Analyse syntaxique

Grammaires

Langage et syntaxe

- Les phrases d'un langage ont une structure déterminée par une grammaire
- **Exemple :** une phrase correcte est constituée par un sujet suivi d'un verbe

```
if (expression) instruction else instruction
```

alternative

- Ceci peut être exprimé par une grammaire

  phrase → sujet verbe

  instr → if (expr) instr else instr

- Si on complète avec deux nouvelle règles

  sujet → pierre | claire
  verbe → court | marche

- 4 phrases possibles :
  pierre court | pierre marche | claire court | claire marche

Exemple 1

Nombre -> Chiffre | Chiffre Nombre

Chiffre -> 0|1|2|3|4|5|6|7|8|9

génère

0

17

547

7350

etc…

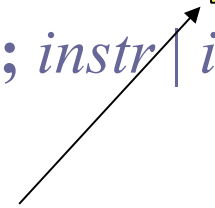
Exemple 2 bloc de code en Java

En Pascal, est une suite d'instructions séparées par des points-virgules entre une { et }

bloc -> { *liste_opt_instr* }

liste_opt_instr -> *liste_instr* | ε

liste_instr -> *liste_instr* ; *instr* | *instr*



liste des symboles vides

Grammaire non contextuelle

Une grammaire non contextuelle comprend quatre composants $G=(\Sigma, V, P, S)$:

- un ensemble d'unités lexicales appelées symboles terminaux Σ (lexèmes ou tokens) ;
- un ensemble de non-terminaux V (disjoint de Σ) ;
- un ensemble de règles de production P où chaque règle est constituée d'un non-terminal, appelé *partie gauche*, d'une flèche appelée *partie droite* de la forme
  $\alpha \rightarrow \beta$ avec $\alpha \in V$, et $\beta \in (\Sigma \cup V)^*$
- La désignation d'un des non-terminaux en tant que *symbole de départ S*.

Grammaire non contextuelle (suite)

- les terminaux sont les symboles de base à partir desquels les chaînes sont formées (⇔unités lexicales)
- les non-terminaux sont les variables syntaxiques qui dénotent un ensemble de chaînes
- l'axiome est un non-terminal particulier. L'ensemble des chaînes que l'on peut obtenir à partir de l'axiome est le langage défini par la grammaire

Grammaire non contextuelle (suite)

Une grammaire définit un langage formé par l'ensemble des séquences finies de symboles terminaux qui peuvent être dérivées du symbole initial par des applications successives de règles

Convention :

- les chiffres, les signes ou les chaînes en **gras** sont des terminaux
- un nom en *italique* désigne un non-terminal

Dérivations

Pour montrer qu'un mot *w* est dans le langage L, il faut exhiber une suite finie d'applications de règles de production partant de S et aboutissant à *w*, appelée dérivation

On peut représenter une dérivation par

$$S = w_0 \xrightarrow{P_1} w_1 \xrightarrow{P_2} \ldots \xrightarrow{P_n} w_n = w$$

Cela peut également se représenter sous la forme d'un arbre que l'on nomme (*arbre de dérivation*)

Exemple de grammaire

phrase → *GN GV*

GN → *Article Nom* | *Article Nom Subord*

Subord → *Pron_relatif GV*

GV → *Verbe GN* | *Verbe GN_prep*

GN_prep → *Prep GN*

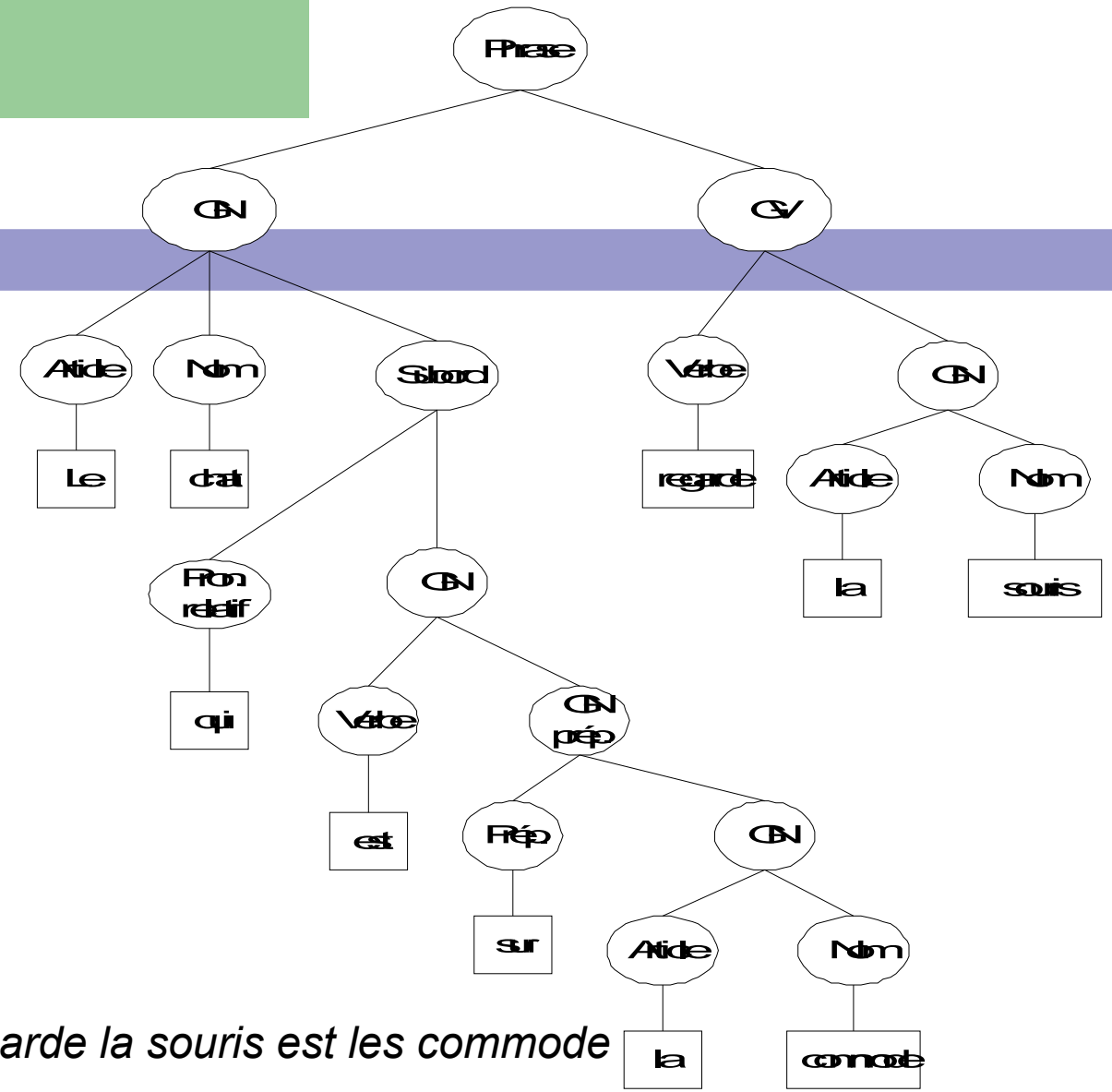
Article → **le** | **la** | **les**

Nom → **chat** | **commode** | **souris**

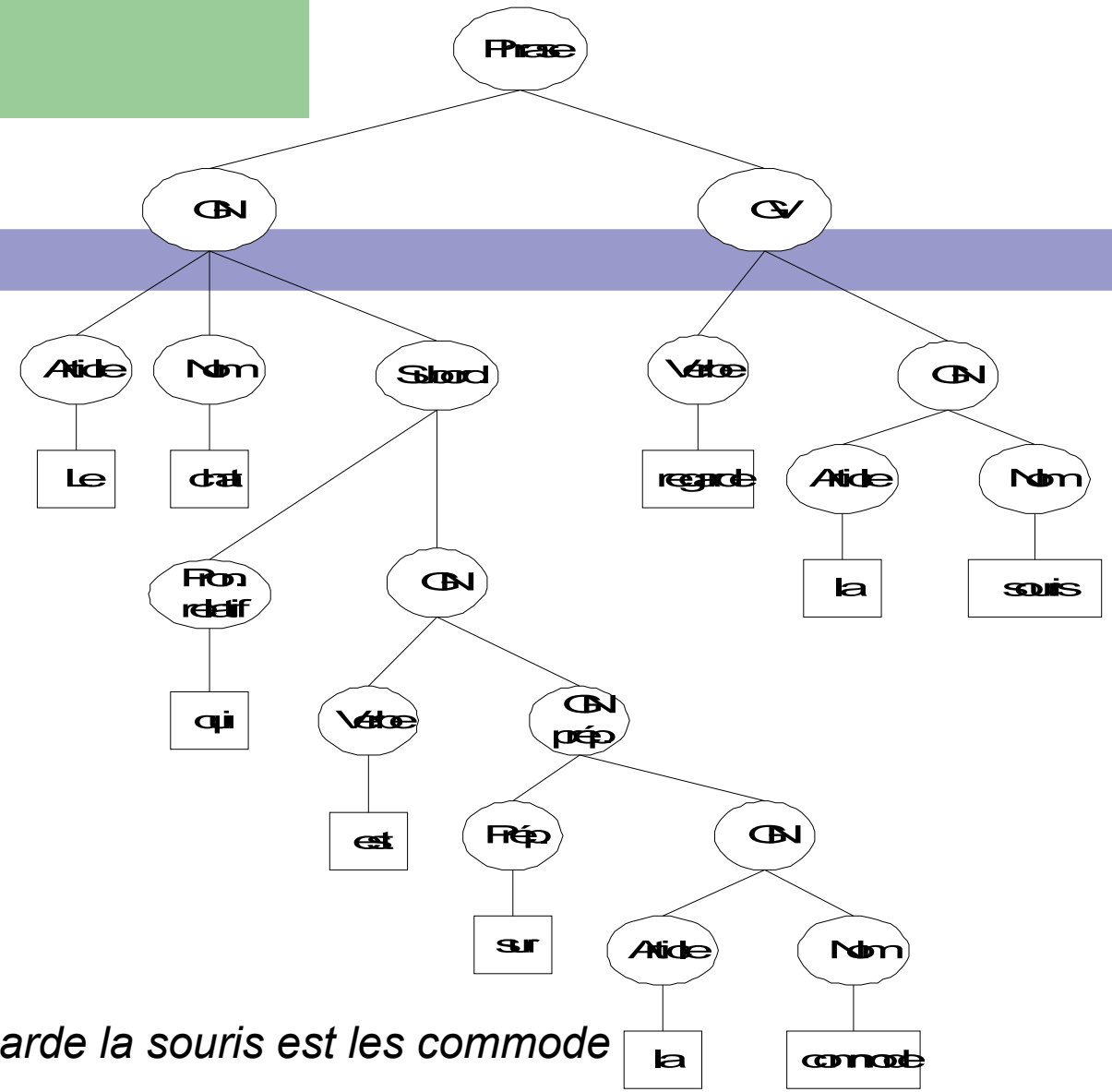
Verbe → **est** | **regarde**

Prep → **sur** | **sous**

Pron_relatif → **qui** | **que** | **dont** | **ou**



le chat que regarde la souris est les commode



le chat que regarde la souris est les commode

Langage engendré

Le langage engendré par une grammaire G est l'ensemble de terminaux que l'on peut dériver à partir de l'axiome

$$\forall w \in \Sigma^* \; w \in L(G) \Leftrightarrow S \overset{+}{\rightarrow} w$$

Dérivation à gauche – à droite

- Une dérivation est dite à gauche si le non terminal le plus à gauche est remplacé

$$\Rightarrow_g$$

- Une dérivation est dite à droite si le non-terminal le plus à droite est remplacé

$$\Rightarrow_d$$

Exemple

G :

E →E + E / E * E / (E) / - E / Id

-(Id + Id) appartient L(G)

E →-E → -(E) → -(E + E)

1er cas :

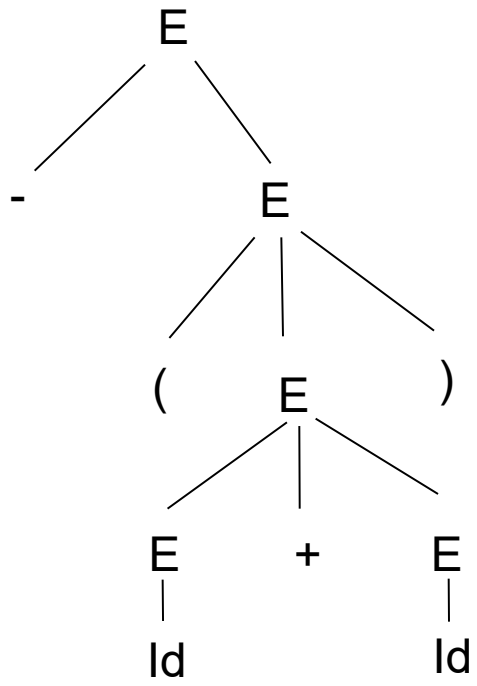
→ - (Id + E) → -(Id + Id)

2ème cas :

→ -(E + Id) → -(Id + Id)

Arbre de dérivation

Un arbre de dérivation est une représentation graphique d'une dérivation.



A chaque arbre d'analyse, est associé une unique dérivation gauche (ou droite)

Les deux suites différentes de dérivations donnent le même arbre de dérivation.

Ambiguïté

Une grammaire est ambiguë si elle produit plus d'un arbre d'analyse (ou d'une dérivation à gauche) pour une phrase donnée.

Remarque : la grammaire précédente **est** ambiguë

Exemple

Soit la grammaire G
E →E + E
E →E * E
E → (E)
E → -E
E →**id**

Il existe deux arbres d'analyse syntaxique pour id+id*id
E →E + E
→**id** + E
→**id** + E * E
→**id** + **id** * E
→**id** + **id** * **id**

ou
E →E * E
→E + E * E
→**id** + E * E
→**id** + **id** * E
→**id** + **id** * **id**

Exemples : expressions arithmétiques

Les terminaux (lexèmes) sont **NUM**, **ID**, + et -

S → *E*

E → **NUM**

E → **ID**

E → *E* **+** *E*

E → *E* **–** *E*

Pour reconnaître l'expression 1 – 1 + x, plusieurs dérivations sont possibles (on parle d'ambiguïté)

Dérivations possibles

$$S \rightarrow E \rightarrow \begin{cases} E \rightarrow \begin{cases} E \rightarrow \text{NUM} \\ - \\ E \rightarrow \text{NUM} \end{cases} \\ + \\ E \rightarrow \text{ID} \end{cases} \qquad S \rightarrow E \rightarrow \begin{cases} E \rightarrow \text{NUM} \\ - \\ E \rightarrow \begin{cases} E \rightarrow \text{NUM} \\ + \\ E \rightarrow \text{ID} \end{cases} \end{cases}$$

Exercice

Trouvez une grammaire G non ambiguë engendrant le même langage que G (* plus prioritaire que +)

E →E + T | T

T →T * **id** | **id**

On obtient une grammaire non ambiguë mais moins lisible

Processus d'analyse

Le but est reconstruire l'arbre syntaxique d'une phrase à partir d'une grammaire non ambiguë. Il existe deux processus :

- l'analyse descendante. Cette analyse correspond à un parcours descendant gauche (on lit de gauche à droite). Il s'agit de deviner à chaque étape de deviner la règle qui sert à engendrer le mot qu'on lit.
- l'analyse ascendante. Dans cette analyse, on essaye d'appliquer le membre droit d'une règle et à le remplacer par le non-terminal gauche correspondant. C'est l'opération inverse de la dérivation.

L'analyse descendante

- Tente de construire un arbre syntaxique pour une chaîne d'entrée de la racine vers les feuilles
  ou
- Tente de déterminer une dérivation gauche associée à une chaîne d'entrée

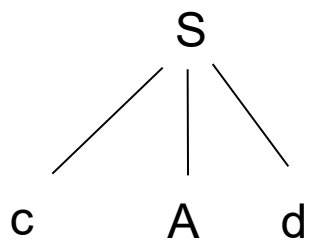
Exemple

$$\begin{cases} S \rightarrow cAd \\ A \rightarrow ab \mid a \\ w = cad \end{cases}$$

c a d

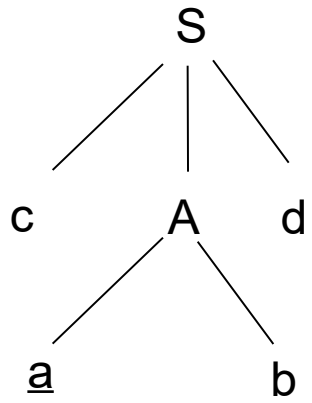
↑ ↑

mémorisé



$S \Rightarrow c\,\underline{A}\,d$

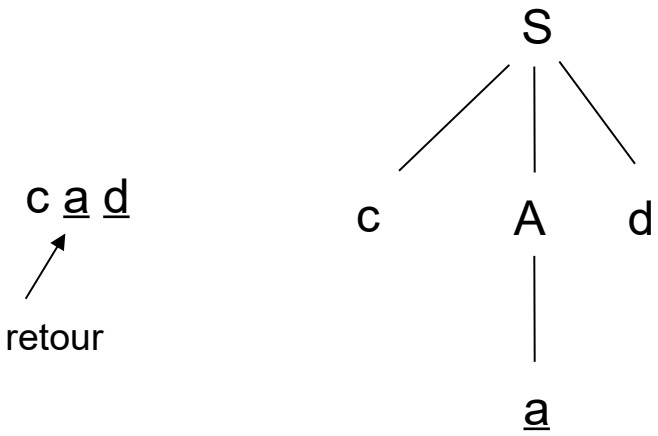
c a d



$\Rightarrow c\,a\,b\,d$

échec
retour au choix précédent

Exemple (suite)



$$S \Rightarrow c\,A\,d \Rightarrow c\,a\,d$$

Dans cet exemple, nous traitons une forme générale d'analyse descendante où des retours arrières sont possibles avec passages répétés sur le texte source. Dans la pratique, le rebroussement est rarement nécessaire, on peut adapter les grammaires pour effectuer une analyse descendante sans rebroussement

Remarque

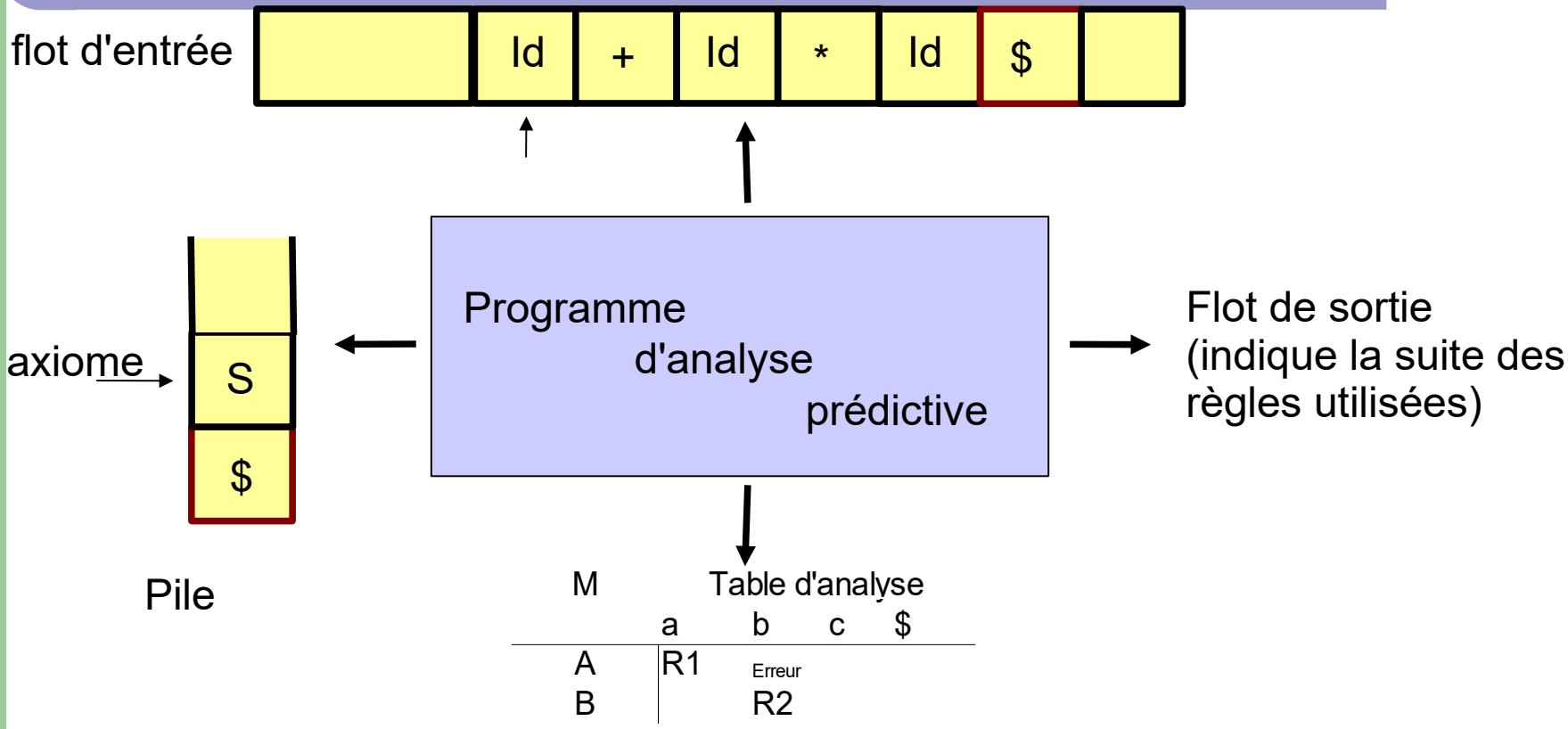
$S \rightarrow aSb|aSc|d$ avec le mot w=aaaaaaadbbcbbbc

- Pour savoir quelle règle utiliser, il faut cette fois-ci connaître aussi la dernière lettre du mot.
- Conclusion : ce qui serait pratique, ça serait d'avoir une table qui nous dit : quand je lis tel caractère et que j'en suis à dériver tel symbole non-terminal, alors j'applique telle règle et je ne me pose pas de questions. Ça existe, et ça s'appelle **une table d'analyse**.

Analyse descendante

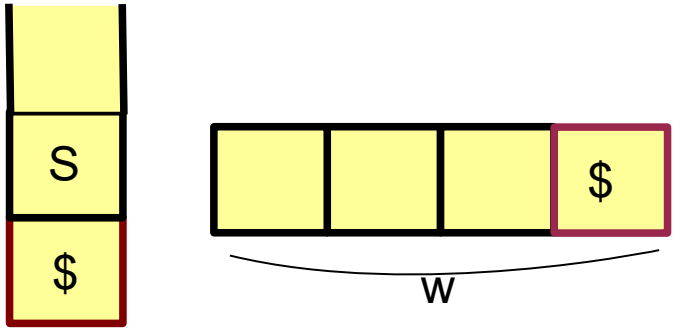
- Analyse prédictive

Implémentation à l'aide d'une pile explicite



Algorithme

au départ :

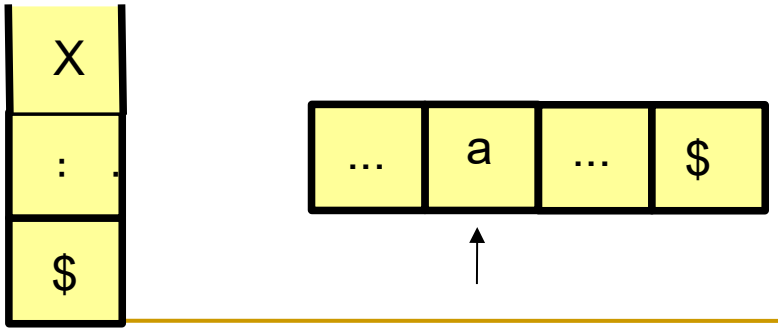



```
si X=a alors Dépiler(X);
   Avancer;
si X est un non terminal
   si M[X,a] contient une
      production X→UVW
      Remplacer X pour WVU
   si M[X,a] vide
      Routine d'erreur
si X=a=$ alors succès
```

terminal ou non-terminal

au fond

en cours :



Exemple

Soit la grammaire suivante :

E → TE' *[r1]*

E' → +TE' *[r2]* | ε *[r3]*

T → FT' *[r4]*

T' → *FT' *[r5]* | ε *[r6]*

F → (E) *[r7]* | nb *[r8]*

Table d'analyse

	nb	+	*	(	)	$
E	*r1*			*r1*		
E'		*r2*			*r3*	*r3*
T	*r4*			*r4*		
T'		*r6*	*r5*		*r6*	*r6*
F	*r8*			*r7*		

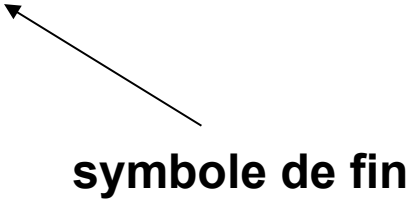
E	2+3*4$	r1
TE'	2+3*4$	r4
FT'E'	2+3*4$	r8
T'E'	2+3*4$	r6
E'	2+3*4$	r2
TE'	2+3*4$	r4
FT'E'	2+3*4$	r8
T'E'	2+3*4$	r5
FT'E'	2+3*4$	r8
T'E'	2+3*4$	r6
E'	2+3*4$	r3
0	succès	

Fonctionnement de la table

Elle permet d'expliciter les dérivations à effectuer pour accepter une phrase d'une grammaire

Exemple :

dérivation de *w* = *2 + 3 * 4* $



Résumé

$$E \Rightarrow \underline{T}\, E' \Rightarrow \underline{F}\, T'\, E' \Rightarrow id\, T'\, E' \Rightarrow id\, E' \Rightarrow$$
$$\Rightarrow id + TE' \Rightarrow id + F\, T'\, E' \Rightarrow id + id\, T'\, E' \Rightarrow id + id * F\, T'\, E'$$
$$\Rightarrow id + id * id\, T'\, E' \Rightarrow id + id * id\, E' \Rightarrow id + id * id$$

Un peu de théorie ☹

les terminaux avant α

grosso modo

définition : pour $\alpha \in (\Sigma \cup V)^*$

$$first(\alpha) = \left\{ x \in \Sigma \middle| \alpha \xrightarrow{*} x\beta \right\} \cup \{ \varepsilon \text{ si } \alpha \text{ est une chaîne nullable} \}$$

ensemble des terminaux qui commencent les chaînes qui se dérivent de α

les terminaux après A

On définit pour $A \in V$:

$$follow(A) = \left\{ x \in \Sigma \middle| S \xrightarrow{*} \alpha A \gamma \text{ avec } x \in first(\gamma) \right\}$$

ensemble des terminaux qui peuvent se trouver immédiatement à droite de A après dérivation. Si A est le symbole le plus à droite, alors $\$ \in \sigma\upsilon\iota\omega\alpha\nu\tau(A)$

Définition de premier

- si X est un terminal premier(X)={X}
- Si $X \rightarrow \varepsilon$ $\varepsilon \in$ premier(X)
- si $X \rightarrow Y_1 Y_2 \ldots Y_k$ premier(Y_1)$\subset$ premier(X),

$si \quad Y_1 \overset{*}{\Rightarrow} \epsilon, \quad first(Y_2) \subset first(x)$

$si \quad Y_2 \overset{*}{\Rightarrow} \epsilon \ldots$

- Pour une chaîne $X_1 X_2 \ldots X_n$ premier($X_1 X_2 \ldots X_n$)=premier(X_1)

si $\varepsilon \in$ premier(X_1) + premier(X_2) si $\varepsilon \in \ldots$

Application

premier(E)={ (, nb }

premier(E')={ +, ε }

premier(T)={ (, nb }

premier(T')={ *, ε }

premier(F)={ (, nb }

Calcul de suivant

1. s'il y a une production A → αBβ, le contenu de premier(*β*), excepté ε, est ajouté à suivant(*B*)
2. mettre $ dans suivant(*S*), où *S* est l'axiome et $ le marqueur de fin de texte
3. Pour calculer suivant(*A*) pour tous les non-terminaux A, appliquer les règles suivantes jusqu'à ce qu'aucun terminal ne puisse être ajouté
4. s'il existe une production A → αB ou A → αBβ tq premier(*β*) contient ε (β→*ε), les éléments de suivant(*A*) sont ajoutés à suivant(*B*)

Définition de suivant

$$\text{Pour l'axiome } S \qquad \$ \in follow(S)$$

$$\text{si } A \to \alpha B \beta \qquad first(\beta) - \{\epsilon\} \subset follow(B)$$

$$\left.\begin{array}{l} \text{si } A \to \alpha B \\ \text{ou } A \to \alpha B \beta \text{ avec } \beta \overset{*}{\Rightarrow} \epsilon \end{array}\right\} follow(A) \subset follow(B)$$

Exemple

E → TE' *[r1]*
E' → +TE' *[r2]* | ε *[r3]*
T → FT' *[r4]*
T' → *FT' *[r5]* | ε *[r6]*
F → (E) *[r7]* | nb *[r8]*

premier(E)=premier(T)=premier(F)={(,nb}
premier(E')={+,ε}
premier(T')={*,ε}
suivant(E)={$,)}
suivant(T)={+,$,)}
suivant(F)={*,+,$,)}
suivant(E')={$,)}
suivant(T')={+,$,)}

règles αBβ
r1 B=T β=E' rajout de +
r2 *idem*
r4 B=F β=T' rajout de *
r5 α=* B=F β=T' *idem*
r7 α=(B=E β=) rajout de)

règles αB ou αBβ β nullable
r1 α=T B=E' rajout de suivant*(E)*
r1 B=T β=E' rajout de suivant*(E)*
r2 α=+ B=T β=E' rajout de suivant*(E')*
r4 α=F β=T' rajout de suivant*(T)*
r4 B=F β=T' rajout de suivant*(T)*
r5 α=*F B=T'
r5 α=* B=F β=T' rajout de suivant*(T')*

Exemple

S → iEtSS' *(r1)* | a *(r2)*

S' → eS *(r3)* | ε *(r4)*

E → b *(r5)*

premier(S)={i,a}

premier(S')={e,ε}

premier(E)={b}

suivant(S)={$,e}

suivant(E)={t}

suivant(S')={$,e}

règles αBβ

r1 B=E, β=tSS' rajout de t

r1 B=S, β=S' rajout de premier*(S')*

règles αB ou αBβ avec β→*ε

r1 α=iEt B=S β=S'

rajout de suivant*(S)* à *suivant(S)*

r1 α=iEtS B=S'

rajout de suivant*(S)* à suivant*(S')*

r3 α=e B=S

rajout de suivant*(S')*

Construction de la table d'analyse

1. Pour chaque règle r_i $\alpha \rightarrow \beta$:
2. Pour chaque $a \in premier(\beta)$, mettre r_i dans T[α,a]
3. Si en plus β est nullable ($\varepsilon \in premier(\beta)$), pour chaque $a \in suivant(\alpha)$ ($a \in \Sigma U \$$), mettre r_i dans T[α,a]

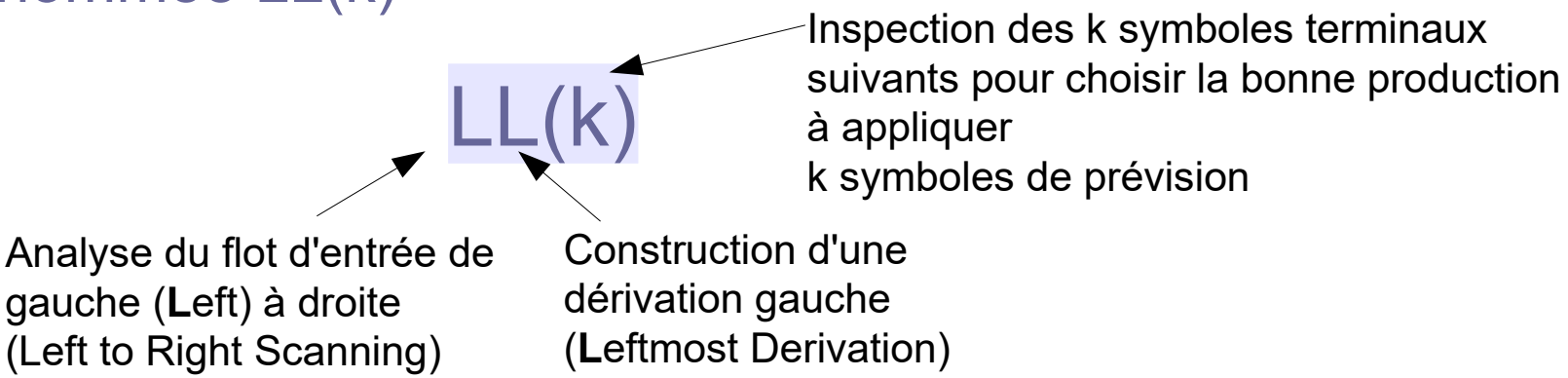
Remarque : nous avons un analyseur prédictif si chaque case la table contient *au plus* une règle de production (pas de conflit)

	a	b	e	i	t	$
S	r2			r1		
S'			r3 r4			r4
E		r5				

ambiguïté de la grammaire

Les grammaires LL(k)

- Il existe une classe de grammaires adaptée à l'analyse descendante déterministe (≈ sans rebroussement) nommée LL(k)



En pratique, si k>1, la réalisation de l'analyseur correspondant est difficile et peu performante. Si k=0, la grammaire est trop restrictive.
Si k=1, notre analyseur est prédictif
LL(1)≈ aucune case de la table d'analyse n'a plusieurs alternatives

Grammaires prédictives LL(1)

Une grammaire G est LL(1) si et seulement si :

$$Pour \quad A \rightarrow \alpha | \beta$$

- il n'existe pas de terminal a tel que :
  $a \in premier(\alpha)$ et $a \in premier(\beta)$
- α et β ne peuvent pas se dériver tous les deux en ε
- Si $\beta \overset{*}{\Rightarrow} \epsilon$, α ne se dérive pas en une chaîne commençant par un terminal de suivant(A)

On appelle grammaire LL(1) une grammaire pour laquelle la table d'analyse décrite précédemment n'a aucune case définie de façon multiple.

Remarque

- Une grammaire **ambiguë** ou **récursive à gauche** ou **non factorisée à gauche** n'est pas LL(1)

Transformation de grammaires

- Deux grammaires sont équivalentes si elles engendrent le même langages
- Selon la méthode d'analyse (ascendante, descendante...) la grammaire doit vérifier certaines contraintes et par conséquent subir éventuellement quelques transformations

Suppression des ambiguïtés

Soit la grammaire suivante :

instr → **si** *expr* **alors** *instr*

| **si** *expr* **alors** *instr* **sinon** *instr*

| *autre*

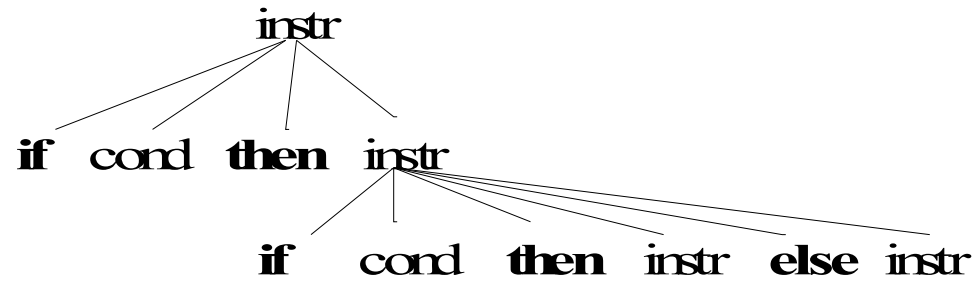
Cette grammaire est ambiguë car la chaîne :

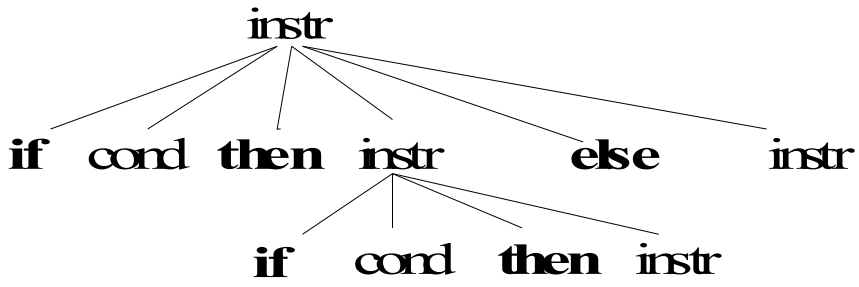
si E1 alors si E2 alors S1 sinon S2

peut être analysée de deux façons différentes

Exemple

phrase : if cond1 then if E2 then I1 else I2





Désambiguïser une grammaire

Il est parfois possible d'enlever les ambiguïtés d'une grammaire. Dans le cas précédent, il suffit d'ajouter des règles de réécriture permettant d'appliquer le principe : "chaque sinon doit être associé au si le plus près".

instruction → *instr_close* | *instr_non_close*

instr_close → **si** *expr* **alors** *instr_close* **sinon** *instr_close* | *autre*

instr_non_close → **si** *expr* **alors** *instr* |
si *expr* **alors** *instr_close* **sinon** *instr_non_close*

Exemple en Java

```
if (i>0)
  if (i>5)
    System.out.print
      ln("erreur")
else
  System.out.println
  ("négatif")
```

```
if (i>0)
{
  if (i>5)
    System.out.print
      ln("erreur");
    }
else
System.out.println("
  négatif")
```

Récursivité à gauche

Une grammaire est récursive à gauche si elle contient un non terminal A tel qu'il existe une dérivation .

$$A \overset{+}{\Rightarrow} A\,\alpha$$

Les méthodes d'analyse descendante ne peuvent pas fonctionner avec des grammaires récursives à gauche

$$A \Rightarrow A\,\alpha \Rightarrow A\,\alpha\,\alpha \Rightarrow A\,\alpha\,\alpha\,\alpha \Rightarrow \ldots$$

Cas d'une récursivité à gauche immédiate

$$A \rightarrow A\,\alpha \mid \beta \longrightarrow \begin{array}{l} A \rightarrow \beta\, A' \\ A' \rightarrow \alpha\, A' \mid \epsilon \end{array}$$

La récursivité à gauche est transformée en récursivité à droite

Ex 1 : $A \rightarrow Aa \mid b$

$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow \ldots \Rightarrow b \ldots aaa$

$$\begin{cases} A \rightarrow bA' \\ A' \rightarrow aA' \mid \epsilon \end{cases}$$

$A \Rightarrow bA' \Rightarrow baA' \Rightarrow baaA' \Rightarrow \ldots \Rightarrow b \ldots aa$

Récursivité à gauche

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

remplacé par :

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon$$

Cas général

$$A \overset{+}{\Rightarrow} A\,\alpha$$

1. Classer les non-terminaux A_1, A_2, ..., A_n
2. Pour i de 1 à n
   - Pour j de 1 à i - 1

     $Remplacer\ A_i \rightarrow A_j \gamma \quad par\ A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \ldots | \delta_k \gamma$

     $où\ A_j \rightarrow \delta_1 | \delta_2 | \ldots \delta_k$

   - Éliminer les récursivités gauches immédiates des A_i

Exemple

$$S \rightarrow Aa|b$$
$$A \rightarrow Ac|Sd|\epsilon$$

- S est récursif à gauche S ⇨Aa ⇨Sda
- A est immédiatement récursif à gauche

1) classer A_1=S, A_2=A

2) pour A_1=S

· vide

· S n'est pas immédiatement récursif

pour A_2=A

· remplacer A_1=S dans A_2=A

$$A \rightarrow Ac \qquad inchangé$$
$$A \rightarrow Aad|bd$$

$$A \rightarrow \epsilon \qquad inchangé$$

Éliminer la récursivité gauche immédiate

$$A \rightarrow bdA'|A'$$
$$A' \rightarrow cA'|adA'|\epsilon$$

On obtient G'

$$S \rightarrow Aa|b$$
$$A \rightarrow bdA'|A'$$
$$A' \rightarrow cA'|adA'|\epsilon$$

Ex 2 : expressions arithmétiques

$$\begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T * Id \mid Id \end{cases}$$

$$\begin{cases} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow Id\,T' \\ T' \rightarrow *Id\,T' \mid \epsilon \end{cases}$$

Récursivité à gauche

Il faut enlever les ambiguïtés

$E \rightarrow E + T \quad | \; T$ boucle infinie

récursivité à gauche

$A \rightarrow A\alpha \mid \beta\bar{x}\bar{x}\mathrm{A} = E, \; \alpha = + \; T$

à remplacer par

$A \rightarrow \beta R$ $\qquad$ $A \rightarrow \mathrm{T}\, R$

$R \rightarrow \alpha R \mid \varepsilon\bar{x}\bar{x}R \rightarrow + \; T \mid \varepsilon$

Factorisation à gauche

C'est une transformation utile pour avoir une grammaire permettant l'analyse prédictive. L'idée est que pour développer un non-terminal *A*, quand il n'est pas évident de choisir l'alternative à utiliser, on ré-écrit les règles de productions

Exemple :

instr → **si** *expr* **alors** *instr* **sinon** *instr*

| **si** *expr* **alors** *instr*

quelle alternative choisir après lecture du si ?

Exemple

instr → **si** *expr* **alors** *instr-suite*

instr-suite → **sinon** *instr* | ε

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 \longrightarrow \begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 | \beta_2 \end{array}$$

Remarque : la factorisation ne supprime pas l'ambiguïté

Factorisation à gauche (suite)

Si les règles de production sont de la forme :

$$A \rightarrow \alpha\beta_1 \left| \alpha\beta_2 \right| \ldots \left| \alpha\beta_n \right| \gamma$$

on transforme en

$$A \rightarrow \alpha A' \left| \gamma \right.$$

$$A' \rightarrow \beta_1 \left| \beta_2 \right| \ldots \left| \beta_n \right.$$

Problèmes rencontrés

- On enlève les ambiguïtés
- On élimine les récursivités à gauche
- On factorise à gauche

On espère obtenir une grammaire LL(1)

Note : les grammaires ambiguës ou récursives à gauche ne sont pas LL(1)

Si la grammaire est ambiguë, on peut choisir arbitrairement la règle à utiliser en cas de conflit mais on affecte le langage

Implémentation à l'aide de procédures récursives

On considère que chaque symbole non-terminal du langage représente une procédure, et on implémente directement cette procédure.

La règle appelée correspond à l'élément présent dans l'ensemble premier

Grammaire LL(1)

$T \rightarrow R \mid aTc$

$R \rightarrow \varepsilon \mid bR$

premier(T) = {a,b,ε} premier(R) = {ε,b}

suivant(T) = {$,c} suivant(R)= {$,c}

	a	b	c	$
T	r2	r1	r1	r1
R		r4	r3	r3

Analyse de aabbbcc$

Exemple

```
fonction parseR()
si next = 'b' alors
   consomme('b') ; parseR()
sinon si next = 'c' ou next = '$' alors
   (* ne rien faire *)
sinon ErreurSyntaxique()

fonction parseT()
si next = 'a' alors
   consomme('a') ; parseT(); consomme('c')
sinon si next = 'b' ou next = 'c' ou next = '$' alors
   parseR()
sinon ErreurSyntaxique()
```

Conclusions

En transformant la grammaire de façon à ce qu'elle soit :

- non ambiguë. Il n'y a pas de méthodes. Une grammaire ambiguë est une grammaire qui a été mal conçue
- factorisée à gauche
- sans récursivité à gauche

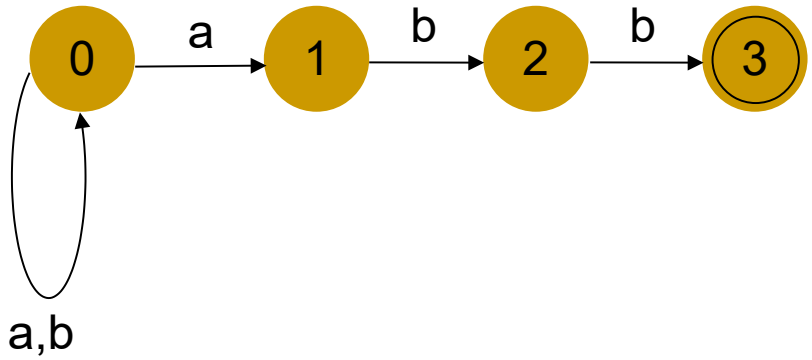
Il ne reste plus qu'a espérer que ça soit LL(1). Sinon, il faut concevoir une autre méthode pour l'analyse syntaxique

On parle d'analyseur prédictif.

$LR \subset LNC$

A partir d'un AFN, on peut facilement définir une grammaire non contextuelle engendrant le même langage

Ex : L=(a|b)*abb



$$A_0 \rightarrow aA_0 | bA_0 | aA_1$$
$$A_1 \rightarrow bA_2$$
$$A_2 \rightarrow bA_3$$
$$A_3 \rightarrow \epsilon$$

•Note :
• LR = Langages réguliers
• LNC = Langages non contextuels
• LC = Langages contextuels

LNC $\not\subset$ LR ?

Un AFND ne sait pas compter

$$L = \{ a^n b^n \mid n \geqslant 1 \quad \}$$
$$G \mid\mid A \rightarrow aAb \mid ab$$

Impossible de construire un AFND qui « compte » le nombre de a pour avoir autant de b après.

LC ?

Certains langages ne peuvent être engendrés par des grammaires non contextuelles

$$L_1 = \{ wcw \mid w \in (a|b)^* \}$$

Ce type de langage pourrait contrôler que les identificateurs sont déclarés avant d'être utilisés
Cette phase est laissée à l'analyse sémantique

$$L_2 = \{ a^n b^m c^n d^m \mid n \geqslant 1,\ m \geqslant 1 \}$$

Ce type de langage pourrait vérifier que le nombre de paramètres formels correspond au nombre de paramètres effectifs (pour deux procédures)

Remarque : des langages très proches de L_1 et L_2 peuvent être engendrés par une grammaire non contextuelle

LC ? (suite)

$$L'_1 = \{ wow^R | w \in (a|b)^* \}$$
$$A \rightarrow aAa | bAb | c$$

$$L'_2 = \{ a^n b^m c^m d^n | n \geqslant 1 \ \ m \geqslant 1 \quad \}$$
$$\begin{cases} A \rightarrow aAd | aA'd \\ A' \rightarrow bA'c | bc \end{cases}$$

Analyse ascendante

- Analyse par décalage réduction

Principes

- Tente de construire un arbre d'analyse par une chaîne d'entrée des feuilles (bas) vers la racine (haut)
- Tente de déterminer une dérivation droite en « sens inverse »

Il s'agit de « réduire » une chaîne w vers l'axiome de la grammaire. On regarde si dans la chaîne à analyser, il existe une sous-chaîne correspondant à la partie droite d'une production et on remplace par le non-terminal de la partie gauche de cette production

Remarques

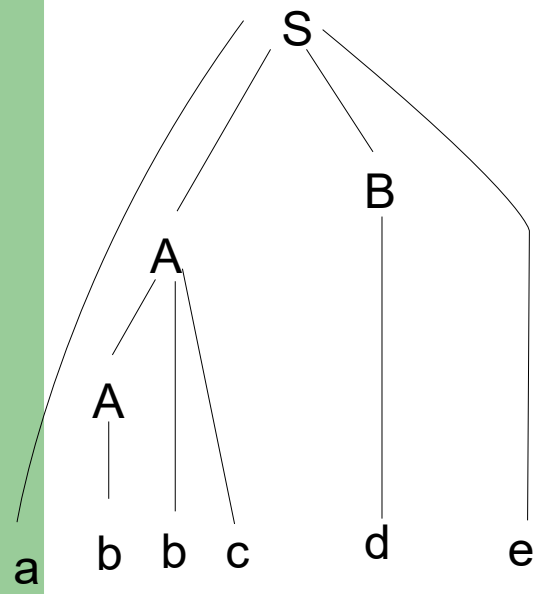
- Malgré les apparences, les analyseurs ascendants sont plus efficaces que les analyseurs descendants. En outre, ils acceptent une classe de langage plus large
- Le principal inconvénient de ces analyseurs est qu'ils nécessitent des tables qu'il est extrêmement difficile de construire à la main

Exemple

$$\begin{cases} S \rightarrow aABe \\ A \rightarrow Abc \mid b \\ B \rightarrow d \end{cases} \qquad et \quad w = abbcde$$

$$S \Rightarrow \underline{aABe} \Rightarrow aA\,\underline{d}\,e \Rightarrow a\,\underline{Abc}\,de \Rightarrow a\,\underline{b}\,bcde$$

On recherche les sous-chaînes correspondants à la partie droite d'une règle de production



On construit ainsi un arbre d'analyse des feuilles vers la racine

Exemple (suite)

La suite des réductions lue à l'envers représente une dérivation droite de l'axiome

$$S \Rightarrow aA\,\underline{B}\,e \Rightarrow a\,\underline{A}\,de \Rightarrow a\,\underline{A}\,bc\,de \Rightarrow abbcde$$

Remarque : le choix de la sous-chaîne à réduire ne doit pas être quelconque. Dans notre exemple :

$$aAAcde \Rightarrow aA\,\underline{b}\,cde \Rightarrow a\,\underline{b}\,bcde$$

bloqué

Définition

On appelle manche de chaîne, une sous-chaîne correspondant à la partie droite d'une production et dont la réduction vers le non-terminal de la partie gauche représente une étape le long de la dérivation droite inverse.

$$\left| \begin{array}{l} si \quad S \overset{*}{\underset{d}{\Rightarrow}} \alpha A w \underset{d}{\Rightarrow} \alpha \gamma w \\ avec \quad \begin{cases} w \in \Sigma^* \\ A \to \gamma \in P \end{cases} \end{array} \right.$$

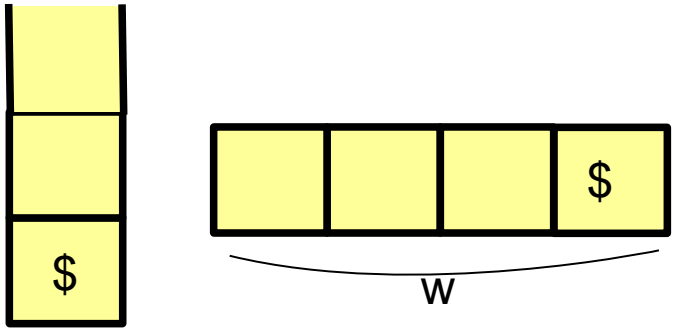
alors γ est un manche de $\alpha\gamma w$

Si la grammaire est ambiguë, il peut exister plusieurs manches dans une chaîne

Mise en place de l'analyse par décalage - réduction

Implémentation à l'aide d'une pile

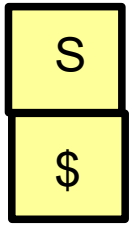
au départ :



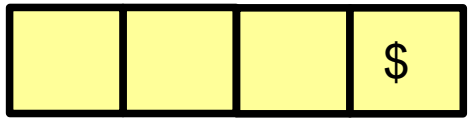
en cours :

On décale de l'entrée vers la pile 0, 1 ou plusieurs symboles jusqu'à ce qu'un manche γ se trouve en sommet de pile. On le remplace par la partie gauche de la production.

On recommence ces « décalages-réductions » jusqu'à ce que l'on détecte une erreur



ou un succès :



Exemple

$$\begin{array}{l} E \to E + E \\ E \to E * E \\ E \to (E) \\ E \to id \end{array}$$

$w = id + id * id$

Cas 1

			$id\ ]$	E	
		$*$	$*$	$*$	
E	E	E	E	E	succès
$\$$	$\$$	$\$$	$\$$	$\$$	

Cas 2

			$id\ ]$	E	
		$+$	$+$	$+$	
$id\ ]$	E	E	E	E	
$\$$	$\$$	$\$$	$\$$	$\$$	

			$id\ ]$	E	
		$*$	$*$	$*$	
E	E	E	E	E	
$+$	$+$	$+$	$+$		
E	E	E	E	E	succès
$\$$	$\$$	$\$$	$\$$	$\$$	

Note : comme la grammaire est ambiguë,
il existe 2 séquences possibles pour l'analyseur

Conflits possibles au cours de l'analyse

- Conflit décaler/réduire : en cours d'analyse, on ne peut pas décider s'il faut réduire le manche présent en sommet de pile ou décaler encore quelques symboles pour faire apparaître un autre manche
  (exemple grammaire précédente)
- Conflit réduire/réduire : le manche au sommet de pile peut être réduit par plusieurs productions (exemple : grammaire où l'appel des procédures avec paramètres à la même syntaxe que les tableaux ou leur indice)

Exemple

```
Proc → id(liste_paramètres)
liste_paramètres → liste_paramètres,paramètre
paramètre → id
tab → id(liste_exprs)
liste_exprs → liste_exprs,expr
expr → id
```

- A l'analyse de l'entrée A(i,j)
- le flot d'unités lexicales est ID(ID,ID)
- Au cours de l'analyse
- ID représente-t-il un indice de tableau ou un paramètre effectif ?

Pour éviter ces conflits, nous allons définir un classe de grammaire LR(k) pour lesquelles l'analyseur décalage/réduction se déroule correctement

Exemple

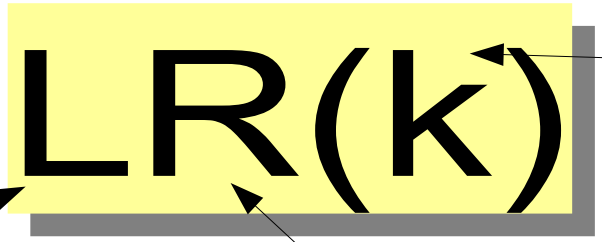
```
Proc → id(liste_paramètres)
liste_paramètres → liste_paramètres,paramètre
paramètre → id
tab → id(liste_exprs)
liste_exprs → liste_exprs,expr
expr → id
```

- A l'analyse de l'entrée A(i,j)
- le flot d'unités lexicales est ID(ID,ID)
- Au cours de l'analyse
- ID représente-t-il un indice de tableau ou un paramètre effectif ?

Pour éviter ces conflits, nous allons définir un classe de grammaire LR(k) pour lesquelles l'analyseur décalage/réduction se déroule correctement

Les analyseurs LR

Analyseurs par décalage réduction sans rebroussement



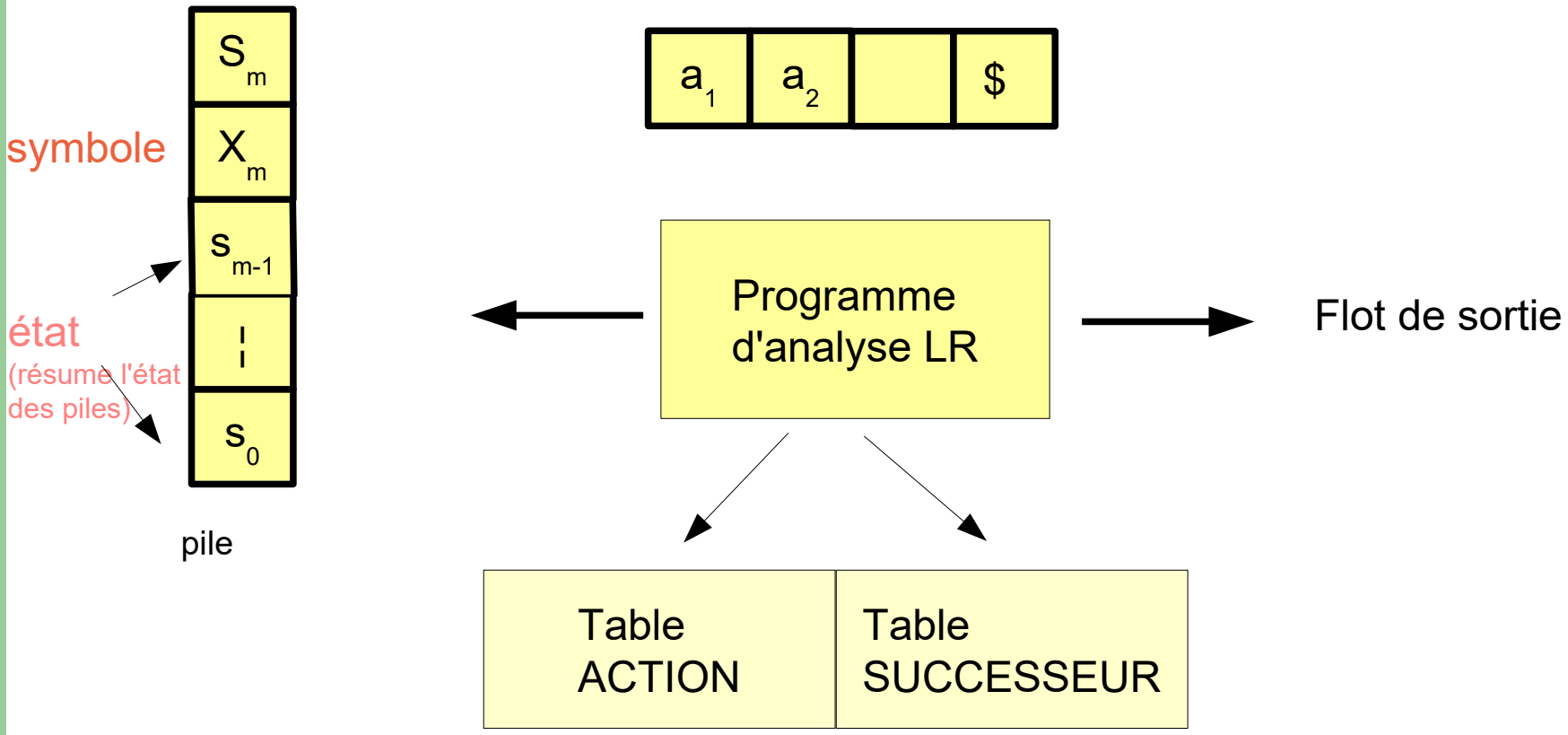
Indique le nombre de symboles de pré-vision pour prendre les décisions d'analyse

Analyse du flot d'entrée de Gauche à Droite (Left to Right scanning)

Construction d'une dérivation Droite inverse (Rightmost derivation in reverse)

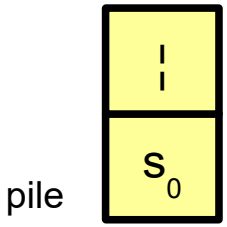
Note : quand k est omis, il est supposé être égal à 1

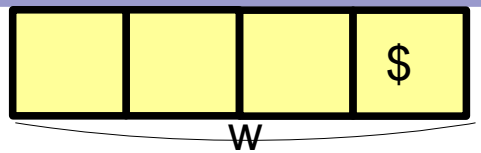
Algorithme d'analyse LR



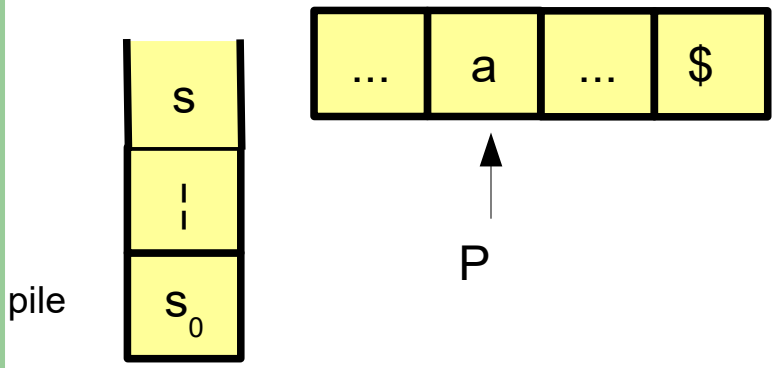
Principe

Au départ :





En cours :



Si Action[s,a]= Décaler s'

$$\begin{bmatrix} empiler(a) \\ empiler(s') \\ avancer(p) \end{bmatrix}$$

si Action[s,a]= Réduire A -> β

$$\begin{bmatrix} Dépiler\ 2*\beta\ symboles \\ Soit\ s'\ l'état\ au\ sommet \\ empiler(A) \\ empiler(Succ[s', A]) \end{bmatrix}$$

si Action[s,a] = Accepter
=> Succès

si Action[s,a] = Erreur
=> Routine de récupération sur erreur

Exemple : grammaire des expressions arithmétiques

notation :

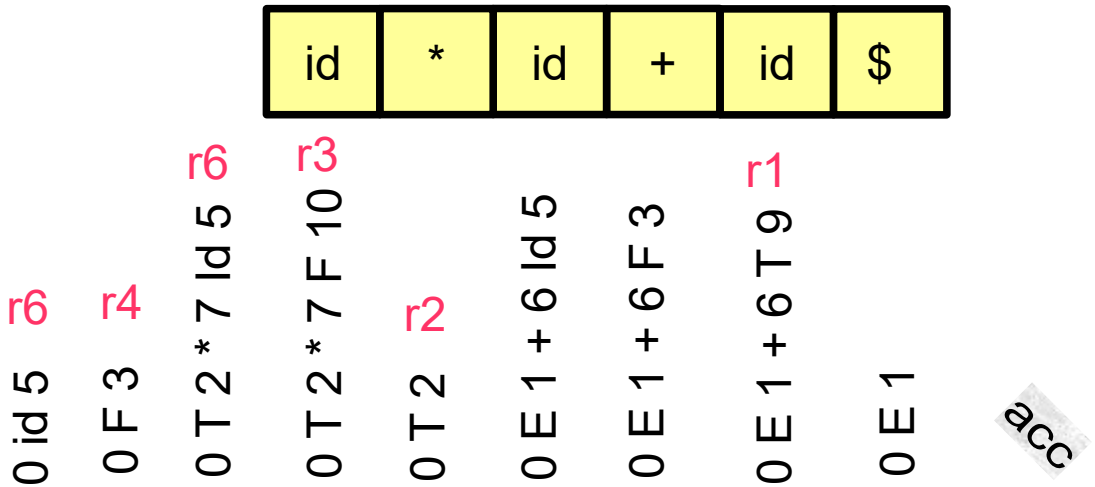
- d_i décaler et empiler l'état i
- r_j réduire avec la règle j
- acc accepter => succès
- 0 erreur

Tables d'analyse

$$
\begin{array}{l}
E \rightarrow E + T \ (1) \\
E \rightarrow T \ (2) \\
T \rightarrow T * F \ (3) \\
T \rightarrow F \ (4) \\
F \rightarrow (E) \ (5) \\
F \rightarrow id \ (6)
\end{array}
$$

Etat	ACTION						SUCC		
	Id	+	*	(	)	$	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			

Exemple (suite)



$$E \Rightarrow E + T' \Rightarrow E + F \Rightarrow E + Id \Rightarrow T + Id$$
$$T + Id \Rightarrow T * F + Id \Rightarrow T * Id * Id \Rightarrow F * Id + Id \Rightarrow Id * Id + Id$$

Conclusion sur les grammaires LR

- Une grammaire pour laquelle nous pouvons construire les tables Action et Successeur d'analyse LR(k) est une grammaire LR(k)
  Il existe des grammaires non contextuelles qui ne sont pas LR, mais tous les langages de programmation classique sont reconnus par des grammaires LR
- Les analyseurs LR sont plus puissants que les analyseurs prédictifs, mais s'il est possible de « programmer à la main » un analyseur prédictif, nous devons utiliser un constructeur d'analyseurs pour les analyseurs LR (par exemple yacc ou bison) car la quantité de travail est trop importante

Construction des tables d'analyse

Il existe 3 méthodes pour construire ces tables d'analyse :

1. SLR (simple LR) : la plus facile mais la moins puissante
2. LR : la plus puissante et la plus coûteuse
3. LALR (Look Ahead LR) : puissante et coût intermédiaire entre les 2 précédentes

Définition

- Un item est une production avec un point repérant une position dans la partie droite

Soit A une règle de production avec $X,Y,Z \in (\Sigma \cup V)$

4 items possibles

$$\begin{vmatrix} A \rightarrow .XYZ \\ A \rightarrow X.YZ \\ A \rightarrow XY.Z \\ A \rightarrow XYZ. \end{vmatrix}$$

interprétation : A →X.YZ

- Nous venons de lire une chaîne dérivée de X
- Nous attendons une chaîne dérivée de YZ

Construction des tables SLR

L'idée est de construire un AFD pour reconnaître les préfixes possibles pour une analyse par décalage – réduction.

Méthode :

1. on augmente la grammaire de la production S'→.S et S' devient le nouvel axiome (à la place de S)
2. pour construire l'automate, on définit 2 fonctions
   Fermeture(I) I étant un ensemble d'items
   - $I \subset Fermeture(I)$
   - $si \quad A \to \alpha . B \beta \in Fermeture(I)$
     $et \quad B \to \gamma \quad est\ une\ production$
     $B \to . \gamma \in Fermeture(I)$

   Transition(I,X) = Δ(I,X) $X \in (T \cup V)$
   est la fermeture de l'ensemble des items de la forme $(A \to \alpha X . \beta) \ tel\ que \ (A \to \alpha . X \beta) \in I$

Exemple de fermeture

- Soit la grammaire :

$$G \begin{cases} E \to E + T \quad (1) \\ E \to T \quad (2) \\ T \to T * F \quad (3) \\ T \to F \quad (4) \\ F \to (E) \quad (5) \\ F \to Id \quad (6) \end{cases}$$

- Et soit l'ensemble d'items

$$I = \{T \to T * . F \, , E \to E . + T\}$$

La fermeture de cet ensemble d'items est :

$$\{T \to T * . F \, , E \to E . + T \, , F \to . Id \, , F \to . (E)\}$$

Exemple de transition

- Sur l'exemple de la grammaire

$$G \begin{cases} E \to E+T \quad (1) \\ E \to T \quad (2) \\ T \to T*F \quad (3) \\ T \to F \quad (4) \\ F \to (E) \quad (5) \\ F \to Id \quad (6) \end{cases}$$

Soit l'ensemble d'items :

$$\begin{Bmatrix} T \to T*.F\,,\, E \to E.+T\,, \\ F \to .Id\,,\, F \to .(E) \end{Bmatrix}$$

- On aura :

$$\Delta(I\,,F)=\{T \to T*F.\}$$

$$\Delta(I\,,+)=\{E \to E+.T\,,T \to .T*F\,,T \to .F\,,F \to .Id\,,F \to .(E)\}$$

Construction des tables SLR (suite)

On part de l'état $I_0 = Fermeture(\{S' \rightarrow .S\})$

Pour chaque symbole X de la grammaire, on effectue la transition (I_0,X) en créant ainsi de nouveaux états I_1, I_2...

On recommence ainsi tant que tous les états n'ont pas été étudiés.

Application :

$$G \left| \begin{array}{ll} E \rightarrow E+T & (1) \\ E \rightarrow T & (2) \\ T \rightarrow T*F & (3) \\ T \rightarrow F & (4) \\ F \rightarrow (E) & (5) \\ F \rightarrow Id & (6) \end{array} \right.$$

Construction des tables SLR (suite)

$$I_0: \left| \begin{array}{l} S' \to .E \\ E \to .E+T \\ E \to .T \\ T \to .T*F \\ T \to .F \\ F \to .(E) \\ F \to .id \end{array} \right.$$

$$I_1: \left| \begin{array}{l} S' \to E. \\ E \to E.+T \end{array} \right.$$

$$I_2: \left| \begin{array}{l} E \to T. \\ T \to T.*F \end{array} \right.$$

$$I_3: T \to F.$$

$$I_4: \left| \begin{array}{l} F \to (.E) \\ E \to .E+T \\ E \to .T \\ T \to .T*F \\ T \to .F \\ F \to .(E) \\ F \to .id \end{array} \right.$$

$$I_5: F \to id.$$

$$I_6: \left| \begin{array}{l} E \to E+.T \\ T \to .T*F \\ T \to .F \\ F \to .(E) \\ F \to .id \end{array} \right.$$

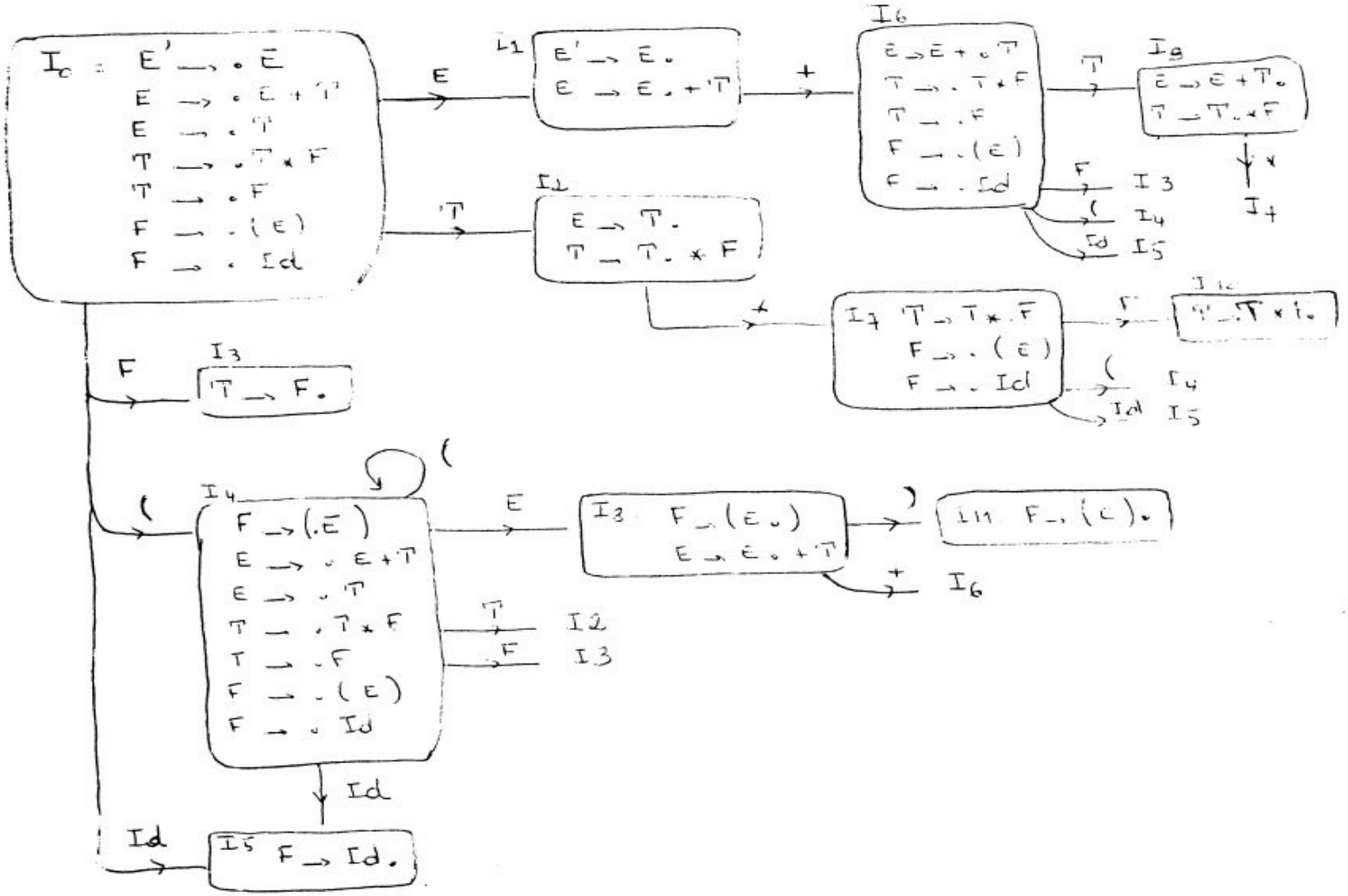
$$I_7: \left| \begin{array}{l} T \to T*.F \\ F \to .(E) \\ F \to .id \end{array} \right.$$

$$I_8: \left| \begin{array}{l} F \to (E.) \\ E \to E.+T \end{array} \right.$$

$$I_9: \left| \begin{array}{l} E \to E+T. \\ T \to T.*F \end{array} \right.$$

$$I_{10}: T \to T*F.$$

$$I_{11}: T \to (E).$$


I0 : E' → . E
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . Id
E
I1
E' → E .
E → E . + T
+
I6
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . Id
T
I9
E → E + T .
T → T . * F
*
I7
T
I2
E → T .
T → T . * F
*
I7 T → T * . F
F → . (E)
F → . Id
F
I10
T → T * F .
F
I3
T → F .
F
I3
(
I4
Id
I5
(
I4
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . Id
E
I8 F → (E .)
E → E . + T
)
I11 F → (E) .
+
I6
T
I2
F
I3
Id
Id
I5 F → Id .

Construction de la table d'analyse

1. Construire la collection d'items $\{I_0, I_1, ..., I_n\}$
2. L'état i est construit à partir de I_i :
   1. Pour chaque $\Delta(I_i,a)=I_j$: mettre décaler j dans M[i,a]
   2. Pour chaque $\Delta(I_i,A)=I_j$: mettre aller en j dans M[i,A]
   3. Pour chaque $A \rightarrow \alpha.$ (sauf A=S') contenu dans I_i : mettre réduire $A \rightarrow \alpha$ dans chaque M[i,a] où $a \in suivant(A)$
   4. Si $S' \rightarrow S. \in I_i$: mettre accepter dans M[i,$]

Construction des tables SLR (suite)

3. Les tables ACTION et SUCCESSEUR

$$\left.\begin{array}{l} si \quad A \to \alpha . a \beta \in I_i \\ et \; transition(I_i, a) = I_j \\ avec \quad a \in T \end{array}\right\} \underline{Action}[i, a] = décaler \; j$$

$$\left.\begin{array}{l} si \quad A \to \alpha . \in I_i \\ et \quad A \neq S' \end{array}\right\} \underline{Action}[i, a] = réduire \; par \quad A \to \alpha \; pour \quad A \in suivant(A)$$

$$si \quad S' \to S. \in I_i \Big\} \underline{Action}[i, \$] = Accepter$$

Toutes les cases d'ACTION non définies sont à « Erreur »

$$\forall A \in V \quad si \; Transition(I_i, A) = I_j \quad alors \quad \underline{Successeur}[i, a] = j$$

	ACTION						SUCCESSEUR		
	Id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6							
2		E→T	d7		E→T	E→T			
3		T→F	T→F		T→F	T→F			
4	d5			d4			8	2	3
5		F→Id	F→Id		F→Id	F→Id			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		E→E+T	d7		E→E+T	E→E+T			
10		T→T*F	T→T*F		T→T*F	T→T*F			
11		F→(E)	F→(E)		F→(E)	F→(E)			

avec suivant(E)={+,),$}

suivant(T)={*,+,),$}

suivant(F)={*,+,),$}

Remarques

- Des conflits peuvent apparaître dans la table d'analyse (grammaire non SLR dans ce cas)
- Pour certaines grammaires, l'analyse SLR n'est pas assez puissante pour se rappeler suffisamment le contexte gauche pour décider de l'action à faire. Pour un item $A \rightarrow \alpha.$, on place cette production dans la table ACTION pour chaque terminal a de suivant(A). Or, il se peut que le manche situé sur la pile à cet état de l'automate ne puisse jamais être suivi de a. L'analyse LR résout ce problème. Elle précise après chaque item, l'ensemble des terminaux qui peuvent effectivement suivre le manche en sommet de pile.

Construction des tables LR

La méthode est similaire à la construction des tables SLR

Fermeture(I)

$$I \subset Fermeture(I)$$
$$si \quad \left.\begin{array}{c}[A \to \alpha . B \beta , a] \in I \\ B \to \gamma \\ b \in Premier(\beta a)\end{array}\right\} \Rightarrow [B \to . \gamma , b] \in Fermeture(I)$$

Transition(I,X) est la fermeture des Items de la forme

$$[A \to \alpha X . \beta , a] \; tel \; que \; [A \to \alpha . X \beta , a] \in I$$

Pour l'automate, on part de I_0= Fermeture([S'→.S,$])

Exemple

$$G \left| \begin{array}{l} S \to CC \\ C \to cC \mid d \end{array} \right.$$

$$first(S) = \{c, d\}$$

$$first(C) = \{c, d\}$$

Fermeture de ([S'→.S,$])

$identification\ avec\ [A \to \alpha . B \beta , a]$
$A = S' \quad \alpha = \epsilon \quad B = S \quad \beta = \epsilon \quad et \quad a = \$$

$Ajout\ de\ [B \to . \gamma , b]$
$\beta \to \gamma \ est\ S \to CC$
$on\ ajoute\ donc\ [S \to . CC , \$]$

$$I_0 \left| \begin{array}{c} S' \to . S , \$ \\ S \to . CC , \$ \\ C \to . cC , c|d \\ C \to . d , c|d \end{array} \right.$$

$identification\ de\ [S \to . CC , \$]\ avec\ [A \to \alpha . B \beta , a]$
$A = S \quad \alpha = \epsilon \quad B = C \quad et \quad a = \$$

$puisque\ C\ ne\ dérive\ pas\ en\ la\ chaîne\ vide, first(C\,\$) = first(C)$
$on\ ajoute\ donc\ [C \to . cC , c], [C \to . cC , d], [C \to . d , d]$

Calcul de transition(I_0,X)

$$pour\ X = S\ \ nous\ fermons\ \{[S' \rightarrow S.,\$]\}$$

$$I_1 : S' \rightarrow S.,\$$$

$$pour\ X = C\ \ nous\ fermons\ \{[C \rightarrow C.C,\$]\}$$

$$I_2 \left| \begin{array}{l} S \rightarrow C.C,\$ \\ C \rightarrow .cC,\$ \\ C \rightarrow .d,\$ \end{array} \right.$$

$$pour\ X = c\ \ nous\ fermons\ \{[C \rightarrow c.C, c|d]\}$$

$$I_3 \left| \begin{array}{l} C \rightarrow c.C, c|d \\ C \rightarrow .cC, c|d \\ C \rightarrow .d, c|d \end{array} \right.$$

$$pour\ X = d$$

$$I_4 : C \rightarrow d., c|d$$

Exemple (suite)

- Aucun nouvel ensemble sur I_1
- I_2 a des transitions sur C, c et d

$$I_5 : S \rightarrow CC.\,,\$$$

- Sur c nous fermons {[C → c.C,$]}

$$I_6 \left| \begin{array}{l} C \rightarrow c.C\,,\$ \\ C \rightarrow .cC\,,\$ \\ C \rightarrow .d\,,\$ \end{array} \right.$$

Exemple (suite)

- Transition (I_2,d)

$$I_7 : C \rightarrow d. , \$$$

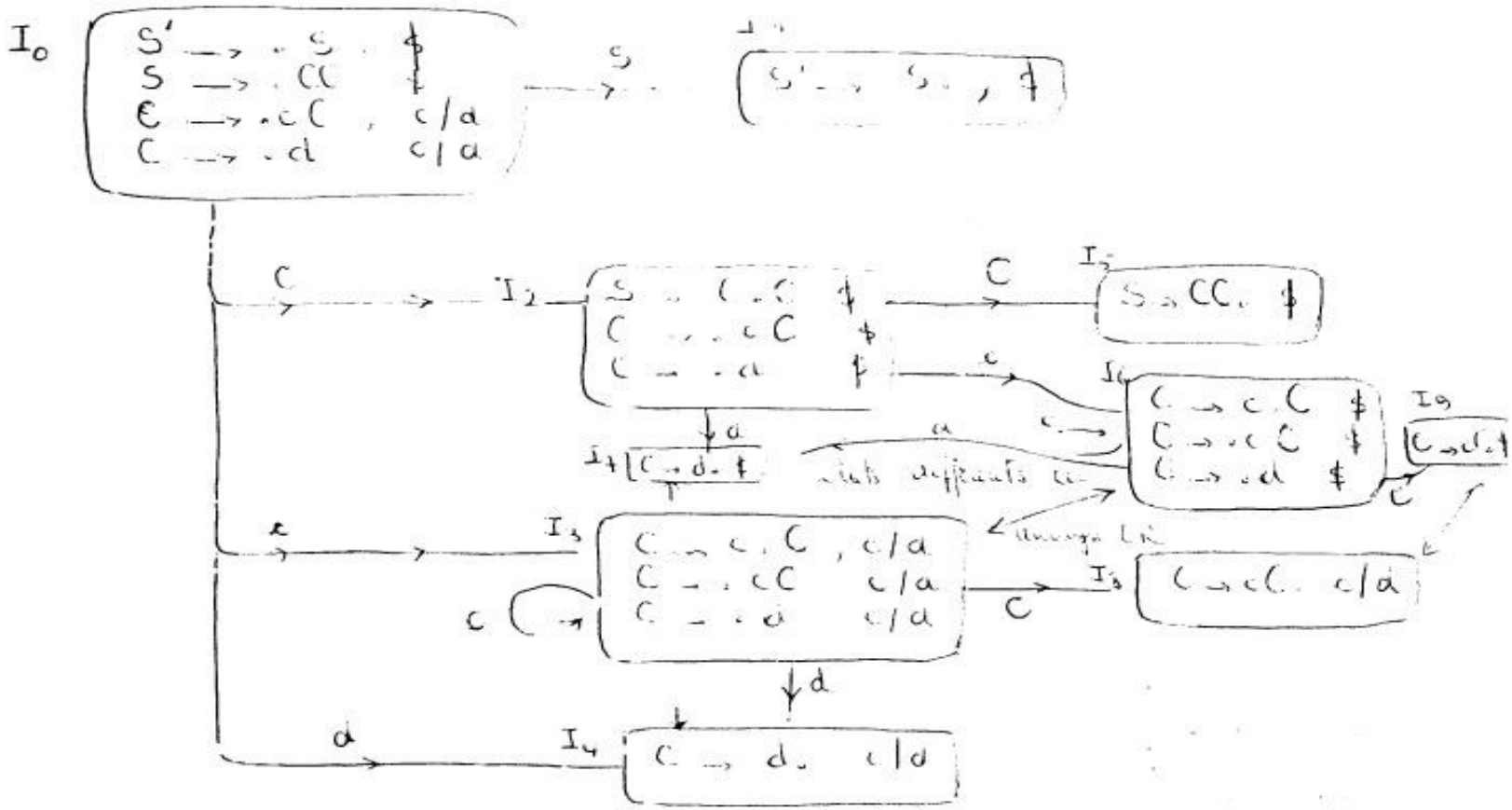
- Transition(I_3,c)=I_3, Transition(I_3,d)=I_4, Transition(I_3,C)=I_8

$$I_8 : C \rightarrow cC. , c|d$$

- Transition pour I_6 sur c et d sont I_6 et I_7 et Transition(I_6,C)

$$I_9 : C \rightarrow cC. , \$$$

Graphe de la fonction Transition



Remplissage de la table ACTION

$$si\ [A \rightarrow \alpha .\ , a] \in I_i \quad A \neq S' \Big\} Action[i, a] = réduire\ A \rightarrow \alpha$$

État	ACTION			SUCCESSEUR	
	c	d	$	S	C
0	d3	d4		1	2
1			Acc		
2	d6	d7			5
3	d3	d4			8
4	C→d	C→d			
5			S→CC		
6	d6	d7			9
7			C→d		
8	C→cC	C→cC			
9			C→cC		

Remarques

- Des conflits peuvent encore apparaître ! Même pour des grammaires non ambiguës. Dans ce cas, la grammaire n'est pas LR (rare pour les langages)
- Toute grammaire SLR est LR mais le nombre d'états de l'analyseur est supérieur au nombre d'états de l'analyseur SLR (la grammaire de l'exemple est SLR et ne nécessite que 7 états dans un analyseur SLR)
- Dans la pratique, un langage comme PASCAL engendre plusieurs centaines d'états en SLR et plusieurs milliers en LR !
- L'analyse LALR évite certains conflits sans engendrer autant d'états que l'analyse (Quelques centaines quand même)

Construction des tables LALR

- LALR : Look Ahead LR
- On construit l'automate d'analyse LR et on regroupe les états ayant un « cœur » commun (les mêmes items indépendamment des terminaux de pré-vision)
- Si aucun conflit ne se produit, la grammaire est LALR

Exemple

Dans l'exemple précédent, on fusionne :

- I_4 et I_7 en I_{47}
- I_3 et I_6 en I_{36}
- I_8 et I_9 en I_{89}

	c	d	$	S	C
0	d36	d47		1	2
1			Acc		
2	d36	d47			5
36	d36	d47			89
47	C→d	C→d	C→d		
5			S→CC		
89	C→cC	C→cC	C→cC		

Cette grammaire est LALR

Il existe un algorithme pour construire efficacement les tables d'analyse LALR (sans passer par les tables LR)

Un générateur d'analyseur syntaxique

- ply.yacc module AS de ply
- import ply.yacc as yacc
- Il est nécessaire de disposer d'une grammaire au format BNF

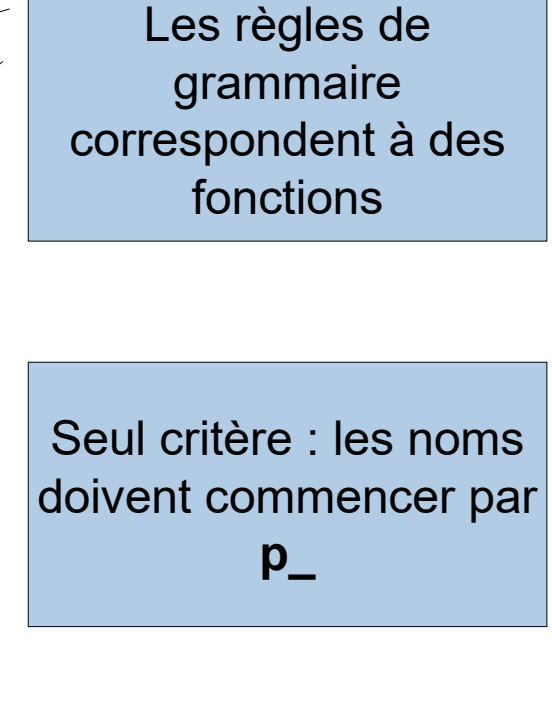
```
Affectation ::= NOM EGAL expr
Expr ::= expr PLUS terme
       | expr MINUS terme
       | terme
terme : := terme FOIS facteur
            | terme DIVISE facteur
            | facteur
facteur ::= NOMBRE
```

Exemple

```
import ply.yacc as yacc

import mylexer # on importe le lexer

tokens = mylexer.tokens # et ses tokens

def p_affectation(p):
    '''affectation : NOM EGAL expr'''

def p_expr(p):
    '''expr : expr PLUS terme
    | expr MOINS terme
    | terme '''


def p_terme(p):
    '''terme : terme FOIS facteur
    | terme DIVISE facteur
    | facteur'''

def p_facteur(p):
    '''facteur : NOMBRE'''
```

Exemple

```
import ply.yacc as yacc

import mylexer # on importe le lexer

tokens = mylexer.tokens # et ses tokens

def p_affectation(p):
    '''affectation : NOM EGAL expr'''

def p_expr(p):
    '''expr : expr PLUS terme
    | expr MOINS terme
    | terme '''


def p_terme(p):
    '''terme : terme FOIS facteur
    | terme DIVISE facteur
    | facteur'''

def p_facteur(p):
    '''facteur : NOMBRE'''
```



Les règles de grammaire correspondent à des fonctions

Seul critère : les noms doivent commencer par **p_**

Exemple

```
import ply.yacc as yacc

import mylexer # on importe le lexer

tokens = mylexer.tokens # et ses tokens

def p_affectation(p):
    '''affectation : NOM EGAL expr'''

def p_expr(p):
    '''expr : expr PLUS terme
    | expr MOINS terme
    | terme '''


def p_terme(p):
    '''terme : terme FOIS facteur
    | terme DIVISE facteur
    | facteur'''

def p_facteur(p):
    '''facteur : NOMBRE'''
```

Ce sont les docstrings qui contiennent les règles au format BNF

Construction du parser

- Parser = yacc.yacc()
- Construction par introspection

Lancement de l'analyse syntaxique

```
parser = yacc.yacc()


data = 'a = 5 +3*2 '
resultat = parser.parse(data)
print(resultat)
```

- Analyse par le parser par invocation des règles

Fonctionnement

- PLY utilise l'analyse LR (comme on a vu dans le cours)
- Plus exactement LALR(1)
- C'est la technique la plus utilisée pour l'analyse syntaxique des langages
- Décalage – réduction
- On remplace la partie droite par la partie gauche
- Analyse ascendante