Chapitre 3 : Analyse syntaxique

1. Introduction

Le rôle principal de l'analyse syntaxique est de vérifier si l'écriture du programme source conforme avec la syntaxe du langage à compilé. Cette dernière est spécifiée à l'aide d'une grammaire hors contexte.

L'analyseur syntaxique reçoit une suite d'unités lexicales de la part de l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage.

Le problème est donc :

- Étant donnée une grammaire G
- Étant donnée un mot m (un programme)

Est ce que m appartient au langage génère par G ? Le principe est d'essayer de construire un **arbre de dérivation.**

Il existe plusieurs méthodes d'analyse appartenant à l'une des deux catégories qui sont l'analyse **descendante** et l'analyse **ascendante**.

Dans l'analyse **descendante** nous essayons de dériver à partir de l'axiome de la grammaire le programme source. D'une façon opposée, l'analyse **ascendante** établit des réductions sur les chaines à analyser pour aboutir à l'axiome de la grammaire.

2. Grammaire et arbre de dérivation

2.1 Définition 1: Une grammaire **hors contexte** est un quadruplet :

G = (T, NT, S, P) où :

- **T** est l'ensemble des **symboles terminaux** du langage. Les symboles terminaux correspondent aux mots découvert par l'analyseur lexical « unité lexicale ».par exemple : if , else sont des terminaux.
- **NT** est l'ensemble des **symboles non-terminaux** du langage. Ces symboles n'apparaissent pas dans le langage mais dans les règles de la grammaire définissant le langage. Ils permettent d'exprimer la structure des règles grammaticales.
- **S ϵ NT** est appelé l'élément de départ de G (ou **axiome de G**). Le langage que G décrit (noté L(G)) correspond à l'ensemble des phrases qui peuvent être dérivées à partir de S par les règles de la grammaire.
- **P** est un ensemble de **production** (ou règles de réécriture) de la forme N-→$\alpha 1\ \alpha 2 \ldots \alpha n$ avec $\alpha i\ \epsilon$ (TUNT)*. C'est à dire que chaque élément de P associe un non terminal à une suite de terminaux et non terminaux. Le fait que les parties gauches des règles ne contiennent qu'un seul non terminal **donne la propriété "hors contexte"à la grammaire.**

Exemple 1:

Symboles terminaux : T= {a,b}

Symboles non terminaux NT={S}

Axiome=S

Règles de production : {S→ ε

S→aSb

Exemple 2 :

Exemple G = (T, NT, S, P) avec

T= {il, elle, parle, est, devient, court, reste, sympa, vite}

NT = {PHRASE, PRONOM, VERBE, COMPLEMENT, VERBETAT, VERBACTION}

S = PHRASE

P = {

PHRASE →PRONOM VERBE COMPLEMENT

PRONOM →il |elle

VERBE→VERBETAT |VERBACTION

VERBETAT→est |devient| reste

VERBACTION→parle |court

COMPLEMENT→sympa | vite}

2.2 Arbre de dérivation

On appelle dérivation l'application d'une ou plusieurs règles à partir d'un mot de(T U NT) +

On notera → une dérivation obtenue par application d'une seule règle de production,

et →* une dérivation obtenue par l'application de n règles de production où n > 0.

Selon la grammaire de l'exemple 1 :

S→ ε

S→aSb

S→* ab

S→* aaabbb

S→* aaSbb

Selon la grammaire de l'exemple 2:

PHRASE→SUJET VERBE COMPLEMENT →*elle VERBETAT sympa

PHRASE →*il parle vite

PHRASE →*elle court sympa

Remarque : il est possible de générer des phrases syntaxiquement correctes mais qui n'ont pas de sens. C'est l'analyse sémantique qui permettra d'éliminer ce problème.

Définition 2: Étant donnée une grammaire G, on note L(G) le langage génère par G et défini par tous les mots composés uniquement de symboles terminaux que l'on peut former à partir de S.

Selon l'exemple 1, nous donnons L(G)= $\{a^n b^n, n>=0\}$

Définition 3: Un arbre de dérivation syntaxique pour la grammaire G de racine Sϵ NT et de feuilles wϵT* est un arbre ordonné dont la racine est S, les feuilles sont étiquetées par des terminaux formant le mot w.

Dérivations gauches et droites :

- Dérivation gauche consiste à réécrire le symbole non-terminal le plus à gauche à chaque étape.
- Dérivation droite consiste à réécrire le symbole non-terminal le plus à droite à chaque étape.

Exemple :

Soit la grammaire ayant S pour axiome et pour règles de production

P ={ S→ aT b /c

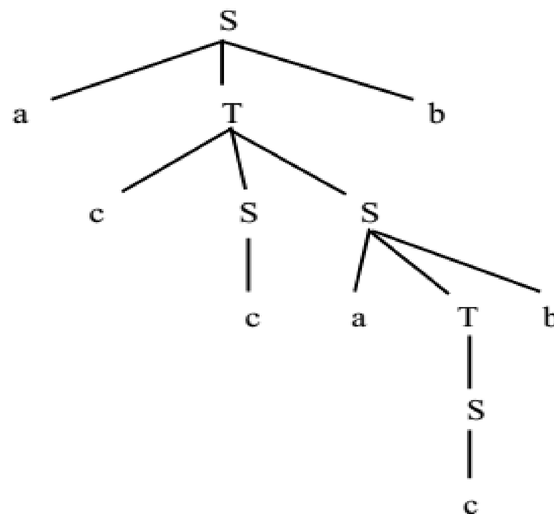
T→cSS /S}

Un arbre de dérivation pour le mot accacbb est :

S→aTb→acSSb→accSb→accaTbb→accaSbb→accacbb (dérivation gauche)

Ou S→aTb →acSSb→acSaTbb→acSaSbb→acSacbb→accacbb (dérivation droite)

Ces deux suites différentes de dérivations donnent **le même arbre de dérivation**



3. Grammaire LL(1) :

Théorème : Une grammaire ambiguë ou récursive à gauche ou non factorisée à gauche n'est pas LL(1).

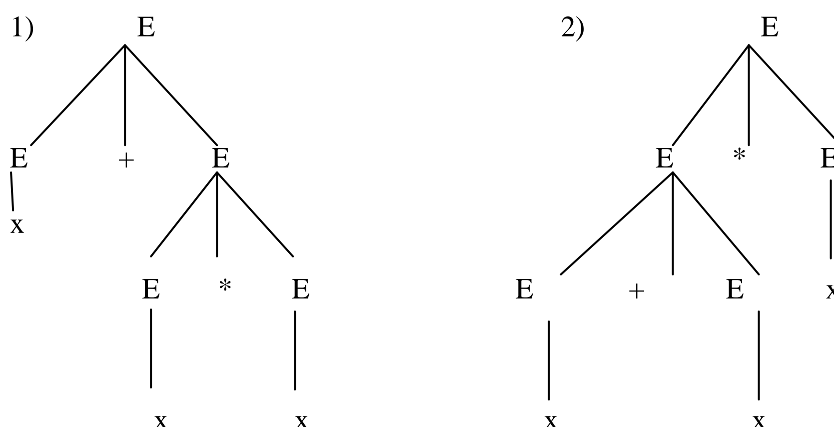
3.1 Ambiguïté

Si une phrase d'un langage L(G) possède deux arbres de dérivations distincts, alors cette phrase est ambiguë et la grammaire associée est elle-aussi ambiguë.

<u>Exemple d'une grammaire ambiguë</u>

La grammaire de l'expression suivantes est ambiguë ; il ya deux arbres syntaxiques possibles pour x+x*x

E →E+E |E*E |x



3.2 Récursivité à gauche

Une grammaire **G** est **Récursive Gauche** (**RG**) s'il existe une dérivation de la forme : $A \rightarrow^{+} Aw$, avec $A \in$ **N** et $w \in$ (**N** ∪ **T**)* ou N (ensemble de symboles ***non-terminaux), T (***symboles ***terminaux)***

Élimination de la récursivité à gauche immédiate

Remplacer toute règle de la forme **$A \rightarrow A\alpha|\beta$** par les deux règles :
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' | \epsilon$
Ou : toute règle de la forme **$A \rightarrow A\alpha_1 | A\alpha_2 | .. A\alpha_n | \beta_1 | \ldots \beta_n$** par les deux règles:
$A \rightarrow \beta_1 A' | \ldots \beta_n A'$
$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots \alpha_n A' | \epsilon$

Élimination de la récursivité à gauche

Ordonner les non-terminaux $A_1, A_2, \ldots A_n$
Pour i=1 à n faire
    pour j=1 à i-1 faire
        remplacer chaque production de la forme **$A_i \rightarrow A_j\alpha$ où $A_j \rightarrow \beta_1 | \ldots | \beta_p$** par **$A_i \rightarrow \beta_1\alpha | \ldots | \beta_p\alpha$**
    fin pour
    éliminer les récursivités à gauche immédiates des productions A_i
fin pour

Exemple

Soit la production suivante:

E→E+T|C

Donc pour éliminer la récursivité à gauche, on obtient :

E→CE'

E'→+TE'| €

3.3 Factorisation à gauche

L'idée de base est que pour développer un non-terminal *A* quand il n'est pas évident de choisir l'alternative à utiliser (c à d quelle production prendre), on doit réécrire les productions de *A* de façon à **différer** la décision jusqu'à ce que suffisamment de texte ait été lu pour faire le bon choix.

Exemple :

$$\begin{cases} S \rightarrow bacdAbd | bacdBcca \\ A \rightarrow aD \\ B \rightarrow cE \\ C \rightarrow \ldots \\ E \rightarrow \ldots \end{cases}$$

Au départ, pour savoir s'il faut choisir $S \rightarrow bacdAbd$ ou $S \rightarrow bacdBcca$, il faut avoir lu la 5ème lettre du mot (un a ou un c). On ne peut donc pas dès le départ savoir quelle production prendre.

Factorisation à gauche :

Pour chaque non-terminal A
trouver le plus long préfixe α commun à deux de ses alternatives ou plus
Si $\alpha \neq \varepsilon$, remplacer $A \rightarrow \alpha\beta_1|\ldots|\alpha\beta_n|\gamma_1|..|\gamma_p$ (où les γ_i ne commencent pas par α) par le deux règles:
$A \rightarrow \alpha A'|\gamma_1|..|\gamma_p$
$A' \rightarrow \beta_1|\ldots|\beta_n$

Fin pour
Recommencer jusqu'à ne plus en trouver.

Exemple

E→abE|abA|abC

Factorisée à gauche, cette production devient :

E→abE'

E'→E|A|C

4. Mise en œuvre d'un analyseur syntaxique

L'analyseur syntaxique reçoit une suite d'unités lexicales (de symboles terminaux) de la part de l'analyseur lexical. Il doit dire si cette suite (ce mot) est syntaxiquement correcte, c'est à dire si c'est un mot du langage généré par la grammaire qu'il possède. Il doit donc essayer de construire l'arbre de dérivation de ce mot. S'il y arrive, alors le mot est syntaxiquement correct, sinon il est incorrect.

Il existe deux approches (deux méthodes) pour construire cet arbre de dérivation : une méthode **descendante** et une méthode **ascendante**.

4.1 Analyse descendante

1. Définition

L'analyse descendante, dans laquelle on construit l'arbre en descendant de la racine (la racine c'est à dire l'axiome de départ) vers les feuilles (les unites lexicales). Il s'agit de deviner à chaque étape quelle est la règle qui sert à engendrer le mot que l'on lit.

Exemple

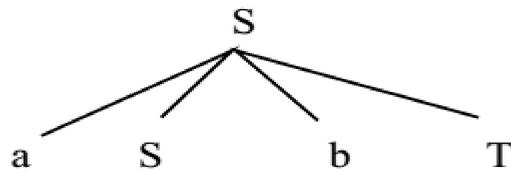
S→aSbT|cT|d avec le mot w=accbbadbc

T→aT|bS|c

On part avec l'arbre contenant le seul sommet S

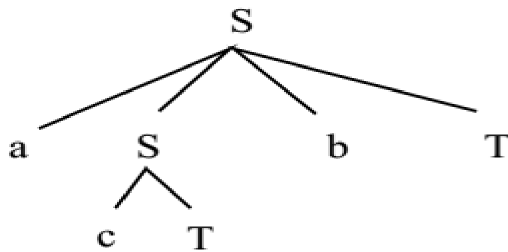
La lecture de la première lettre du mot (a) nous permet d'avancer la construction

W=accbbadbc

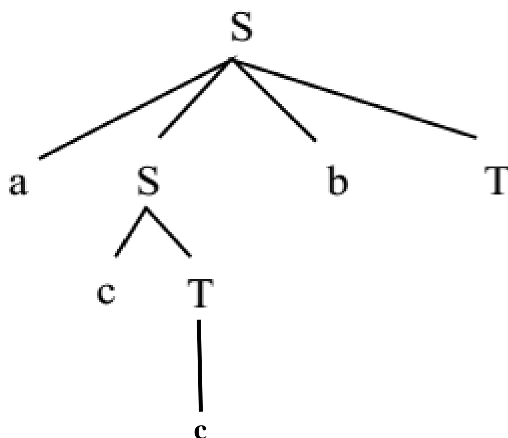


Puis la deuxième lettre nous amène à :

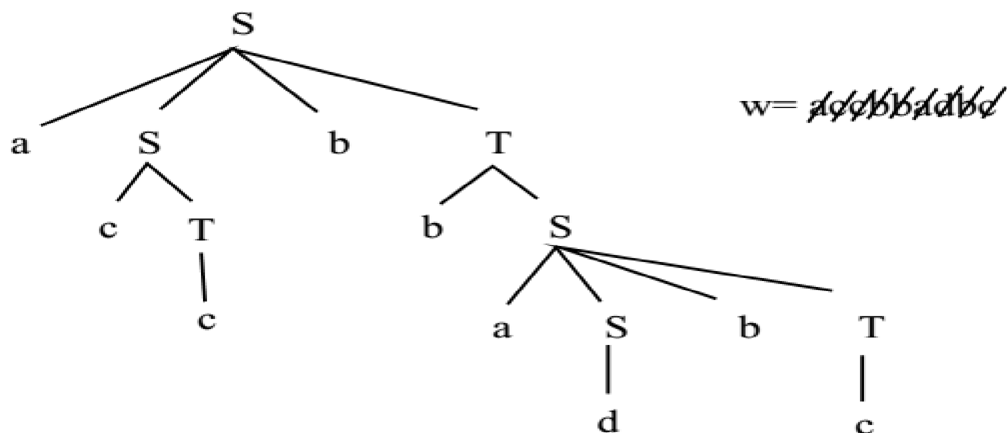
W= accbbadbc



Puis la troisième lettre nous amène à : **W= accbbadbc**



Et ainsi de suite jusqu'à :



On a trouvé un arbre de dérivation, donc le mot appartient au langage.

Remarque : Si notre grammaire est LL(l), l'analyse syntaxique peut se faire par l'analyse descendante. Mais comment savoir que notre grammaire est LL(1) ?

Étant donnée une grammaire

1- la rendre non

2- éliminer la récursivité à gauche si nécessaire

3- la factoriser à gauche si nécessaire

4- construire la table d'analyse

2. Table d'analyse LL(1)

Pour construire une table d'analyse, on a besoin des ensembles **PREMIER** et **SUIVANT**.

1) Calcul de PREMIER :

Pour toute chaine composée de symboles terminaux et non_terminaux, on cherche PREMIER(α) l'ensemble de tous les terminaux qui peuvent commencer une chaine qui se dérive de α.

Exemple :

On la grammaire suivant :

$$\begin{cases} E \to TE' \\ E' \to +TE' \mid -TE' \mid \varepsilon \\ T \to FT' \\ T' \to *FT' \mid /FT' \mid \varepsilon \\ F \to (E) \mid \text{nb} \end{cases}$$

$$\begin{aligned} &\text{PREMIER}(E) &&= \text{PREMIER}(T) &&= \{(, \text{nb}\} \\ &\text{PREMIER}(E') &&= \{+, -, \varepsilon\} \\ &\text{PREMIER}(T) &&= \text{PREMIER}(F) &&= \{(, \text{nb}\} \\ &\text{PREMIER}(T') &&= \{*, /, \varepsilon\} \\ &\text{PREMIER}(F) &&= \{(, \text{nb}\} \end{aligned}$$

2) Calcul de SUIVANT

Pour tout non_terminal A, on cherche SUIVANT(A), l'ensemble de tous les symboles terminaux a qui peuvent apparaitre immédiatement à droite de A dans une dérivation.

Algorithme de construction des ensembles SUIVANT :

1. Ajouter un marqueur de fin de chaîne (symbole \$ par exemple) à SUIVANT(S) (où S est l'axiome de départ de la grammaire)
2. Pour chaque production $A \to \alpha B \beta$ où B est un non-terminal, alors ajouter le contenu de PREMIER(β) à SUIVANT(B), **sauf** ε
3. Pour chaque production $A \to \alpha B$, alors ajouter SUIVANT(A) à SUIVANT(B)
4. Pour chaque production $A \to \alpha B \beta$ avec $\varepsilon \in$PREMIER(β), ajouter SUIVANT(A) à SUIVANT(B)

Recommencer à partir de l'étape 3 jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles SUIVANT.

Exemple : Toujours avec la même grammaire **:**

$$\begin{cases} E \to TE' \\ E' \to +TE' \mid -TE' \mid \varepsilon \\ T \to FT' \\ T' \to *FT' \mid /FT' \mid \varepsilon \\ F \to (E) \mid \text{nb} \end{cases}$$

$$\begin{aligned} \text{SUIVANT}(E) &= \{ \ \$,) \ \} \\ \text{SUIVANT}(E') &= \{ \ \$,) \ \} \\ \text{SUIVANT}(T) &= \{ \ +, -, \ ), \$ \ \} \\ \text{SUIVANT}(T') &= \{ \ +, -, \ ), \$ \ \} \\ \text{SUIVANT}(F) &= \{ \ *, /, \ ), +, -, \$ \ \} \end{aligned}$$

3) Construction de la table d'analyse LL(1)

Une table d'analyse est un tableau M a deux dimensions qui indique pour chaque symbole non_terminal « **A** » et chaque symbole terminal « **a** » ou symbole **$** la règle de production à appliquer.

Algorithme de construction de la table d'analyse LL (1):

- Pour chaque production $A \to \alpha$ faire
  1. pour tout $a \in$PREMIER(α) (et $a \neq \varepsilon$), rajouter la production $A \to \alpha$ dans la case $M[A, a]$
  2. si $\varepsilon \in$PREMIER(α), alors pour chaque $b \in$SUIVANT(A) ajouter $A \to \alpha$ dans $M[A, b]$
- Chaque case $M[A, a]$ vide est une erreur de syntaxe

Exemple : selon la même grammaire précédente, la table d' analyse LL (1) est comme suit:

	nb	+	−	*	/	(	)	$
E	$E \to TE'$					$E \to TE'$		
E'		$E' \to +TE'$	$E' \to -TE'$				$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$					$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$	$T' \to /FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ nb					$F \to (E)$		

3. Analyseur syntaxique

Maintenant qu'on a la table, comment l'utiliser pour déterminer si un mot **m** donné est tel que

S→*m ? , On utilise une pile.

Algorithme :

Données : mot m termine par $, table d'analyse M

Initialisation de la pile :

S
$\$$

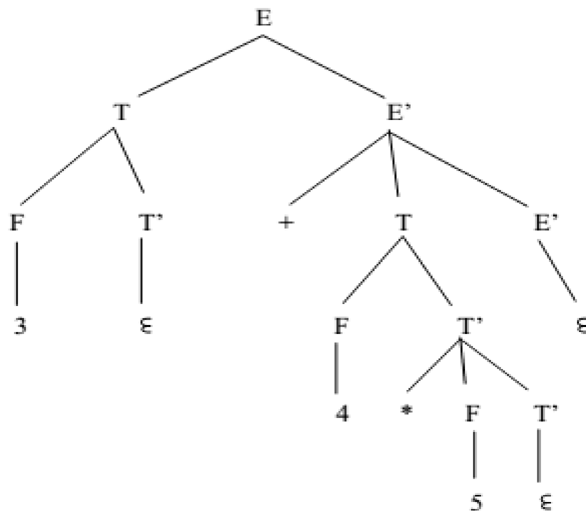
et un pointeur ps sur la 1 ère lettre de m

```
repeter
        Soit X le symbole en sommet de pile
        Soit a la lettre pointée par ps
        Si X est un non terminal alors
                Si M[X, a] = X → Y1 ...Yn alors
                        enlever X de la pile
                        mettre Yn puis Yn−1 puis ...puis Y1 dans la pile
                        émettre en sortie la production X → Y1 ...Yn
                sinon           (case vide dans la table)
                        ERREUR
                fin si
        Sinon
                Si X =$ alors
                        Si a = $ alors ACCEPTER
                        Sinon ERREUR
                        fin si
                Sinon
                        Si X = a alors
                                enlever X de la pile
                                avancer ps
                        sinon
                                ERREUR
                        fin si
                fin si
        finsi
jusqu'à ERREUR ou ACCEPTER$
```

Exemple 1 : analysez le mot : m= 3 *4 +5, en utilisant la table d'analyse de l'exemple précédent :

PILE	Entrée	Sortie
$\$ E$	$3 + 4 * 5\$$	$E \to TE'$
$\$ E'T$	$3 + 4 * 5\$$	$T \to FT'$
$\$ E'T'F$	$3 + 4 * 5\$$	$F \to \text{nb}$
$\$ E'T'3$	$3 + 4 * 5\$$	
$\$ E'T'$	$+4 * 5\$$	$T' \to \varepsilon$
$\$ E'$	$+4 * 5\$$	$E' \to +TE'$
$\$ E'T+$	$+4 * 5\$$	
$\$ E'T$	$4 * 5\$$	$T \to FT'$
$\$ E'T'F$	$4 * 5\$$	$F \to \text{nb}$
$\$ E'T'4$	$4 * 5\$$	
$\$ E'T'$	$*5\$$	$T' \to *FT'$
$\$ E'T'F*$	$*5\$$	
$\$ E'T'F$	$5\$$	$F \to \text{nb}$
$\$ E'T'5$	$5\$$	
$\$ E'T'$	$\$$	$T' \to \varepsilon$
$\$ E'$	$\$$	$E' \to \varepsilon$
$\$$	$\$$	ACCEPTER (analyse syntaxique réussie)

On obtient donc l'arbre syntaxique suivant:



Exemple 2 : analysez le mot : m= (7+3)5, en utilisant la même table d'analyse.

PILE	Entrée	Sortie
$\$ E$	$(7+3)5\$$	$E \rightarrow TE'$
$\$ E'T$	$(7+3)5\$$	$T \rightarrow FT'$
$\$ E'T'F$	$(7+3)5\$$	$F \rightarrow (E)$
$\$ E'T')E($	$(7+3)5\$$	
$\$ E'T')E$	$7+3)5\$$	$E \rightarrow TE'$
$\$ E'T')E'T$	$7+3)5\$$	$T \rightarrow FT'$
$\$ E'T')E'T'F$	$7+3)5\$$	$F \rightarrow 7$
$\$ E'T')E'T'7$	$7+3)5\$$	
$\$ E'T')E'T'$	$+3)5\$$	$T' \rightarrow \varepsilon$
$\$ E'T')E'$	$+3)5\$$	$E' \rightarrow$ * T E'
$\$ E'T')E'T+$	$+3)5\$$	
$\$ E'T')E'T$	$3)5\$$	$T \rightarrow FT'$
$\$ E'T')E'T'F$	$3)5\$$	$F \rightarrow 3$
$\$ E'T')E'T'3$	$3)5\$$	
$\$ E'T')E'T'$	$)5\$$	$T' \rightarrow \varepsilon$
$\$ E'T')E'$	$)5\$$	$E' \rightarrow \varepsilon$
$\$ E'T')$	$)5\$$	
$\$ E'T'$	$5\$$	**ERREUR !!**

Donc ce mot n'appartient pas au langage généré par cette grammaire.

Remarque importante:

L'algorithme de l'analyse descendante ne peut pas être appliqué à toutes les grammaires. En effet, si la table d'analyse comporte des entrées multiples (plusieurs productions pour une même case M [A,a], on ne pourra pas faire une telle analyse descendante car on ne pourra pas savoir quelle production appliquer.

Donc : grammaire LL (1) est une grammaire pour laquelle la table d'analyse décrite précédemment n'a aucune case définie de façon multiple (chaque case contient au plus une règle de production).

Par exemple : on a la grammaire suivant :

$$\begin{cases} S \to aAb \\ A \to cd|c \end{cases}$$

Nous avons PREMIER (S)= {a}, PREMIER(A)={c}, SUIVANT(S)= {$},et SUIVANT(A)={b}, ce qui donne la table d'analyse.

	a	c	b	d	$\$$
S	$S \to aAb$				
A		$A \to cd$ $A \to c$			

Il y a deux réductions pour la case M [A,c], donc ce n'est pas une grammaire LL(1). On ne peut pas utiliser cette méthode d'analyse. Donc, pour pouvoir choisir entre la production A→cd et la production A→c, il faut lire la lettre qui suit celle que l'on pointe (donc deux symboles de prévision sont nécessaires).

4.2 Analyse ascendante

Construire un arbre de dérivation du bas (les feuilles, ou les unités lexicales) vers le haut (la racine, ou l'axiome de départ).

Le modèle général utilisé est le modèle par **décalages–réductions**. C'est à dire que l'on ne s'autorise que deux opérations :

- ✓ **Décalage (Shift)** : décaler d'une lettre le pointeur sur le mot en entrée
- ✓ **Réduction (reduce)** : réduire une chaîne (suite consécutive de terminaux et non terminaux à gauche du pointeur sur le mot en entrée et finissant sur ce pointeur) par un non-terminal en utilisant une des règles de production.

Exemple :

Soit la grammaire G ayant les règles de production suivantes :

S→aABe

A→Abc| b

B→d

On se propose d'analyser la chaîne abbcde de manière ascendante :

a b b c d e décaler

a b b c d e réduire

a A b c d e décaler

a A b c d e décaler

a A b c d e réduire

a A d e décaler

a A d e réduire

a A B e décaler

a A B e réduire

S Analyse réussie

1. Les analyseurs LR

L'analyse LR (ou LR(k)) est une technique générale efficace d'analyse syntaxique ascendante, basée sur le modèle par décalage-réduction et qui peut être utilisée pour analyser une large classe de grammaires non contextuelles.

L : Left to right scanning (on parcourt ou on analyse la chaîne en entrée de la gauche vers la droite)

R : constructing a Rightmost derivation in reverse (en construisant une dérivation droite inverse)

k : on utilise k symboles d'entrée de prévision à chaque étape nécessitant la prise d'une décision d'action d'analyse.

Les analyseurs LR comportent :

- Un algorithme d'analyse commun aux différentes méthodes LR.
- Une table d'analyse dont le contenu diffère selon le type d'analyseur LR.

On distingue trois techniques de construction de tables d'analyse LR pour une grammaire donnée :

Simple LR (SLR), LR canonique, LookAhead LR (LALR).

1.1 L'analyse SLR

Un analyseur SLR ou LR simple (left to right, rightmost derivation) c'est un analyseur pour les grammaires non contextuelles qui lit l'entrée de gauche à droite et produit une dérivation droite.

La construction d'une table d'analyse SLR (SLR(1)) à partir d'une grammaire est la méthode la plus simple à implémenter. Elle constitue un bon point de départ pour l'étude de l'analyse LR.

Le principe de la méthode SLR est de construire, à partir de la grammaire un automate fini déterministe qui reconnaît les préfixes viables de G (préfixes pouvant apparaître sur la pile) puis de transformer cet automate en une table d'analyse.

La construction des analyseurs SLR, pour une grammaire donnée, est basée **sur la collection d'ensembles d'items LR(0)**. Pour construire cette collection, on définit une grammaire augmentée, **une fonction fermeture** et une **fonction transition**.

Les étapes à suivre :

- Augmentation de la grammaire S'→S
- On pose l'item I={ S'→.S}
- Calcul de l'item I_0 : I_0 =Fermeture(I)

- Calcul de la transition dans I_0 : $\Delta(I_0,X)$ pour construire des nouveaux items.
- Arrêt jusqu'à ce qu'aucun nouveaux item ne peut plus trouver.
- Construction de la table d'analyse SLR(1)

➢ Collection des items LR(0) d'une grammaire

Exemple

Soit la grammaire G des expressions arithmétiques numérotées comme suit:

1. E → E+ T
2. E → T
3. T →T* F
4. T → F
5. F → (E)
6. F → nb

Étape 1 : Augmentation de la grammaire :

S→E

E → E+ T
E → T
T →T* F
T → F
F → (E)
F → nb

Étape 2 : calcul de l'item I

I={S→.E}

Étape 3 : calcul de l'item I_0 :I_0=Fermeture(I)

I_0=Fermeture{S→.E}

Comment calculer la Fermeture d'un ensemble d'items I ?

Fermeture d'un ensemble d'items LR(0):

1- Mettre chaque item de I dans Fermeture(I)
2- Pour chaque item i de Fermeture(I) de la forme $A \rightarrow \alpha.B\beta$
    pour chaque production $B \rightarrow \gamma$
        rajouter (s'il n'y est pas déja) l'item $B \rightarrow .\gamma$ dans Fermeture(I)
    finpour
  finpour
3- Recommencer 2 jusqu'à ce qu'on n'ajoute rien de nouveau

Donc d'après l'algorithme, on obtient :
I_0={S→.E, E → .E+ T, E → .T, T →.T* F, T → .F, F → .(E), F → .nb}

Étape 4 :

- Calcul de la transition dans I_0 : $\Delta(I_0,X)$:

Transition par X d'un ensemble d'items I :

$\Delta(I,X)$ =Fermeture(tous les items $A \rightarrow \alpha X.\beta$) où $A \rightarrow \alpha.X\beta \in I$

On a : I_0={S→.**E**, E → .**E**+ T, E → .**T** , T →.**T*** F, T → .**F**, F → .(E), F → .**nb**} Avec : X={E,T,F,(, nb}, donc :

$\Delta(I_0,E)$= {S→E., E →E.+T}=$\mathbf{I_1}$
$\Delta(I_0,T)$= {E → T., T→T.*F}=$\mathbf{I_2}$
$\Delta(I_0,F)$= {T→ F.}=$\mathbf{I_3}$
$\Delta(I_0,()$= {F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=$\mathbf{I_4}$
$\Delta(I_0,nb)$={ F →nb.}=$\mathbf{I_5}$

- Calcul de la transition dans I_1 : $\Delta(I_1,X)$:

I_1={S→E., E →E.+T}, avec X={+}
$\Delta(I_1,+)$= {E →E+.T, T→.T*F, T→. F, F →.nb, F →.(E) }=$\mathbf{I_6}$

- Calcul de la transition dans I_2 : $\Delta(I_2,X)$:

I_2={E → T., T→T.*F}, avec X={*}
$\Delta(I_2,*)$={ T→T*.F, F →.nb, F →.(E)}=$\mathbf{I_7}$

- Calcul de la transition dans I_3 : $\Delta(I_3,X)$

I_3={T→ F.} Ensemble vide de transition

- Calcul de la transition dans I_4 : $\Delta(I_4,X)$

I_4={F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)} avec X={E,T,F,(,nb}
$\Delta(I_4, E)$= {F→(E.), E→E.+T}=$\mathbf{I_8}$
$\Delta(I_4, T)$={ E→T., T→T.*F }=$\mathbf{I_2}$
$\Delta(I_4, F)$={ T→F. }=$\mathbf{I_3}$
$\Delta(I_4,nb)$={ F→nb. }=$\mathbf{I_5}$
$\Delta(I_4,()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=$\mathbf{I_4}$

- Calcul de la transition dans I_5 : $\Delta(I_5,X)$

I_5= {F→nb.} Ensemble vide de transition

- Calcul de la transition dans I_6 : $\Delta(I_6,X)$

I_6={E →E+.T, T→.T*F, T→. F, F →.nb, F →.(E) } avec X={T,F,(,nb}
$\Delta(I_6, T)$= {E →E+T., T→T.*F }=$\mathbf{I_9}$
$\Delta(I_6, F)$={ T→F. }=$\mathbf{I_3}$
$\Delta(I_6, nb)$={ F→nb. }=$\mathbf{I_5}$
$\Delta(I_6, ()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=$\mathbf{I_4}$

- Calcul de la transition dans I_7 : $\Delta(I_7,X)$ avec X={F,(,nb}

$\Delta(I_7, F)$={ T→T*F.}=$\mathbf{I_{10}}$
$\Delta(I_7, nb)$={ F→nb. }=$\mathbf{I_5}$
$\Delta(I_7, ()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=$\mathbf{I_4}$

- Calcul de la transition dans I_8 : $\Delta(I_8,X)$ avec X={),+}

$\Delta(I_8,))$= {F→(E).}=$\mathbf{I_{11}}$
$\Delta(I_8, +)$= {F→E+.T, T→.T*F, T→. F, F →.nb, F →.(E) }=$\mathbf{I_6}$

- Calcul de la transition dans I_9 : $\Delta(I_9,X)$:

I_9= {E →E+T., T→T.*F }, avec X={*}
$\Delta(I_9,*)$={ T→T*.F, F →.nb, F →.(E)}=$\mathbf{I_7}$

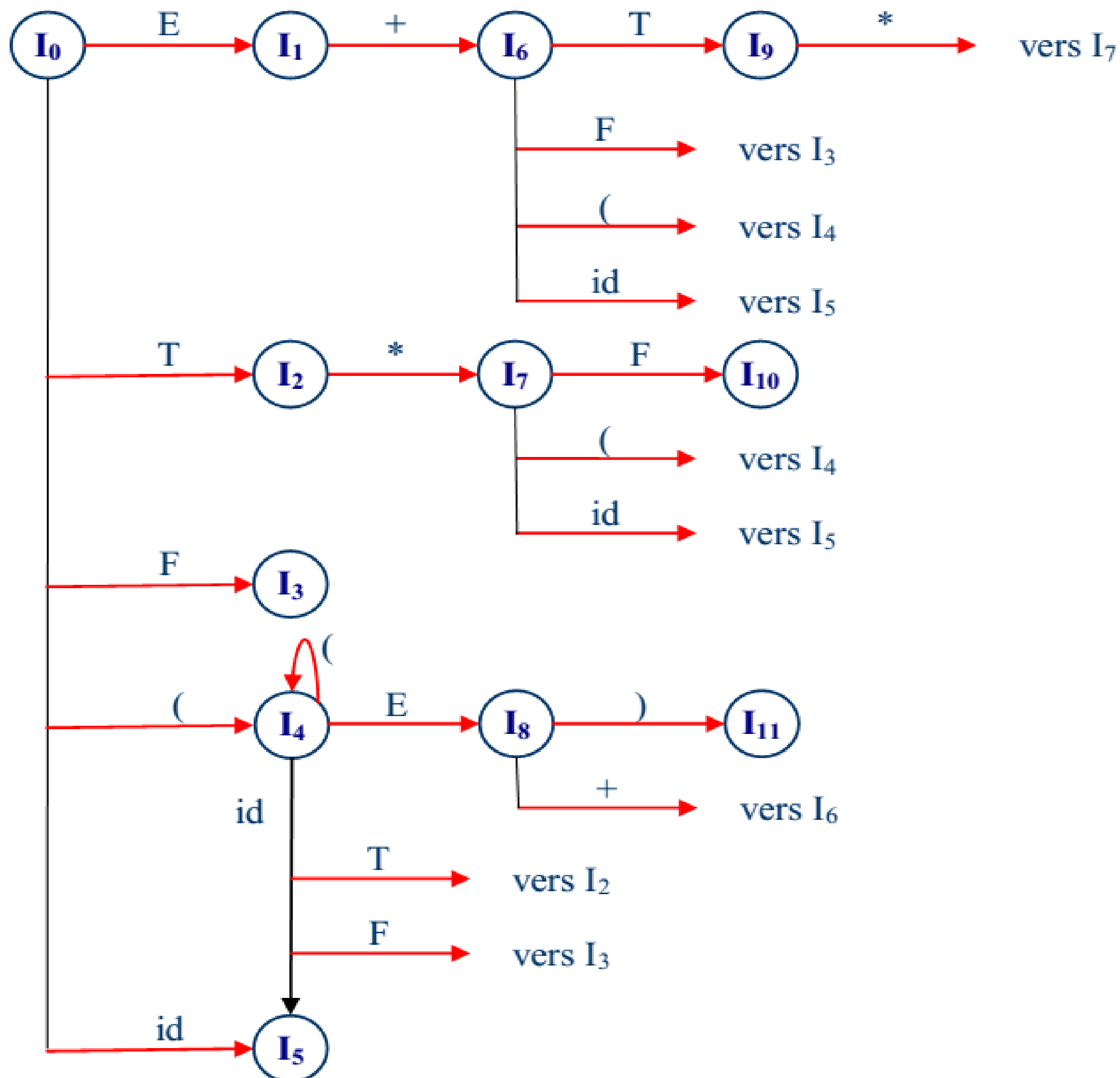
- Calcul de la transition dans I_{10} : $\Delta(I_{10},X)$

I_{10}= { T→T*F.}: Ensemble vide de transition

- Calcul de la transition dans I_{11} : $\Delta(I_{11},X)$

I_{11}= {F→(E).}: Ensemble vide de transition

La fonction de transition pour l'ensembles d' items de la grammaire augmentée considérée dans cet exemple est donnée sous forme d'un AFD représenté comme suit :



➢ **Construction de la table d'analyse SLR :**

Cette table va nous dire ce qu'il faut faire quand on lit une lettre ***a*** et qu'on est dans un état ***i***.

- Soit on **décale**. Dans ce cas, on empile la lettre lue et on va dans un autre état *j*. On note ça **d*j***
- Soit on **réduit** par la règle de production numéro ***p***, c à d qu'on remplace la chaîne en sommet de pile par le non-terminal de la partie gauche de la règle de production, et on va dans l'état *j* qui dépend du non-terminal en question. On note ça **r*p***.
- Soit on **accepte** le mot. Ce qui sera noté ACC.
- Soit c'est une **erreur**. c à d case vide.

L'algorithme pour construire la table d'analyse SLR :

1- Construire la collection d'items $\{I_o, \ldots, I_n\}$
2- l'état i est contruit à partir de I_i :
 a) pour chaque $\Delta(I_i, a) = I_j$: mettre **decaller j** dans la case $M[i, a]$
 b) pour chaque $\Delta(I_i, A) = I_j$: mettre **aller en j** dans la case $M[i, A]$
 c) pour chaque $A \to \alpha.$ (sauf $A = S'$) contenu dans I_i :
  mettre **reduire** $A \to \alpha$ dans **chaque** case $M[i, a]$ où $a \in$SUIVANT(A)
 d) si $S' \to S. \in I_i$: mettre **accepter** dans la case $M[i, \$]$

Toujours le même exemple : l'ensemble des SUIVANT et PREMIER :

	PREMIER	SUIVANT
E	nb (	$ +)
T	nb (	$ + *)
F	nb (	$ + *)

Et donc la table d'analyse SLR de cette grammaire est :

état	nb	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				ACC			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

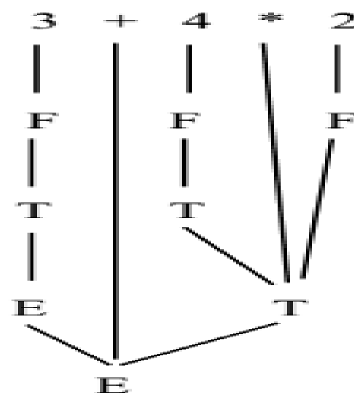
➢ **Analyseur syntaxique SLR** :

On part dans l'état 0, et on empile et dépile non seulement les symboles (comme lors de l'analyseur LL) mais aussi les états successifs.

L'analyse du mot 3+4*2$ est comme suit :

pile	entrée	action
$ 0	3 + 4 * 2$	d5
$ 0 3 5	+4 * 2$	r6 : F → nb
$ 0 F	+4 * 2$	je suis en 0 avec F : je vais en 3
$ 0 F 3	+4 * 2$	r4 : T → F
$ 0 T	+4 * 2$	je suis en 0 avec T : je vais en 2
$ 0 T 2	+4 * 2$	r2 : E → T
$ 0 E	+4 * 2$	je suis en 0 avec E : je vais en 1
$ 0 E 1	+4 * 2$	d6
$ 0 E 1 + 6	4 * 2$	d5
$ 0 E 1 + 6 4 5	*2$	r6 : F → nb
$ 0 E 1 + 6 F	*2$	je suis en 6 avec F : je vais en 3
$ 0 E 1 + 6 F 3	*2$	r4 : T → F
$ 0 E 1 + 6 T	*2$	en 6 avec T : je vais en 9
$ 0 E 1 + 6 T 9	*2$	d7
$ 0 E 1 + 6 T 9 * 7	2$	d5
$ 0 E 1 + 6 T 9 * 7 2 5	$	r6 : F → nb
$ 0 E 1 + 6 T 9 * 7 F	$	en 7 avec F : je vais en 10
$ 0 E 1 + 6 T 9 * 7 F 10	$	r3 : T → T * F
$ 0 E 1 + 6 T	$	en 6 avec T : je vais en 9
$ 0 E 1 + 6 T 9	$	r1 : E → E + T
$ 0 E	$	en 0 avec E : je vais en 1
$ 0 E 1	$	ACCEPTÉ !!!

Donc l'arbre de dérivation du ce mot 3+4*2 est comme suit :



2. Utilisation des priorités et des associativités pour résoudre les actions conflictuelles d'analyse :

Les grammaires ambigües provoquent des conflits dans la table d'analyse:

- Conflit décalage /réduction : on ne peut pas décider à la lecture du terminal 'a' s'il faut réduire une production S→σ ou décaler le terminal
- Conflit réduction /réduction : on ne peut pas décider à la lecture du terminal 'a' s'il faut réduire une production S→σ ou une production T→B.

→On doit alors résoudre les conflits en donnant des **priorités** aux actions (décaler ou réduire) et aux productions.

Par exemple, soit la grammaire : E→E+E|E*E|nb

1) Soit à analyser 3+4+5. Lorsqu'on lit le 2iéme + on a le choix entre :

- ✓ réduire ce qu'on a déjà lu par : E→E+E. Ce qui nous donnera finalement le calcul (3+4) +5
- ✓ décaler ce +, ce qui nous donnera finalement le calcul : 3+ (4+5).

→ + est associatif à gauche, donc on préférera réduire.

2) Soit à analyser 3+4*5. Lorsqu'on lit le * on a encore un choix shift/reduce. Si l'on réduit on calcule (3+4)*5, si on décale on calcule 3+ (4*5) : donc Il faut décaler →car le * est prioritaire que +.

3) Soit à analyser, 3*4+5 : donc il faut réduire→ il faut mettre quelque part dans l'analyseur le fait que * est prioritaire sur +.

5. Erreurs syntaxiques

Beaucoup d'erreurs sont par nature syntaxique, Le gestionnaire d'erreur doit :

- indiquer la présence de l'erreur de façon claire et précise.
- traiter l'erreur rapidement pour continuer l'analyse.
- traiter l'erreur le plus efficacement possible de manière à ne pas en créer de nouvelles.

Heureusement les erreurs communes (confusion entre deux séparateurs par exemple entre, et ; oubli de ;) sont simples et un mécanisme simple de traitement en général.

Il existe plusieurs stratégies de récupération sur erreur : mode panique, au niveau du syntagme, productions d'erreur, correction globale.

- ✓ **Récupération en mode panique :**

C'est la méthode la plus simple à implanter. Quand il découvre une erreur, l'analyseur syntaxique élimine les symboles d'entrée les un après les autres jusqu'à en rencontrer un qui appartienne à un ensemble d'unités lexicales de synchronisation. Bien que cette méthode saute en générale une partie considérable du texte source sans en vérifier la validité, elle a l'avantage de la simplicité et ne peut pas entrer dans une boucle infinie.

- ✓ **Récupération au niveau du syntagme**

Quand une erreur est découverte, l'analyseur syntaxique peut effectuer des corrections locales (Par exemple : remplacer une, par un ; un wihle par un while_ insérer un ; ou une)….Le choix de la modification à faire n'est pas évident du tout du tout en général. En outre, il faut faire attention à ne pas faire de modifications qui entrainerait une boucle infinie.

On implante cette récupération sur erreur en remplissant les cases vides des tables d'analyse par des pointeurs vers des routines d'erreur. Ces routines remplacent, insèrent ou suppriment des symboles d'entrée et émettent les messages appropriées.

- ✓ **Productions d'erreur :**

Si l'on a une idée assez précise des erreurs courantes qui peuvent être rencontrées, il est possible d'augmenter la grammaire du langage avec des productions qui engendrent les constructions erronées.

Par exemple (pour un compilateur C) :

I→if E I (erreur : il manque les parenthèses)

I→ if (E) then I (erreur : il n' y a de then en C).

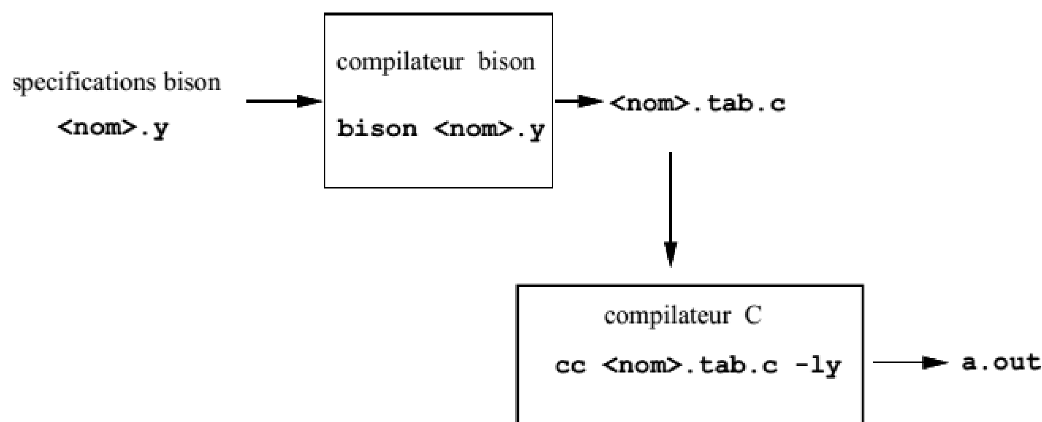
✓ **Correction globale:**

Dans l'idéal, il est souhaitable que le compilateur effectue aussi peu de changements que possible. Il existe des algorithmes qui permettent de choisir une séquence minimale de changements correspondant globalement au cout de correction le plus faible. Malheureusement, ces méthodes sont trop couteuses en temps et en espace pour être implantées en pratique et ont donc uniquement un intérêt théorique. En outre, le programme correct le plus proche n'est pas forcément celui que le programmeur avait en tête.

6. Générateur d'analyseur syntaxique : YACC

De nombreux outils ont été batis pour construire des analyseurs syntaxiques à partir de grammaires. C'est à dire des outils qui construisent automatiquement une table d'analyse à partir d'une grammaire donnée.

Yacc/bison accepte en entrée la description d'un langage sous forme de règles de productions et produit un programme écrit dans un langage de haut niveau (ici le langage C) qui une fois compilé, reconnait des phrases de ce langage (ce programme est donc un analyseur syntaxique).



1) Le format d'une spécification YACC est le suivant :

%{

Déclarations : déclaration(en C) de variables, constantes, inclusions de fichiers

%}

Déclarations des unités lexicales utilisées

Déclarations de priorités et de types

%%

Règles de production et actions sémantiques

%%

Bloc principal

- **Partie Déclaration**

La partie déclarations contient les déclarations "C" (entre %{ et %}) ainsi que la déclaration des noms de tokens :

%token name1 name2

Tout nom non-défini dans la partie déclarative est considéré comme un symbole non terminal. Tout symbole non-terminal doit apparaître dans la partie gauche d'une règle au moins une fois.

Lorsque les tokens correspondent à des opérateurs pour lesquels on souhaite spécifier une propriété d'associativité, on utilise *left* et *right* à la place de token.

Exemple : on écrira :

%left '+' '-' /* addition et soustraction associatives à gauche */

%right '^' /* exponentiation associative à droite */

Les priorités des symboles sont données par l'ordre dans lequel apparait leur déclaration d'associativité, les premiers ayant la plus faible priorité. Lorsque les symboles sont dans la même déclaration d'associativité, ils ont la même priorité.

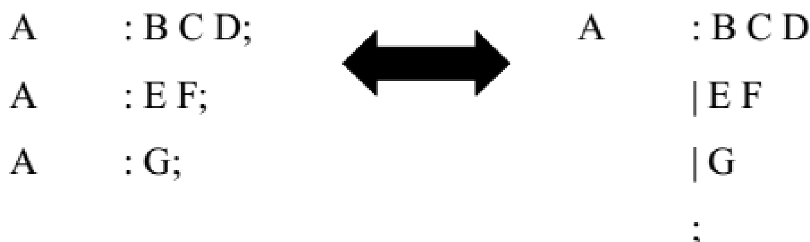
- **Les règles ont le format suivant :**

A : CORPS;

"A" représente un symbole non-terminal, et **"CORPS"** représente une chaîne de noms et lettres. Les symboles **":"** et **";"** sont des délimiteurs. Les noms peuvent être des symboles terminaux ou non-terminaux. YACC nécessite que les noms des symboles terminaux soient déclarés ainsi dans la partie de déclarations.

Si plusieurs règles ont la même partie gauche, on peut utiliser le caractère " | ".

Exemple :



Parmi tous les symboles non-terminaux il ya un seul symbole de début. L'analyseur syntaxique reconnaît ce symbole. Par défaut, le symbole de début (axiome) est considéré comme étant le premier

nom de côté gauche de la première règle. Si on désire, on peut déclarer ce symbole explicite en utilisant le mot clé: %start

Exemple: %start symbole.

- **Actions**

A chaque règle syntaxique on associe des actions. Ces actions peuvent retourner différentes valeurs et peuvent utiliser les valeurs retournées par des autres actions. Une action est un fragment de code en C, placé entre accolades.

G : S B 'X' { printf("mot reconnu");}

- **Le programme en C**

Il doit contenir le programme principal main () qui doit en général faire un appel à la fonction **yyparse()**. Ce fonction est créée par Yacc et doit contenir une fonction yylex effectuant l'analyse lexicale du texte, car l'analyseur syntaxique l'appelle chaque fois qu'il a besoin du terminal suivant.

2) Communication avec l'analyseur lexical : yylval

L'analyseur syntaxique et l'analyseur lexical peuvent communiquer entre eux par l'intermédiaire de la variable ***int yylval***.

Dans une action lexicale (donc dans le fichier lex par exemple), l'instruction *return(ul)* permet de renvoyer à l'analyseur syntaxique l'unité lexicale *ul*. La valeur de cette unité lexicale peut être rangée dans *yylval*.

L'analyseur syntaxique prendra automatiquement le contenu de *yylval* comme valeur de l'attribut associé à cette unité lexicale. La variable yylval est de type YYSTYPE (déclaré dans la bibliothèque yacc/bison) qui est une *int* par défaut. On peut changer ce type par:

```
#define YYSTYPE autre_type_C
```

Ou encore par :

%union {champs d'une union C}

Par exemple

```
%union {
        int entier;
        double reel;
        char * chaine;
      }
```

Permet de stocker dans yylval à la fois des int, des double et des char*.

➔ Pour faciliter la communication entre actions et l'analyseur syntaxique on utilise **le symbole $** de la manière suivante :

- ✓ A chaque symbole (terminal ou non) est associe une valeur (de type entier par défaut). Cette valeur peut être utilisée dans les actions sémantiques.
- ✓ Le symbole **$$** référence la valeur de l'attribut associe au non terminal de la partie gauche.
- ✓ **$$_i$** référence la valeur associée au i-éme symbole (terminal ou non terminal), ou action sémantique de la partie droite.

3) Variables, fonctions et actions prédéfinies

YYPARSE() : appel de l'analyseur syntaxique.

YYACCEPT : instruction permettant de stopper l'analyseur syntaxique.

YYABORT : instruction qui permet également de stopper l'analyseur. yyparse retourne alors 1, ce qui peut être utilisé pour signifier l'echec de l'analyseur.

main() : le main par défaut se contente d'appeler yyparse(). L'utilisateur peut écrire sa propre main dans la partie du bloc principal.

yyerror() fonction définie dans la bibliothèque de YACC, qui indique qu'une erreur de syntaxe a été rencontrée.

Exemple :

La source Lex du mini-interprète d'expressions :

```
%{
#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}
chiffre   [0-9]
entier    {chiffre}+
exposant  [eE][+-]?{entier}
reel    {entier}("."{entier})?{exposant}?
%%
{reel}    {
      yylval=atof(yytext);
      return(NOMBRE);
    }
"+"   return(PLUS);
```

```
"-"   return(MOINS);
"*"   return(FOIS);
"/"   return(DIVISE);
"^"   return(PUISSANCE);
"("   return(PARENTHESE_GAUCHE);
")"   return(PARENTHESE_DROITE);
"\n"  return(FIN);
```

Le source Yacc du mini-interprète d'expressions :

```
%{
#include "global.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}
%token  NOMBRE
%token  PLUS  MOINS FOIS  DIVISE  PUISSANCE
%token  PARENTHESE_GAUCHE PARENTHESE_DROITE
%token  FIN
%left PLUS MOINS
%left FOIS DIVISE
%left NEG
%right PUISSANCE
%start Input
%%
Input:
   /* Vide */
  | Input Ligne
  ;
```

```
Ligne:
   FIN
  | Expression FIN    { printf("Resultat : %f\n",$1); }
  ;
Expression: NOMBRE     { $$=$1; }
| Expression PLUS Expression {$$=$1+$3; }
| Expression MOINS Expression {$$=$1-$3; }
 | Expression FOIS Expression {$$=$1*$3; }
  | Expression DIVISE Expression {$$=$1/$3; }
  | MOINS Expression %prec NEG {$$=-$2; }
  | Expression PUISSANCE Expression {$$=pow($1,$3); }
  | PARENTHESE_GAUCHE Expression PARENTHESE_DROITE {$$=$2;}
  ;
%%
int yyerror(char *s) {
printf("%s\n",s);
}
int main(void) {
yyparse();
}
```

Bibliographie

Mohamed BOUHDADI, « Compilation théorie des langages », Support du cours, Université Mohammed V - Agdal,Faculté des Sciences, Rabat.

SAOUDI Lalia, (2008), « Analyse syntaxique», Support du cours.

Souici-Meslati L, (2013), « Compilation », Support de cours, Université d'Annaba.

Aho A., Sethi R, (2000), « Compilateurs : principes, techniques et outils », Inter-éditions,