# Langages et Compilation

## Grammaires formelles

# Grammaire

*Un ensemble de règles qui donne une énumération récursive des mots du langage.*

Une **grammaire** est un quadruplet $\mathbf{G} = (\boldsymbol{\Sigma}, N, S, P)$ où :

$\boldsymbol{\Sigma}$ est un alphabet de lettres dites terminales

$N$ est un alphabet de lettres dites non terminales ou variables

$S \in N$ est la variable de départ ou axiome

$P$ est une famille finie de productions, i.e. de règles de dérivation $\alpha \to \beta$ où $\alpha$, $\beta$ sont des mots de $(\boldsymbol{\Sigma} \cup N)^*$ et $\alpha$ contient au moins une variable.

$\boldsymbol{\Sigma} = \{a, b\},\ N = \{S, T, U\},\ S,$

$$P = \left\{ \begin{array}{lcl} S & \to & TU \\ T & \to & aTb \\ T & \to & \varepsilon \\ U & \to & bUa \\ U & \to & \varepsilon \end{array} \right. \quad \text{ou de façon concise} \quad \left\{ \begin{array}{lcl} S & \to & TU \\ T & \to & aTb \mid \varepsilon \\ U & \to & bUa \mid \varepsilon \end{array} \right.$$

# Dérivation

Soit $\mathbf{G} = (\boldsymbol{\Sigma}, \boldsymbol{N}, \boldsymbol{S}, \boldsymbol{P})$ une grammaire.

$(\{a, b\}, \{S, T, U\}, S, \{S \to TU, T \to aTb \mid \varepsilon, U \to bUa \mid \varepsilon\})$

Pour toutes chaînes $\boldsymbol{u}, \boldsymbol{v} \in (\boldsymbol{\Sigma} \cup \boldsymbol{N})^*$,

> *on dit que* $\boldsymbol{u}$ **se dérive en** $\boldsymbol{v}$ *et l'on note* $\boldsymbol{u} \to \boldsymbol{v}$*,*
> *s'il existe une production* $\boldsymbol{\alpha} \to \boldsymbol{\beta} \in \boldsymbol{P}$ *telle que*
> $\boldsymbol{u} = \boldsymbol{\gamma\alpha\delta}$ *et* $\boldsymbol{v} = \boldsymbol{\gamma\beta\delta}$*.*

$aaTbbbUa \to aaaTbbbbUa \qquad T \to aTb \in P,\ \gamma = aa, \delta = bbbUa$

$aaTbbbUa \to aaTbbbbUaa \qquad U \to bUa \in P,\ \gamma = aaTbbb, \delta = a$

On note $\boldsymbol{u} \xrightarrow{\boldsymbol{k}} \boldsymbol{v}$ s'il existe des chaînes $\boldsymbol{u_1, u_2, \cdots, u_{k-1}}$ définissant une suite de $\boldsymbol{k}$ dérivations $\boldsymbol{u} \to \boldsymbol{u_1} \to \boldsymbol{u_2} \to \cdots \to \boldsymbol{u_{k-1}} \to \boldsymbol{v}$.

On note $\boldsymbol{u} \xrightarrow{*} \boldsymbol{v}$ s'il existe un $\boldsymbol{k}$ tel que $\boldsymbol{u} \xrightarrow{\boldsymbol{k}} \boldsymbol{v}$.

$TU \xrightarrow{2} \varepsilon \qquad TU \to U \to \varepsilon$

$U \xrightarrow{4} bbbaaa \qquad U \to bUa \to bbUaa \to bbbUaaa \to bbbaaa$

# Langage engendré

Pour toute chaîne $\alpha \in (\Sigma \cup V)^*$, on note par $\mathcal{L}_{\mathrm{G}}(\alpha)$ l'ensemble des chaînes terminales qui dérivent de $\alpha$ :

$$\mathcal{L}_{\mathrm{G}}(\alpha) = \{u \in \Sigma^* \colon \alpha \xrightarrow{*} u\}$$

Le **langage engendré par une grammaire** $\mathrm{G} = (\Sigma, N, S, P)$ est l'ensemble des chaînes terminales qui dérive de l'axiome $S$ :

$$L(\mathrm{G}) = \mathcal{L}_{\mathrm{G}}(S) = \{u \in \Sigma^* \colon S \xrightarrow{*} u\}$$

$$\mathrm{G} = \big(\{a, b\}, \{S, T, U\}, S, \{S \to TU, T \to aTb \mid \varepsilon, U \to bUa \mid \varepsilon\}\big)$$

$$\mathcal{L}_{\mathrm{G}}(T) = \{a^n b^n \colon n \in \mathbb{N}\}$$

$$T \to \varepsilon,\ T \to aTb \to ab,\ T \to aTb \to aaTbb \to aabb,\ \ldots$$

$$\mathcal{L}_{\mathrm{G}}(U) = \{b^n a^n \colon n \in \mathbb{N}\}$$

$$L(\mathrm{G}) = \{a^n b^{n+m} a^m \colon n, m \in \mathbb{N}\}$$

# Classification de Chomsky

On définit différents types de grammaire suivant la forme des règles.

**type 0** Toute grammaire

**type 1** Les grammaires **contextuelles**
Les règles sont de la forme $\alpha \rightarrow \beta$ avec $|\alpha| \leq |\beta|$

**type 2** Les grammaires **hors contexte** ou **algébriques**
Les règles sont de la forme $A \rightarrow \beta$ avec $A \in N$

**type 3** Les grammaires **régulières**

- les grammaires **linéaires à droite**
  Les règles sont de la forme $A \rightarrow uB$ ou $A \rightarrow u$
  avec $A, B \in N$ et $u \in \Sigma^*$
- Les grammaires **linéaires à gauche**
  Les règles sont de la forme $A \rightarrow Bu$ ou $A \rightarrow u$
  avec $A, B \in N$ et $u \in \Sigma^*$

Hiérarchie stricte

# Exemple

**Grammaire régulière qui engendre le langage régulier**

$$(a+b)^*(aa+abaaa+abab)$$

l'ensemble des terminaux $\Sigma = \{a, b\}$

l'ensemble des variables $N = \{S, T, U\}$

l'axiome $S$

les productions $P$

$$\begin{cases} S & \rightarrow & aS \mid bS \mid aT \\ T & \rightarrow & baU \mid a \\ U & \rightarrow & aa \mid b \end{cases}$$

$\mathcal{L}_{\mathrm{G}}(U)$: $aa+b$

$\mathcal{L}_{\mathrm{G}}(T)$: $ba(aa+b)+a$

$L(\mathrm{G})$: $(a+b)^*(aa+abaaa+abab)$

# Exemple

**Grammaire hors contexte qui engendre les expressions arithmétiques totalement parenthésées**

l'ensemble des terminaux $\boldsymbol{\Sigma = \{+, -, *, /, (, ), nb\}}$

l'ensemble des variables $\boldsymbol{N = \{Exp, Op\}}$

l'axiome ***Exp***

les productions ***P***

$$\begin{cases} \boldsymbol{Exp} & \rightarrow & \boldsymbol{(Exp\ Op\ Exp) \mid nb} \\ \boldsymbol{Op} & \rightarrow & \boldsymbol{+ \mid - \mid * \mid /} \end{cases}$$

Dérivation de $(nb - (nb * nb))$

$$Exp \rightarrow (Exp\ Op\ Exp) \rightarrow (nb\ Op\ Exp) \rightarrow (nb - \ Exp)$$
$$\rightarrow (nb - (nb\ Op\ Exp)) \rightarrow (nb - (nb * \ Exp)) \rightarrow (nb - (nb * nb))$$

## Exemple

**Grammaire sensible au contexte qui engendre le langage $\{a^n b^n c^n : n \in \mathbb{N}\}$**

l'ensemble des terminaux $\boldsymbol{\Sigma = \{a, b, c\}}$

l'ensemble des variables $\boldsymbol{N = \{S, T\}}$

l'axiome $\boldsymbol{S}$

les productions $\boldsymbol{P}$

$$\begin{cases} S & \rightarrow & aTc \mid \varepsilon \\ aT & \rightarrow & aaTTc \mid ab \\ bT & \rightarrow & bb \\ cT & \rightarrow & Tc \end{cases}$$

$S \rightarrow aTc \rightarrow abc$

$S \rightarrow aTc \rightarrow aaTTcc \rightarrow aabTcc \rightarrow aabbcc$

$S \rightarrow aTc \rightarrow aaTTcc \rightarrow aaaTTcTcc \rightarrow aaabTcTcc \rightarrow$
$aaabbcTcc \rightarrow aaabbTccc \rightarrow aaabbbccc$

## Classification de Chomsky

Ici, les deux types de grammaire qui nous intéressent sont :

- les grammaires régulières qui caractérisent les langages réguliers,
- les grammaires hors contexte ou algébriques qui définissent les langages algébriques.

Objectif : développer les outils algorithmiques qui, étant donnée une telle grammaire, déterminent si un mot est engendré ou non par cette grammaire.

# Grammaire régulière

Une grammaire **régulière** est

soit une grammaire **linéaire à droite** et toutes ses productions sont de la forme $\begin{cases} A \to uB \\ A \to u \end{cases}$ avec $A, B \in N$ et $u \in \Sigma^*$.

soit une grammaire **linéaire à gauche** et toutes ses productions sont de la forme $\begin{cases} A \to Bu \\ A \to u \end{cases}$

**Proposition**

Les langages engendrés par les grammaires régulières sont exactement les langages réguliers.

On hérite ainsi de tous les outils algorithmiques développés pour les AF.

# Grammaire régulière

## De grammaire régulière vers AF

$\{a, b, c\}$, $N = \{S, T\}$, $S$ l'axiome, $P = \left\{ \begin{array}{lcl} S & \rightarrow & abS \mid cT \\ T & \rightarrow & cS \mid bacT \mid c \end{array} \right.$

Comment mimer une dérivation par un calcul sur AF ?

$abcbac \in L(\mathrm{G})$ :

$$S \rightarrow abS \rightarrow abcT \rightarrow abcbacT \rightarrow abcbacc$$

$$S \xrightarrow{ab} S \xrightarrow{c} T \xrightarrow{bac} T \xrightarrow{c} \text{un état final}$$

Étape préliminaire : se ramener à une grammaire telle que les productions soient de la forme $\left\{ \begin{array}{lcl} \boldsymbol{A} & \rightarrow & \boldsymbol{aB} \\ \boldsymbol{A} & \rightarrow & \varepsilon \end{array} \right.$ où $\boldsymbol{a}$ est une simple lettre terminale.

$S \rightarrow abS \quad \leadsto \quad S \rightarrow aX,\ X \rightarrow bS$

$T \rightarrow bacT \quad \leadsto \quad T \rightarrow bY_1,\ Y_1 \rightarrow aY_2,\ Y_2 \rightarrow cT$

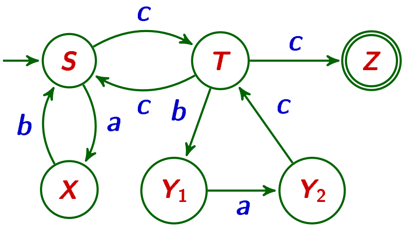
$T \rightarrow c \quad \leadsto \quad T \rightarrow cZ,\ Z \rightarrow \varepsilon$

# Grammaire régulière

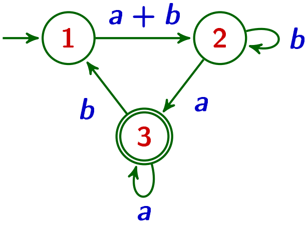
## De grammaire régulière vers AF

À partir de la grammaire $(\Sigma, N, S, P)$ dont les règles sont de la forme $A \to aB$ et $A \to \varepsilon$,
on construit l'AFN $(\Sigma, N, \delta, S, F)$ où
$\delta(A, a) = \{B : A \to aB \in P\}$ et $F = \{A : A \to \varepsilon\}$.

$$\begin{cases} S & \to & aX \mid cT \\ T & \to & cS \mid bY_1 \mid cZ \\ X & \to & bS \\ Y_1 & \to & aY_2 \\ Y_2 & \to & cT \\ Z & \to & \varepsilon \end{cases}$$



Pour tout AFD $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$, la grammaire linéaire à droite $(\Sigma, Q, q_0, P)$ où
$P = \{q \to ar : \delta(q, a) = r\} \cup \{q \to \varepsilon : q \in F\}$,
engendre $L(\mathcal{A})$.




les terminaux : $a$, $b$

les variables : $1$, $2$, $3$

l'axiome : $1$

les productions :

$$\begin{cases} 1 & \to & a\,2 \mid b\,2 \\ 2 & \to & a\,3 \mid b\,2 \\ 3 & \to & a\,3 \mid b\,1 \mid \varepsilon \end{cases}$$

# Grammaire algébrique ou hors contexte

Les règles sont de la forme $A \rightarrow \beta$ avec $A \in N$

Ces grammaires sont suffisamment expressives pour décrire la syntaxe des langages de programmation (du moins la plupart) et suffisamment simples (modulo quelques hypothèses supplémentaires) pour admettre des algorithmes efficaces qui détermine si un programme est syntaxiquement correct et en crée une représentation sous forme d'arbre

La grammaire d'un langage de programmation est classiquement représentée sous format **BNF** (Backus-Naur Form) qui permet quelques variantes sur la notation des grammaires algébriques.

# Grammaire algébrique et format BNF

## Un extrait de la grammaire de JAVA

```
Blocks and Commands
    <block> ::=  <block statements>?
    <block statements> ::= <block statement> | <block statements> <block statement>
    <block statement> ::= <local variable declaration statement> | <statement>
    <local variable declaration statement> ::= <local variable declaration> ;
    <local variable declaration> ::= <type> <variable declarators>
    <statement> ::= <statement without trailing substatement> | <labeled statement> |
        <if then statement> | <if then else statement> | <while statement> |
        <for statement>
    <statement no short if> ::= <statement without trailing substatement> |
        <labeled statement no short if> | <if then else statement no short if> |
        <while statement no short if> | <for statement no short if>
    <statement without trailing substatement> ::= <block> | <empty statement> |
        <expression statement> | <switch statement> | <do statement> | <break statement> |
        <continue statement> | <return statement> | <synchronized statement> |
        <throws statements> | <try statement>
    <empty statement> ::= ;
    <labeled statement> ::= <identifier> : <statement>
    <labeled statement no short if> ::= <identifier> : <statement no short if>
    <expression statement> ::= <statement expression> ;
    <statement expression> ::= <assignment> | <preincrement expression> |
        <postincrement expression> | <predecrement expression> |
        <postdecrement expression> | <method invocation> |
        <class instance creation expression>
    <if then statement>::= if ( <expression> ) <statement>
    <if then else statement>::= if ( <expression> ) <statement no short if> else <statement>
```

```
<if then else statement no short if> ::= if ( <expression> ) <statement no short if>
    else <statement no short if>
<switch statement> ::= switch ( <expression> ) <switch block>
<switch block> ::=  <switch block statement groups>? <switch labels>?
<switch block statement groups> ::= <switch block statement group> |
    <switch block statement groups> <switch block statement group>
<switch block statement group> ::= <switch labels> <block statements>
<switch labels> ::= <switch label> | <switch labels> <switch label>
<switch label> ::= case <constant expression> : | default :
<while statement> ::= while ( <expression> ) <statement>
<while statement no short if> ::= while ( <expression> ) <statement no short if>
<do statement> ::= do <statement> while ( <expression> ) ;
<for statement> ::= for ( <for init>? ; <expression>? ; <for update>? ) <statement>
<for statement no short if> ::= for ( <for init>? ; <expression>? ; <for update>? )
    <statement no short if>
<for init> ::= <statement expression list> | <local variable declaration>
<for update> ::= <statement expression list>
<statement expression list> ::= <statement expression> |
    <statement expression list> , <statement expression>
<break statement> ::= break <identifier>? ;
<continue statement> ::= continue <identifier>? ;
<return statement> ::= return <expression>? ;
<throws statement> ::= throw <expression> ;
<synchronized statement> ::= synchronized ( <expression> ) <block>
<try statement> ::= try <block> <catches> | try <block> <catches>? <finally>
<catches> ::= <catch clause> | <catches> <catch clause>
<catch clause> ::= catch ( <formal parameter> ) <block>
<finally > ::= finally <block>
```

# Arbre de dérivation

Soit une grammaire $\mathrm{G} = (\Sigma, N, S, P)$.

Un arbre de **dérivation** ou arbre d'**analyse** ou arbre de **syntaxe** pour $\mathbf{G}$ est un arbre étiqueté par $\Sigma \cup N \cup \{\varepsilon\}$ tel que

- la racine est étiquetée par l'axiome $S$
- les nœuds internes sont étiquetés par les variables
- les feuilles sont étiquetées par les terminaux ou $\varepsilon$
- si $A$ est l'étiquette d'un nœud interne qui a $k$ fils étiquetés $x_1$, ..., $x_k$ alors $A \to x_1 \cdots x_k$ est une règle de $\mathbf{G}$.

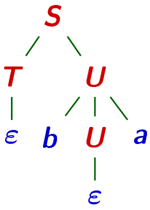
La **frontière** de l'arbre est le mot formé des feuilles de l'arbre (lues de gauche à droite).

Le langage engendré par $\mathbf{G}$ est l'ensemble des mots $u \in \Sigma^*$ tels qu'il existe un arbre de dérivation de frontière $u$.

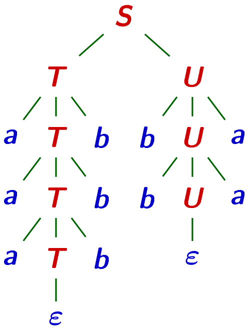
# Arbre de dérivation

Des arbre de dérivation de la grammaire

$$(\{a, b\}, \{S, T, U\}, S, \{S \to TU, T \to aTb \mid \varepsilon, U \to bUa \mid \varepsilon\})$$



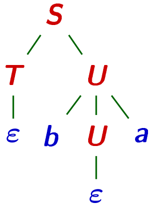

frontière : $\boldsymbol{ba}$




frontière : $\boldsymbol{a^3 b^5 a^2}$

# Arbre de dérivation

Chaque arbre de dérivation peut synthétiser plusieurs dérivations.




$S \to TU \to U \to bUa \to ba$

$S \to TU \to TbUa \to bUa \to ba$

$S \to TU \to TbUa \to Tba \to ba$

Une dérivation est dite **gauche** si c'est la variable la plus à gauche qui est dérivée à chaque étape.

Elle est **droite** si c'est toujours la variable la plus à droite qui est dérivée.

$S \to TU \to U \to bUa \to ba$ est une dérivation gauche

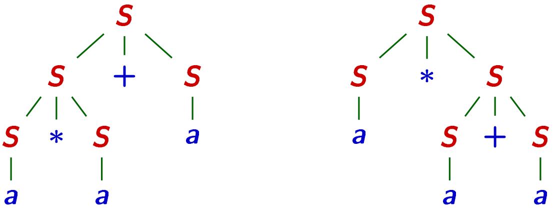
$S \to TU \to TbUa \to Tba \to ba$ est une dérivation droite

À un arbre d'analyse correspond exactement une dérivation gauche et une dérivation droite.

# Notion d'ambiguïté

Une grammaire $\mathbf{G}$ est **ambiguë** s'il existe un mot $w \in L(\mathbf{G})$ qui admet plusieurs arbres de dérivation.

$\mathrm{G} = (\{a, +, *\}, \{S\}, S, \{S \to S + S \mid S * S \mid a\})$ est ambiguë :
le mot $a * a + a$ admet deux arbres d'analyse.



le mot $a * a + a$ admet deux dérivations gauches :

$$S \to S + S \to S * S + S \to a * S + S \to a * a + S \to a * a + a$$
$$S \to S * S \to a * S \to a * S + S \to a * a + S \to a * a + a$$

et deux dérivations droites :

$$S \to S + S \to S + a \to S * S + a \to S * a + a \to a * a + a$$
$$S \to S * S \to S * S + S \to S * S + a \to S * a + a \to a * a + a$$

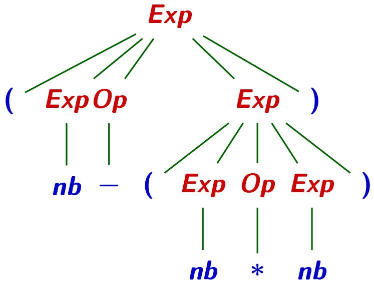
# Notion d'ambiguïté

Tout mot d'une grammaire non ambiguë admet exactement

- un arbre d'analyse,
- une dérivation gauche,
- une dérivation droite.

$(\{+, -, *, \backslash, (, ), nb\}, \{Exp, Op\}, Exp, Exp \rightarrow (Exp\ Op\ Exp) \mid nb,$
$Op \rightarrow + \mid - \mid * \mid \backslash\})$ qui engendre les expressions totalement parenthésées, n'est pas ambiguë.

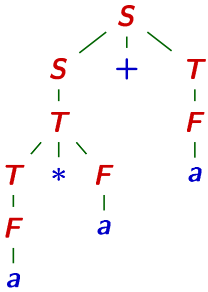
À partir de l'expression
$(nb - (nb * nb))$
on peut reconstruire
l'unique arbre associé



# Notion d'ambiguïté

$\mathrm{G}' = (\{a, +, *\}, \{S, T, F\}, S, \{S \to S + T \mid T,\ T \to T * F \mid F,$
$F \to a\})$ engendre le même langage que la grammaire ambiguë
$\mathrm{G} = (\{a, +, *\}, \{S\}, S, \{S \to S + S \mid S * S \mid a\})$,
mais n'est pas ambiguë.

$a * a + a$ définit un unique arbre de dérivation




L'unique dérivation gauche
$S \to S + T \to T + T \to T * F + T \to F * F + T \to a * F + T \to a * a + T \to a * a + F \to a * a + a$

# Notion d'ambiguïté

Un langage est **inhéremment ambigu** lorsque toute grammaire qui l'engendre est ambiguë.

$L = \{a^n b^m c^p : n = m \text{ ou } m = p\}$ est inhéremment ambigu.
Quelque soit la grammaire qui engendre $L$, on peut montrer que le mot $a^n b^n c^n$ pour $n$ assez grand admet toujours deux arbres de dérivation.

Le langage engendré par les expressions arithmétiques sur l'alphabet $\{+, *, a\}$ n'est pas inhéremment ambigu.

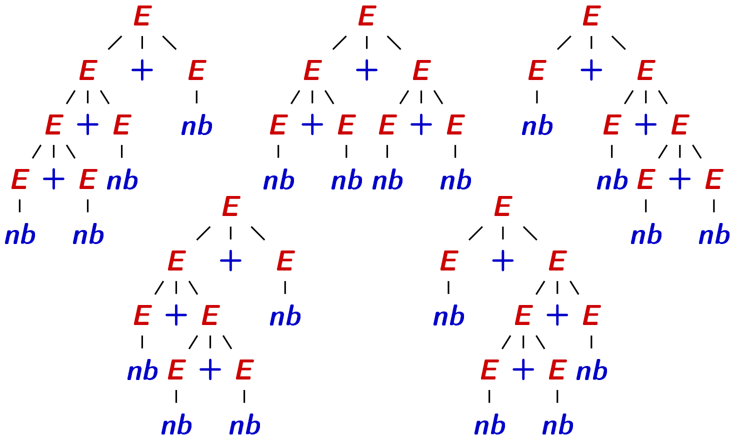
$\Rightarrow$ Modifier une grammaire pour la rendre non ambiguë n'est pas toujours possible.

# Lever certaines ambiguïtés

## Ambiguïté due à l'associativité

$\left(\{+, (, ), nb\}, \{E\}, E, \{E \to E + E \mid (E) \mid nb\}\right)$ est ambiguë.

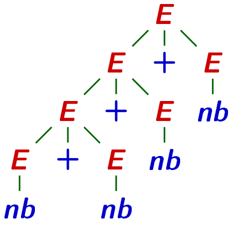
$nb + nb + nb + nb$ admet plusieurs arbres de dérivation

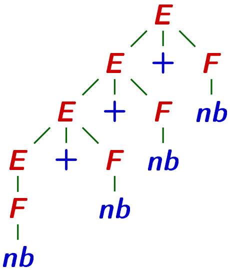


# Lever certaines ambiguïtés

## Ambiguïté due à l'associativité

Usuellement, on effectue les additions de la gauche vers la droite (associativité gauche).
Ceci correspond au premier arbre.




Pour forcer l'associativité à gauche, on introduit une nouvelle variable.

$$\begin{cases} E & \rightarrow \quad E + F \mid F \\ F & \rightarrow \quad (E) \mid nb \end{cases}$$

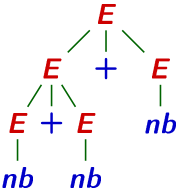
La grammaire modifiée n'est plus ambiguë.

# Lever certaines ambiguïtés

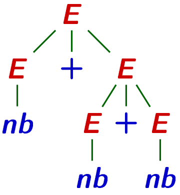
## Ambiguïté due à l'associativité et aux priorités des opérations

$\big(\{+, *, (, ), nb\}, \{E\}, E, \{E \to E + E \mid E * E \mid (E) \mid nb\}\big)$ est ambiguë.

Ambiguïté liée à l'associativité : $nb + nb + nb$ admet deux arbres de dérivation

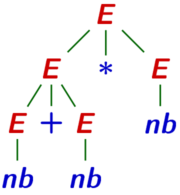



$\{nb + nb\} + nb$

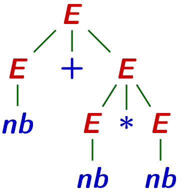



$nb + \{nb + nb\}$

Ambiguïté sur la priorité des opérations : $nb + nb * nb$ admet deux arbres de dérivation




$\{nb + nb\} * nb$



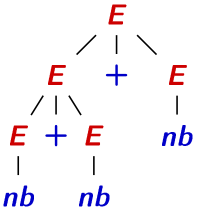

$nb + \{nb * nb\}$

# Lever certaines ambiguïtés

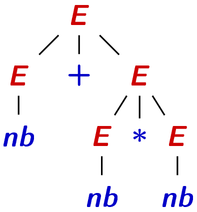
## Ambiguïté due à l'associativité et aux priorités des opérations

Pour supprimer les ambiguïtés, on choisit suivant les règles usuelles de privilégier

- l'associativité à gauche

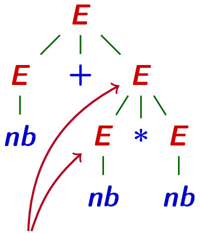



- la priorité de la multiplication sur l'addition



# Lever certaines ambiguïtés

## Ambiguïté due à l'associativité et aux priorités des opérations



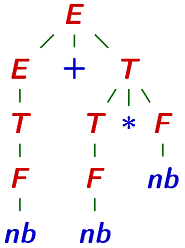

Empêcher les expressions non parenthésées avec $+$ ou $*$

$\Rightarrow$ Ajouter deux nouvelles variables $T$ et $F$

Empêcher les expressions non parenthésées avec $+$

$$\begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid nb \end{cases}$$

La grammaire modifiée n'est plus ambiguë.



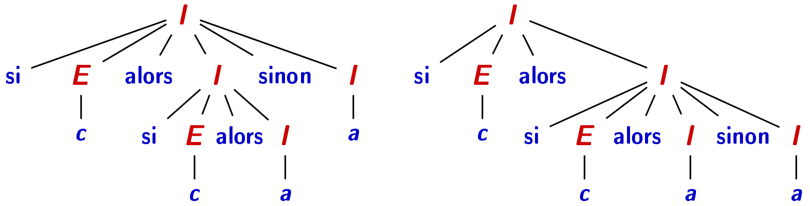
# Lever certaines ambiguïtés

## Ambiguïté du “ si alors sinon ”

La grammaire des instructions de branchements conditionnels

$$\begin{cases} I & \rightarrow & si\ E\ alors\ I\ sinon\ I \ \mid \ si\ E\ alors\ I \mid a \\ E & \rightarrow & c \end{cases}$$ est ambiguë.

`si c alors si c alors a sinon a` admet deux arbres d'analyse




Usuellement, le sinon est associé au si précédent le plus proche.
Ce qui correspond au deuxième arbre.

On modifie la grammaire de sorte à n'avoir que des instructions qui ont autant de **si** que de **sinon**, entre tout **alors** et **sinon**.

# Lever certaines ambiguïtés

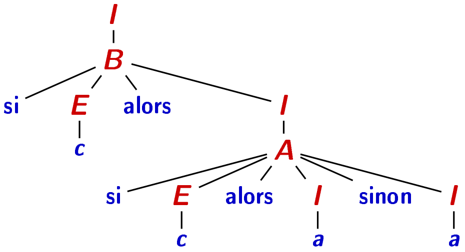
## Ambiguïté du “ si alors sinon ”

$$\left\{\begin{array}{lcl} I & \rightarrow & A \mid B \\ A & \rightarrow & si\ E\ alors\ A\ sinon\ A \mid a \\ B & \rightarrow & si\ E\ alors\ I \mid si\ E\ alors\ A\ sinon\ B \\ E & \rightarrow & c \end{array}\right.$$

**A** : instructions qui ont autant de **si** que de **sinon**.

**B** : instructions qui ont plus de **si** que de **sinon**.

Entre tout **alors** et **sinon**, des instructions **A** uniquement.



# Grammaire réduite

*Éliminer les variables inutiles*

Une grammaire $\mathbf{G} = (\boldsymbol{\Sigma}, N, S, P)$ est dite **réduite** si

- toute variable est **productive** :
  chaque variable $A$ engendre au moins un mot terminal,
  $$L_{\mathrm{G}}(A) \neq \emptyset$$
- toute variable est **accessible** à partir de l'axiome :
  pour chaque variable $A$, il existe $u, v \in (N \cup \Sigma)^*$ tels que
  $$S \xrightarrow{*} uAv$$

Chaque variable d'une grammaire réduite apparaît dans au moins un arbre de dérivation.

**Fait**

Pour toute grammaire **G**, on peut construire de manière effective une grammaire réduite équivalente.

# Grammaire réduite

L'algorithme se fait en deux étapes.

**Étape 1. Suppression des variables improductives**

On construit par récurrence sur $i$, l'ensemble $\mathbf{Prod}_i$ des variables $A$ qui sont racines d'un arbre de dérivation de hauteur $i$.

$$\mathbf{Prod}_1 = \{A \in N : A \to u \in P \text{ et } u \in \Sigma^*\}$$

$$\mathbf{Prod}_{i+1} = \{A \in N : A \to \alpha \in P \text{ et } \alpha \in (\mathbf{Prod}_i \cup \Sigma)^*\}$$

Arrêt quand $\mathbf{Prod}_{i+1} = \mathbf{Prod}_i$ (au bout d'au plus $|N|$ étapes)

$$\mathrm{G} = \left(\{a,b\}, \{S,T,U,V\}, S, \begin{cases} S & \to & aT \mid bTU \mid abTS \mid UV \\ T & \to & aU \mid bT \mid a \\ U & \to & aU \mid bU \\ V & \to & aT \mid bS \mid a \end{cases}\right)$$

$Prod_1 = \{T, V\}$ , $Prod_2 = \{T, V, S\}$ , $Prod_3 = Prod_2$

$\mathrm{G}_1 = \left(\{a,b\}, \{S,T,V\}, S, \{S \to aT \mid abTS,\ T \to bT \mid a,\ V \to aT \mid bS \mid a\}\right)$

NB. On a un algo qui détermine si $L(G) = \emptyset$.

# Grammaire réduite

## Étape 2. Suppression des variables inaccessibles

On construit par récurrence sur $i$, l'ensemble $\mathbf{Acc}_i$ des variables $A$ qui sont nœuds d'un arbre de dérivation de racine $S$ et de hauteur $i$.

$$\mathbf{Acc}_0 = \{S\}$$

$$\mathbf{Acc}_{i+1} = \{A \in N\colon B \to \alpha A \beta \in P \text{ et } B \in \mathbf{Acc}_i\}$$

Arrêt lorsque $\mathbf{Acc}_{i+1} = \mathbf{Acc}_i$ (au bout d'au plus $|N|$ étapes)

$$\mathrm{G}_1 = \left(\{a, b\}, \{S, T, V\}, S, \left\{\begin{array}{lcl} S & \to & aT \mid abTS \\ T & \to & bT \mid a \\ V & \to & aT \mid bS \mid a \end{array}\right.\right)$$

$Acc_0 = \{S\}$ , $Acc_1 = \{S, T\}$ , $Acc_2 = Acc_1$.

$\leadsto \big(\{a, b\}, \{S, V\}, S, \{S \to aT \mid abTS,\ T \to bT \mid a\}\big)$

L'étape 1 doit être effectuée avant l'étape 2. La suppression de variables improductives peut introduire de nouvelles variables inaccessibles.