# COURS COMPILATION

## MASTER 1 TIC GI

## 2018-2019

**Dr. Michele Mukeshimana**

# ANALYSE SYNTAXIQUE

March,2020

# L'analyse syntaxique

1. Introduction
2. Grammaires, langages, arbres syntaxiques, ambiguïtés
3. Grammaires non contextuelles
4. Analyse descendante (top-down parsing)
5. Analyse ascendante (bottom-up parsing)

# Introduction

- **De part sa conception, chaque langage de programmation a des règles bien précises qui régissent la structure syntaxique (grammaticale) d'un programme bien formé.**
- **En langage C :**
  - **un programme → de fonctions,**
  - **les fonctions →déclaration et instructions,**
  - **les instructions → expressions**
  - **ainsi de suite.**

# Introduction

- Les **grammaires** sont utiles pour les **concepteurs des langages** et les **auteurs des compilateurs**.
- **L'analyse syntaxique** est la **deuxième** étape de la partie analyse d'un compilateur.
  - Elle vise à structurer les mots découverts par l'analyseur lexical sous forme de phrases.

# Introduction

- L'analyse syntaxique prend les mots que produit l'analyse lexicale et
- les regroupe jusqu'à former des phrases du langage en appliquant des règles de grammaire.

➢ Dans le cas des compilateurs, ce regroupement se fait dans la plupart des cas sous la forme d'un arbre syntaxique.

# Introduction

- **L'analyse syntaxique ou analyse grammaticale (le « parser») transforme une suite de lexèmes en un arbre de syntaxe abstraite.**

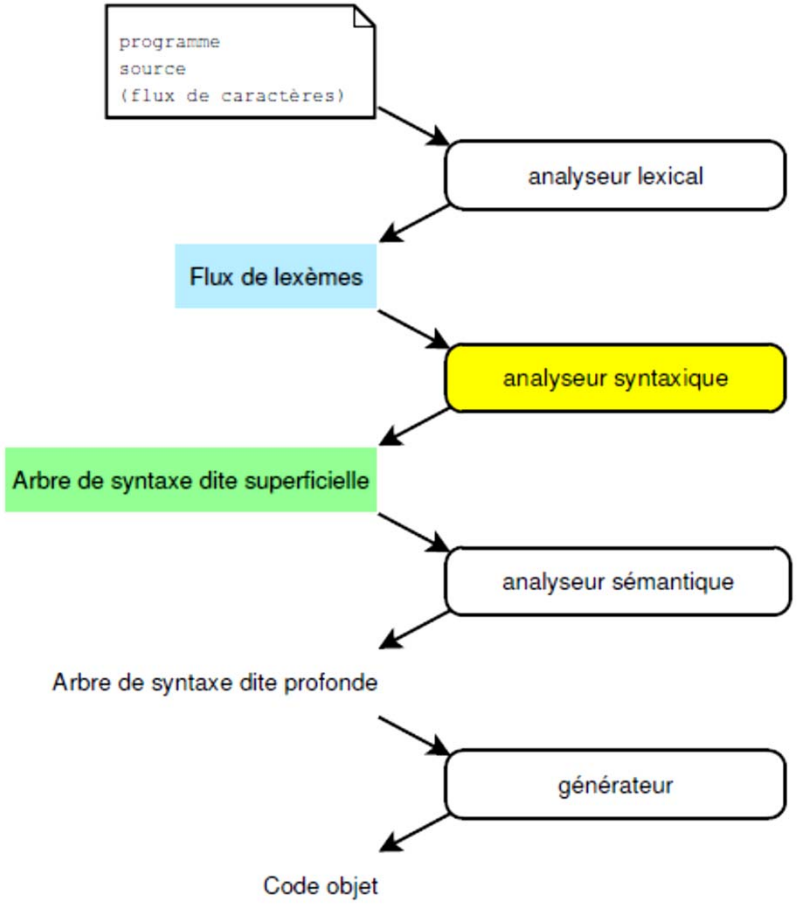
# Introduction

- **L'analyse syntaxique donc, a pour but**
  - **d'organiser les lexèmes des programmes en une représentation compréhensible,**
  - **de sorte a pouvoir la traiter systématiquement par le reste de la chaîne de compilation.**

# Introduction

- **Elle a aussi pour rôle de refuser les programmes (syntaxiquement) mal construits, auxquels le reste de la chaîne ne saurait donner de sens.**
- **La figure suivante rappelle la place de l'analyse syntaxique dans la chaîne de compilation.**

# Introduction



# Introduction

- La grammaire permet l'évolution et le développement itératif d'un langage en y ajoutant de nouvelles constructions pour effectuer de nouvelles taches.
- Ces nouvelles constructions peuvent être facilement intégrées dans l'implémentation que suivit la structure grammaticale d'un langage.

# Grammaires, langages, arbres syntaxiques, ambiguïtés

- **Une grammaire est un ensemble de règles permettant de dire si une phrase( une suite de mots ou lexèmes) est correcte ou non,**
  - **elle est la donnée des éléments constituant le langage (les mots ou lexèmes), et les différentes règles d'assemblage de ces mots.**
  - **Ces règles d'assemblage définissent la plupart du temps des catégories syntaxiques (comme expression ou instruction).**

# Grammaires, langages, arbres syntaxiques, ambiguïtés

- **Les grammaires étudiées dans le cours de compilation ne ressemblent pas formidablement à celle de notre langue maternelle.**
- **Elles sont constituées d'un ensemble de règles qui décrivent complètement la structure de toutes les phrases d'un langage.**

# Grammaires, langages, arbres syntaxiques, ambiguïtés

- **Par exemple une définition générative est la suivante :**

➢ ***Une phrase se compose d'une proposition sujet, d'un verbe et d'une proposition complément.***

- **Alors qu'une définition de la forme :**

❖ ***La subordonnée relative complète un nom ou un groupe nominal appartenant à la proposition principale***

➢ **N'est pas générative : elle ne nous dit pas comment effectivement construire une subordonnée relative.**

# Grammaires, langages, arbres syntaxiques, ambiguïtés

- **Une grammaire G d'un langage est un quadruplet (V, Σ, R, S) où :**

➢ **V : est un alphabet (un ensemble fini de symboles)**

➢ **$\Sigma \subseteq V$ : est l'ensemble de symboles terminaux**

➢ **$S \in V-\Sigma$ : Symbole de départ**

➢ **$R \subseteq (V^{+} \times V^{*})$ : ensemble fini de règles de production**

# Les règles de grammaires

- Le principe de base des grammaires est de donner un ensemble de règles pour engendrer les mots d'un langage.
- Les mots générés par une grammaire sont ceux qui peuvent être obtenus en appliquant ces règles.
- Les règles de grammaires sont formées avec une partie gauche et une partie droite, pour décrire une des constructions du langage.

# Les règles de grammaires

- **La définition générale d'une grammaire devient alors :**
- **Une grammaire G est un quadruplet (N, T, P, S) où :**
  - √ **N est un ensemble fini de symboles dits non terminaux**
  - √ **T est un ensemble de symboles dits terminaux**
  - √ **P est l'ensemble des productions : c'est un sous-ensemble de $(N \cup T)^*N\,(N \cup T)^* \times (N \cup T)^*$ ; on note $\alpha \rightarrow \beta$ un élément $(\alpha, \beta)$ de P**
  - √ **S est un symbole particulier de N et est appelé le symbole de départ.**

# Les règles de grammaires

- Les règles sont notées $\alpha \rightarrow \beta$ où $\alpha$ et $\beta$ sont des séquences de terminaux et de non-terminaux.
- Les symboles terminaux sont ceux qui font partie de l'alphabet sur lequel le langage généré par la grammaire est défini,
  - ce sont les types de mot que renvoie la fonction d'analyse lexicale.

# Les règles de grammaires

- Les symboles non terminaux sont ceux qui n'apparaissent pas dans les mots générés mais qui sont utilisés pour la génération.
- On abrège souvent ces noms en terminaux et non terminaux, sans préciser qu'il s'agit d'un symbole.

# Les règles de grammaires

- **On remarquera que le membre gauche d'une production doit comporter au moins une occurrence de non terminal.**
- **Intuitivement, c'est la catégorie syntaxique associée à ce non terminal que cette production contribue à définir.**

# Les règles de grammaires

- **Généralement, les non terminaux sont représentés par des lettres majuscules, les terminaux par des minuscules.**
- **Exemple :**

  **Soit G la grammaire suivante : G=(N={S,a,b}, T={a,b}, P={S→ε, S→aSb},S)**

  **G définit le langage $\{a^n b^n, n \geq 0\}$ avec un seul non terminal S qui est aussi le symbole de départ.**

# Les règles de grammaires

- **Le mot aabb fait partie du langage défini par cette grammaire car :**

$$
\begin{array}{ll}
S & \\
aSb & (S \rightarrow aSb) \\
aaSbb & (S \rightarrow aSb) \\
aabb & (S \rightarrow \epsilon)
\end{array}
$$

# Les arbres syntaxiques

- **On représente souvent le résultat du travail de l'analyseur syntaxique sous la forme d'un arbre syntaxique : par exemple, considérons la grammaire :**
- **S →aTb ;**

  **T →c S ;**

  **T →d ;**

# Les arbres syntaxiques

- Elle se compose de trois règles, pour un langage dont les seuls mots sont a, b, c et d.
- A partir de ces quatre mots, elle permet de construire une infinité de phrase adb, acadbb, acacadbbb, etc.
- La première règle décrit comment est construit un S, (s→aTb)
- la deuxième et la troisième montrent deux manières différentes de construire un T (T→cS et T→d)

# Les arbres syntaxiques

- **A partir de la grammaire simple**

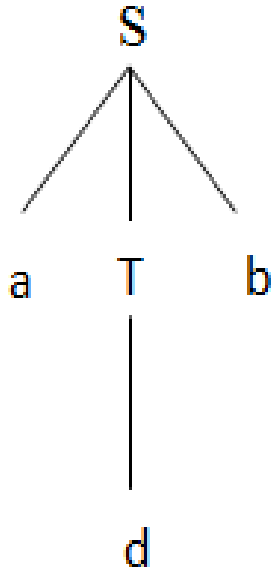
**S →aTb ;**

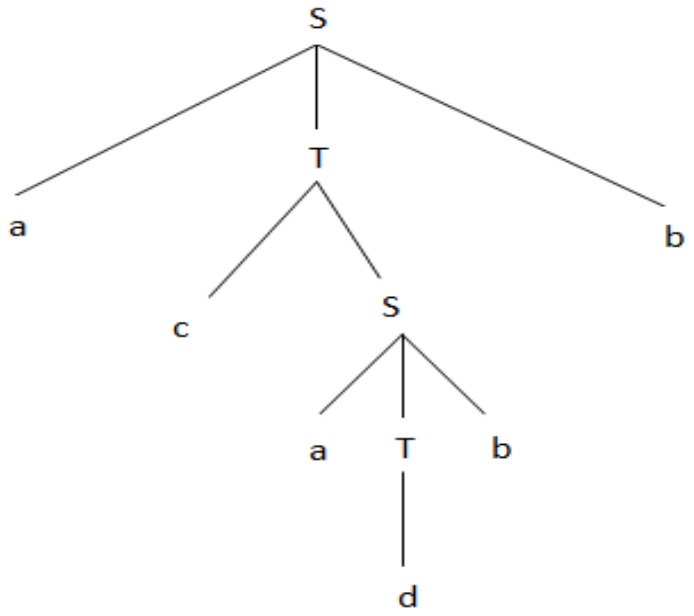
**T →c S ;**

**T →d ;**

- **Deux arbres syntaxiques qui rendent compte de la structure des deux phrases les plus simples du langage adb et du langage acadbb**

# Les arbres syntaxiques



# Les arbres syntaxiques

- La construction de l'arbre syntaxique par l'analyseur s'appelle **une dérivation**.

# Les arbres syntaxiques

- **Quand on construit un nœud de l'arbre syntaxique de type S à partir des mots T et P, on dit qu'on réduit T et P en S.**
- **Le symbole qui doit se trouver à la racine de l'arbre s'appelle le symbole de départ.**
- **Les feuilles de l'arbre correspondent aux terminaux, et les nœuds non-terminaux aux catégories grammaticales.**

# Les grammaires ambiguës, l'associativité et la précédence

- **La grammaire ambiguë**
- **Quand il est possible de dériver plusieurs arbres syntaxiques différents à partir d'une même phrase, on dit que la grammaire est ambiguë.**
- **Les ambiguïtés dans les grammaires proviennent principalement de deux sources :**
  - **soit on peut dériver un mot ou un groupe de mots en non terminaux différents (en utilisant donc des règles différentes),**
  - **soit on peut construire un arbre avec une forme différente en utilisant les mêmes règles.**

# La grammaire ambiguë

- **Pour un exemple de la première sorte, considérons la grammaire élémentaire pour un langage qui ne comprend que la phrase a :**

  **S → T**

  **S → P**

  **T → a**

  **P→ a**

# La grammaire ambiguë

- **Sur a, la phrase unique de la grammaire, on peut dériver deux arbres syntaxiques différents en appliquant des règles différentes :**

➢ **celui où la racine S est constituée d'un T qui lui même est fait d'un a et**

➢ **celui où la racine S est constituée d'un P lui aussi fait d'un a.**

# La grammaire ambiguë

- **Il n'y a pas de manière de déterminer en examinant la grammaire laquelle des deux interprétations doit être préférée.**

# L'associativité

- **Pour un exemple de la seconde sorte, regardons une grammaire pour un langage dont les phrases sont constituées d'un nombre quelconque de a :**

  **S → a**

  **S → SS**

# L'associativité

- Pour la phrase a, il n'y a un qu'un seul arbre : celui où le a est réduit en S par la première règle.
- Pour la phrase aa, chaque a est réduit en S (par la première règle) puis les deux S sont réduits en S par la seconde règle.
- En revanche pour la phrase aaa, il existe deux dérivations possibles (figure ci-dessous).

# L'associativité

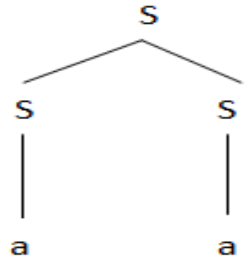
- **Chacune de ces dérivations emploie les mêmes règles mais les applique dans un ordre différent.**

a) $S \rightarrow a$

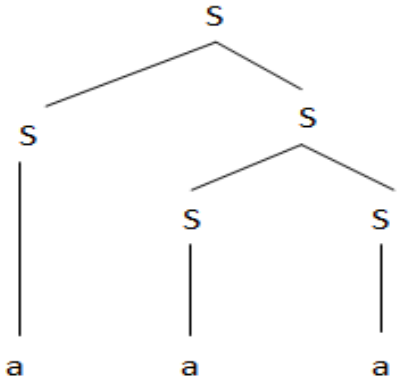
$S : SS$

b)

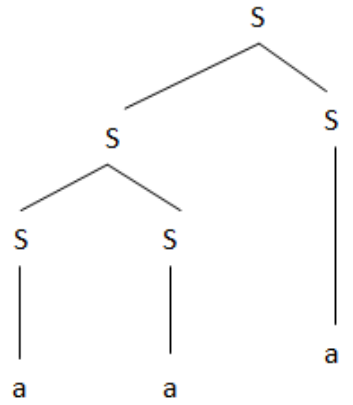

c)




d1)




d2)



# L'associativité

- **Sur cette figure :A partir de la grammaire simple a), on ne peut dériver qu'un seul arbre syntaxique pour la phrase a en b) ou aa (en c) ; en revanche, il y a deux arbres syntaxiques différents possibles pour la phrase aaa (en d1 et d2) : la grammaire est ambiguë.**

# L'associativité

- **On peut rencontrer ce genre de construction dans une grammaire des expressions arithmétiques ; si on se limite aux nombres et à + :**

  **EXPR → nbre**

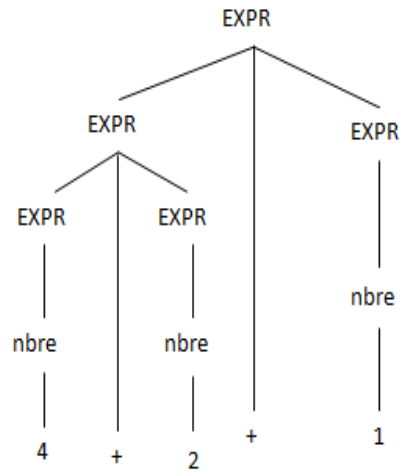
  **EXPR→ EXPR '+' EXPR**

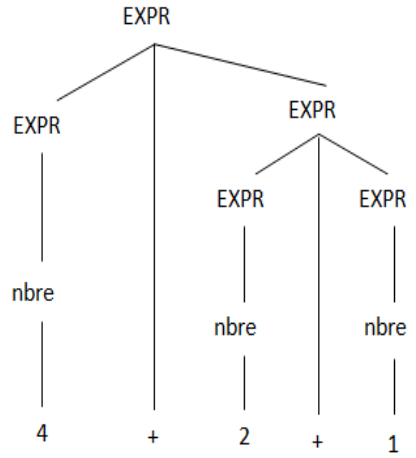
# L'associativité

- **De la même manière que la précédente, cette grammaire permet de dériver deux arbres différents pour une phrase qui contient trois nombres, comme dans la Figure suivante.**
- **Le problème ici est celui de l'associativité de l'opérateur + : l'arbre de gauche de la figure correspond à un + associatif à gauche, celui de droite à un + associatif à droite.**

# L'associativité

- **Deux arbres syntaxiques différents pour la même expression arithmétiques 4+2+1, dérivables à partir de notre grammaire.**

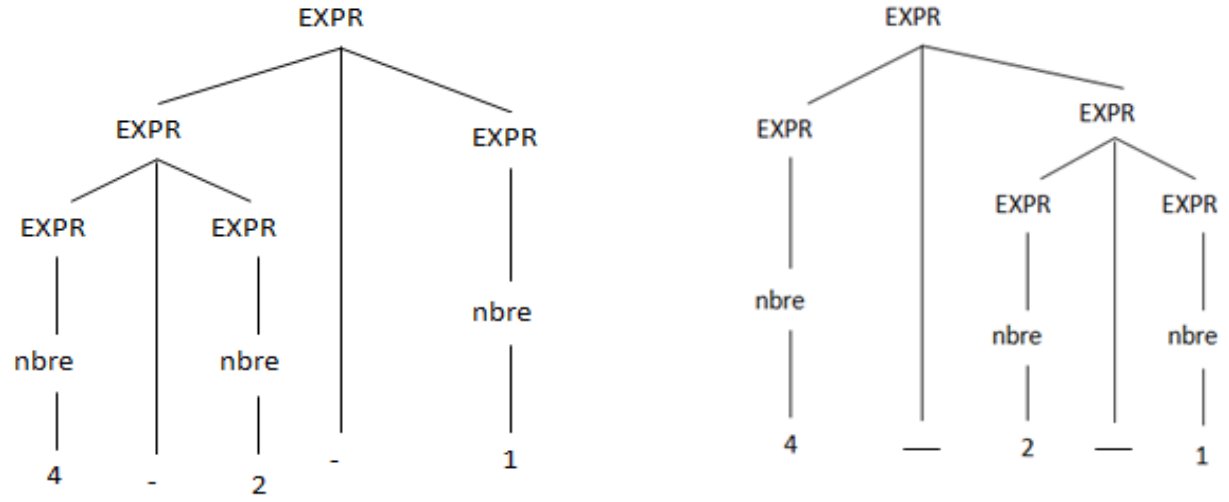



**Chacun des arbres rend compte d'une interprétation différente de la phrase, mais avec les deux arbres expression résultat vaut 7 grâce aux propriétés du +.**

# L'associativité

- **Si on remplace le + par un - dans notre grammaire, les deux arbres correspondront à deux interprétations de l'expression arithmétique qui donnent des résultats différents.**




**Les mêmes arbres syntaxiques que dans la fig.3 : du fait des propriétés du -, l'expression arithmétique dérivée comme dans l'arbre de gauche vaut 1 alors que dérivée comme dans l'arbre de droite elle vaut 3.**

# La précédence

- Avec l'interprétation usuelle des expressions arithmétiques, nous savons qu'une expression comme 1 + 2 *3 s'interprète de manière à ce que sa valeur soit 7.
- Une grammaire possible serait

EXPR → nbre

EXPR → EXPR + EXPR

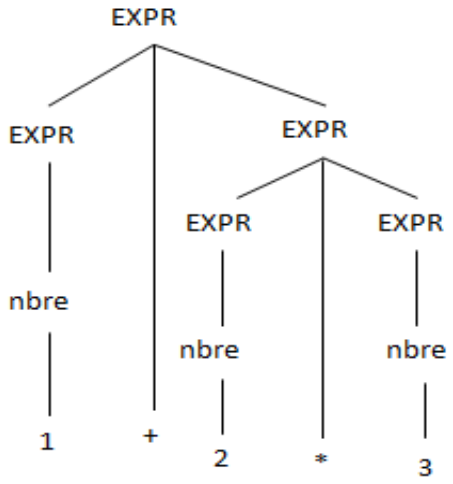
EXPR → EXPR * EXPR

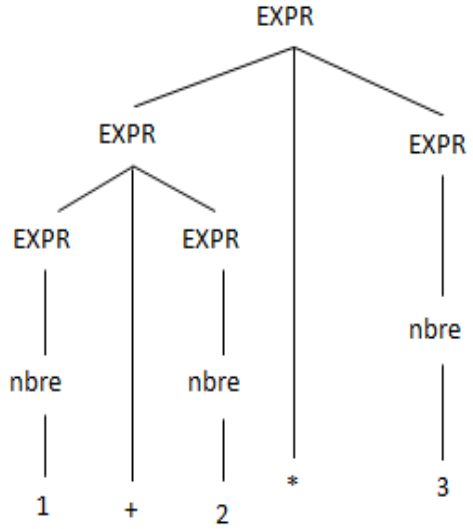
# La précédence

- A partir de cette grammaire, il est possible de dériver plusieurs arbres syntaxiques différents pour une expression arithmétique qui contient à la fois des + et des * .
- Comme dans la section précédente, il s'agit des mêmes règles de grammaires et la question porte sur la manière de les employer.

# La précédence

- **À partir de la phrase 1+2*3, on peut dériver deux arbres syntaxiques différents.**




**Celui de gauche donne une valeur de 9, celle de droite une valeur (habituelle) de 7. La question est celle de la précédence des opérateurs + et *.**

# Réécritures de grammaires

- **Il est toujours possible de réécrire une grammaire pour enlever les ambiguïtés dues aux questions de précédence et d'associativité. Quand une règle présente une ambiguïté due à une question d'associativité comme dans**

**S → a**

**S → SS**

| On peut la réécrire comme | Ou comme |
|---|---|
| S → a | S → a |
| S → Sa | S → aS |

# Réécritures de grammaires

- **Pour forcer l'associativité à gauche ou à droite.**
- **Quand une règle présente une ambiguïté due à la précédence deux opérateurs comme dans**

**X → a**

**X → X b a**

**X → X c a**

# Réécritures de grammaires

- On peut lever l'ambiguïté en introduisant un type de nœud supplémentaire, comme dans

$X \rightarrow X_1$

$X \rightarrow X\ b\ X_1$

$X_1 \rightarrow a$

$X_1 \rightarrow X_1\ c\ X_1$

# Réécritures de grammaires

- Toutes les occurrences de l'opérateur C seront réduites dans des applications de la dernière règle de grammaire avant de réduire les occurrences de l'opérateur B par la deuxième règle.

# Réécritures de grammaires

- C'est pour fixer la précédence des opérateurs d'addition et de multiplication qu'on voit souvent la grammaire des expressions arithmétiques écrite comme :

EXPR → TERME
TERME → FACTEUR
TERME → TERME + FACTEUR

FACTEUR → nbre
FACTEUR → FACTEUR * nbre
FACTEUR → (EXPR)

# Typologie des grammaires

**On distingue 4 types de grammaires:**

- **Type 0 : il n'y a aucune restriction sur les règles de production.**

- **Type 1 : grammaires sensibles au contexte**
  - **Dans une règle, la partie droite doit contenir au moins autant de symboles que la partie gauche.**
  - **Formellement : $|\alpha| \leq |\beta|$**

# Typologie des grammaires

- Ces grammaires sont dites sensibles au contexte car elles permettent d'avoir des règles du genre : aAb→aβb où :
  - a, b sont des terminaux,
  - A est un non-terminal, et
  - β est une chaîne (non vide) dont les éléments peuvent être terminaux ou non-terminaux

# Typologie des grammaires

- Avec une telle règle, A peut être remplacé par β seulement dans le contexte où il est entouré de a et b
- La règle $S \rightarrow \varepsilon$ (où S est le symbole de départ) est permise si aucune des autres règles ne contient S sur sa partie droite

# Typologie des grammaires

- **Type 2 : Grammaires hors-contexte (non contextuelles)**
- **Les règles sont de la forme A→β où :**
  - **A est un non-terminal et**
  - **β est une chaîne (possiblement vide) dont les éléments peuvent être terminaux ou non-terminaux**

# Typologie des grammaires

- **Avantage :**
- **Les grammaires non contextuelles permettent de développer des analyseurs syntaxiques performants**

# Typologie des grammaires

- **Inconvénient :**
- **Langages de programmation ne peuvent pas être entièrement décrits par ce type de grammaire, car ils possèdent des parties sensibles au contexte, telles que :**
  - **règles de compatibilité des types**
  - **correspondance entre les paramètres formels et effectifs d'une procédure**
  - **une variable ne peut pas être déclarée plus d'une fois dans une même portée...**

# Typologie des grammaires

- **Pour résoudre l'inconvénient :**

√**on décrit formellement un langage par une grammaire non contextuelle**

√**les parties sensibles au contexte sont décrites d'une manière informelle**

# Typologie des grammaires

- **Type 3 : Grammaires régulières**
- **Les règles sont de la forme A→ wB ou A →w où :**
  - **A est un non-terminal**
  - **w est une chaîne de terminaux**
  - **B est un non-terminal**

# Typologie des grammaires

- Un langage généré par une grammaire régulière est appelé **langage régulier** et peut être décrit par **une expression régulière**
- Une grammaire régulière peut donc être représentée par un **automate à états fini** (AEF)

# Typologie des grammaires

- **Exemple :**
- **Ensemble des identificateurs commençant par une lettre et constitués d'une séquence de chiffres et de lettres peut être décrit par la grammaire suivante :**

**$V_T$ = {A, B, …, Z, a, b, …, z, 0, 1, …, 9}**

**$V_N$ = {lettre, chiffre, id}**

**S = id**

# Typologie des grammaires

**P = {**

**lettre →( A | B | … | Z | a | b | … | z ),**

**chiffre →( 0 | 1 | … | 9 )**

**id →lettre(lettre|chiffre)***

**}**

- **La grammaire n ’est pas régulière (à cause de la dernière règle)**

# Typologie des grammaires

- La grammaire précédente est équivalente à la suivante :
- $V_T$ = {A, B, …, Z, a, b, …, z, 0, 1, …, 9}
- $V_N$ = {chaîne, id}
- S = id
- P = { chaîne → (A | B | … | Z | a | b | … | z | 0 | 1 | … | 9)
  chaîne → (A chaîne | B chaîne | … | Z chaîne)
  chaîne → (a chaîne | b chaîne | … | z chaîne)
  chaîne→ (0 chaîne | 1 chaîne | … | 9 chaîne)
  id → (A | B | … | Z | a | b | … | z | 0 | 1 | … | 9)
  id → (A chaîne | B chaîne | … | Z chaîne)
  id→ (a chaîne | b chaîne | … | z chaîne) }

# Typologie des grammaires

- La grammaire obtenue est bien régulière puisque les parties droites des règles sont de la forme w ou wB où : B est un non-terminal et w est un terminal

# Typologie des grammaires

- Les grammaires des langages de programmation sont le plus souvent des grammaires de type 2, dont les catégories syntaxiques sont constituées indépendamment du contexte où elles apparaissent.

# Exercices d'application :

1. **Que vaut l'expression 4-2-1en C?**
2. **Soit la grammaire :**

**P : a b**

**P : a b P**

a) **Dériver l'arbre syntaxique de ababab.**

b) **Décrire d'une phrase (en français !) le langage décrit par cette grammaire.**

c) **La grammaire est-elle ambiguë ?**

# Exercices d'application :

3. **Soit la grammaire :**

**P : a b**

**P : a P b**

a) **Générer à partir de cette grammaire une phrase de 2 mots, de 4 mots, de 6 mots, de 8 mots.**

b) **Décrire d'une phrase le langage décrit par cette grammaire.**

# Exercices d'application :

4. **Soit la grammaire :**

**P : /* rien */**

**P : a P b P**

a) **Générer à partir de cette grammaire toutes les phrases de 2 mots, de 4 mots, de 6 mots.**

b) **On peut remplacer le mot a par une parenthèse ouvrante et le mot b par une parenthèse fermante. Qu'obtient-on alors ?**

# Grammaires non contextuelles

- **Les langages de programmation sont souvent définis par des règles *récursives*, comme :**
  - **« on a une *expression* en écrivant successivement un terme, '+' et une *expression* » ou**
  - **« on obtient une *instruction* en écrivant à la suite si, une expression, alors, une *instruction* et, éventuellement, sinon et une *instruction* ».**

# Grammaires non contextuelles

- **Les grammaires non contextuelles sont un formalisme particulièrement bien adapté à la description de telles règles.**

# Définitions

- **Une grammaire non contextuelle, on dit parfois *grammaire BNF* (pour Backus-Naur form):**
- **un quadruplet $G = (V_T ; V_N; S_0; P)$ formé de:**
  - **Un ensemble $V_T$ de *symboles terminaux*,**
  - **Un ensemble $V_N$ de *symboles non terminaux*,**
  - **Un symbole $S_0 \in V_N$ particulier, appelé symbole de départ ou axiome,**

# Définitions

- **Un ensemble *P* de *productions*, qui sont des règles de la forme** $S \rightarrow S_1 S_2 \ldots S_k$

  **avec** $S \in V_N$ **et** $S_i \in V_N \cup V_T$

- **Dans le cadre de l'écriture de compilateurs, on peut expliquer ces éléments de la manière suivante :**

1. **Les symboles terminaux sont les symboles élémentaires qui constituent les chaines du langage, les phrases: donc les unités lexicales**

# Définitions

- Ces unités lexicales sont extraites du texte source par l'analyseur lexical
  - l'analyseur syntaxique ne connait pas les caractères dont le texte source est fait, il ne voit ce dernier que comme une suite d'unités lexicales.
  - C'est donc un mot qui peut faire partie d'une phrase acceptée par le langage.

# Définitions

2. **Les symboles non terminaux sont des variables syntaxiques désignant des ensembles de chaines de symboles terminaux.**

- **Un symbole non-terminal peut être:**
  - **un morceau de phrase ou**
  - **une phrase complète.**

# Définitions

3) Le **symbole de départ** est un symbole non terminal particulier qui désigne le **langage en son entier**.

4) Les **productions** peuvent être interprétées de deux manières :

- comme des **règles d'écriture** (on dit plutôt de **réécriture**), permettant d'engendrer toutes les chaînes correctes.

# Définitions

- De ce point de vue, la production $S \rightarrow S_1 S_2 \ldots S_k$ se lit « pour produire un $S$ correct [de toutes les manières possibles] il faut produire un $S_1$ [de toutes les manières possibles] suivi d'un $S_2$ [de toutes les manières possibles] suivi d'un . . . suivi d'un $S_k$ [de toutes les manières possibles] »,

# Définitions

- comme des règles d'analyse, on dit aussi reconnaissance.

  La production $S \rightarrow S_1 S_2 \ldots S_k$ se lit alors « pour reconnaître un $S$, dans une suite de terminaux donnée, il faut reconnaître un $S_1$ suivi d'un $S_2$ suivi d'un . . . suivi d'un $S_k$ »

# Définitions

- La définition d'une grammaire devrait donc commencer par l'énumération des ensembles $V_T$ et $V_N$.
- En pratique on se limite à donner la liste des productions, avec une convention typographique pour distinguer les symboles terminaux des symboles non terminaux,

# Définitions

- **On convient que :**

➢ **$V_T$ est l'ensemble de tous les symboles terminaux apparaissant dans les productions,**

➢ **$V_N$ est l'ensemble de tous les symboles non terminaux apparaissant dans les productions,**

➢ **Le symbole de départ est le membre gauche de la première production.**

# Définitions

- **En outre, on allège les notations en décidant que si plusieurs productions ont le même membre gauche**

$S \rightarrow S_{1,1} S_{2,1} \ldots S_{1,k1}$

$S \rightarrow S_{2,1} S_{2,2} \ldots S_{2,k2}$

…

$S \rightarrow S_{n,1} S_{n,2} \ldots S_{n,kn}$

- **Alors on peut les noter simplement**

$S \rightarrow S_{1,1} S_{2,1} \ldots S_{1,k1} | S_{2,1} S_{2,2} \ldots S_{2,k2} | \ldots | S \rightarrow S_{n,1} S_{n,2} \ldots S_{n,kn}$

# Définitions

- **Exemple:**
- **voici la grammaire *G*1 définissant le langage dont les chaînes sont les expressions arithmétiques formées avec**
  - **des nombres,**
  - **des identificateurs et**
  - **les deux opérateurs + et ***

  **comme « 60*vitesse+ 200 ».**

# Définitions

- **On a:**

➢ **les symboles non terminaux qui sont : EXPRESSION, TERME et FACTEUR ;**

➢ **le symbole de départ est EXPRESSION**

➢ **les règles de production :**

  ➢ ***EXPRESSION →EXPRESSION "+" TERME|TERME***

  ➢ ***TERME → TERME "*" FACTEUR |FACTEUR* (*G*1)**

  ➢ ***FACTEUR* → nombre | identificateur | "(" *EXPRESSION* ") »**

# Dérivations et arbres de dérivation

- Le processus par lequel une grammaire définit un langage s'appelle **dérivation**.
- Il peut être formalisé de la manière suivante :
- Soit $G = (V_T ; V_N ; S_0 ; P)$ une grammaire non contextuelle,
  - $A \in V_N$ un symbole **non terminal** et
- $\gamma \in (V_T \cup V_N)^*$ une suite de symboles, tels qu'il existe dans $P$ une **production** $A \rightarrow \gamma$.

# Dérivations et arbres de dérivation

- **Quelles que soient les suites de symboles α et β, on dit que αAβ se dérive en une étape en la suite αγβ ce qui s'écrit :**

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

- **Cette définition justifie la dénomination grammaire non contextuelle (on dit aussi grammaire indépendante du contexte ou context free).**

# Dérivations et arbres de dérivation

- En effet, dans la suite $\alpha A \beta$ les chaines $\alpha$ et $\beta$ sont le contexte du symbole $A$.
- Ce que cette définition dit, c'est que le symbole $A$ se réécrit dans la chaine $\gamma$ quel que soit le contexte $\alpha$, $\beta$ dans le lequel $A$ apparait.
-

# Dérivations et arbres de dérivation

- Si $\alpha_0 => \alpha_1 => \ldots => \alpha_n$ on dit que $\alpha_0$ se dérive en $\alpha_n$ en **n étapes**, et on écrit :

$$\alpha_0 \overset{n}{\Rightarrow} \alpha_n$$

- Enfin, si $\alpha$ se dérive en $\beta$ en un **nombre quelconque**, éventuellement nul, d'étapes on dit simplement que $\alpha$ se dérive en $\beta$ et on écrit

$$\alpha \overset{*}{\Rightarrow} \beta$$

# Dérivations et arbres de dérivation

- Soit $G = \{V_T ; V_N; S_0; P\}$ une grammaire non contextuelle ; le langage engendre par *G* est l'ensemble des chaînes de symboles terminaux qui dérivent de $S_0$ :
- $$L(G) = \left\{ w \in V_T^* | S_0 \overset{*}{\Rightarrow} w \right\}$$

# Dérivations et arbres de dérivation

- Si $w \in L(G)$ on dit que $w$ est une phrase de $G$.
- Plus généralement, si $\gamma \in (V_T \cup V_N)^*$ est tel que $S_0 \rightarrow \gamma$ alors on dit que $\gamma$ est une proto-phrase de $G$.
- Une proto-phrase dont tous les symboles sont terminaux est une phrase.

# Dérivations et arbres de dérivation

- **Exemple: Soit la grammaire *G*1 :**

***E* → *E* "+" *T* | *T***

***T*→T "*" *F*|*F* (*G*1)**

***F*→nombre | identificateur | "( *E* )“**

- **Considérons la chaine "60 * vitesse + 200" qui, une fois lue par l'analyseur lexical, se présente ainsi :**
- ***w* = (nombre "*" identificateur "+" nombre).**

# Dérivations et arbres de dérivation

- Nous avons *expression :w*, c'est-a-dire $w \in L(G1)$ ;
- En effet, nous pouvons exhiber la suite de dérivations en une étape :

$E \rightarrow E$ "+" $T$

$\rightarrow T$ "+" $T$

$\rightarrow T$ "*" $F$ "+" $T$

$\rightarrow F$ "*" $F$ "+" $T$

$\rightarrow$nombre "*" $F$ "+" $T$

$\rightarrow$nombre "*" identificateur "+" $T$

$\rightarrow$ nombre "*" identificateur "+" $F$

$\rightarrow$nombre "*" identificateur "+" nombre

# Dérivation gauche

- La dérivation précédente est appelée une **dérivation gauche** car elle est entièrement composée de dérivations en une étape dans lesquelles à chaque fois c'est **le non-terminal le plus a gauche qui est réécrit**.

# Dérivation droite

- **On peut définir de même une dérivation droite, où à chaque étape c'est le non-terminal le plus à droite qui est réécrit.**
- $E \rightarrow E$ "+" $T$
  $\rightarrow E$ "+" $F$
  $\rightarrow T$ "+" $F$
  $\rightarrow T$ "*" $F$ "+" $F$
  $\rightarrow F$ "*" $F$ "+" $F$
  $\rightarrow F$"*" $F$ "+" nombre "
  $\rightarrow F$ "*" identificateur "+" nombre
  $\rightarrow$nombre "*" identificateur "+" nombre

# Arbre de dérivation.

- Soit *w* une chaine de symboles terminaux du langage $L(G)$ ; il existe donc une dérivation telle que $S_0 \rightarrow w$.
- Cette dérivation peut être représentée graphiquement par un arbre, appelé *arbre de dérivation.*

# Qualités des grammaires en vue des analyseurs

- Etant donnée une grammaire $G = \{V_T ; V_N; S_0; P\}$, faire l'analyse syntaxique d'une chaîne $w \in V_T^*$ : c'est répondre à la question « $w$ appartient-elle au langage $L(G)$ ? ».
-

# Qualités des grammaires en vue des analyseurs

- **Parlant strictement, un analyseur syntaxique est donc un programme qui n'extrait aucune information de la chaîne analysée, il ne fait qu'accepter (par défaut) ou rejeter (en annonçant une erreur de syntaxe) cette chaîne.**

# Qualités des grammaires en vue des analyseurs

- En réalité on ne peut pas empêcher les analyseurs d'en faire un peu plus car, pour prouver que $w \in L(G)$ il faut exhiber une dérivation $S_0 \rightarrow w$, c'est-à-dire construire un arbre de dérivation dont la liste des feuilles est $w$.

# Qualités des grammaires en vue des analyseurs

- Or, cet arbre de dérivation est déjà une première information extraite de la chaîne source, un début de compréhension de ce que le texte signifie.
- Nous examinons ici des qualités qu'une grammaire doit avoir et des défauts dont elle doit être exempte pour que la construction de l'arbre de dérivation de toute chaîne du langage soit possible et utile.

# Qualités des grammaires en vue des analyseurs

- Une grammaire est ambiguë s'il existe plusieurs dérivations gauches différentes pour une même chaîne de terminaux.
- Deux grammaires sont dites *équivalentes* si elles engendrent le même langage.
- Il est souvent possible de remplacer une grammaire ambiguë par une grammaire non ambiguë équivalent, mais il n'y a pas une méthode générale pour cela.

# Ce que les grammaires non contextuelles ne savent pas faire

- **Les grammaires non contextuelles sont un outil puissant, en tout cas plus puissant que les expressions régulières, mais il existe des langages (pratiquement tous les langages de programmation,... !) qu'elles ne peuvent pas décrire complètement.**

# Qualités des grammaires en vue des analyseurs

- On démontre par exemple que le langage

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

où a, b et c sont des terminaux, ne peut pas être décrit par une grammaire non contextuelle.

- L est fait de phrases comportant deux chaînes de a et b identiques, séparées par un c, comme ababcabab.

# Qualités des grammaires en vue des analyseurs

- L'importance de cet exemple provient du fait que L modélise l'obligation, qu'ont la plupart des langages, de vérifier que les identificateurs apparaissant dans les instructions ont bien été préalablement déclarés (la première occurrence de w dans wcw correspond à la déclaration d'un identificateur, la deuxième occurrence de w à l'utilisation de ce dernier).

# Qualités des grammaires en vue des analyseurs

- Autrement dit, l'analyse syntaxique ne permet pas de vérifier que les identificateurs utilisés dans les programmes font l'objet de déclarations préalables.
- Ce problème doit nécessairement être remis à une phase ultérieure d'analyse sémantique.

# Analyse descendante (top-down parsing)

- **Une fois bien défini le langage à analyser, c'est à dire une fois posée une grammaire G.**

➢ **On veut d'abord vérifier qu'un mot w de $\Sigma^*$ (une suite de lexèmes) appartient bien à L(G).**

# Analyse descendante (top-down parsing)

- Une première intuition est la suivante : G' n'est pas ambiguë (et L(G) = L(G')), il existe donc un unique arbre de dérivation de w.
- Exemple d'une grammaire non-ambiguë pour les expressions arithmétiques :

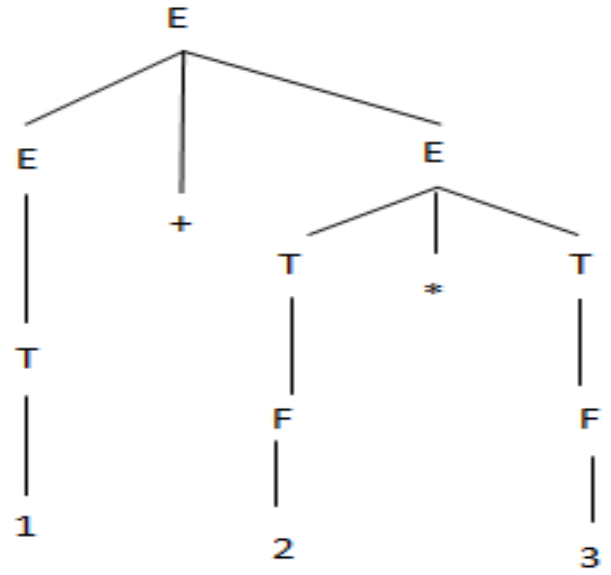
$$E \rightarrow E + T \qquad E \rightarrow E - T \qquad E \rightarrow T$$

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

$$F \rightarrow (E) \qquad F \rightarrow \text{int}$$

# Analyse descendante (top-down parsing)

- **Dans ce cas l'arbre de dérivation de la phrase 1+2*3, le voici :**



# Analyse descendante (top-down parsing)

- En ce qui concerne la réalisation de l'analyse syntaxique, il y a une différence notable entre le schéma fonctionnel de la chaîne de compilation et l'organisation des rapports entre les analyseurs lexical et grammatical.
- La chaîne de compilation fait apparaître deux phases successives :
  - d'abord l'analyse lexicale qui produit une suite de lexèmes,
  - puis l'analyse grammaticale consomme cette suite.

# Analyse descendante (top-down parsing)

- Mais en pratique les appels à l'analyseur lexical sont opérés par l'analyseur grammatical en fonction de ses besoins.
  - L'analyseur lexical consomme les caractères de l'entrée un par un à la demande, c'est la boite flux qui offre cette interface, et de même l'analyseur lexical montre un flux de lexèmes à l'analyseur grammatical.

# Analyse descendante (top-down parsing)

- Le principal impact de cette technique en deux flux est que la mémoire nécessaire pour stoker les lexèmes utiles à un instant donné est constante.
- Si on produisait d'abord une liste de tous les lexèmes, l'analyse demanderait nécessairement une taille mémoire proportionnelle à la longueur de l'entrée.

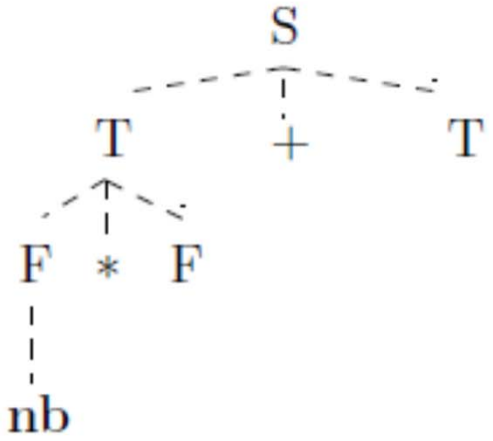
# Analyse descendante (top-down parsing)

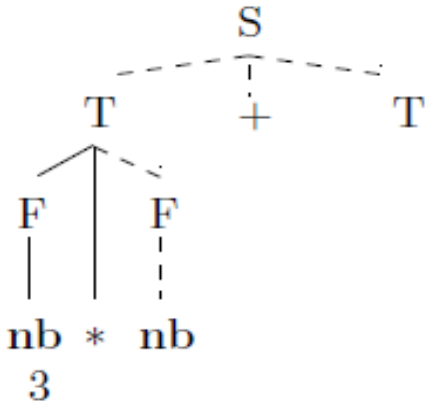
- **L'analyse descendante part du symbole de départ (de la racine) et tente de reconstituer (virtuellement) l'arbre de dérivation par un parcours gauche-droite préfixé ou,**
- **en d'autres termes, elle essaye de dériver le départ par une suite de dérivations gauches.**

# Analyse descendante (top-down parsing)

- L'analyse descendante essaie peut être vu comme une façon de trouver une dérivation à gauche d'une chaîne donnée.
- Par exemple avec {S→T + T, T→F * F, F = nb} et la phrase 3*4+5*7
- La méthode utilisée est la méthode dite **LL**

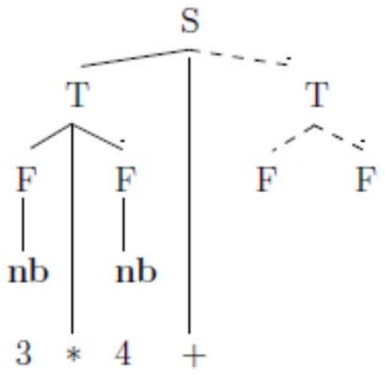
# Analyse descendante (top-down parsing)



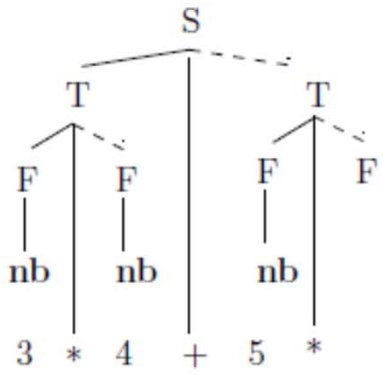




$$S \underset{lm}{\Rightarrow} T + T \underset{lm}{\Rightarrow} F * F + T \underset{lm}{\Rightarrow} nb * F + T \underset{lm}{\Rightarrow} nb * nb + T$$

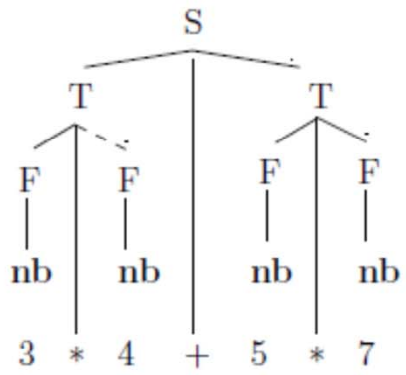
# Analyse descendante (top-down parsing)




$nb * nb + T \underset{lm}{\Rightarrow} nb * nb + F * F$




$\underset{lm}{\Rightarrow} nb * nb + nb * F$




$S \underset{lm}{\Rightarrow} T + T \underset{lm}{\Rightarrow} F * F + T \underset{lm}{\Rightarrow} nb * F + T \underset{lm}{\Rightarrow} nb * nb + T \underset{lm}{\Rightarrow} nb * nb + F * F \underset{lm}{\Rightarrow} nb * nb + nb * F \underset{lm}{\Rightarrow} nb * nb + nb * nb$

# Méthode LL

- LL signifie “Left scanning, Leftmost dérivation.”
- L'analyse d'un programme se fait toujours en lisant les caractères un à un à partir du début de celui-ci (left scanning).
- Lors de cette analyse la construction de l'arbre de dérivation est envisagée à partir de la racine.
- La dérivation est une dérivation la plus à gauche (Leftmost derivation).

# Méthode LL

- **Exemple**
- **On considère la grammaire :T={a, c, d} V={S,T}**

  **P= {**

  ***S → aSbT 2. S → cT  3. S → d  4. T → aT  5. T → bS  6. T → c}***
- **On considère la chaîne : *w= accbbadbc***
- **L'analyse LL consiste à établir l'enchaînement des règles suivant :**

  **S → aSbT →acTbT → accbT → accbbS → accbbaSbT →accbbadbT → accbbadbc**

# Les problèmes rencontrés

- Pour que l'on puisse appliquer une analyse descendante récursive, il faut que toutes les règles de la forme:
- $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$
  puissent être telles que l'on sache toujours quelle règle choisir.
- Il faut que pour un non-terminal donné, toutes les règles associées commencent par un symbole terminal.

# Les problèmes rencontrés

- Les règles doivent donc être de la forme:
- $A \rightarrow a1\alpha_1 \mid a2\ \alpha_2 \mid \ldots \mid a3\ \alpha_n$
- où
- $A \rightarrow \alpha$ (pas d'autre règle ayant A comme membre gauche).

# Les problèmes rencontrés

- En pratique, les langages de programmation « bien écrits », des mots clés (ou des classes lexicales) permettent de distinguer parmi les règles sans problèmes.
- Par exemple, dans :

<instruction> → for <var> := <expr> do <instr> |
if <cond> then <expr> else <expr> |
while <cond> do <instruction> |
begin <instr-list> end |
<procedure>(<arg-list>) |<ident> := <expr>

# Les problèmes rencontrés

- **les mots clés permettent de reconnaître sans ambiguïté le type d'instruction.**
- **Il en est de même des mots clés procedure, function, var, type, const qui sont placés devant ce qu'ils introduisent.**
- **Les classes lexicales <procedure> et <ident> agissent comme des mots clés pour lever l'ambiguïté sur les règles.**

# Les problèmes rencontrés

- **Mais cela ne suffit pas toujours. Il ya un ensemble des opérations que l'on doit appliquer aux règles pour qu'elles aient le bon format.**
  - **Factorisation à gauche**
  - **Expansion partielle**
  - **Traitement des règles vides**
  - **Récursion**

# Factorisation à gauche

- **Les règles de la forme : $A \rightarrow \alpha\beta \mid \alpha\delta$ sont transformées en :**

  $A \rightarrow \alpha A'$

  $A' \rightarrow \alpha \mid \beta$

# Expansion partielle

- **Principe: lorsque des règles commencent par des non-terminaux dans le membre droit, on expanse toutes ces règles jusqu'à obtenir des règles qui commencent par des terminaux, et ce de manière récursive.**

**Ainsi, les règles de la forme:**

**A → Bα | Cβ**
**B → aB' | B"**
**B" → bD | D'**
**D' → eD"**
**C → cC' | dC"**

**doivent être transformées en:**

**A → aB'α | bDα | eD"α | cC'β | dC"β**

**et on élimine les règles:**

**B → aB' | B"**
**B" → bD | D'**
**D' → eD"**
**C → cC' | dC"**

# Expansion partielle

- **Par exemple, les règles suivantes:**

**<instruction> → <instr-if> <instr-for> <instr-while>**
**<affect> <proc-call>**

**<instr-if> → if <cond> then <expr> else <expr>**

**<instr-for> → for <var> := <expr> do <instr>**

**<instr-while> → while <cond> do <instruction>**

**<instr-bloc> → begin <instr-list> end**

**<instr-callproc> → <procedure>(<arg-list>) |**

**<instr-affect> → <ident> := <expr>**

# Expansion partielle

- **doivent être transformées en l'instruction supérieure.**

**<instruction> → for <var> := <expr> do <instr> |**
**if <cond> then <expr> else <expr> |**
**while <cond> do <instruction> |**
**begin <instr-list> end |**
**<procedure>(<arg-list>) |**
**<ident> := <expr>**

# Traitement des règles vides

- Les règles de type A → ε posent un problème particulier, car il faut tenir compte, lors de l'expansion partielle de ce qui suit ces non-terminaux.
- Par exemple une grammaire de la forme

A → BB' | cD

B → ε | dE

B' → mM

- se transforme en :

A → cD | dE | mM // cette dernière à cause de la règle B → ε.

# Récursion

- **Ex:**

**A → aB | CB'**

**C → bA | c**

**B → (A)**

**B' → d**

- **se transforme en**
- **A → a(A) | bA | c(A) | cd**

# Application au if

- Il est possible de transformer le "if" en une règle un peu plus complexe.
- Cette transformation n'est pas applicable dans le cas d'une analyse descendante récursive, car il n'existe pas de mécanisme d'expansion partielle et de factorisation à gauche permettant de lever l'ambiguïté.

# Application au if

<instruction> → <instr. assoc.> | <instr. non-assoc>

<instr. assoc.> → if <condit> then <instr. assoc.> else
<instr. assoc> |
<autres instr.>

<instr. non-assoc> → if <condit> then <instruction> |
if <condit> then <instr. assoc.> else
<instr. non-assoc>

# Application au if

- La technique utilisée est en général assez simple: elle revient tout simplement à traiter l'instruction if de la manière suivante:

<instr-if> → if <cond> then <expr> else <expr> |
if <cond> then <expr>

- puis par factorisation à gauche d'obtenir:

<instr-if> → if <cond> then <expr> <suite-if>

<suite-if> → else <expr> | ε

# Application au if

- puis de considérer, directement dans l'analyseur, que le <suite-if> se rattache au dernier <instr-if> rencontré.
- Cela n'est pas très propre, mais on fait de bons compilateurs avec de telles techniques.
- En fait, cela n'est pas trop grave du fait qu'on fait attention à ne traiter que des grammaires non-ambiguë sauf dans le cas du if qui est traité à part.

# Exercice

1. **Soit la grammaire G= {S→T + T, T→F * F, F = nb}, donner la dérivation de la phrase 3*4+5*7 par l'analyse descendante**

2. **Soit la grammaire non contextuelle**

**S→SS+|SS*|a**

**Et la phrase aa+a***

a) **Donner la dérivation a gauche**
b) **Donner la dérivation a droite**
c) **Donner l'arbre de syntaxe**
d) **La grammaire est-elle ambiguë? Justifiez**
e) **Decrivez le langage engendre par cette grammaire**

# L’analyse ascendante (bottom-up parsing)

- **L’analyse ascendante (ou par décalage-réduction) construit (virtuellement) l’arbre de dérivation à partir du bas ; ce qui revient à suivre à l’envers une chaîne de dérivations droites. La méthode la plus puissante utilisée est la méthode dite : méthode LR(Left scanning, Rightmost derivation)**

# Méthode LR

- LR signifie Left scanning, Rightmost derivation.
- L'analyse d'un programme se fait toujours en lisant les caractères un à un à partir du début de celui-ci (left scanning).
- Lors de cette analyse la construction de l'arbre de dérivation est envisagé à partir des feuilles.
- La dérivation est une dérivation la plus à droite (Rightmost derivation).

# Exemple :

$V_t = \{id, *, +\}$

$V_n = \{E, T, F\}$

$S = E$

$P = \{ E \rightarrow T, E \rightarrow E+T, T \rightarrow F,$
$T \rightarrow T*F, F \rightarrow id \}$

# Exemple :

- La phrase p à analyser est id + id
- Séquence de dérivation à droite
- E →E + T →E + F →E + id →T + id →F + id →id + id
- Séquence de réductions à droite correspondante est
- id + id ←F + id← T + id ←E + id ←E + F ←E + T ←E

# Exemple :

- La phrase p à analyser est id + id * id
- Une séquence de dérivations permettant d'obtenir le symbole de départ E par la méthode LR, est la suivante :
- id+id*id ← F+id*id ← t+id*id ←E+id*id ← E+F*id ← E+T*id ← E+T*F ← E+T ← E

# Exercice

1. **Soit la grammaire G= {S→T + T, T→F * F, F = nb}, donner la dérivation de la phrase 3*4+5*7 par l'analyse ascendante**