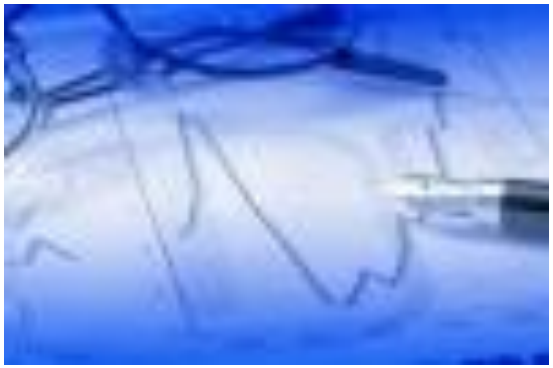
SUPINFO International University

Sign of Success

L’analyse syntaxique

2CMP : Language and translator - Compilation

SUPINFO Official Document



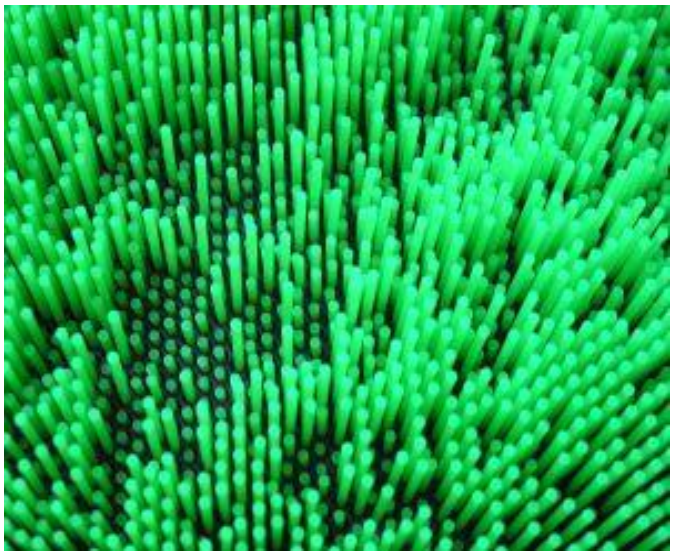
Objectifs de ce chapitre

En suivant ce chapitre vous allez:

- **Comprendre** l'analyse syntaxique au travers d’une analyse descendante et d’une analyse ascendante.
- **Voir** la coopération entre l'analyseur lexical et l’analyseur syntaxique.

Plan du chapitre

Voici les chapitres que nous allons aborder:



- **Les concepts de base.**
- **L’analyse descendante : La méthode LL.**
- **L’analyse ascendante : La méthode LR.**

L’analyse syntaxique

Les concepts de base

Plan de la partie

Voici les parties que nous allons aborder:

- Présentation.
- Rappel sur les méthodes d’analyse ascendantes et descendantes.
- Les méthodes d’analyse.
- Les grammaires LL.

Présentation

La deuxième phase d'un compilateur est : L'**analyse syntaxique**.

Elle consiste à contrôler la bonne forme d'une séquence de **terminaux**, pour déterminer si elle constitue une phrase du langage.

Présentation

Par construction, tout langage de programmation comprend des règles précises, pour bien former nos programmes.

En langage C, par exemple, un programme **main()** comporte, des déclarations, des initialisations, puis des instructions diverses (appel à fonctions, affectations, boucles, conditions, …).

La syntaxe de tel programme peut et sera décrite à l'aide de grammaires **non-contextuelles** (souvenez-vous des BNF).

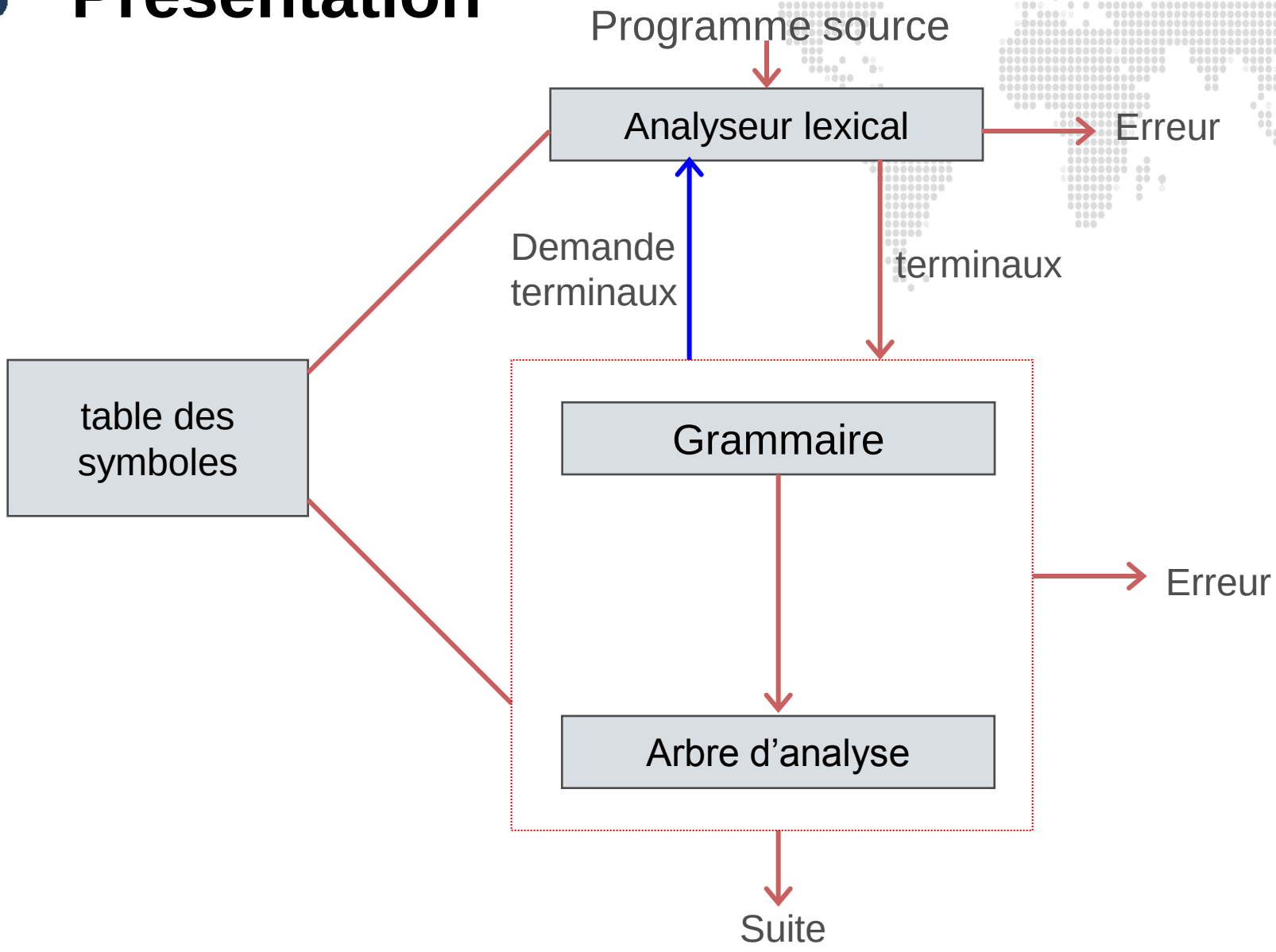
Présentation

Un **analyseur syntaxique** (parseur) est basé sur une notion d'**acceptation** : C'est une fonction booléenne indiquant si le texte source est conforme aux règles de la grammaire définissant le langage.

Un texte de plusieurs milliers de lignes sera perçu comme une seule phrase au sens de la grammaire du langage.

Seule la forme est prise en compte dans l'analyse syntaxique : le fond, la signification du code source, relève de l'**analyse sémantique**.

Présentation



Présentation

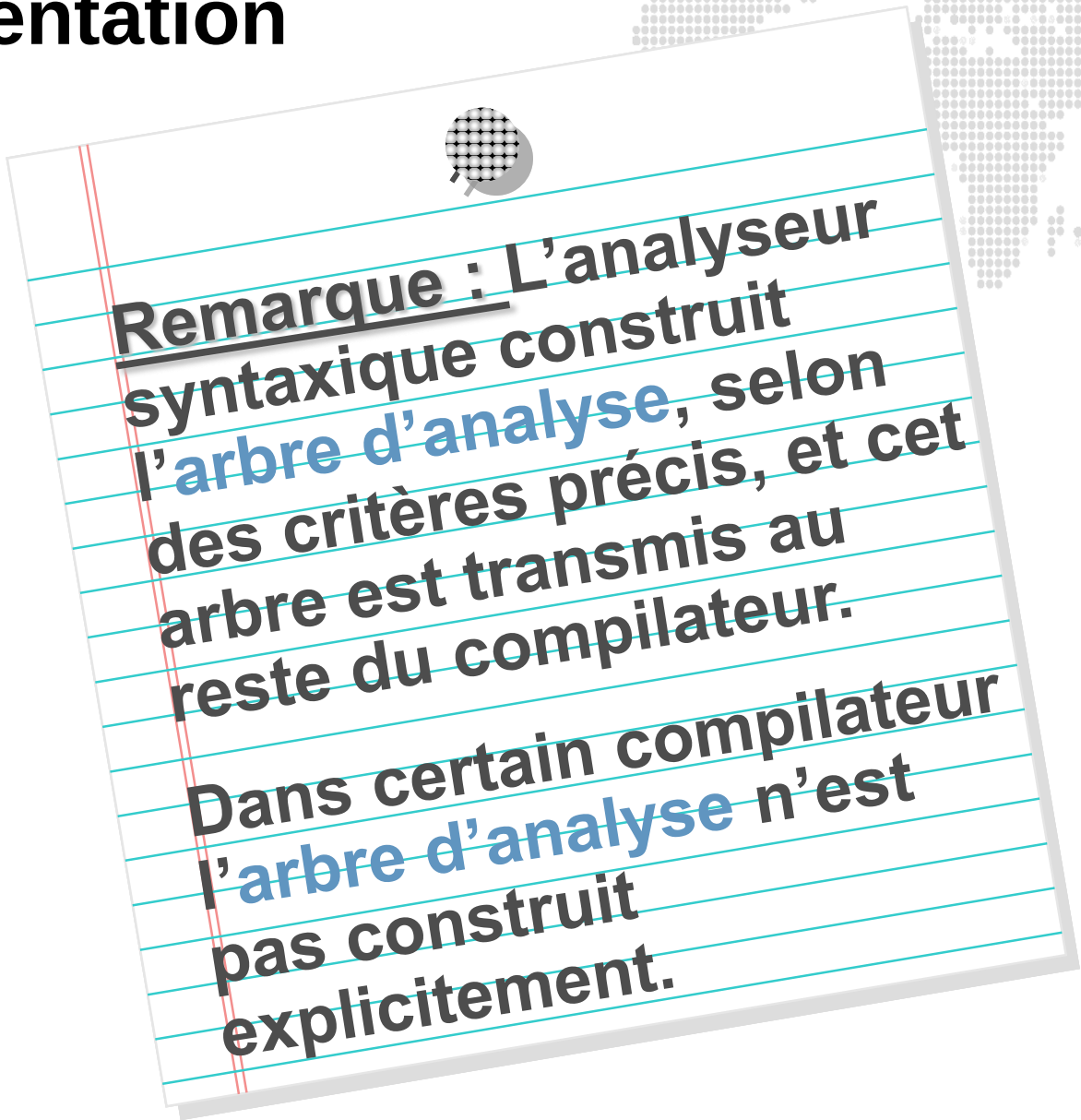
Dans notre modèle, l'analyseur syntaxique reçoit une phrase (séquence de **terminaux**), de l'analyseur lexical et vérifie que cette séquence peut être produite par la grammaire du langage source.

Nous attendons de cet analyseur qu'il signale les erreurs.

Pour cela, lors de l'**analyse syntaxique** d'une phrase, un **arbre d'analyse** est produit.

Cet **arbre d'analyse** est perçu comme un sous-produit de l'**acceptation** de la séquence.

Présentation



Rappels sur les analyses ascendantes et descendantes

Exemple : Nous allons voir, par rapport aux analyses précédentes (chapitre 3), si nous pouvons produire un **arbre d'analyse** pour dériver la séquence de **terminaux** : a + d, selon la grammaire G suivante (manifestement une phrase du langage) :

G = ({ expr ➔ term | term "+" expr | term "-" expr

term ➔ fact | fact "*" term | fact "/" term

fact ➔ "a" | "b" | "c" | "d" | ("expr") | fact "^" fact }

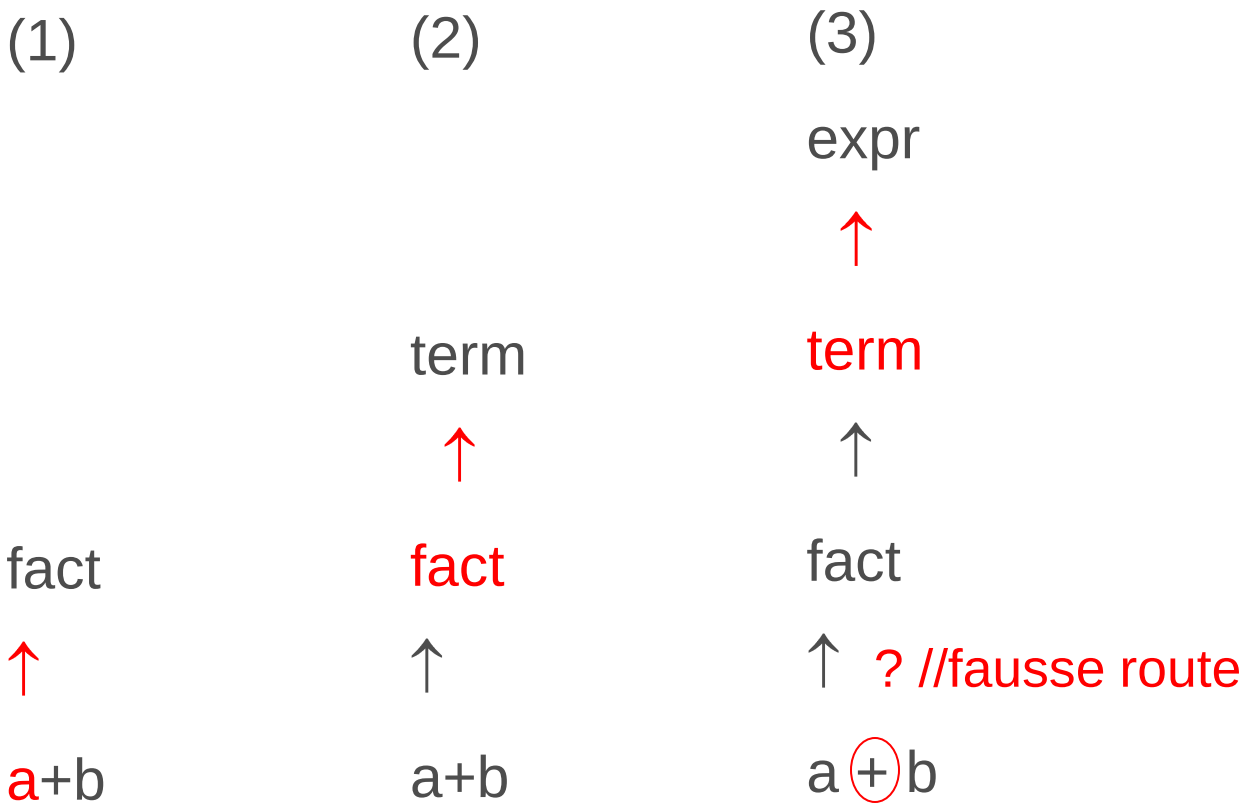
expr /*axiome*/)

- Que donne l'analyse ascendante ?
- Que donne l'analyse descendante ?

Rappels sur les analyses ascendantes et descendantes

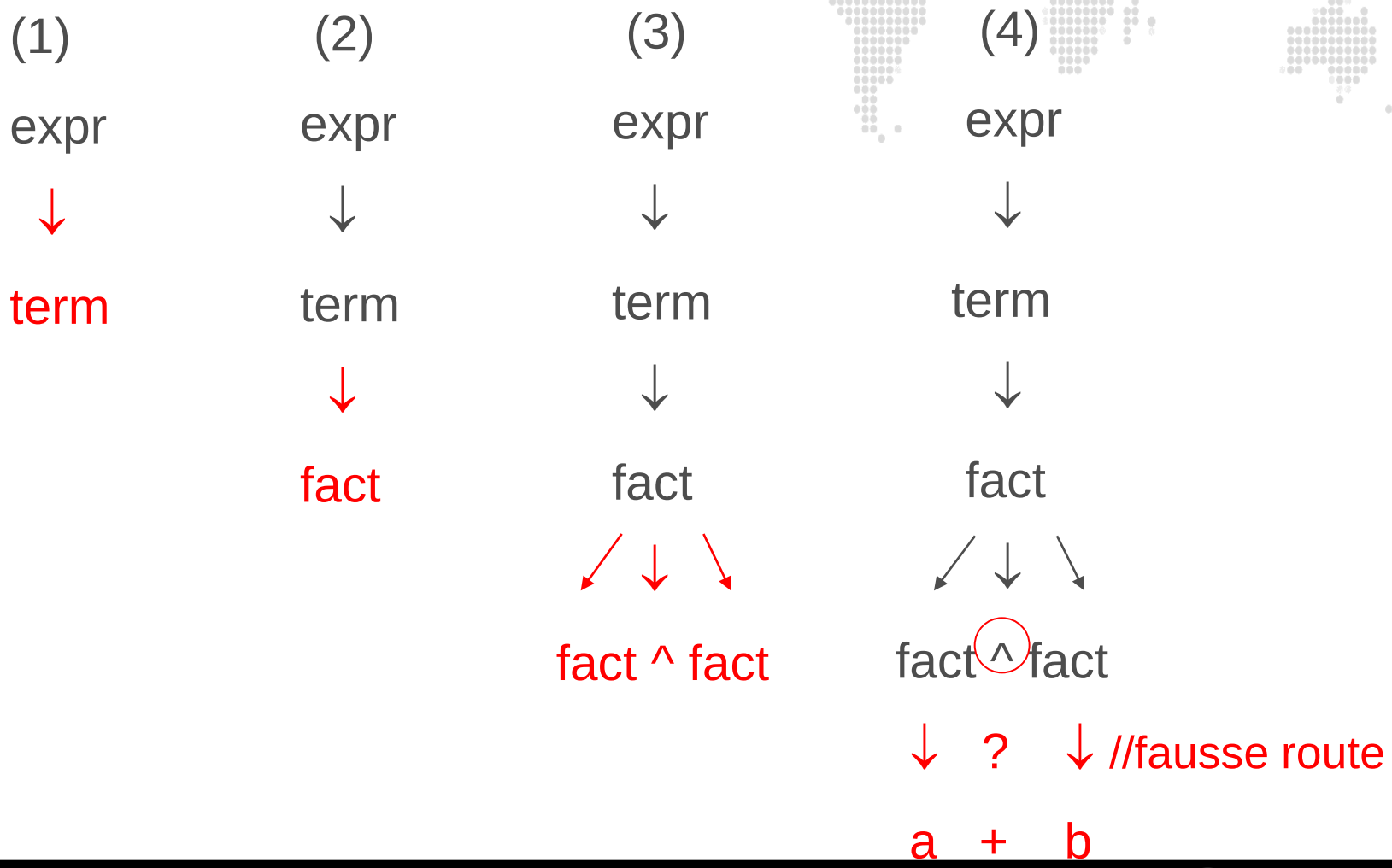
Analyse ascendante :

(1) Règle de production : fact → “a“
(2) Règle de production : term → fact
(3) Règle de production : expr → term



Rappels sur les analyses ascendantes et descendantes

Analyse descendante :



Les méthodes d'analyse

Il y a 3 grandes familles d'analyseurs syntaxiques pour les grammaires d'analyse syntaxique :

1. Les analyseurs **universels**.
2. Les analyseurs **descendants**.
3. Les analyseurs **ascendants**.

Les techniques **universelles** supposent des grammaires quelconques. Cependant ces méthodes ne sont pas assez efficaces (Algorithme de Coke-Younger-Kasami ou l'algorithme de Earley).

Les méthodes d'analyse

Étant donné un langage engendré par une grammaire, nous appelons analyseur d'une phrase de ce langage un **algorithme qui détermine si cette phrase satisfait la grammaire**.

Pour classer les méthodes d'analyse, nous utilisons :

1. **Le critère du sens de parcours de la suite analysée** (de gauche à droite ou de droite à gauche).

2. **Le sens d'application des règles** de la grammaire donnée (par l'avant : dérivation, par l'arrière : réduction).

Les méthodes d'analyse

Le sens gauche → droite est noté par la lettre **L** (Left), inversement le sens droite → gauche est noté par la lettre **R** (Right).

A partir des lettres **L** et **R**, 4 combinaisons sont alors possibles :

LL	analyse descendante
LR	analyse ascendante
RL	analyse descendante
RR	analyse ascendante

La première lettre est le **sens du parcours** de la séquence à analyser, la seconde est le **sens d'application des règles**.

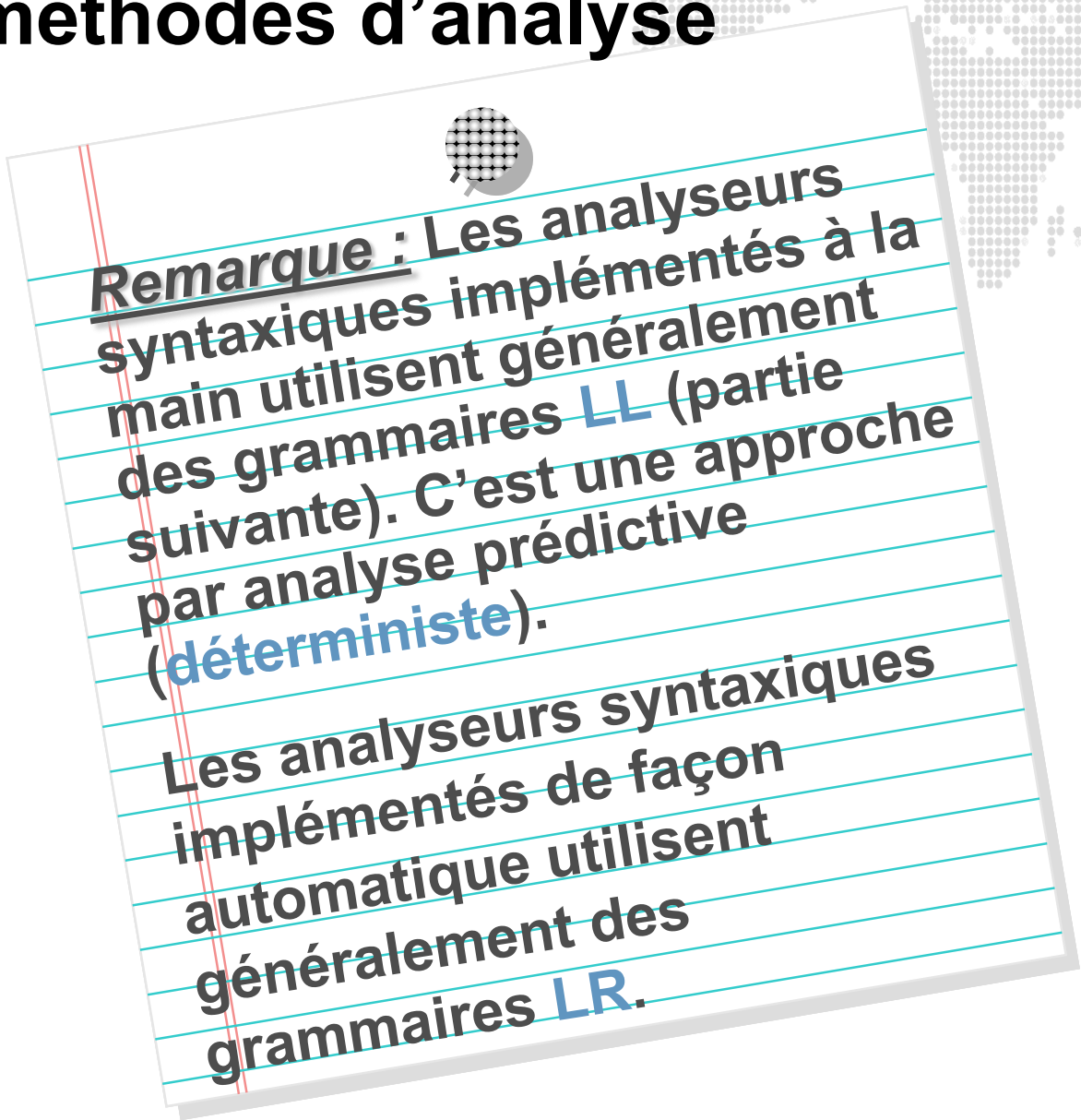
Les méthodes d'analyse

Les méthodes utilisées dans les compilateurs sont classées en méthodes d'analyse **ascendantes** et d'analyse **descendantes**.

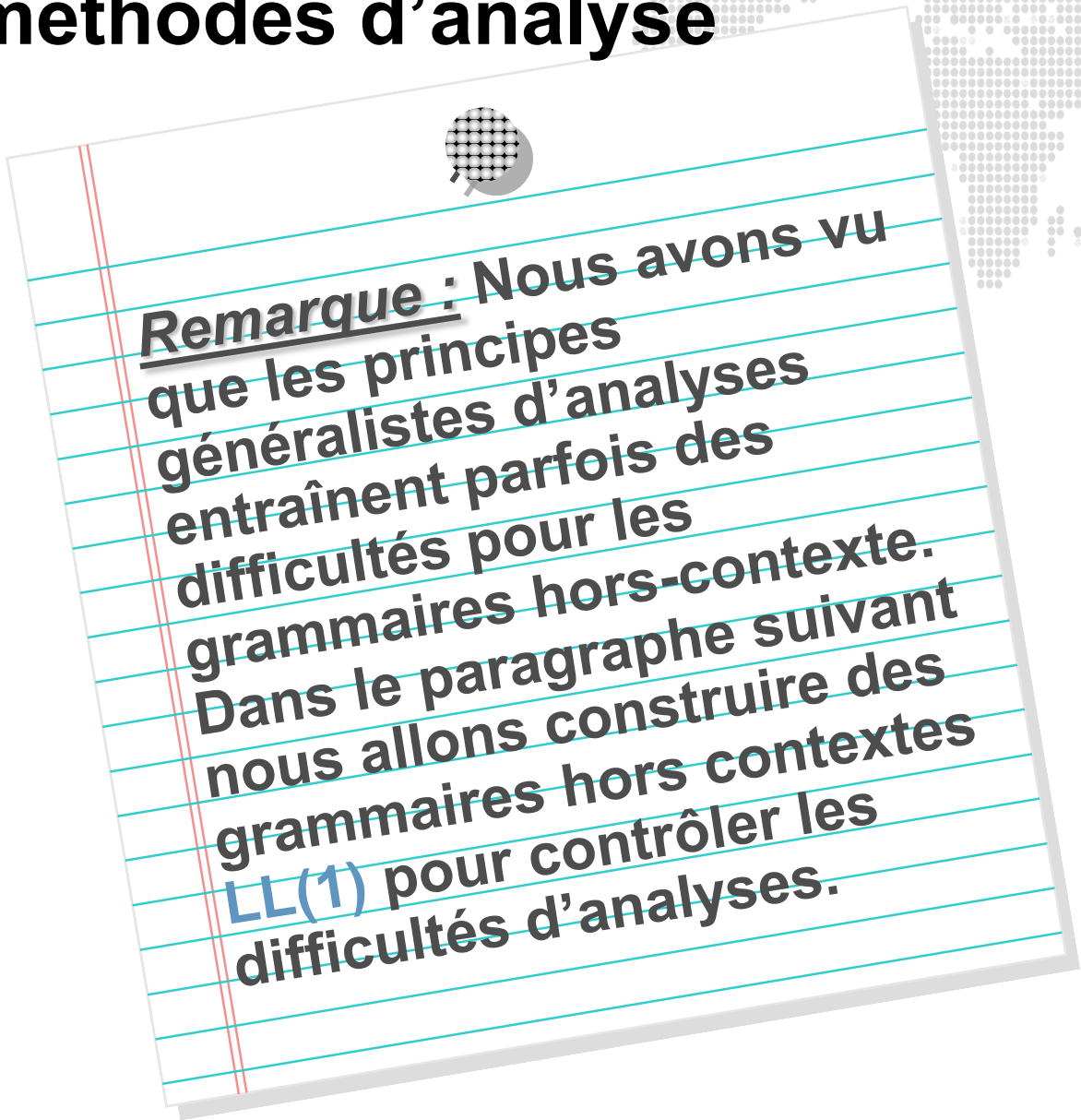
Les techniques d'analyse **ascendantes** et d'analyse **descendantes** les plus efficaces ne fonctionnent que pour certaines sous-classes de grammaires.

Ces sous-classes de grammaires dites **LL** et **LR** sont suffisamment expressives pour décrire la majorité des constructions syntaxiques des langages de programmation.

Les méthodes d'analyse



Les méthodes d'analyse



Les grammaires LL

Une grammaire est dite **LL(n)** si, et seulement si, elle peut être analysée en ne disposant, à chaque instant, que des **n** prochains **terminaux** non encore consommés.

Le premier **L** signifie **left** : provient du sens de parcours de l'entrée. Nous analysons la phrase de gauche à droite en consommant un **terminal** après l'autre, dans l'ordre où ils sont produits par l'analyse lexicale.

Le second **L** signifie **leftmost derivation** (dérivation la plus à gauche) : Nous construisons l'arbre de dérivation de gauche à droite, en dérivant en premier le **non-terminal** le plus à gauche d'une production.

Les grammaires LL

Le **n** appelé lookahead (regarder en avant) indique le nombre de **terminaux** qu'il faut avoir lus sans les avoir encore consommés pour décider quelle dérivation faire.

Par exemple, L'analyse d'une grammaire **LL(3)** impose de gérer 3 variables contenant les 3 prochains **terminaux** non encore consommés à chaque instant, et d'effectuer des permutations circulaires de deux d'entre elles lors de chaque lecture d'un **terminal** : ce n'est pas très efficace.

Les grammaires LL

La classe des grammaires **LL(n)** est assez riche pour couvrir la plupart des constructions des langages de programmation à condition d'être rigoureux dans leur écriture.

Dans la pratique, nous nous limitons aux grammaires **LL(1)**, l'emploi d'un **n** supérieur étant moins efficace que la gestion d'une variable contenant que le prochain **terminal** lu non consommé.

Pour définir les grammaires **LL(1)** nous allons définir la notion d'ensemble de **productions** des règles de réécritures. Il s'agit de définir l'ensemble des **terminaux** pour trouver la reconnaissance des **non-terminaux** utilisés dans les règles.

Les grammaires LL

Pour les grammaires **LL(1)** lorsqu'il y a plusieurs règles associées à un **terminal**, leurs ensembles de productions sont disjoints. Ce qui nous permet de choisir une est une seule étape pour faire l'analyse descendante.

Notons :

productions(A → T) : L'ensemble des productions de la règle A. Ou T dénote l'ensemble de toutes les réécritures de A. A → T1 | T2 | …. | Tn. Avec la propriété : **∩Productions(A→ Ti) =** ∅ pour i allant de 1 à n.

Premier(T) : L'ensemble des **terminaux** qui débute la phrase dérivée de T.

Suivant(N) : L'ensemble des **terminaux** suivant le **non-terminal** N.

Les grammaires LL

Une grammaire G est dite **LL(1)**, si est seulement si, pour toute paire de productions distinctes A → T | F de G, les conditions suivantes sont vérifiées :

1. Il n'y a aucun **terminal a** tel que T et F dérivent toutes deux des chaînes commençant par **a**. Premier(T)=a ou Premier(F)=a, mais Premier(T)=Premier(F)=a n'existe pas .

2. T et F ne peuvent dériver toutes les deux la chaîne vide.

Remarque : Le fait de ne pas avoir à faire de réécriture se dénote vide et peut s'écrire **ε.**

Les grammaires LL

Une grammaire G est dite **LL(1)** (suite) :

3. Si F➔vide (par dérivations successives), alors T ne dérive aucune chaîne commençant par un **terminal** qui est dans la production de A (Suivant(A)=Premier(F), Suivant(A)≠Premier(T)) , inversement pour T➔vide.

Dans ce cas, un **corps de production** au plus peut **engendrer le vide** (sinon toutes les règles engendrant le vide seraient sélectionnables dès que l'une le serait, ce qui n'est pas **déterministe**). Si un corps peut engendrer le vide, aucune autre production ne peut engendrer un **terminal** pouvant suivre cette notion.

Les grammaires LL

La grammaire S → a | bS est **LL(1)**:

Productions(S→a) = Premier(a) = {a}
Productions(S→ bS) = Premier(bS) = Premier(b) = {b}

La grammaire S → a | aS n'est pas **LL(1)**:

Productions(S→a) = Premier(a) = {a}
Productions(S→ aS) = Premier(aS) = Premier(a) = {a}

La grammaire (A→ ε | Exp A, Exp → a | (Exp A)) est **LL(1)**:

Productions(Exp → a) = Premier(Exp) = {a}
Productions(Exp → (Exp A)) = Premier(Exp) = {(}
Productions(A → ε) = Suivant(A)= Suivant(Exp)={)}
Productions(A → Exp A) = Premier (Exp) = {a,(}

Les grammaires LL

Nous avons alors pour l'ensemble premier :

Productions(S→a) = **Premier**(a) si a n'est pas vide.

- Si a est de la forme aS ou a est un **terminal** alors nous avons **Premier**(aS) = {a}.
- Si a est de la forme AS ou A est un **non-terminal** alors nous avons **Premier**(AS) = **Premier**(A).

Nous avons alors pour l'ensemble suivant :

Productions(S→a) = **Suivant**(S) si a est vide.

- Si une règle est de la forme S → aTb alors l'ensemble **Suivant**(T) contient l'ensemble **Premier**(b).
- Si une règle est de la forme A→ aT alors l'ensemble **Suivant**(T) contient l'ensemble **Suivant**(A).

Les grammaires LL

Une grammaire **hors-contexte** de type 2 est dite **LL(1)** si et seulement si :

- Lorsque plus d'une production existe pour une notion **non-terminale** donnée, alors toutes les séquences non vides de **terminaux**, dérivables par les membres droits de ces productions, diffèrent par leur premier **terminal**.

Les grammaires LL

Une grammaire contenant une production récursive à gauche ne peut pas être **LL(1)**.

Exemple : Soit la règle (d'une grammaire de type 2) récursive à gauche suivante :

expression ➔ expression "+" expression | (autres productions)

Tout terminal commençant par expression peut être considéré par :

- Débutant expression figurant avant "+"
- Débutant le corps de l'une des autres productions défini par expression

Nous avons donc 2 dérivations possibles par expression ou par expression sans que nous puissions déterminer, laquelle choisir ???

Les grammaires LL

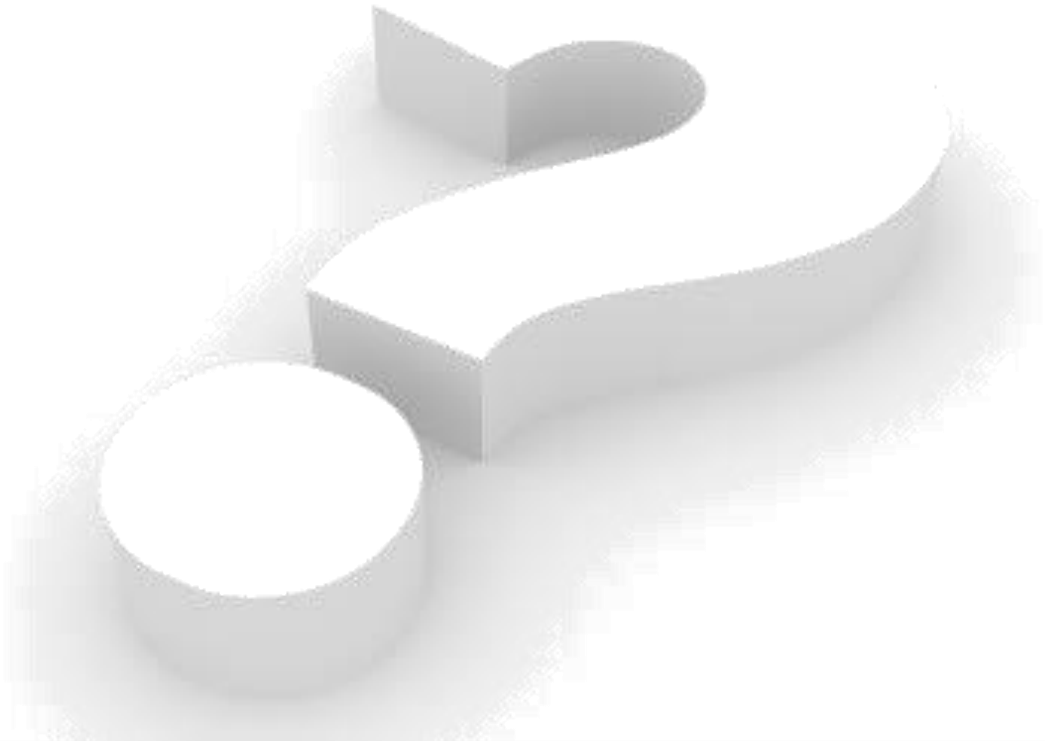
Une grammaire **ambiguë** ne peut pas être **LL(1)** :

1. Une seule alternative sélectionnable à chaque choix de production pour une dérivation
2. Un seul arbre de dérivation pour chaque production sélectionnable.

Une grammaire **LL(1)** ne peut pas être **ambiguë**, et réciproquement, une **grammaire ambiguë** ne peut pas être **LL(1)**.

Pause-réflexion sur cette 1ère partie

Avez-vous des questions ?



L’analyse syntaxique

L’analyse descendante : La méthode LL

Plan de la partie

Voici les parties que nous allons aborder:

- Présentation.
- Exemples.
- Synthèse.

Présentation

Un programme d’analyse syntaxique par **descente récursive** est constitué d’un ensemble de procédures, une pour chaque **non-terminal**.

L’exécution commence par la procédure de l’**axiome** et se termine avec succès si elle parcourt l’intégralité de la séquence de **terminaux**.

Présentation

Nous retrouvons alors dans l'analyse syntaxique par **descente récursive** l'image exacte des productions grammaticales.

Les notions **non-terminales** deviennent des noms de fonctions, et le corps de ces notions devient les instructions de ces fonctions.

La notion d'**accepteur** est sous-entendue : Un booléen implicite de contrôle permet de sortir prématurément ou non de l'exécution : la poursuite de l'analyse indique l'**acceptation**.

Présentation

Les analyseurs syntaxiques par **descente récursive** qui ne nécessitent pas de retour arrière, peuvent être écrits pour des grammaires **LL(n)**. Ces analyseurs sont dits des **analyseurs** syntaxiques prédictifs.

La classes des grammaires **LL(n)** est assez riche pour couvrir la plupart des constructions des langages de programmation à condition d’être rigoureux dans leur écriture.

Exemples

Exemple : Déterminons un programme permettant une analyse syntaxique en reprenant la grammaire G précédente :

G = ({ expr ➔ term | term "+" expr | term "-" expr

term ➔ fact | fact "*" term | fact "/" term

fact ➔ "a" | "b" | "c" | "d" | ("expr") | fact "^"fact}

expr /*axiome*/)

Pour l'exemple, nous décrirons seulement les règles de productions de fact.

Exemples

Pour la fonction fact, nous obtenons alors :

```
Void MonAnalyseur :: _fact()
{
If (sTerminal == “ ( “) {  /*commençons par “(“ expr “)“ */
        _expr();
        If (sTerminal != “ ) “)
                {_ErrSyntax(‘’après expr “ ) “ attendu’’);}
        else
                {_avancer();}
} /*fin de “(“ expr “)“ */
else
{
```

Exemples

```
{
If (sTerminal == (“a“ || “b“ || “c“ || “d“) {
                    _avancer()
}
else
{
          _fact();
          If (sTerminal != “^“)
             {_ErrSyntax(‘’après FACT “^“ attendu ’’);}
          else     {
                    _fact();
                    _avancer();
                    } /*fin de fact “^“ fact*/
}
} /*Fin de la fonction pour les règles de fact*/
```

Exemples

Nous venons de produire un programme permettant de construire uniquement une analyse de la production Fact. Mais pouvons-nous être plus général pour l’ensemble des constructions ?

Pour l’analyse **descendante**, il existe une méthode dite d’**analyse prédictive**.

L’**analyse prédictive** se fait à l’aide d’une **table d’analyse syntaxique prédictive**.

Exemples

Exemple d'utilisation d'une table syntaxique prédictive :

Prenons la grammaire G suivante :

S → Term Exp
Exp → "+"Term Exp | ε
Term → Entier ou "0" | "1" | .. | "9"

A noter : Il est important pour une analyse **LL** de ne pas avoir de dérivation gauche. S est l'axiome.

Pour la grammaire G donnée construisons la **table d'analyse syntaxique prédictive**.

Exemples

Exemple d'utilisation d'une table syntaxique prédictive (suite):

Non-terminal	Symboles d'entrée		
	+	Entier	$
S		S → Term Exp	
Exp	Exp → +Term Exp		Exp → ε
Term		Term → Entier	

$ indique l'élément reconnaisseur de séquence. $ sera placé en début d'analyse pour être atteint en fin d'analyse permettant ainsi l'**acceptation** ou non de la séquence analysée.

A partir de la grammaire et de la **table**, analysons maintenant 1 + 3, pour cela construisons les piles d'analyses :

Exemples

Exemple des piles d'analyses pour l'entrée 1 + 3 (suite) :

nous partons de l'**axiome S** et avec l'indicateur de bas de pile **$**.

Reconnu	Pile	Entrée	Action
	S$	1 + 3$	
	Term Exp$	1 + 3$	S → Term Exp
	EntierExp$	1 + 3$	Term → Entier
1	Exp$	+ 3$	Entier = 1
1	+Term Exp$	+ 3$	Exp → +Term Exp
1+	Term Exp$	3$	+ = +
1+	Entier Exp$	3$	Term → Entier
1+3	Exp$	$	Entier = 3
	$	$	Exp → ε

En fin $ = $ implique que 1+3 a été reconnu comme séquence du langage engendré par la grammaire G.

Synthèse

Les grammaires **LL(1)** sont adaptées aux langages procéduraux, l'analyse se fait :

- de haut en bas par les appels de fonctions,
- et de gauche à droite par le séquencement

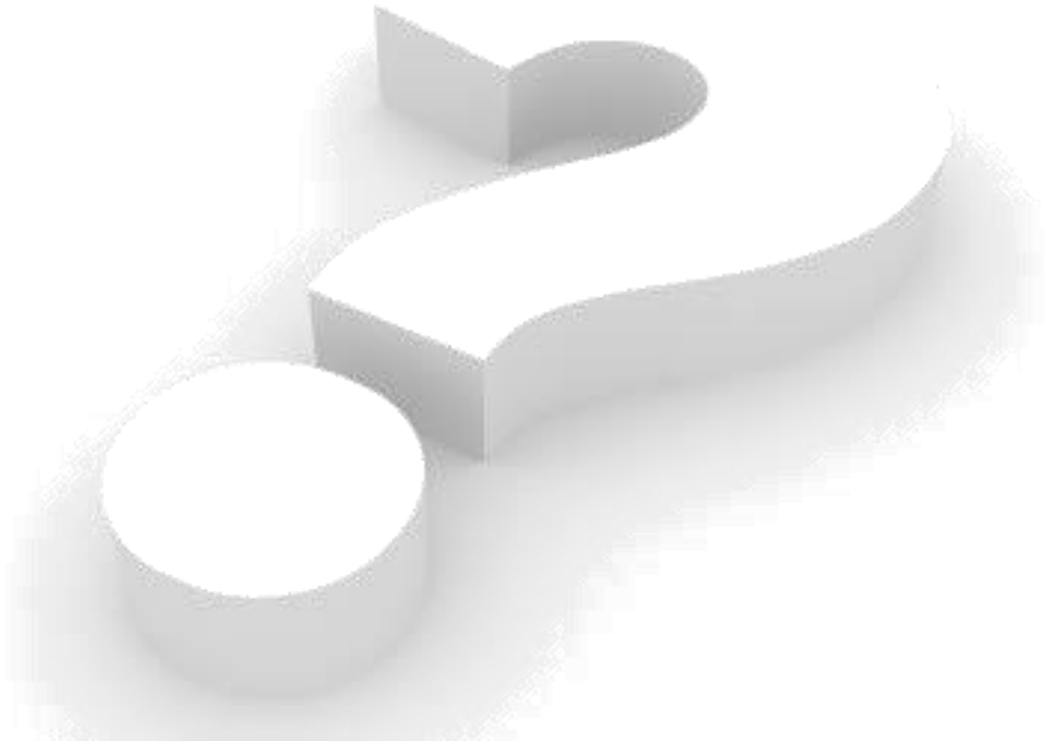
Le **déterminisme** vient de ce que ces langages ne gèrent pas le retour arrière.

La restriction sur la **récursivité à gauche** vient de ce qu'il y a une récursivité infinie dans un code de type :

```
void expression()
{
        expression();
        …
}
```

Pause-réflexion sur cette 2ème partie

Avez-vous des questions ?



L’analyse syntaxique

L’analyse ascendante : La méthode LR

Plan de la partie

Voici les parties que nous allons aborder:

- Présentation.
- Exemples.
- Synthèse.

Présentation

Cette classe de méthodes **ascendantes** due à Donald Knuth (mathématicien américain né en 1938, membre de l'Académie des Sciences) couvre la méthode d'analyse déterministe la plus générale connue applicable aux grammaires non ambiguës.

Elle présente les avantages suivants :

- Détection des erreurs de syntaxe le plus tôt possible, en lisant les **terminaux** de gauche à droite.
- Analyse de toutes les constructions syntaxiques des langages courants.

Présentation

avantages (suite) :

- C'est la méthode la plus générale d'analyse syntaxique par décalage-réduction sans retour-arrière.
- Nous pouvons construire des analyseurs **LR** reconnaissant quasiment toutes les constructions des langages.
- La classe des grammaires analysées est un sur-ensemble de la classe des grammaires analysées en **LL**.

inconvénient :

- Il est compliqué de construire à la main. Heureusement, il existe des constructeurs comme **YACC**.

Présentation

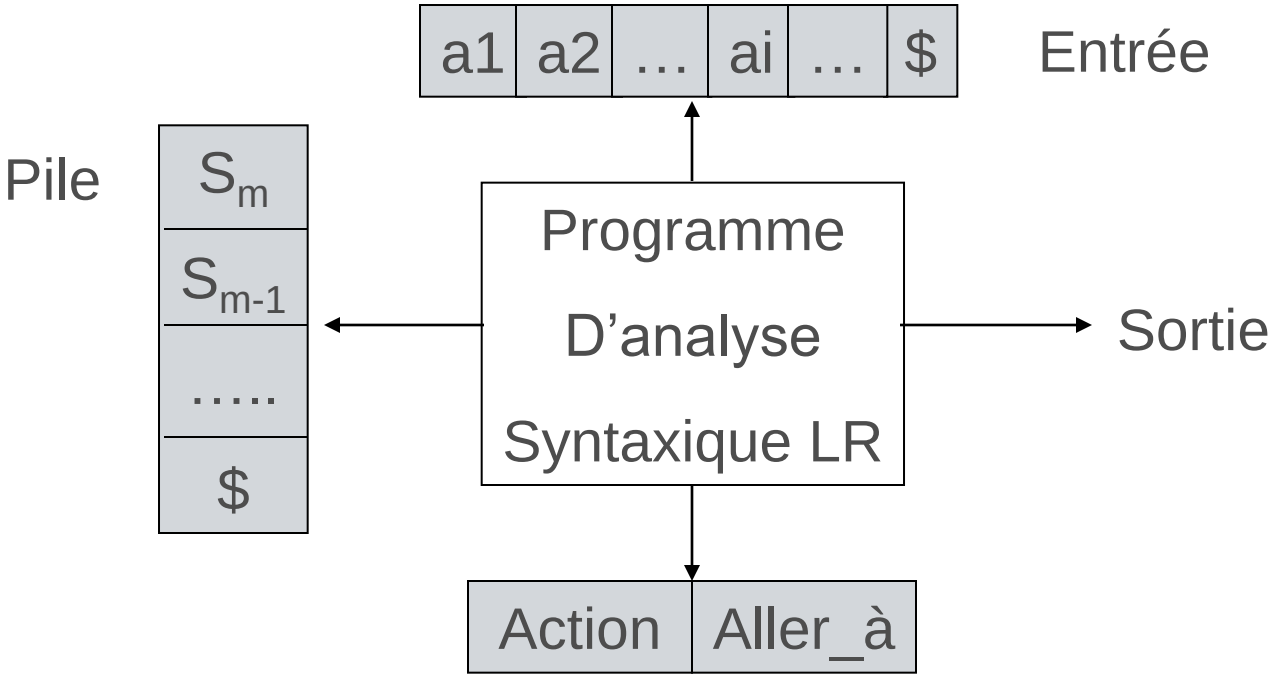
Une grammaire pour laquelle nous pouvons construire une **table d’analyse** en utilisant une méthode d’analyse ascendante sera dite : **LR**

Le **L** signifie **left** : Nous analysons de gauche à droite, comme dans la descente récursive.

Le **R** signifie **rightmost derivation** : Nous construisons l’arbre de dérivation de droite à gauche, en dérivant le premier non terminal le plus à droite dans le corps de production.

Présentation

L'architecture générale d'un analyseur syntaxique **LR** est constitué d'une entrée, d'une sortie, d'une pile, d'un programme de pilotage et d'une table d'**analyse syntaxique**.



Présentation

Définition : L'architecture générale est un 8-uplet {S, Σ, α, P, T, q0, $, F}.

S est l'ensemble fini des états.

Σ est l'alphabet fini d'entrée.

α est l'alphabet fini de sortie.

P est l'alphabet de symboles de pile.

T est la fonction de transition :

$$S \times (\Sigma \cup \{\varepsilon\}) \times P \rightarrow (S \times P^* \times \alpha^*).$$

$q0 \in S$ est l'état initial.

$\$ \in P$ est le symbole de fond de pile.

$F \subseteq S$ est l'ensemble fini des états d'acceptation.

Présentation

Pour cette analyse nous devons construire :

1. Un **automate LR** pour produire les étapes de l'analyse. La Pile (précédente) stocke une séquence d'états de l'automate **LR** s_0 s_1 … s_m où s_m est un sommet. Cet automate est donné à partir de la grammaire.
2. La lecture de l'automate construira la **table d'analyse**. Par construction chaque état de l'automate correspondra à un symbole grammatical.
3. La lecture de la table d'analyse permettra l'**analyse syntaxique**. Les méthodes **LR** construisent l'**arbre d'analyse** de dérivation en ordre inverse, en partant des feuilles.

Présentation

Une position d'analyse **LR** placé dans le corps de chaque production de la grammaire est schématisée par un point •.

Ce • indique que nous avons accepté ce qui précède dans la production, et que nous sommes prêts à accepter ce qui suit le point.

Exemple de positionnement du • :

expression ➔ expression • '' + '' terme

L'idée centrale de la méthode **LR** est : Étant donnée une position d'analyse •, nous cherchons à obtenir par **fermeture transitive** toutes les possibilités de continuer l'analyse du texte source, en tenant compte de toutes les productions de la grammaire par décalage ou réduction.

Présentation

Une position d'analyse est de la forme:

notion ➔ préfixe • non-terminal suffixe

Sa **fermeture transitive** (transitive closure) se construit suivant toutes les productions définissant la notion **non-terminal** de la forme :

Non-terminal ➔ corps

Nous ajoutons à l'état d'analyse le point • pour marquer son début :

Non-terminal ➔ • corps

Présentation

Si corps débute lui-même par une notion de **non-terminal**, alors nous faisons le **fermeture transitive** de cette notion également, et ainsi de suite, jusqu'à **saturation.**

Définition de **fermeture transitive** :

- **Transitive** signifie que nous propageons la connaissance que nous avons de la position d'analyse en tenant compte des productions définissant la notion non terminale que nous sommes prêts à accepter.
- **Fermeture** signifie que nous faisons cette propagation de toutes les manières combinatoires possibles, jusqu'à saturation.

Exemple

Exemple d'analyse avec la méthode LR: Soit la grammaire G, avec des productions récursives à gauche suivantes :

exp_bis → exp

exp →exp + term | term

term → term * fact | fact

fact → (exp) | Entier

L'**axiome** exp_bis permet d'avoir qu'un seul état accepteur (méthode ascendante).

Exemple

Au début de l'analyse nous nous trouvons dans la position initiale : noté Etat_0 dans l'automate LR.

exp_bis → • exp

Nous n'avons encore rien consommé et nous sommes prêt à accepter une exp :

exp → • exp + term

exp → • term

D'après les productions de notre grammaire nous sommes dans l'une des positions d'analyse initiales suivantes:

term → • term * fact

term → • fact

fact → • (exp)

fact → • Entier

Exemple

De **l'état_0** initial, nous pouvons accepter tout ce qui se trouve à droite du point d'analyse ; nous nous retrouvons alors dans un des états suivants :

Etat_1 // accepter exp

exp_bis → exp •

exp → exp • + term

Etat_2 // accepter term

exp → term •

term → term • * fact

Etat_2 // accepter fact

term → fact •

Exemple

Exemple : Prenons la grammaire G suivante :

Exp_bis → Exp
Exp → Exp "+" T | T
T → Entier ou "0" | "1" | .. | "9"

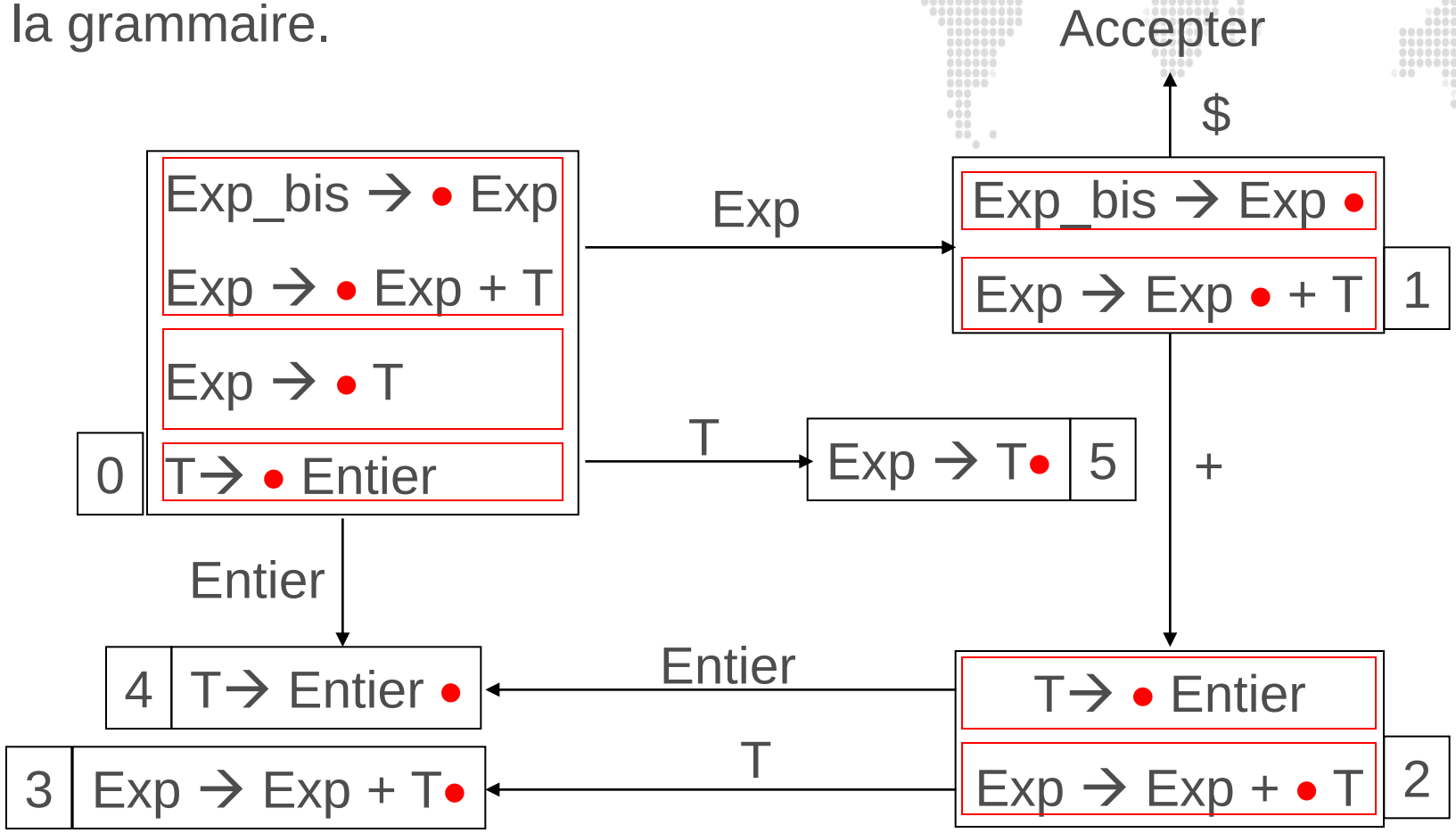
A noter : Un axiome Exp_bis distinct permet de ne pas avoir d'**ambiguïté** pour l'acceptation de l'analyse.

Illustrons notre analyse avec : 1 + 3 soit Entier + Entier

Pour comprendre l'analyse, construisons d'abord l'automate qui décrit les différents états (état_0, état_1, …)

Exemple

Faisons d'abord l'automate des symboles de la grammaire.



Exemple

A partir de l'automate **LR**, nous pouvons ensuite construire la table d'analyse suivante :

État	Actions			Aller_a	
	Entier	+	$	Exp	T
0	d4			1	5
1		d2	acc		
2	d4				3
3	r1		r1		
4	r5		r5		
5	r1		r1		

Exemple

Nous pouvons ensuite analyser Entier + Entier :

Pile états	Symboles	Entrée	Action
0	$	Entier + Entier$	décalage etat4
0 4	$Entier	+ Entier$	réduction T→Entier
0 5	$T	+ Entier$	réduction Exp →T
0 1	$Exp	+ Entier$	décalage etat2
0 1 2	$Exp +	Entier$	décalage etat4
0 1 2 4	$Exp + Entier	$	réduction T→Entier
0 1 2 3	$Exp + T	$	réduction Exp→Exp+T
0 1	$Exp	$	accepter

Synthèse

Dans la terminologie **LR** nous disons que nous avons construit une **table des états d’analyse** indiquant une transition d’un état d’analyse **LR** à un autre par une **réduction** (acceptation) d’un **non terminal** ou par la consommation d’un **terminal**.

L’analyse consistera à effectuer des transitions en fonction des **terminaux** successifs rencontrés jusqu’à arriver à un état acceptant ou une erreur.

Synthèse

Les méthodes **LR** sont les plus générales, au prix de tables d'analyse volumineuses. La méthode **LR** comprend plusieurs cas particuliers, correspondant au même algorithme d'analyse :

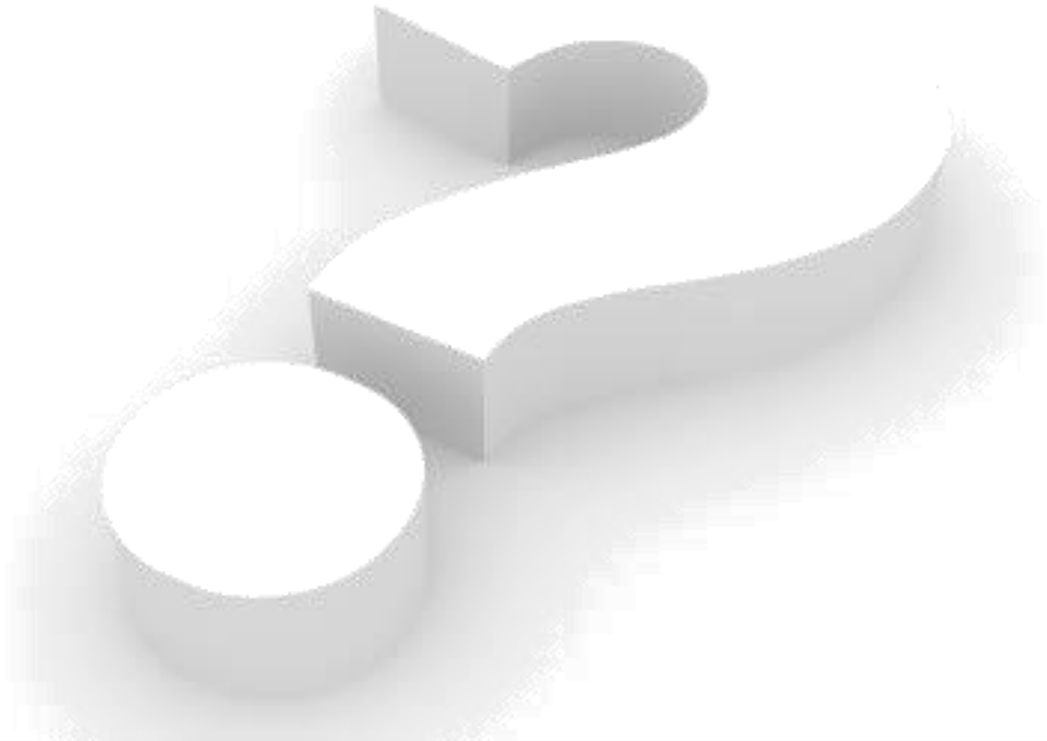
SLR où **S** signifie **simple** : c'est la construction de l'automate **LR** à partir de la grammaire. Transitions données uniquement par Aller_à.

LALR où **LA** signifie lookahead : ce cas couvre beaucoup de langages, avec une taille de table d'analyse de la même taille que **SLR**. L'analyse **LALR (YACC/Bison)** améliore la sélectivité d'un analyseur syntaxique **LR**.

Les méthodes **LR** construisent l'**arbre d'analyse** de dérivation en ordre inverse, en partant des feuilles.

Pause-réflexion sur cette 3ème partie

Avez-vous des questions ?



Résumé du module

Les concepts de base

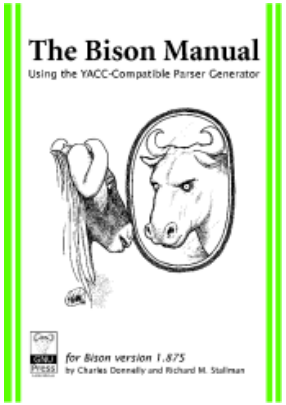
L’analyse ascendante : La méthode LR.

L’analyse descendante : La méthode LL.

Pour aller plus loin…

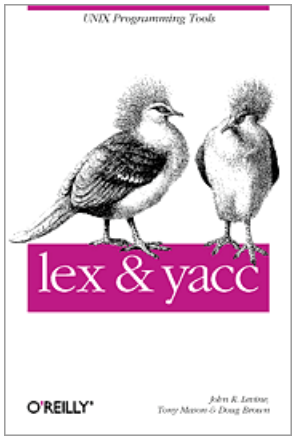
Si vous voulez approfondir vos connaissances:

Publications



The Bison Manual: Using the Yacc-Compatible Parser Generator

C Dennely &Al.



Lex & Yacc

Doug Brown, John Levine, Tony Mason

Fin du module

SUPINFO International University

Sign of Success

Merci de votre attention