):

- On calcule les contextes gauches et droits des différentes règles de production de la grammaire.
- On vérifie si la grammaire est LR(k) en commençant par k=0 puis 1, 2.... jusqu'à l'obtention d'un k telle que la condition LR(k) sur les contextes soit vérifiée.
- On construit l'automate déterministe correspondant aux CTXLR(K).
- On construit la table d'analyse T correspondante à cet automate comme suit:
  - *Si on a une transition* $E_i \xrightarrow{A} E_j$ *alors* $T(E_i,A) = E_j$
  - *Si on a une transition* $E_i \xrightarrow{a} E_j$ *dans l'automate*

    alors si E_j *non final*

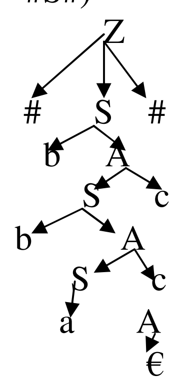
    alors $T(E_i,A)$ = *Décalage*, E_j

    sinon $T(E_i,A)$= *Réduction* $A \to \alpha$ *où cette règle correspond à la règle du CTX reconnu dans l'automate*

Pour analyser une chaine, on utilise cette table pour vérifier si cette chaine appartient au langage ou non.

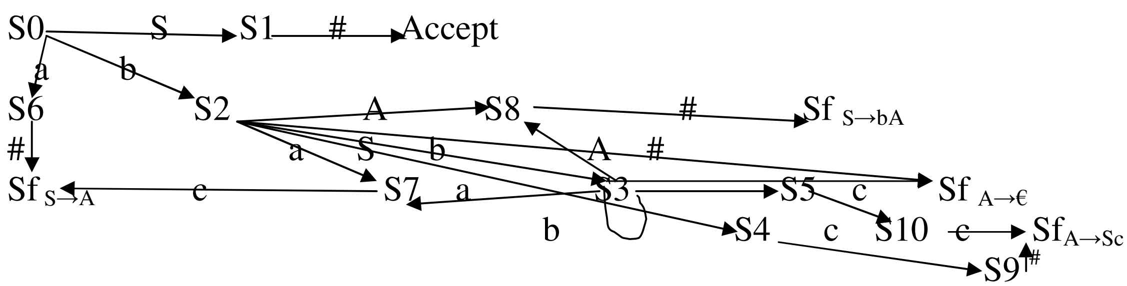
Exemple: Soit la grammaire $S \to bA$ / a $A \to Sc$ / € (on ajoute $Z \to \#S\#$)

Regle	**CTXG**	**CTXD**	**CTXLR(1)**
$Z \to S$	#S	#	#S#
$S \to bA$	$\#b^ibA$	$c^i\#$ $i \geq 0$	$\#b^ibAc$ $i \geq 1$
			#bA#
$S \to a$	$\#b^ia$	$c^i\#$ $i \geq 0$	$\#b^iac$ $i \geq 1$
			#a#
$S \to Sc$	$\#b^iSc$	$c^{i-1}\#$ $i \geq 1$	#bSc#
			$\#b^iScc$ $i \geq 1$
$A \to$ €	$\mathbf{\#b^i}$	$c^{i-1}\#$ $i \geq 1$	#b#
			$\#b^ic$ $i \geq 1$



On vérifie tout d'abord si elle est LR(0), c-à-d on regarde uniquement les contextes gauches. On remarque qu'on a un contexte gauche qui est sous mot d'un autre contexte: pour $\#b^i$ on a les CTXG #b #bb #bbb .... et donc elle n'est

pas LR(0). On détermine les contextes droits puis on calcule les contextes LR(1), en rajoutant le 1er car du CTXD au CTXG (faire attention à l'indice i), on obtient les CTX LR(1) donnés ci-dessus. On remarque qu'il n'y a pas un CTX qui est sous mot gauche d'un autre CTX et donc elle LR(1).
On dessine l'automate correspondant à ces contextes.



	A	b	c	#	S	A	B
0	6	2			1		
1				accept			
2	7	3		A→€	4	8	
3	7	3	A→€		5	8	
4			9				
5			10				
6				S→a			
7			S →a				
8				S →bA			
9				A →Sc			
10			A →Sc				

Analyse de la chaine bac#

#0	bac#	D,2 on fait un décalage et on va S0 par b.
#0b2	ac#	d,7
#0b2a7	c#	Réduire S → a et aller à S2 par S
#0b2S4	c#	D,9
#0b2S4c9	#	Réduire A →Sc
#0b2A8	#	Réduire S →bA
#0S1	#	Accepter

<u>I2) Par les items:</u> Un item est une production avec un point représentant une position dans un MDP α d'une règle $A \rightarrow \alpha$. Il est de la forme:

[$A \rightarrow \alpha.\beta,\psi$] où α correspond à ce qui a été déjà réduit,

β ce qui reste à construire avant la réduction de $\alpha\beta$ en A pour une règle $A \rightarrow \alpha\beta$. Cette réduction n'est possible que si ψ € $suiv_k(A)$

On commence avec l'item I_0=[Z →.S,$\#^k$] puis on fait la fermeture de cet élément. (On prend k=1)

Fermeture d'Item: On définit la fermeture d'un item par:
Pour chaque $[A \rightarrow \alpha.B\varphi, w]$ *avec* $B \in N$, $w \in (T \cup \#)^*$ *et* $\alpha, \varphi \in (T \cup N)^*$
faire pour chaque règle $B \rightarrow \beta$
faire ajouter $[B \rightarrow .\beta, \sigma]$ *avec* $\sigma \in deb_k(\varphi w)$
répéter ceci jusqu'à ce qu'il n'y ait aucun changement dans l'ensemble.

Exemple: soit la grammaire S → Sbc / a
I_0={[Z→.S,#], /* on ajoute ceux de S→ ... avec deb_1(#) */
[S→.Sbc,#],[S→.a,#], /*on rajoute ceux de S→ ... avec deb_1(bc#)*/
[S→.Sbc,b],[S→.a,b]} /*plus aucun changement, on arrete.*/

Fonction goto: On définit aussi une fonction goto(I_i, t) qui permet de changer l'état = décalage ou changement d'état avec $t \in T \cup N \cup \#$.
si I_i *contient* $[A \rightarrow \alpha . X\beta, w]$ *avec* $X \in (T \cup N)$
alors $I_j = goto(I_i, X) = fermeture([A \rightarrow \alpha X . \beta, w])$ /*on fait avancer le point*/

Exemple: On reprend la grammaire ci_dessus;
I_1=goto(I_0,S)=[Z→S.,#],[S→S.bc,#],[S→S.bc,b]
I_2=goto(I_0,a)= [S→a.,#], [S→a.,b].

Table LR: On construit la table LR(1) par les items de la manière suivante:
si $I_j = goto(I_i, A)$ *alors* $T(I_i, A) = I_j$ $A \in N$
si $I_j = goto(I_i, a)$ $a \in T$
alors pour chaque $[A \rightarrow \alpha . a\beta, w]$ *de* I_i
faire pour chaque élément de aw' de $deb_k(a\beta w)$
faire $T(I_i, aw')$ = *Décalage*, I_j
si I_i *contient* $[A \rightarrow \alpha., w]$ *alors* $T(I_i, w)$= *réduction de* $A \rightarrow \alpha$
Si cette table T est mono définie alors la grammaire est LR

Exemple: On reprend l'exemple ci-dessus et on continue à générer les goto.
I_3=goto(I_1,b)=[S→Sb.c,#],[S→Sb.c,b]
I_4=goto(I_3,c)=[S→Sbc.,#],[S→Sbc.,b] on arrete.

	a	b	c	#	S
I_0	2				1
I_1		3		Accepté	
I_2		S→a		S→a	
I_3			4		
I_4		S→Sbc		S→Sbc	

La table est mono définie alors la grammaire est LR(1).

Exemple analyse: Analyser la chaine abc#

Pile	Chaine	Action
0	abc#	Décalage et aller à 2
0a2	bc#	Réduire a en S puis aller à I0 par S c-a-d 1
0S1	bc#	Décalage et aller à 3
0S1b3	c#	Décalage et aller à 4
0S1b3c4	#	Réduire Sbc par S puis aller à I0 par S
0S1	#	Acceptée

Passage à LR(k): Si on veut travailler sur une grammaire LR(k) avec k≥2, le raisonnement est le meme. Les différences sont:
- Au lieu de raisonner avec $deb_1(\alpha.suiv_1(\alpha))$ on prend $deb_k((\alpha.suiv_k(\alpha))$
- La table contient $(\text{nombre de terminaux}+1)^k$ + nbre de Non terminaux colonne

Multi définitions: Il est à noter que des fois on peut enlever les multi définitions dans la table en prenant certaines conventions, pour rendre la grammaire LR(k)

Cas de multi définitions:

a) décalage / réduction

	a
I_i	Dec, I_j Red $A \rightarrow \alpha$

ceci correspond au cas Ii contient [B →β.aφ, t] et [A →α., a] là on décide est ce qu'on fait une réduction ou on continue le décalage (on définit une convention dans le langage).

b) Réduction/réduction

	a
I_i	Red $B \rightarrow \beta$ Red $A \rightarrow \alpha$

ceci correspond au cas où on a Ii contient [A →α., a] et [B →β., a]. Là aussi on définit des conventions pour décider est ce qu'on réduit α en A ou β en B.

c) Pour le cas Décalage/décalage, impossible à avoir.

II) Méthode d'analyse SLR:

II1) Avec les contextes *:* Soit θ le contexte gauche d'une règle $A \rightarrow \alpha$ défini précédemment, et $w \in suiv_k(A)$ alors θw est un contexte SLR(k) de $A \rightarrow \alpha$.

Pour construire la table d'analyse avec les contextes SLR(k), on procède de la même manière que pour la méthode LR en utilisant les contextes SLR(k) définis ci-dessus, c-à-d:

- *On cherche les CTX SLR(K)*
- *On vérifie si on n'a pas de CTX sous mot gauche d'un autre CTX*
- *On construit l'automate des CTX SLR(K)*
- *On construit la table d'analyse SLR(K)*

II2) Par les items, on reprend les items LR(0). L'item LR(0) de l'item LR(1) $[A \rightarrow \alpha.\beta,\ w]$ est l'item $[A \rightarrow \alpha.\beta]$). On produit tous les items comme dans la méthode LR en utilisant la fonction goto et la fermeture. La différence c'est au niveau de la table.

Table SLR: On construit la table SLR(k) par les items de la manière suivante: (c'est la même table que LR sauf au niveau des réductions où on regarde avec les $suiv_k$ des non terminaux).

si $I_j = goto(I_i, A)$ alors $T(I_i, A) = I_j$ $\quad A \in N$
si $I_j = goto(I_i, a)$ $\quad a \in T$
alors pour chaque $[A \rightarrow \alpha\,.\,a\beta]$ de I_i
faire pour chaque élément de aw' de $deb_k(a\beta)$
faire $T(I_i, aw') =$ Décalage, I_j
si I_i contient $[A \rightarrow \alpha.]$ alors $T(I_i, suiv_k(A)) =$ réduction de $A \rightarrow \alpha$

Si la table T obtenue est mono définie alors la grammaire est SLR(k)

III) Méthode d'analyse LALR: Pour la méthode LALR(k) on reprend les items LR(k) comme précédemment puis on regroupe dans la table d'analyse les items ayant le même corps d'items. 2 éléments d'items ont le même corps s'ils sont de la forme : $[A \rightarrow \alpha,\ w1]$ et $[A \rightarrow \alpha,\ w2]$ (la seule différence entre ces 2 éléments items est w1 et w2). En rassemblant les ensembles d'items ayant les meme corps, on obtient une table de taille éventuellement inférieur à celle de LR. Si cette table est mono définie alors la grammaire est LALR.

Analyse :

Pour l'analyse des chaines, elle se fait de la même manière pour les 3 méthodes en utilisant la table d'analyse :

Pile	**Chaine #**	**Action**
#0	………#	…….
…..	……..	……
#...E_i	tc…….#	Soit Blocage si $T(E_i,tc)=\emptyset$ Soit décalage, E_i Soit réduction de α par A

Jusqu'à l'épuisement de la chaine et de la pile.

<u>Génération de la forme intermédiaire:</u> C'est une forme intermédiaire qui permet de faciliter l'écriture des programmes. C'est la forme que génèrent les routines sémantiques. On trouve plusieurs formes intermédiaires:

- La forme post fixée (opérateur après les opérandes)
- la forme pré fixée (opérateur avant les opérandes)
- les quadruplets (champs de 4 éléments)
- les triplets (champs de 3 éléments)
- l'arbre abstrait (sous forme d'arbre)

<u>a)La forme Post-fixée :</u> C'est une forme intermédiaire dans laquelle, l'opérateur est écrit après les opérandes. Nous allons donner quelques exemples de formes post fixée (dans la suite fp(w) désigne forme post fixée de w):

Element	**sa forme post fixée**
Var	*var*
Cte	*cte*
Begin	*block*
End	*blockend*
*Opérateur + - / * exp1 op exp2*	*fp(exp1)fp(exp2)op*
Branchement vers une position pos	*pos BR*
Var := expr	*var fp(expr) :=*
Goto etiq	*etip BRL*

Opérateurs *Op1 op2 BP* *Branchement à op2 si op1 >0 (de meme BM BZ BNZ BMZ BPZ pour < = ≤..)*

If exp then inst1 else inst2 fp(exp) pos1 BZ fp(inst1) pos2 BR fp(inst2)

Après inst1 on saute inst2 à pos2
on va à pos1=début de inst2 si exp fausse

While exp do inst fp(expr) pos1 BZ fp(inst) pos2 BR

Si expr fausse (BZ) aller à la fin
Revenir à l'évaluation de l'expr

$A[L_1:N_1, L_2:N_2, \ldots\ L_n:N_n]$ $fp(L_1)fp(N_1)\ldots fp(L_n)fp(N_n)$ A adec

$A[exp_1,\ldots, exp_n)$ $fp(exp_1)fp(exp_2)\ldots fp(exp_n)$ A subs

On peut utiliser aussi les formes fp(op1) fp(op2) op BG pour exprimer : se brancher à op si op1>op2 (de meme pour BGE, BE, BNE, BL, BLE).

Exemple: Donner la forme post fixée de: var := expr arithmétique.

$x:=(a+b*c) + ((d+b*c)-f*g) + f*g*d$. Sa forme post fixée est:

11 2 1 7 4 3 6 5 10 8 9 (ordre des opérateurs)

$x\ a\ bc* + d\ \ bc* + fg* - + fg*d*+ :=$

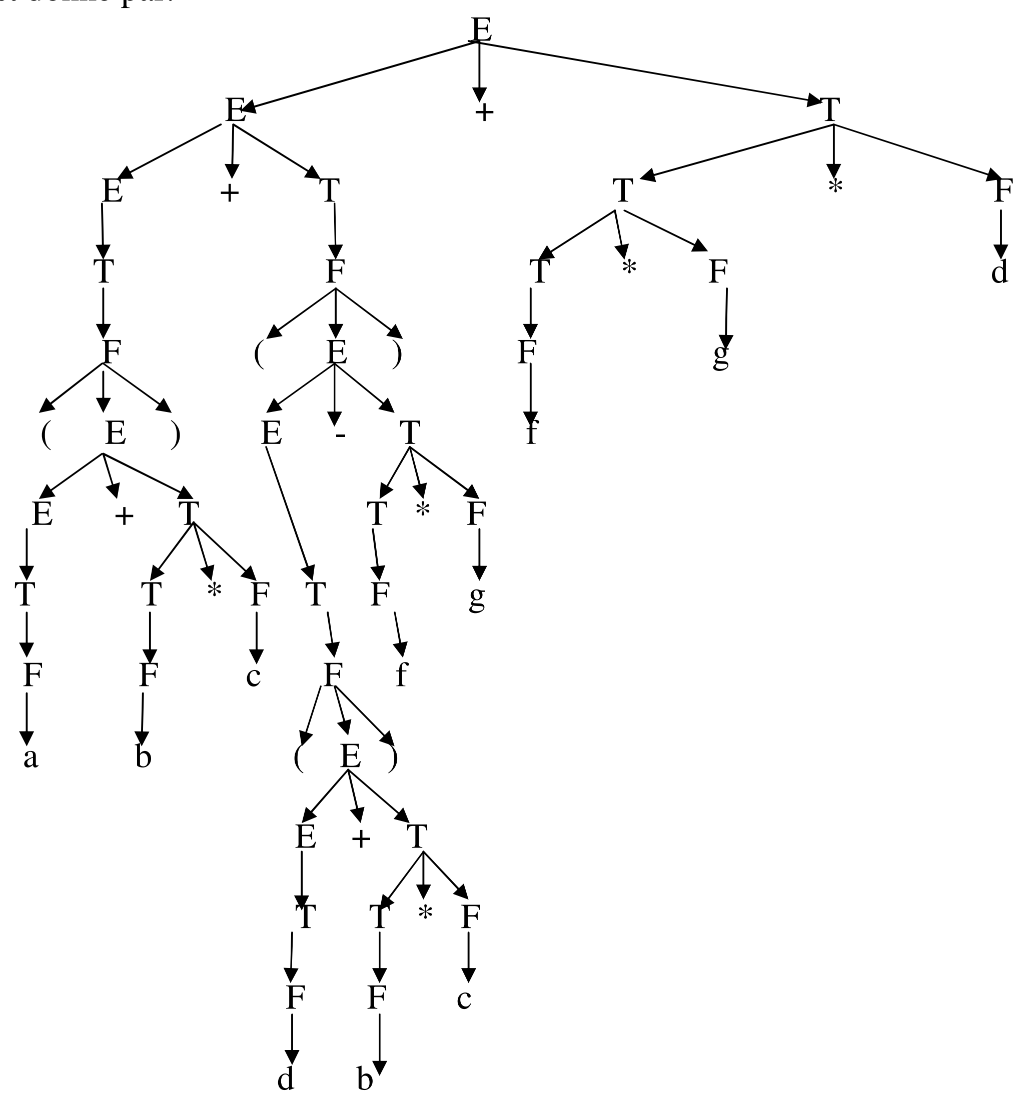
l'arbre syntaxique de l'expression arithmétique et en utilisant la grammaire des expressions arithmétiques:

$E \rightarrow E + T / E - T / T$

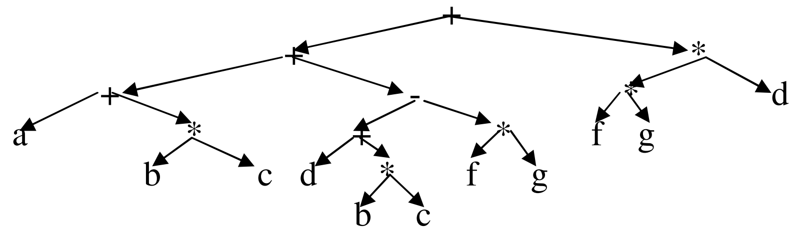
$T \rightarrow T * F / T/F / F$

$F \rightarrow id / (E)$

est donné par:



En omettant les parenthèses, et les noms inutiles on obtient l'arbre syntaxique:



On obtient rapidement les formes post-fixées (coté gauche puis coté droit puis opérateur), infixée (coté gauche opérateur coté droit) et préfixée (opérateur coté gauche coté droit) en parcourant cet arbre syntaxique.

Exemple2: donner la forme post fixée du programme suivant:

*case b*b-4*a*c of*
0: x:=b/2;
1,2,3: begin $x:=b-x_2;x_2:=c/x;$*end;*
otherwise x:=a+b+c;

On peut transformer (pour faciliter la compréhension) cette instruction en:

*if b*b-4*a*c - 0=0*
then x:=b/2
*else if (b*b-4*a*c - 1=0) or (b*b-4*a*c - 2=0)or(b*b-4*a*c - 3=0)*
then begin $x:=b-x_2;x_2:=c/x;$*end*
else x:=a+b+c;

sa forme post fixée est: else

*bb*4a*c*- 0 - else bnz xb2/:= fin BR bb*4a*c*-1 vrai BZ bb*4a*c*-2 vrai BZ*
*bb*4a*c*-3 vrai BZ xab+c+:= fin BR block* xbx_2*-:=*x_2cx*/:= blockend*

vrai fin

où else, fin, vrai sont des positions dans la chaine post fixée.

<u>b)La forme pré fixée :</u> c'est la meme chose que la forme post fixée sauf la position de l'opérateur qui est au début au lieu de la fin.

<u>c) Les quadruplets :</u> La forme des quadruplets est un ensemble de 4 champs.

Instruction	**Forme**
exp_1 *op* exp_2	*(op, resultat(fp(*exp_1*)), resultat(fp(*exp_2*)), Tmp) où op est un Opérateur arithmétique*
Aller I	*(BR, i, ,)*
Aller à i si T=0	*(BZ, i, T,) de meme pour BMZ, BNZ,…*
Aller à i si T1>T2	*(BG, i, T1, T2) de meme pour BGE, BL, BLE, BE, BNE,etc…*
Goto i	*(BRL, i, ,)*

Var := exp *(:=, resultat(fp(exp)), var,)*
Pour les tableaux *on utilise les déclarations (bounds,T_{i1},T_{i2},) puis (adec,A, ,)*
A[exp_1,…,exp_n] *quadruplets(exp_1) résultat dans T_1*
…….
quadruplet(exp_n) résultat dans T_n
(A[T_1,…,T_n], , ,)
(block, , ,)
(blockend, , ,)

Exemple: Donner les quadruplets générés par l'instruction suivante:
(a+b*c) + ((d+b*c)-f*g) +f*g*d

(*, b, c, T1)
(+, a, T1, T2)
(*, b, c, T3)
(+, d, T3, T4)
(*, f, g, T5)
(-, T4, T5, T6)
(+, T2, T6, T7)
(*, f, g, T8)
(*, T8, d, T9)
(+, T7, T9, T10)

d) **Triplet :** C'est la meme forme que les quadruplets sauf qu'on manipule 3 champs uniquement. L'ordre d'exécution des triplets est l'ordre de leur écriture. Au lieu de faire référence à un temporaire comme dans les quadruplets, on fait référence au N° du triplet.

Exemple: Donner les triplets de: (a+b*c) + ((d+b*c)-f*g) +f*g*d
1) (*, b, c)
2) (+, a, (1))
3) (*, b, c)
4) (+, d, (3))
5)(*, f, g)
6) (-, (4), (5))
7) (+, (2), (6))
8) (*, f, g)
9) (*, (8), d)
10) (+, (7), (9))
le résultat est dans le triplet (10)

e) **Triplet indirecte :** C'est la meme chose que les triplets, sauf qu'on ajoute un vecteur contenant l'ordre d'éxécution des triplets. L'avantage de ce mode est qu'il est inutile de réecrire les triplets identiques. (Les triplets 1 et 3, ainsi que 5 et 8 sont identiques).

f) **<u>Arbre syntaxique :</u>** c'est un arbre syntaxique qui exprime les différentes instructions, c'est un arbre réduit où on élimine les temporaires.

Expressions arithmétique — opérateur

Arbre(op1) arbre(op2)

If exp then inst1 else inst2 — if

Arbre(Exp) arbre(inst1) arbre(inst2)

Begin inst1 inst2 ...... instn end — begin

arbre(inst1) .... arbre(instn)

while exp_bool do inst — while

arbre(exp_bool) arbre(inst)

goto etip — goto

etiq

déclaration tableau array $A[L_1:N_1, ..., L_n:N_n]$ — Array

A L_1 N_1 ..... L_n N_n

<u>Parcours de l'arbre abstrait</u>: Le parcours d'un arbre abstrait peut se faire de plusieurs manières:

Post fixée: sous-arbre gauche , sous-arbre droit, racine
pré fixée: racine, sous-arbre gauche, sous-arbre droit
infixée: sous-arbre gauche, racine, sous-arbre droit