

Langages et Compilation

Grammaires formelles

Grammaire

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

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

Σ 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
 $\alpha \rightarrow \beta$ où α, β sont des mots de $(\Sigma \cup N)^*$ et α contient au moins une variable.

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

$$P = \begin{cases} S & \rightarrow TU \\ T & \rightarrow aTb \\ T & \rightarrow \epsilon \\ U & \rightarrow bUa \\ U & \rightarrow \epsilon \end{cases} \quad \text{ou de façon concise} \quad \begin{cases} S & \rightarrow TU \\ T & \rightarrow aTb \mid \epsilon \\ U & \rightarrow bUa \mid \epsilon \end{cases}$$

Dérivation

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

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

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

on dit que u se dérive en v et l'on note $u \rightarrow v$,

s'il existe une production $\alpha \rightarrow \beta \in P$ telle que

$u = \gamma\alpha\delta$ et $v = \gamma\beta\delta$.

$aaTbbbUa \rightarrow aaaTbbbbUa \quad T \rightarrow aTb \in P, \gamma = aa, \delta = bbbUa$

$aaTbbbUa \rightarrow aaTbbbbUaa \quad U \rightarrow bUa \in P, \gamma = aaTbbb, \delta = a$

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

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

$TU \xrightarrow{2} \varepsilon \quad TU \rightarrow U \rightarrow \varepsilon$

$U \xrightarrow{4} bbbaaa \quad U \rightarrow bUa \rightarrow bbUaa \rightarrow bbbUaaa \rightarrow bbbaaa$

Langage engendré

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

$$\mathcal{L}_G(\alpha) = \{\mathbf{u} \in \Sigma^* : \alpha \xrightarrow{*} \mathbf{u}\}$$

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

$$L(G) = \mathcal{L}_G(\mathbf{S}) = \{\mathbf{u} \in \Sigma^* : \mathbf{S} \xrightarrow{*} \mathbf{u}\}$$

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

$$\mathcal{L}_G(T) = \{a^n b^n : n \in \mathbb{N}\}$$

$$T \rightarrow \varepsilon, T \rightarrow aTb \rightarrow ab, T \rightarrow aTb \rightarrow aaTbb \rightarrow aabb, \dots$$

$$\mathcal{L}_G(U) = \{b^n a^n : n \in \mathbb{N}\}$$

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

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

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

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

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

l'axiome S

les productions P

$$\begin{cases} S \rightarrow aS \mid bS \mid aT \\ T \rightarrow bU \mid a \\ U \rightarrow ab \mid b \end{cases}$$

$$\mathcal{L}_G(U): ab + b$$

$$\mathcal{L}_G(T): b(ab + b) + a$$

$$L(G): (a + b)^*(aa + abab + abb)$$

Exemple

Grammaire hors contexte qui engendre les expressions arithmétiques totalement parenthésées

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

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

l'axiome Exp

les productions P

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

Dérivation de $(nb - (nb * nb))$

$$\begin{aligned} Exp &\rightarrow (Exp Op Exp) \rightarrow (nb Op Exp) \rightarrow (nb - Exp) \\ &\rightarrow (nb - (nb Op Exp)) \rightarrow (nb - (nb * Exp)) \rightarrow (nb - (nb * nb)) \end{aligned}$$

Exemple

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

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

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

l'axiome S

les productions P

$$\left\{ \begin{array}{ll} S & \rightarrow aTc \mid \varepsilon \\ aT & \rightarrow aaTTc \mid ab \\ bT & \rightarrow bb \\ cT & \rightarrow Tc \end{array} \right.$$

$S \rightarrow aTc \rightarrow abc$

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

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

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 grammaire hors contexte, 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 $\begin{cases} A \rightarrow uB \\ A \rightarrow u \end{cases}$ avec $A, B \in N$ et $u \in \Sigma^*$.

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

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

$$\{a, b, c\}, N = \{S, T\}, \text{ l'axiome } S, P = \begin{cases} S \rightarrow abS \mid cT \\ T \rightarrow cS \mid bacT \mid c \end{cases}$$

Comment mimer une dérivation par un calcul sur AF ?

$abcbac \in L(G)$:

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

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

Première étape : normaliser la grammaire

on transforme la grammaire initiale en une grammaire équivalente dont les productions sont de la forme

$$\begin{cases} A \rightarrow aB \\ A \rightarrow \epsilon \end{cases} \quad \text{où } a \text{ est une simple lettre terminale.}$$

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

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

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

Grammaire régulière

De grammaire régulière vers AF

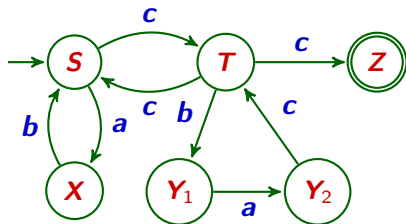
Deuxième étape À partir de la grammaire normalisée (Σ, N, S, P) qui a des productions de la forme $A \rightarrow aB$ et $A \rightarrow \epsilon$,

on construit l'AFN $(\Sigma, N, \delta, S, F)$ où

$$\delta(A, a) = \{B : A \rightarrow aB \in P\}$$

$$F = \{A : A \rightarrow \epsilon \in P\}.$$

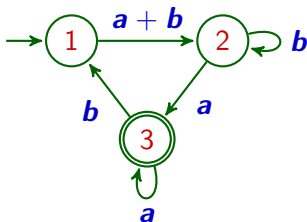
$$\left\{ \begin{array}{lcl} S & \rightarrow & aX \mid cT \\ T & \rightarrow & cS \mid bY_1 \mid cZ \\ X & \rightarrow & bS \\ Y_1 & \rightarrow & aY_2 \\ Y_2 & \rightarrow & cT \\ Z & \rightarrow & \epsilon \end{array} \right.$$



Grammaire régulière

D'AF vers grammaire régulière

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



les terminaux : a, b

les variables : $1, 2, 3$

l'axiome : 1

les productions :

$$\left\{ \begin{array}{l} 1 \rightarrow a2 \mid b2 \\ 2 \rightarrow a3 \mid b2 \\ 3 \rightarrow a3 \mid b1 \mid \varepsilon \end{array} \right.$$

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 déterminent 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 $G = (\Sigma, N, S, P)$.

Un arbre de **dérivation** ou arbre d'**analyse** ou arbre de **syntaxe** pour G est un arbre étiqueté par $\Sigma \cup N \cup \{\epsilon\}$ 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 ϵ

- si A est l'étiquette d'un nœud interne qui a k fils étiquetés x_1, \dots, x_k alors $A \rightarrow x_1 \cdots x_k$ est une règle de G .

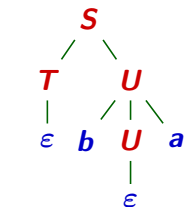
La **frontière** de l'arbre est le mot formé des feuilles de l'arbre (lues de gauche à droite).

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

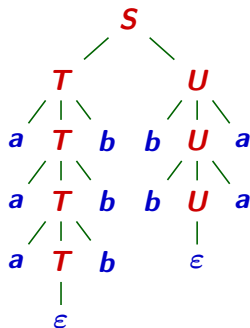
Arbre de dérivation

Des arbre de dérivation de la grammaire

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



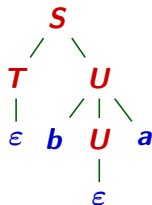
frontière : **ba**



frontière : **a³b⁵a²**

Arbre de dérivation

Chaque arbre de dérivation peut synthétiser plusieurs dérivations.



$S \rightarrow TU \rightarrow U \rightarrow bUa \rightarrow ba$

$S \rightarrow TU \rightarrow TbUa \rightarrow bUa \rightarrow ba$

$S \rightarrow TU \rightarrow TbUa \rightarrow Tba \rightarrow ba$

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.

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

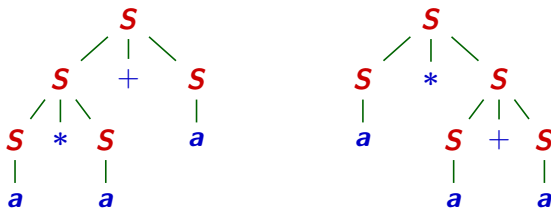
$S \rightarrow TU \rightarrow TbUa \rightarrow Tba \rightarrow ba$ est une dérivation droite

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

Notion d'ambiguïté

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

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



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

$$\begin{aligned} S &\rightarrow S + S \rightarrow S * S + S \rightarrow a * S + S \rightarrow a * a + S \rightarrow a * a + a \\ S &\rightarrow S * S \rightarrow a * S \rightarrow a * S + S \rightarrow a * a + S \rightarrow a * a + a \end{aligned}$$

et deux dérivations droites :

$$\begin{aligned} S &\rightarrow S + S \rightarrow S + a \rightarrow S * S + a \rightarrow S * a + a \rightarrow a * a + a \\ S &\rightarrow S * S \rightarrow S * S + S \rightarrow S * S + a \rightarrow S * a + a \rightarrow a * a + a \end{aligned}$$

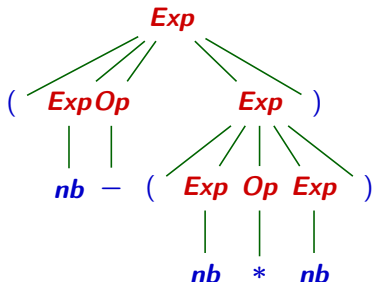
Notion d'ambiguïté

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.

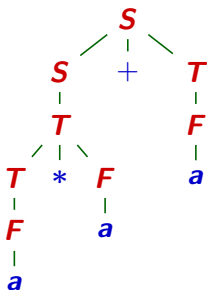
À partir de l'expression
 $(nb - (nb * nb))$
on peut reconstruire
l'unique arbre associé



Notion d'ambiguïté

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

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



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

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.

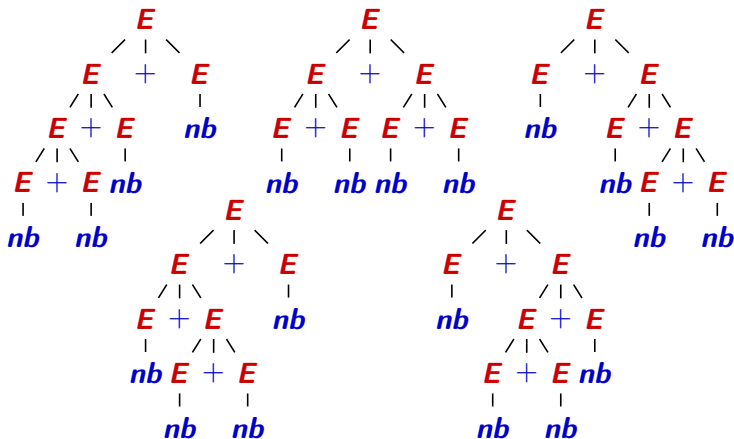
⇒ Modifier une grammaire pour la rendre non ambiguë n'est pas toujours possible.

Lever certaines ambiguïtés

Ambiguïté due à l'associativité

$(\{+, (,), nb\}, \{E\}, E, \{E \rightarrow E + E \mid (E) \mid nb\})$ est ambiguë.

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

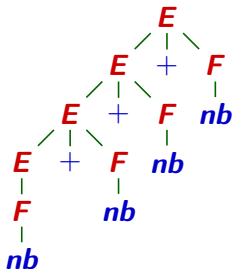
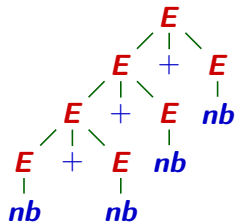


Lever certaines ambiguïtés

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.

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

La grammaire modifiée n'est plus ambiguë.

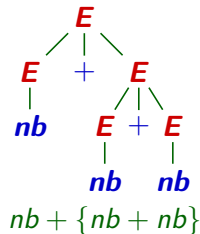
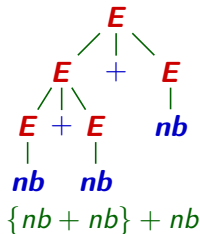
Lever certaines ambiguïtés

Ambiguïté due à l'associativité et aux priorités des opérations

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

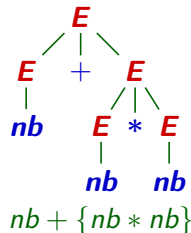
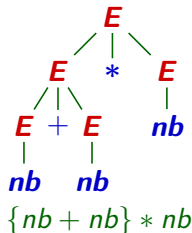
Ambiguïté liée à
l'associativité :

$nb + nb + nb$ admet
deux arbres de dérivation



Ambiguïté sur la priorité
des opérations :

$nb + nb * nb$ admet
deux arbres de dérivation

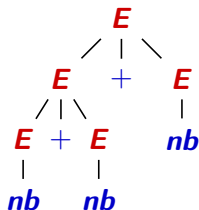


Lever certaines ambiguïtés

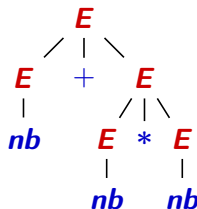
Ambiguïté due à l'associativité et aux priorités des opérations

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

- l'associativité à gauche

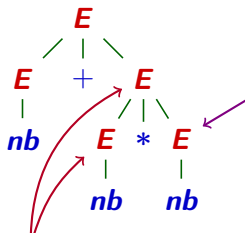


- la priorité de la multiplication sur l'addition



Lever certaines ambiguïtés

Ambiguïté due à l'associativité et aux priorités des opérations



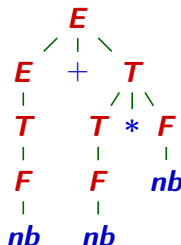
Empêcher les expressions non parenthésées avec $+$ ou $*$

\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.



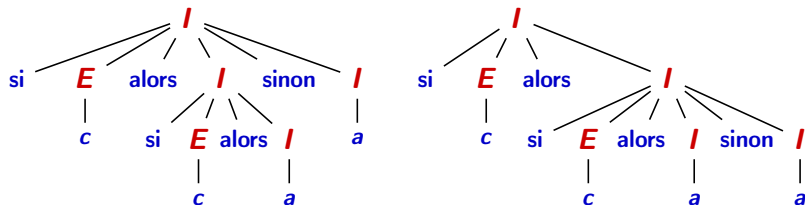
Lever certaines ambiguïtés

Ambiguïté du " si alors sinon "

La grammaire des instructions de branchements conditionnels

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

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

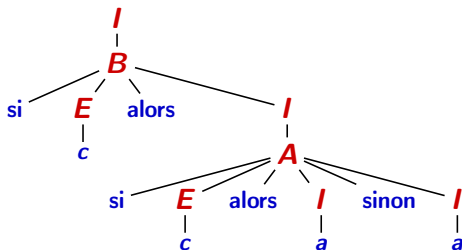
Ambiguïté du “ si alors sinon ”

$$\begin{cases} I & \rightarrow A \mid B \\ A & \rightarrow \text{si } E \text{ alors } A \text{ sinon } A \mid a \\ B & \rightarrow \text{si } E \text{ alors } I \mid \text{si } E \text{ alors } A \text{ sinon } B \\ E & \rightarrow c \end{cases}$$

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 $G = (\Sigma, N, S, P)$ est dite **réduite** si

- toute variable est **productive** :
chaque variable A engendre au moins un mot terminal,
 $L_G(A) \neq \emptyset$
- 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 \mathbf{A} qui sont racines d'un arbre de dérivation de hauteur i .

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

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

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

$$G = (\{a, b\}, \{S, T, U, V\}, S, \left\{ \begin{array}{l} S \rightarrow aT \mid bTU \mid abTS \mid UV \\ T \rightarrow aU \mid bT \mid a \\ U \rightarrow aU \mid bU \\ V \rightarrow aT \mid bS \mid a \end{array} \right. \right)$$

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

$$G_1 = (\{a, b\}, \{S, T, V\}, S, \{S \rightarrow aT \mid abTS, T \rightarrow bT \mid a, V \rightarrow aT \mid bS \mid a\})$$

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 \mathbf{A} qui sont nœuds d'un arbre de dérivation de racine \mathbf{S} et de hauteur i .

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

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

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

$$G_1 = (\{a, b\}, \{S, T, V\}, S, \left\{ \begin{array}{l} S \rightarrow aT \mid abTS \\ T \rightarrow bT \mid a \\ V \rightarrow aT \mid bS \mid a \end{array} \right\})$$

$$\mathbf{Acc}_0 = \{S\}, \mathbf{Acc}_1 = \{S, T\}, \mathbf{Acc}_2 = \mathbf{Acc}_1.$$

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

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