Chapitre 1 : Les grammaires formelles (Semaines 2 & 3)

Etienne Kouokam

Département d'Informatique
Université de Yaoundé I, Cameroun

Année académique 2019-2020
Univ de Ydé I : Mars-Juin 2020

Plan

Plan

Une grammaire c'est quoi ?

4 composantes : G = (V_n, V_t, S, P). On pose $\Sigma = V_n \cup V_t$ où :

1. $V_t =$ Alphabet des symboles terminaux : Les éléments du langage (variable, identificateur, ...). Ils sont notés en minuscules. a, b, c ...
2. $V_n =$ Symboles non-terminaux : symboles auxiliaires dénotant les types de construction (boucle, expression booléenne, ...). Ils sont notés en majuscules. A, B, C ...
3. S = Le but (symbole de départ) appelé axiome : dénote n'importe quelle phrase.
4. P = Productions : Les règles de réécriture utilisées pour reconnaître et générer des phrases. Elles sont de la forme
   - $\alpha \rightarrow \beta$
     avec $\alpha \in \Sigma^* V_n \Sigma^*$ et $\beta \in \Sigma^*$
   - (α, β), où $\alpha \in \Sigma^+$ et $\beta \in \Sigma^*$
5. Quelques notions de décidabilité

Définition

4 composantes : G = (V_n, V_t, S, P). On pose $\Sigma = V_n \cup V_t$ où :

Dérivation

- Les ensembles de symboles V_t et V_n doivent être disjoints
- Seuls les symboles terminaux formes les mots du langage
- Les symboles non-terminaux sont là juste pour aider à générer le langage
- La génération du langage commence toujours par le symbole de départ S

Dérivation

Souvent, les lettres grecques sont utilisées pour désigner les chaînes construites d'éléments terminaux ou non terminaux comme dans la production précédente.

Dérivation & réécriture

Dérivation

La grammaire G permet de dériver v de u en une étape (notation $u \Rightarrow v$ si et seulement si :

- $u = xu'y$ (u peut être décomposé en trois parties x, u' et y ; les parties x et y peuvent éventuellement être vides),
- $v = xv'y$ (v peut être décomposé en trois parties x, v' et y),
- $u' \to v'$ (la règle (u', v') est dans P).

$\to^+$ désigne la fermeture transitive de $\to$

Une grammaire G défini un langage L(G) sur l'alphabet Σ dont les éléments sont les mots engendrés par dérivation à partir du symbole de départ S.

$$L(G) = \{w \in \Sigma^* S \to^+ w\}$$

Hiérarchie de Chomsky

Classe et complexité des langages

On détermine la classe et la complexité des langages (et des grammaires) en fonction d'un certain nombre de contraintes sur la forme des règles de production. 4 ou 5 types de grammaire.

Types de grammaire

- Type 0
- Type 1
- Type 2
- Type 3
- (Type 4)

Hiérarchie de Chomsky

Grammaire de type 0 ou grammaire générale

Les règles ne sont sujettes à aucune restriction. Il suffit que chaque règle fasse intervenir (au moins) un non terminal à gauche.

Exemple / Contre exemple

Exemple	$aAbb \rightarrow ba$ ou $aAbB \rightarrow \epsilon$
Contre-exemple	$ab \rightarrow ba$ ou $\epsilon \rightarrow aa$

- Elle correspond aux langages récursivement énumérables
- Le problème de l'analyse pour de tels langages est indécidable. Un langage est décidable si pour toute phrase on peut savoir en temps fini si elle est du langage ou pas
- Les langages correspondants sont reconnus par des machines de Turing.

Hiérarchie de Chomsky

Exemple 1 : Grammaire de type 0 ou grammaire générale

Une grammaire qui engendre tous les mots qui contiennent un nombre égal de a, b et c.

$$G_1 : \left\{ \begin{array}{lllllllll} S & \rightarrow & SABC & AC & \rightarrow & CA & A & \rightarrow & a \\ S & \rightarrow & \epsilon & CA & \rightarrow & AC & B & \rightarrow & b \\ AB & \rightarrow & BA & BC & \rightarrow & CB & C & \rightarrow & c \\ BA & \rightarrow & AB & CB & \rightarrow & BC & & & \end{array} \right.$$

Cette grammaire fonctionne en produisant des chaînes de la forme $(ABC)^n$ puis en permutant les non terminaux, et enfin en produisant les chaînes terminales.

Hiérarchie de Chomsky

Exemple 2 : Les mots jumeaux

Langage vu plus pour les machines de Turing.

$$G_2 : \left\{ \begin{array}{lllllllll} S & \rightarrow & \$S'\$ & Aa & \rightarrow & aA & \$a & \rightarrow & a\$ \\ S' & \rightarrow & aAS'\$ & Ab & \rightarrow & bA & \$b & \rightarrow & b\$ \\ S' & \rightarrow & bBS'\$ & Ba & \rightarrow & aB & A\$ & \rightarrow & \$a \\ S' & \rightarrow & \epsilon & Bb & \rightarrow & bB & B\$ & \rightarrow & \$b \\ \$\$ & \rightarrow & \# & & & & & & \end{array} \right.$$

Hiérarchie de Chomsky

Grammaire de type 1 ou grammaire sensible au contexte (context-sensitive) ou monotone

Les règles sont de la forme $\alpha \rightarrow \beta$ avec $|\alpha| \leq |\beta|$. On dit alors que le langage engendré est propre. Exceptionnellement, afin d'engendrer le mot vide, on introduit $S \rightarrow \epsilon$ pour autant que S n'apparaisse pas dans le membre de droite d'une production.

Définition alternative des grammaires de type 1

Les règles sont de la forme $\alpha A\gamma \rightarrow \alpha\beta\gamma$ avec $\beta \in (V_t \cup V_n)$
Ou bien de la forme $S \rightarrow \epsilon$ et S n'apparaît dans aucun membre de droite. On préfère souvent $S' \rightarrow S + \epsilon$ (langage propre).

- Autrement dit, toute règle comprend un non-terminal entouré de deux mots qui décrivent le contexte dans lequel la variable peut être remplacée.

Définition alternative des grammaires de type 1

- Grammaires dites contextuelles car le remplacement d'un élément non-terminal peut dépendre des éléments autour de lui : son contexte.
- Les langages contextuels qui en résultent sont exactement ceux reconnus par une machine de Turing non déterministe à mémoire linéairement bornée, appelés couramment automates linéairement bornés.

Exemple de grammaire de type 1

Grammaire contextuelle pour le langage $L(G_3) = \{a^{2^n} | n > 0\}$ Langage vu plus pour les machines de Turing.

$$G_3 : \begin{cases} S & \rightarrow & DT & T & \rightarrow & XT & Xaa & \rightarrow & aaXa \\ S & \rightarrow & AA & T & \rightarrow & AF & XaF & \rightarrow & aaF \\ Daaa & \rightarrow & aaDaa & DaaF & \rightarrow & aaaa & & & \end{cases}$$

Hiérarchie de Chomsky

Grammaire de type 2 : Grammaire hors-contexte (context-free) ou algébrique

Règles de la forme $A \rightarrow \beta$ où $A \in V_n$ et pas de restriction sur β i.e $\beta \in \Sigma^*$

- Elles sont particulièrement étudiées
- Les langages algébriques correspondants sont reconnus par des automates à pile non-déterministes.
- L'analyse de ces langages est polynomiale.

Exemple

$$P = \{S \rightarrow aSb, S \rightarrow ab\}$$

$$\Longrightarrow = \{S \rightarrow a^n b^n | n > 0\}$$

Hiérarchie de Chomsky

Grammaire de type 3 : Grammaire régulière

Les règles peuvent prendre deux formes :

- linéaire à gauche : $\alpha \rightarrow \beta$ où $\alpha \in V_n$ et $\beta \in V_n V_t^*$.
  i-e $A \rightarrow Bw$ ou $A \rightarrow w$ avec $A, B \in V_n$ et $w \in V_t^*$
- linéaire à droite : $\alpha \rightarrow \beta$ où $\alpha \in V_n$ et $\beta \in V_t^* V_n$.
  i-e $A \rightarrow wB$ ou $A \rightarrow w$ avec $A, B \in V_n$ et $w \in V_t^*$

- Les langages réguliers correspondants sont construits et reconnus par des automates finis
- L'analyse de ces langages est polynomiale.

Exemple

$$P = \{A \rightarrow aB, A \rightarrow a, B \rightarrow b\}$$

Hiérarchie de Chomsky

Grammaire de type 4 : Grammaire à choix finis

Les parties droites de toutes les règles sont des terminaux. Les règles sont de la forme $X \rightarrow \alpha$ où $X \in V_n$ et $\alpha \in V_t^*$.

- Une telle grammaire ne fait qu'énumérer les phrases de son langage sur V_t.

Hiérarchie de Chomsky

Théorème : Chomsky

- Les langages réguliers (type 3) sont strictement contenus dans les langages algébriques (type 2).
- Les langages algébriques propres (type 2) sont strictement contenus dans les langages contextuels (type 1).
- les langages contextuels (type 1) sont strictement contenus dans les langages récursifs.
- les langages récursifs sont strictement contenus dans les langages récursivement énumérables (type 0).

Théorème : Chomsky

Au final, on a :

$$Types4 \subset Type3 \subset Type2 \subset Type1 \subset Type0$$

Syntaxe spécifiée par des règles de grammaire (algébrique)

Les grammaires

- Terminaux (symboles minuscules) : alphabet noté A ou V_t
- Variables (symboles majuscules) : alphabet X ou V_n
- Mots de $V_t \cup V_n$: lettres de l'alphabet grec.
- Règles de grammaire $x \rightarrow w$ où $x \in X$ et $w \in (A \cup X)$. w est donc un mot quelconque, même vide
- Axiome : souvent S (mais peut changer)

Exemples

- instr $\rightarrow$ **si** expr **alors** instr **sinon** instr
- phrase $\rightarrow$ sujet verbe complément
- $P = \left\{ \begin{array}{lll} S & \rightarrow & aSb \\ S & \rightarrow & ab \end{array} \right.$

Langage engendrés

Langage engendré = mots terminaux dérivant de l'axiome.

Opération dans les grammaires

L'unique opération autorisée, dans les grammaires, est la réécriture d'une séquence de symboles par application d'une production :

Dérivation immédiate

On définit la relation $\Rightarrow_G$ (lire : dérive immédiatement) sur l'ensemble $(V_n \cup V_t)^* \times (V_n \cup V_t)^*$ par $\gamma\alpha\delta \Rightarrow_G \gamma\beta\delta$ si et seulement si $\alpha \rightarrow \beta$ est une production de G.

La notion de dérivation immédiate se généralise à la dérivation en un nombre quelconque d'étapes.

Dérivation

On définit la relation $\Rightarrow_G^*$ sur l'ensemble $(V_n \cup V_t)^* \times (V_n \cup V_t)^*$ par $\alpha_1 \Rightarrow_G^* \alpha_n$ si et seulement si $\exists \alpha_2, \alpha_3, \ldots, \alpha_{n-1}$ dans $(V_n \cup V_t)^*$ tels que $\alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \ldots \Rightarrow_G \alpha_n$

Opération dans les grammaires

Exemple :

Partant de la grammaire G_1 définie par $P = \left\{ \begin{array}{lcl} S & \rightarrow & aSb \\ S & \rightarrow & ab \end{array} \right.$

$$S \Rightarrow_{G_1} aSb \Rightarrow_{G_1} aaSbb \Rightarrow_{G_1} aaaSbbb \Rightarrow_{G_1} aaaaSbbbb$$

permettant de déduire le fait que

$$S \Rightarrow^{*}_{G_1} aaaaSbbbb$$

Arbre de dérivation

Toute dérivation peut être représentée graphiquement par un arbre appelé arbre de dérivation, défini de la manière suivante :

- la racine de l'arbre est le symbole de départ (S)
- les nœuds intérieurs sont étiquetés par des symboles non terminaux (V_n)
- si un nœud intérieur e est étiqueté par le symbole S et si la production $S \to S_1, S_2, \ldots, S_k$ a été utilisée pour dériver S alors les fils de e sont des nœuds étiquetés, de la gauche vers la droite, par $S_1, S_2, \ldots, S_k$
- les feuilles sont étiquetées par des symboles terminaux (V_t) et, si on allonge verticalement les branches de l'arbre (sans les croiser) de telle manière que les feuilles soient toutes à la même hauteur, alors, lues de la gauche vers la droite, elles constituent la chaîne w

Arbre de dérivation

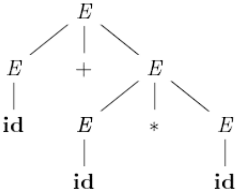
- Grammaire $G_A = (V_n, V_t, S, P)$ pour les expressions arithmétiques où :
  $V_n = \{E\}$; $V_t = \{int, (,), +, -, *, /\}$, S = E,
  $$P = \begin{cases} E \rightarrow E+E, & E \rightarrow E-E, & E \rightarrow E*E \\ E \rightarrow E/E, & E \rightarrow (E), & E \rightarrow id \end{cases}$$
- Dérivation : $id + id * id \in L(G_A)$ comme le montrent les dérivations suivantes :
  - $E \rightarrow E+E \rightarrow E+E*E \rightarrow id+E*E \rightarrow id+id*E \rightarrow id+id*id$
  - $E \rightarrow E+E \rightarrow id+E \rightarrow id+E*E \rightarrow id+id*E \rightarrow id+id*id$
  - $E \rightarrow E*E \rightarrow E+E*E \rightarrow id+E*E \rightarrow id+id*E \rightarrow id+id*id$

Remarque

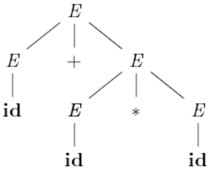
Les deux premières dérivations correspondent à la même façon de comprendre le mot, en voyant + reconnu plus haut que *, ce qui n'est pas le cas pour la dernière.

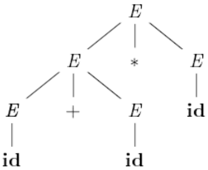
Ambigüité

- Exemple : Arbre de dérivation pour l'expression $id + id * id \in L(G_A)$ pour la dérivation 1 (et 2).



- S'il existe différentes dérivations gauches pour une même chaîne de terminaux, la grammaire est dite ambigüe.





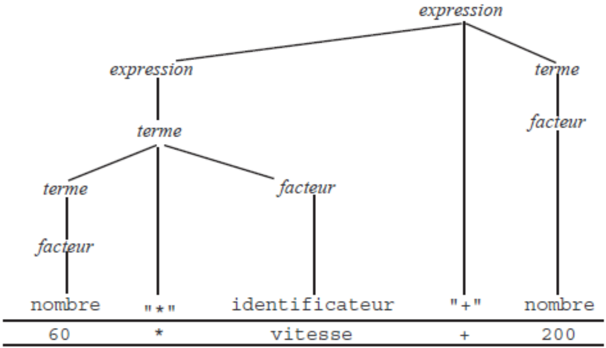
Exemple 1 (1/3) :

- Grammaire $G_1 = (V_n, V_t, S, P)$ définie par :

$$P = \left\{ \begin{array}{lcl} E & \rightarrow & E + T|T \\ T & \rightarrow & T * F|F \\ F & \rightarrow & (E)|id|nb \end{array} \right\} \text{où} \left\{ \begin{array}{lcl} E & \equiv & expression \\ T & \equiv & terme \\ F & \equiv & facteur \\ id & \equiv & identificateur \\ nb & \equiv & nombre \end{array} \right.$$

- L'analyse lexicale de "200 + 60 * vitesse" produit w = ("nombre" + "nombre" * "identificateur") $\in L(G_1)$:

$$\left\{ \begin{array}{lcl} E & \rightarrow & E + T|T \\ & \rightarrow & T + T \\ & \rightarrow & T * F + T \\ & \rightarrow & F * F + T \\ & \rightarrow & nb * F + T \\ & \rightarrow & nb * id + T \\ & \rightarrow & nb * id + F \\ & \rightarrow & nb * id + nb \end{array} \right.$$



Exemple 1 (2/3) :

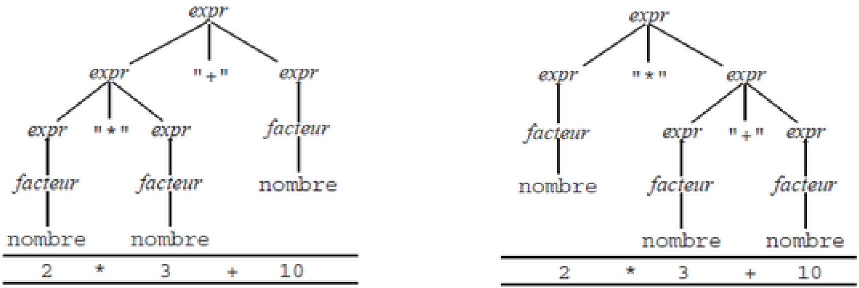
La dérivation précédente est appelée dérivation gauche car entièrement composée de dérivations en une étape où à chaque fois c'est le non-terminal le plus à gauche qui est réécrit. On définit de même une dérivation droite : à chaque étape c'est le non-terminal le plus à droite qui est réécrit.

Exemple 1 (3/3) :

- Grammaire $G_2 = (V_n, V_t, S, P)$ définie par :

$$P = \left\{ \begin{array}{lcl} E & \rightarrow & E+E|E*E|F \\ F & \rightarrow & nb|id|(E) \end{array} \right\} \text{où} \left\{ \begin{array}{lcl} E & \equiv & \textit{expression} \\ F & \equiv & \textit{facteur} \\ id & \equiv & \textit{identificateur} \\ nb & \equiv & \textit{nombre} \end{array} \right.$$

- 2 arbres de dérivation distincts pour la chaîne "2 * 3 + 10" :



Ambigüité & Suppression de l'ambigüité

Quelques remarques

- L'ambiguïté est une propriété des grammaires et non des langages.
- Idéalement, pour permettre le parsing, une grammaire ne doit pas être ambiguë. En effet, l'arbre de dérivation détermine le code généré par le compilateur.
- On essayera donc de modifier la grammaire pour supprimer ses ambiguïtés.
- Il n'existe pas d'algorithme pour supprimer les ambiguïtés d'une grammaire context-free.
- Pour un même langage, certaines grammaires peuvent être ambiguës, d'autres non.

En bref

- D'ailleurs, certains langages context-free sont ambigus de façon inhérente (toutes les grammaires définissant ce langage sont ambiguës). Un tel langage est dit intrinsèquement ambigü.
- L'ambiguïté est, pour nous, nuisible. Il est donc souhaitable de l'éviter. De plus, un analyseur syntaxique déterministe sera plus efficace. Malheureusement, déterminer si une grammaire algébrique est ou non ambiguë est un problème indécidable.
- L'analyse syntaxique consiste, étant donné un mot, à dire s'il est engendré par une grammaire donnée. Si oui, à produire un arbre de dérivation. Les techniques classiques d'analyse descendante et ascendante ne s'appliquent qu'à des grammaires non ambiguës.

Exemples de langages intrinsèquement ambigüs

1. $L = L_1 \cup L_2 = \{a^n b^n c^n\} \cup \{a^m b^n c^n\} = \{a^n b^m c^p,\ m = n|\ m = p\}$ Décrire L_1 demande d'introduire une récursion centrale, pour apparier les a et les b ; Idem pour L_2 quant aux b et les c. On peut montrer que pour toute grammaire hors-contexte engendrant ce langage, tout mot de la forme $a^n b^n c^n$ aura une double interprétation, selon que l'on utilise le mécanisme d'appariement (de comptage) des a et des b ou celui qui contrôle l'appariement des b et des c.
2. $L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$

$$P = \left\{ \begin{array}{llllllll} S & \rightarrow & AB|CF & A & \rightarrow & aAb|ab & B & \rightarrow & cBd|cd \\ C & \rightarrow & aCd|ad & C & \rightarrow & bCc|bc & & & \end{array} \right\}$$

Considérant les mots $a^i b^i c^i d^i, \forall i \geq 0$, G a 2 arbres de dérivation. On peut ensuite démontrer que toute autre Grammaire Hors-contexte pour L est ambiguë.

Opération dans les grammaires

Langage engendré & Équivalence entre Grammaires

- On appelle langage engendré par G, noté $L(G)$, le sous-ensemble de Σ^* défini par $\{w \in \Sigma^*, S \Rightarrow_G w\}$
- G_1 et G_2 sont deux grammaires équivalentes si et seulement si elles engendrent le même langage. Si, de plus, pour tout mot du langage, les arbres de dérivation dans G_1 et dans G_2 sont identiques, on dit que G_1 et G_2 sont fortement équivalentes. Dans le cas contraire, on dit que les grammaires G_1 et G_2 sont faiblement équivalentes.

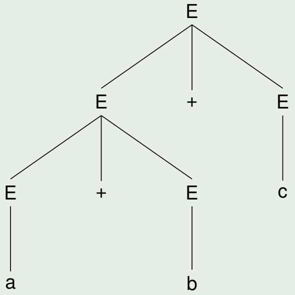
Priorité et associativité : causes d'ambigüité

Lorsque le langage définit des chaînes composés d'instructions et d'opérations, l'arbre syntaxique (qui va déterminer le code produit par le compilateur) doit refléter

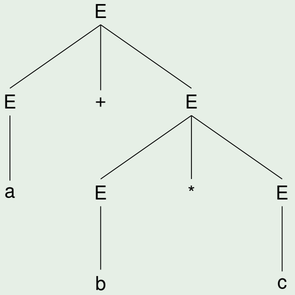
- les priorités et
- les associativités

Arbres associés à des expressions

a+b+c



a+b*c



Priorité et associativité

- Pour respecter l'associativité à gauche, on n'écrit pas

$$E \rightarrow E + E|T \text{ mais } E \rightarrow E + T|T$$

- Pour respecter les priorités on définit plusieurs niveaux de variables / règles (le symbole de départ ayant le niveau 0) : les opérateurs les moins prioritaires sont définis à un niveau plus bas (plus proche du symbole de départ) que les plus prioritaires. On écrit en 2 niveaux :

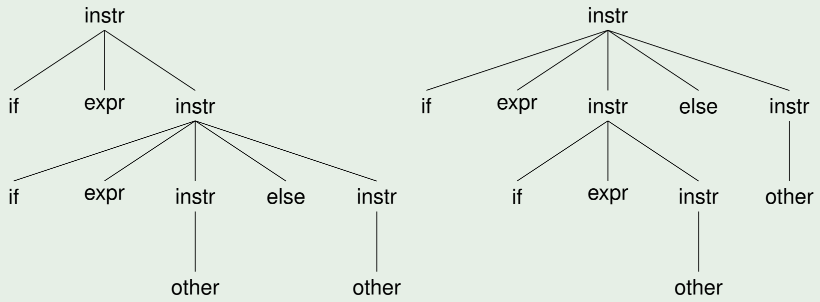
$$\text{Non pas } P = \left\{ \begin{array}{lll} E & \rightarrow & T + E|T * E|T \\ T & \rightarrow & id|(E) \end{array} \right\} \text{ mais plutôt } ???$$

$$P = \left\{ \begin{array}{lll} E & \rightarrow & T + E|T \\ T & \rightarrow & F * T|F \\ F & \rightarrow & id|(E) \end{array} \right\}$$

Dérivation

Associativité de l'instruction If

instr $\rightarrow$ *if expr instr* | *if expr instr else instr* | *other* est ambigüe.



Dans les langages impératifs habituels c'est l'arbre de gauche qu'il faut générer. Grammaire pouvant être transformée . . .

Dérivation

Associativité de l'instruction If

$instr \rightarrow$ *if expr instr* | *if expr instr else instr* | *other* est ambigüe.

1. En réalité, par convention, on fait correspondre chaque sinon avec le si qui le précède, le plus proche et sans correspondant.
2. L'idée est qu'une instruction apparaissant entre un alors et un sinon doit être "close", i-e qu'elle ne doit pas se terminer par un alors sans correspondant.
3. Une instruction close est soit une instruction si-alors-sinon ne contenant pas d'instruction non close, soit une instruction non conditionnelle quelconque.

$$\text{Et on a : } P = \left\{ \begin{array}{lcl} instr & \rightarrow & open|close \\ close & \rightarrow & if\ expr\ close\ else\ close|other \\ open & \rightarrow & if\ expr\ instr \\ open & \rightarrow & if\ expr\ close\ else\ open \end{array} \right\}$$

Simplification des grammaires

Processus de simplification des grammaires

Il existe des algorithmes pour transformer une grammaire hors-contexte en une grammaire équivalente plus simple. Cela se passe en trois étapes :

Etape 1 : Élimination des symboles inutiles.

Etape 2 : Suppression des ϵ-productions.

Etape 3 : Productions unitaires.

Étape 1 : Élimination des symboles inutiles

1. Eliminer les variables d'où ne dérive aucun mot en symboles terminaux. Pour cela
    1.1 Les variables dont une production au moins ne contient que des terminaux sont utiles
    1.2 Les variables dont une production au moins ne contient que des terminaux et des symboles utiles sont utiles.
2. Eliminer tous les symboles (terminaux ou non) n'appartenant à aucun métamot dérivé de S.
    2.1 L'axiome est utile
    2.2 Les symboles apparaissant dans les productions de symboles utiles sont utiles.

A chaque étape (1 et 2), les symboles non retenus sont inutiles, on les enlève et on a encore une grammaire équivalente.
L'ordre a de l'importance.

Application

Exemple (1/2) :

- Grammaire $G_2 = (V_n, V_t, S, P)$ définie par :

$$P = \begin{cases} S \rightarrow AB|CA, & A \rightarrow a, \quad B \rightarrow AB|EA \\ C \rightarrow aB|b, & D \rightarrow aC, \quad E \rightarrow BA \end{cases}$$

- Qu'obtient-on à l'étape 1 ? ? ?
  Il reste donc : $S \rightarrow CA, A \rightarrow a, C \rightarrow b, D \rightarrow aC$
- Qu'obtient-on à l'étape 2 ? ? ?
- Que reste-t-il finalement ? ? ?

Application

Exemple (2/2) :

- Grammaire $G_2 = (V_n, V_t, S, P)$ définie par :

$$P = \left\{ \begin{array}{lllllllll} S & \to & AB|CA, & A & \to & a, & B & \to & AB|EA \\ C & \to & aB|b, & D & \to & aC, & E & \to & BA \end{array} \right.$$

- Qu'obtient-on à l'étape 1 ? ? ?
  Il reste donc :

$$S \to CA, A \to a, C \to b, D \to aC$$

- L'étape 2 de l'algo élimine D
- Il reste finalement :

$$S \to CA, A \to a, C \to b$$

Etape 2 : Suppression des ϵ-productions.

1. Les ϵ-productions ont comme inconvénient que, dans une dérivation, la longueur des métamots peut décroître. Pour des besoins d'analyse, il est préférable d'éviter cette situation. Si ϵ appartient à L(G), il n'est bien sûr pas possible d'éviter toutes les ϵ-productions.
2. On traite donc uniquement les langages non contextuels dont on a éventuellement enlevé le mot vide.
3. On cherche récursivement les variables annulables, celles d'où dérive le mot vide ; on part d'une grammaire sans symbole inutile :
   - Les variables qui se réécrivent ϵ sont annulables ;
   - Les variables dont une production au moins ne contient que des variables annulables sont annulables.
4. L'ensemble ANNUL(G) des variables annulables de G étant déterminé, on modifie les productions contenant des variables annulables.

Application

Exemple :

- La grammaire suivante engendre les mots bien parenthésés :

$$S \rightarrow S(S)|\epsilon$$

- Celle qui suit engendre les mêmes mots, sauf le mot vide.

$$S \rightarrow S(S)|(S)|S()|()$$

- Pour obtenir le mot vide, on tolère une seule production vide, sur l'axiome, et sans autoriser celui-ci à apparaître en partie droite de production. On rajoute donc un nouvel axiome, ici T :

$$T \rightarrow S|\epsilon, S \rightarrow S(S)|(S)|S()|()$$

Etape 3 : Productions unitaires.

1. Il s'agit des productions $A \to B$, où B est une variable.
2. On suppose que G n'a aucune variable inutile, ni production vide.
3. On cherche toutes les dérivations de la forme $A \Rightarrow^* B$.
   Cela se fait récursivement à partir des productions unitaires. Chaque fois qu'une telle dérivation est obtenue, on ajoute aux productions de A toutes les productions non unitaires de B. Enfin, on efface les productions unitaires.
4. La grammaire ainsi obtenue a peut-être des symboles inutiles, qu'on supprime. La grammaire finale est équivalente à la grammaire de départ, n'a pas de symboles inutiles, de productions vides ni de productions unitaires.

Application

Exemple :

Considérant la grammaire précédente :

$$T \rightarrow S|\epsilon, S \rightarrow S(S)|(S)|S()|()$$

On obtiendrait alors :

$$P' = \begin{cases} T & \rightarrow & \epsilon|S(S)|(S)|S()|() \\ S & \rightarrow & S(S)|(S)|S()|() \end{cases}$$

Récursivité à gauche

1. Une grammaire est récursive à gauche s'il existe un non-terminal A et une dérivation de la forme $A \Rightarrow^* A\alpha$ où α est une chaîne quelconque. Une récursivité à gauche est simple si la grammaire possède une production $A \to A\alpha$.

La grammaire G_1 est récursive à gauche, et même simplement

$$P = \begin{cases} \textit{expression} & \to \quad \textit{expression} \,{}''+{}''\, \textit{terme} \mid \textit{terme} \\ \textit{terme} & \to \quad \textit{terme} \,{}''*{}''\, \textit{facteur} \mid \textit{facteur} \\ \textit{facteur} & \to \quad \textit{nombre} \mid \textit{identificateur} \mid {}''({}''\, \textit{expression} \,{}'')'' \end{cases}$$

2. Une grammaire G est non ambiguë, si pour tout mot terminal il existe au plus une dérivation gauche.
3. Idéalement, pour permettre le parsing, une grammaire ne doit pas être ambiguë. En effet, l'arbre de dérivation détermine le code généré par le compilateur.
4. Souvent, on essaie de modifier la grammaire pour supprimer ses ambiguïtés.

Élimination de la récursion gauche

1. Il existe une méthode pour obtenir une grammaire non récursive à gauche équivalente à une grammaire donnée.
2. Dans le cas de la récursivité à gauche simple, cela consiste à remplacer une production telle que $A \rightarrow A\alpha|\beta$

   par les deux productions $A \rightarrow \beta A'$ et $A' \rightarrow \alpha A'|\epsilon$
3. En appliquant à G_1 caractérisée par P, on a P'.

$$\text{Si } P = \left\{ \begin{array}{lll} \textit{expression} & \rightarrow & \textit{expression}\ "+"\ \textit{terme} \mid \textit{terme} \\ \textit{terme} & \rightarrow & \textit{terme}\ "*"\ \textit{facteur} \mid \textit{facteur} \\ \textit{facteur} & \rightarrow & \textit{nombre} \mid \textit{identificateur} \mid "(" \textit{expression} ")" \end{array} \right.$$

Que vaut donc P'???

Élimination de la récursion gauche

1. Il existe une méthode pour obtenir une grammaire non récursive à gauche équivalente à une grammaire donnée.
2. Dans le cas de la récursivité à gauche simple, cela consiste à remplacer une production telle que $A \to A\alpha|\beta$

   par les deux productions $A \to \beta A'$ et $A' \to \alpha A'|\epsilon$
3. En appliquant à G_1 caractérisée par P, on a P'.

$$\text{Si } P = \left\{ \begin{array}{lll} \textit{expression} & \to & \textit{expression} \text{ "} + \text{" } \textit{terme} \mid \textit{terme} \\ \textit{terme} & \to & \textit{terme} \text{ "} * \text{" } \textit{facteur} \mid \textit{facteur} \\ \textit{facteur} & \to & \textit{nombre} \mid \textit{identificateur} \mid \text{"("} \textit{expression} \text{")"} \end{array} \right.$$

$$P' = \left\{ \begin{array}{lll} \textit{expression} & \to & \textit{terme fin\_expression} \\ \textit{fin\_expression} & \to & \text{"} + \text{" } \textit{terme fin\_expression} \mid \epsilon \\ \textit{terme} & \to & \textit{facteur fin\_terme} \\ \textit{fin\_terme} & \to & \text{"} * \text{" } \textit{facteur fin\_terme} \mid \epsilon \\ \textit{facteur} & \to & \textit{nombre} \mid \textit{identificateur} \mid \text{"("} \textit{expression} \text{")"} \end{array} \right.$$

Élimination de la récursion gauche

1. Dans le cas de la récursivité à gauche indirecte, comme dans l'exemple $S \rightarrow Aa \mid b, A \rightarrow Ac \mid Sd \mid \epsilon$, on peut appliquer l'algorithme suivant :

Algorithme de suppression de la récursivité à gauche

Ranger les non-terminaux dans un ordre $A_1, \ldots, A_n$
for *(i in 1..n)* **do**
    for *(j in 1..i-1)* **do**
        Remplacer chaque production de la forme $A_i \rightarrow A_j\gamma$ par les productions $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\ldots|\delta_k\gamma$ où $A_j \rightarrow \delta_1|\delta_2|\ldots|\delta_k$
    end
    Eliminer les récursions simples parmi les production A_i
end

Algorithm 1: Suppression de la récursivité

Élimination de la récursion gauche

1. Un analyseur syntaxique étant prédictif, à tout moment le choix entre productions qui ont le même membre gauche doit pouvoir se faire, sans risque d'erreur, en comparant le symbole courant de la chaîne à analyser avec les symboles susceptibles de commencer les dérivations des membres droits des productions en compétition.
2. Une grammaire contenant des productions $A \to \alpha\beta_1 \mid \alpha\beta_2$ viole ce principe car lorsqu'il faut choisir entre les productions $A \to \alpha\beta_1$ et $A \to \alpha\beta_2$ le symbole courant est un de ceux qui peuvent commencer une dérivation de α et on ne peut choisir à coup sûr entre $\alpha\beta_1$ et $\alpha\beta_2$
3. La factorisation à gauche est donc cette transformation simple qui corrige ce défaut (si les symboles susceptibles de commencer une réécriture de β_1 sont distincts de ceux pouvant commencer une réécriture de β_2) :

Exemples

- Classique : Les grammaires de la plupart des langages de programmation définissent ainsi l'instruction conditionnelle

$$instr\_si \rightarrow \; si \; expr \; alors \; instr \mid si \; expr \; alors \; instr \; sinon \; instr$$

Pour avoir un analyseur prédictif dans ce cas, il faudra opérer une factorisation à gauche pour avoir :

$$P = \left\{ \begin{array}{lcl} instr\_si & \rightarrow & si \; expr \; alors \; instr \; fin\_intsr\_si \\ fin\_intsr\_si & \rightarrow & sinon \; instr \mid \epsilon \end{array} \right\}$$

- La factorisation de la grammaire $S \rightarrow ABCD|ABaD|Aba$ aboutit à

$$P = \left\{ \begin{array}{lcl} S & \rightarrow & AE \\ E & \rightarrow & BF \\ F & \rightarrow & CD \mid aD \end{array} \right\}$$

Intérêts des formes normales

- La notion de forme normale d'une grammaire répond à la nécessité, pour un certain nombre d'algorithmes de parsage, de disposer d'une connaissance a priori sur la forme des productions de la grammaire.
- Cette connaissance est exploitée pour simplifier la programmation d'un algorithme de parsage, ou encore pour accélérer l'analyse.
- Les algorithmes de mise sous forme normale construisent des grammaires faiblement équivalentes à la grammaire d'origine : les arbres de dérivation de la grammaire normalisée devront donc être transformés pour reconstruire les dérivations (et les interprétations) de la grammaire originale.

Définition

- Une grammaire hors-contexte est sous Forme Normale de Chomsky (CNF) si ses règles ont l'une des deux formes :

$$\left.\begin{array}{lcl} X & \rightarrow & YZ \\ X & \rightarrow & a \end{array}\right\} \text{avec} \left\{\begin{array}{lcl} X, Y, Z & \in & V_n \\ a & \in & V_t \end{array}\right.$$

- Une grammaire hors-contexte est sous Forme Normale de Chomsky étendue si ses règles peuvent également prendre les formes :

$$\left.\begin{array}{lcl} X & \rightarrow & YZ \\ X & \rightarrow & Y \\ X & \rightarrow & a \end{array}\right\} \text{avec} \left\{\begin{array}{lcl} X, Y, Z & \in & V_n \\ a & \in & V_t \end{array}\right.$$

Mise sous CNF

Théorème

Pour toute grammaire hors-contexte, il existe une grammaire hors-contexte faiblement équivalente sous forme normale.
Si, de surcroît, $S \Rightarrow_G^* \epsilon$, alors la forme normale contient également $S \to \epsilon$

Concepts & Définition

- Deux grammaires sont dites équivalentes si elles peuvent produire les mêmes chaînes de symboles terminaux.
- Grammaires de type 2 : passage d'un arbre syntaxique à un arbre équivalent sous CNF = facile

Mise sous CNF (Méthode en 3 temps)

1. Suppression des règles de type : $X \to \alpha t_i \beta$ où t_i est un terminal et α et/ou β est/sont non vide(s)
   1.1 Créer un non-terminal T_i
   1.2 Ajouter la règle $T_i \to t_i$
   1.3 Remplacer la règle $X \to \alpha t_i \beta$ par $X \to \alpha T_i \beta$
2. Suppression des règles de type : $X \to Y$
   2.1 Pour chaque règle $Z \to \alpha X \beta$, ajouter une règle $Z \to \alpha Y \beta$
   2.2 Supprimer la règle $X \to Y$
3. Suppression des règles de type : $X \to YZ\alpha$
   3.1 Créer un nouveau non-terminal X_i
   3.2 Ajouter la règle $X_i \to Z\alpha$
   3.3 Remplacer la règle $X \to YZ\alpha$ par $X \to YX_i$

Cette méthode de mise sous CNF augmente considérablement le nombre de non-terminaux et de règles.

Application

Exemple

$$R = \left\{ \begin{array}{lllllll} P & \rightarrow & SN\ SV, & SN & \rightarrow & Det\ N \\ SN & \rightarrow & Det\ N\ SP, & SP & \rightarrow & Prep\ SN \\ SV & \rightarrow & V, & SV & \rightarrow & V\ SN \\ SV & \rightarrow & V\ SN\ SP, & SV & \rightarrow & mange \end{array} \right\} \quad (1)$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$

$$R = \left\{ \begin{array}{llllll} P & \rightarrow & SN\ SV, & SN & \rightarrow & Det\ N \\ SN & \rightarrow & Det\ N\ SP, & SP & \rightarrow & Prep\ SN \\ SV & \rightarrow & V, & SV & \rightarrow & V\ SN \\ SV & \rightarrow & V\ SN\ SP, & SV & \rightarrow & mange \end{array} \right\}$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \rightarrow Det\ N\ SP$ $SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow V$	$R_{1.2}$:	$P \rightarrow SN\ V$
R_5 :	$SV \rightarrow V\ SN$	R_6 :	$SV \rightarrow V\ SN$
R_7 :	$SV \rightarrow V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \rightarrow SN\ SP$ $SV \rightarrow V\ X_2$
R_8 :	$SV \rightarrow mange$	R_8 :	$SV \rightarrow mange$

$$R = \left\{ \begin{array}{lllllll} P & \to & SN\ SV, & SN & \to & Det\ N \\ SN & \to & Det\ N\ SP, & SP & \to & Prep\ SN \\ SV & \to & V, & SV & \to & V\ SN \\ SV & \to & V\ SN\ SP, & SV & \to & mange \end{array} \right\}$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \to SN\ SV$	R_1 :	$P \to SN\ SV$
R_2 :	$SN \to Det\ N$	R_2 :	$SN \to Det\ N$
R_3 :	$SN \to Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \to Det\ N\ SP$ $SN \to Det\ X_1$
R_4 :	$SP \to Prep\ SN$	R_4 :	$SP \to Prep\ SN$
R_5 :	$SV \to V$	$R_{1.2}$:	$P \to SN\ V$
R_5 :	$SV \to V\ SN$	R_6 :	$SV \to V\ SN$
R_7 :	$SV \to V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \to SN\ SP$ $SV \to V\ X_2$
R_8 :	$SV \to mange$	R_8 :	$SV \to mange$

$$R = \left\{ \begin{array}{lllllll} P & \rightarrow & SN\ SV, & SN & \rightarrow & Det\ N \\ SN & \rightarrow & Det\ N\ SP, & SP & \rightarrow & Prep\ SN \\ SV & \rightarrow & V, & SV & \rightarrow & V\ SN \\ SV & \rightarrow & V\ SN\ SP, & SV & \rightarrow & mange \end{array} \right\}$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \rightarrow Det\ N\ SP$ $SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow V$	$R_{1.2}$:	$P \rightarrow SN\ V$
R_5 :	$SV \rightarrow V\ SN$	R_6 :	$SV \rightarrow V\ SN$
R_7 :	$SV \rightarrow V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \rightarrow SN\ SP$ $SV \rightarrow V\ X_2$
R_8 :	$SV \rightarrow mange$	R_8 :	$SV \rightarrow mange$

$$R = \left\{ \begin{array}{llllll} P & \to & SN\ SV, & SN & \to & Det\ N \\ SN & \to & Det\ N\ SP, & SP & \to & Prep\ SN \\ SV & \to & V, & SV & \to & V\ SN \\ SV & \to & V\ SN\ SP, & SV & \to & mange \end{array} \right\}$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \to SN\ SV$	R_1 :	$P \to SN\ SV$
R_2 :	$SN \to Det\ N$	R_2 :	$SN \to Det\ N$
R_3 :	$SN \to Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \to Det\ N\ SP$ $SN \to Det\ X_1$
R_4 :	$SP \to Prep\ SN$	R_4 :	$SP \to Prep\ SN$
R_5 :	$SV \to V$	$R_{1.2}$:	$P \to SN\ V$
R_5 :	$SV \to V\ SN$	R_6 :	$SV \to V\ SN$
R_7 :	$SV \to V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \to SN\ SP$ $SV \to V\ X_2$
R_8 :	$SV \to mange$	R_8 :	$SV \to mange$

$$R = \left\{ \begin{array}{lllllll} P & \rightarrow & SN\ SV, & SN & \rightarrow & Det\ N \\ SN & \rightarrow & Det\ N\ SP, & SP & \rightarrow & Prep\ SN \\ SV & \rightarrow & V, & SV & \rightarrow & V\ SN \\ SV & \rightarrow & V\ SN\ SP, & SV & \rightarrow & mange \end{array} \right\}$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \rightarrow Det\ N\ SP$ $SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow V$	$R_{1.2}$:	$P \rightarrow SN\ V$
R_5 :	$SV \rightarrow V\ SN$	R_6 :	$SV \rightarrow V\ SN$
R_7 :	$SV \rightarrow V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \rightarrow SN\ SP$ $SV \rightarrow V\ X_2$
R_8 :	$SV \rightarrow mange$	R_8 :	$SV \rightarrow mange$

$$R = \left\{ \begin{array}{lllllll} P & \to & SN\ SV, & SN & \to & Det\ N \\ SN & \to & Det\ N\ SP, & SP & \to & Prep\ SN \\ SV & \to & V, & SV & \to & V\ SN \\ SV & \to & V\ SN\ SP, & SV & \to & mange \end{array} \right\}$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \to SN\ SV$	R_1 :	$P \to SN\ SV$
R_2 :	$SN \to Det\ N$	R_2 :	$SN \to Det\ N$
R_3 :	$SN \to Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \to Det\ N\ SP$ $SN \to Det\ X_1$
R_4 :	$SP \to Prep\ SN$	R_4 :	$SP \to Prep\ SN$
R_5 :	$SV \to V$	$R_{1.2}$:	$P \to SN\ V$
R_5 :	$SV \to V\ SN$	R_6 :	$SV \to V\ SN$
R_7 :	$SV \to V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \to SN\ SP$ $SV \to V\ X_2$
R_8 :	$SV \to mange$	R_8 :	$SV \to mange$

$$R = \left\{ \begin{array}{lllllll} P & \rightarrow & SN\ SV, & SN & \rightarrow & Det\ N \\ SN & \rightarrow & Det\ N\ SP, & SP & \rightarrow & Prep\ SN \\ SV & \rightarrow & V, & SV & \rightarrow & V\ SN \\ SV & \rightarrow & V\ SN\ SP, & SV & \rightarrow & mange \end{array} \right\}$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \rightarrow Det\ N\ SP$ $SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow V$	$R_{1.2}$:	$P \rightarrow SN\ V$
R_5 :	$SV \rightarrow V\ SN$	R_6 :	$SV \rightarrow V\ SN$
R_7 :	$SV \rightarrow V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \rightarrow SN\ SP$ $SV \rightarrow V\ X_2$
R_8 :	$SV \rightarrow mange$	R_8 :	$SV \rightarrow mange$

$$R = \left\{ \begin{array}{llllll} P & \rightarrow & SN\ SV, & SN & \rightarrow & Det\ N \\ SN & \rightarrow & Det\ N\ SP, & SP & \rightarrow & Prep\ SN \\ SV & \rightarrow & V, & SV & \rightarrow & V\ SN \\ SV & \rightarrow & V\ SN\ SP, & SV & \rightarrow & mange \end{array} \right\}$$

Règle initiale	**Forme initiale**	**Règle utilisée**	**Forme Normale de Chomsky**
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \rightarrow Det\ N\ SP$ $SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow V$	$R_{1.2}$:	$P \rightarrow SN\ V$
R_5 :	$SV \rightarrow V\ SN$	R_6 :	$SV \rightarrow V\ SN$
R_7 :	$SV \rightarrow V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \rightarrow SN\ SP$ $SV \rightarrow V\ X_2$
R_8 :	$SV \rightarrow mange$	R_8 :	$SV \rightarrow mange$

Forme Normale de Greibach (GNF) & Théorème

À la différence de la forme normale de Chomsky, l'existence de la forme normale de Greibach est plus utilisée pour démontrer divers résultats théoriques que pour fonder des algorithmes d'analyse.

Définition

Une grammaire hors-contexte est sous Forme Normale de Greibach (GNF) si ses règles sont de la forme : $X \rightarrow a\,\alpha$ où $a \in V_t$ et $\alpha \in V_n^*$

Théorème

Tout langage L non contextuel sans le mot vide peut être engendré par une grammaire sans symbole inutile sous Forme Normale de Greibach. Si $\epsilon \in L$, le résultat reste valide et on ajoute $S \rightarrow \epsilon$

Pour la démonstration, on part d'une grammaire G sous forme normale de Chomsky.
On ordonne (arbitrairement) les variables : $A_1 A_2 \ldots A_m$: c'est une bonne idée de supposer que la première est l'axiome.

Les productions considérées seront de trois types possibles :

type 1 : $A_i \rightarrow a\,\alpha$ où $\alpha \in \{A_i + B_i\}^*$;

type 2 : $A_i \rightarrow A_j\,\alpha$ où $\alpha \in \{A_i\}\{A_i + B_i\}^*$ et $j > i$;

type 3 : $A_i \rightarrow A_j\,\alpha$ où $\alpha \in \{A_i\}\{A_i + B_i\}^*$ et $j \leq i$;

Au départ, les trois types sont les seuls présents (Chomsky). Dans un premier temps, on va éliminer les productions de type 3, quitte à introduire de nouvelles variables B_i et des productions de la forme $B_j \rightarrow A_j\,\beta$ où $\beta \in \{A_i + B_i\}^*$.

Supposons qu'on a pu éliminer les productions de type 3 pour les variables jusqu'à A_{i-1} en introduisant les B_j correspondantes.

On considère les productions de type 3 dans l'ordre croissant des indices j : $A_i \rightarrow A_1\alpha$ et on remplace A_1 par ses productions.

On fait ainsi apparaître des productions des trois types, mais plus aucune de type 3 commençant par A_1. Puis on fait de même pour $A_2, \ldots, A_{i-1}$. Les seules productions de type 3 restantes sont directement récursives à gauche.

Les productions de A_i sont des types :

type 1 : $A_i \to a\,\alpha$ où $\alpha \in \{A_i + B_j\}^*$:

type 2 : $A_i \to A_j\,\alpha$ où $\alpha \in \{A_i\}\{A_i + B_i\}^*$ et $j > i$:

type 3 : $A_i \to A_i\,\gamma$ où $\gamma \in \{A_i\}\{A_i + B_j\}^*$

On supprime le type 3 par élimination de la récursivité directe à gauche : on introduit donc la variable Bi et on obtient des productions :
$A_i \to a\,\alpha \mid a\,\alpha\,B_i, A_i \to A_j\,\alpha \mid A_j\,\alpha\,B_i, B_i \to \gamma \mid \gamma B_i$.
Les productions restantes sont bien de types 1 ou 2, et les productions de B_i commencent bien par une variable A_j.
On considère maintenant les productions de type 2 par ordre décroissant des indices.
La dernière variable A_m ne peut en avoir. Dans les productions de A_{m-1} de type 2, seule apparaît A_m, qu'on remplace par ses productions et ainsi de suite jusqu'à A_1.

- Maintenant, toutes les productions des A_i sont de type 1. Dans les productions des B_i , on remplace le premier symbole, qui est un A_i , par les productions de celui-ci. La grammaire obtenue est sous forme normale de Greibach.

Remarques :

- Cela accroît le nombre de productions de façon exponentielle.
- Dans la pratique, il n'est pas indispensable que la grammaire soit sous forme normale de Chomsky, mais le déroulement de l'algorithme peut en être perturbé.

Exemple :

On considère la grammaire initiale :

$$\left\{ \begin{array}{lcl} A_1 & \rightarrow & A_2\ A_3 \\ A_2 & \rightarrow & A_3\ A_1 \mid b \\ A_3 & \rightarrow & A_1\ A_2 \mid a \end{array} \right.$$

- Première étape : A_1 et A_2 sans changements ;
  Pour A_3, garder $A_3 \to a$; puis $A_3 \to A_2A_3A_2$, mais aussi $A_3 \to A_3A_1A_3A_2 \mid bA_3A_2$; les productions de A_3 sont donc : $A_3 \to A_3A_1A_3A_2$ et $A_3 \to bA_3A_2 \mid a$.
- Élimination de la récursivité directe à gauche :
  $A_3 \to bA_3A_2 \mid a\ bA_3A_2B_3 \mid aB_3$ et $B_3 \to A_1A_3A_2 \mid A_1A_3A_2B_3$.
- Comme prévu, les productions de A_3 sont de type 2. Utilisons-les pour récrire les productions de A_2 :
  $A_2 \to bA_3A_2A_1 \mid aA_1 \mid bA_3A_2B_3A_1 \mid aB_3A_1 \mid b$.
- Réécrivons maintenant les productions de A_1 :
  $A_1 \to bA_3A_2A_1A_3 \mid aA_1A_3 \mid bA_3A_2B_3A_1A_3 \mid aB_3A_1A_3 \mid bA_3$
- puis les productions de B_3 :
  $B_3 \to bA_3A_2A_1A_3A_3A_2 \mid aA_1A_3A_3A_2 \mid bA_3A_2B_3A_1A_3A_3A_2$
  $B_3 \to aB_3A_1A_3A_3A_2 \mid bA_3A_3A_2 \mid bA_3A_2A_1A_3A_3A_2B_3$
  $B_3 \to aA_1A_3A_3A_2B_3 \mid bA_3A_2B_3A_1A_3A_3A_2B_3$
  $B_3 \to aB_3A_1A_3A_3A_2B_3 \mid bA_3A_3A_2B_3$

Lemme de pompage

Les grammaires algébriques sont intrinsèquement limitées dans la structure des mots qu'elles engendrent.
Appréhension de la nature de cette limitation $\Rightarrow$ Introduction de quelques notions nouvelles.

Quelques définitions

Soit a un symbole terminal d'une grammaire G.

- Lignée de a : Séquence de règles utilisée pour produire a à partir de S. Chaque élément de la lignée est une paire (P, i), où P est une production, et i l'indice dans la partie droite de P de l'ancêtre de a.
- Un symbole est original si tous les couples (P, i) qui constituent sa lignée sont différents.
- Par extension, Mot original = mot dont tous les symboles qui le composent sont originaux.

Lemme de pompage

Exemple :

- Considérant de nouveau la grammaire pour le langage $a^n b^n$ et une dérivation de aaabbb dans cette grammaire, la lignée du second a de aaabbb correspond à la séquence $(S \rightarrow aSb, 2), (S \rightarrow aSb, 1)$.
- Contrairement au premier et au second a de aaabbb, le troisième a n'est pas original, puisque sa lignée est $(S \rightarrow aSb, 2), (S \rightarrow aSb, 2), (S \rightarrow aSb, 1)$.

Remarques

Une grammaire algébrique, même lorsqu'elle engendre un nombre infini de mots, ne peut produire qu'un nombre fini de mots originaux.

- En effet, puisqu'il n'y a qu'un nombre fini de productions, chacune contenant un nombre fini de symboles dans sa partie droite, chaque symbole terminal ne peut avoir qu'un nombre fini de lignées différentes.
- Les symboles étant finis, il existe donc une longueur maximale pour un mot original et donc un nombre fini de mots originaux.
- À quoi ressemblent alors les mots non-originaux ?
  Soit s un tel mot, il contient nécessairement un symbole a, qui, étant non-original, contient deux fois le même ancêtre (A, i).
  La dérivation complète de s pourra donc s'écrire :

  $$S \Rightarrow_G^* uAy \Rightarrow_G^* uvAxy \Rightarrow_G^* uvwxy$$

  où u, v, w, x, y sont des séquences de terminaux, la séquence w contenant le symbole a.
- Il est alors facile de déduire de nouveaux mots engendrés par la grammaire, en remplaçant w (qui est une dérivation possible de A) par vwx qui est une autre dérivation possible de A.

Ce processus peut même être itéré, permettant de construire un nombre infini de nouveaux mots, tous non-originaux, et qui sont de la forme : $uv^n wx^n z$.

Lemme de pompage

Pour montrer qu'un langage n'est pas hors-contexte, on exploite parfois le lemme de pompage (pumping lemma) associé aux langages hors-contextes.

Lemme de pompage pour les Langages Hors-contexte

Si L est un langage algébrique. Alors il existe un entier p (appelé longueur de pompage) tel que tout mot $t \in L$ de taille $|t| > p$, t se décompose en 5 facteurs $t = uvxyz$ avec $u, v, w, x, y \in \Sigma^*$, tels que :

(i) $|vx| \geq 1$ i-e l'un au moins de v et x est $\neq \epsilon$

(ii) $|vwx| < p$

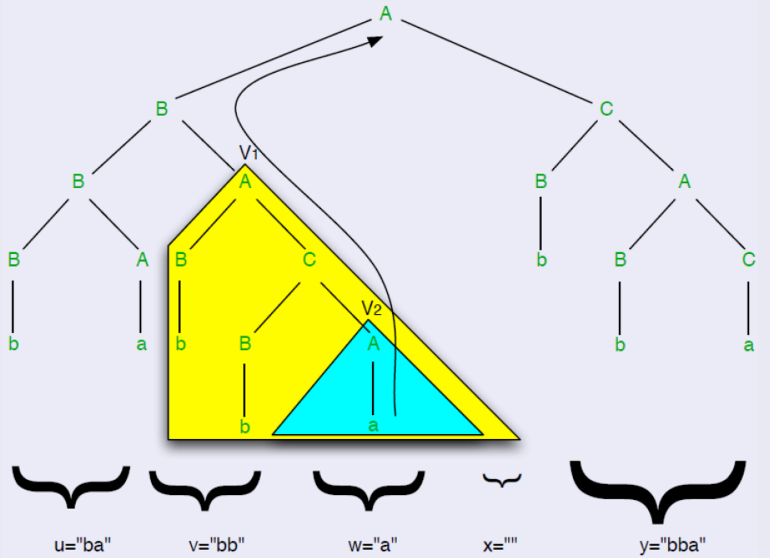
(iii) $\forall i \geq 0, uv^iwx^iy \in L$

Lemme de pompage pour les CFG

Démonstration

- Soit L un langage algébrique
- L (ou $L \setminus \{\epsilon\}$)est reconnu par une CFG en Forme Normale de Chomsky
- Soit k le nombre de variables de G
- Prenons $p = 2^k$ et $|z| > p$
- Considérons un des plus longs chemins de l'arbre de dérivation de z : plusieurs noeuds du chemin on le même label (une variable)
- Dénotons par v_1 et v_2 2 sommets sur ce chemin étiquetés par la même variable avec v_1 ancêtre de v_2 et on choisit v_1 "le plus bas possible"
- Découpons z comme indiqué sur la figure

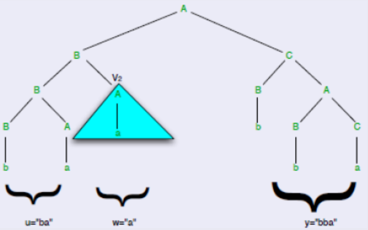
Lemme de pompage pour les CFG

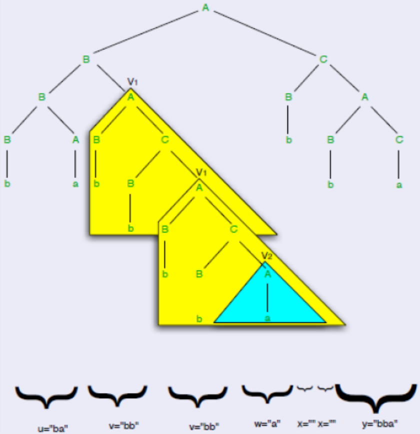


Démonstration (Suite)

- Et donc z=uvwxy
- La hauteur maximale de l'arbre de racine v_1 est k+1 et génère un mot vwx avec $|vwx| \leq 2^k = p$
- De plus comme G est en forme normale de Chomsky, v_2 est soit dans le fils droit soit dans le fils gauche de l'arbre de racine v_1
- De toute façon, soit donc $v \neq \epsilon$ soit $x \neq \epsilon$
- On peut enfin voir que $uv^i wx^i y \in L(G)$
    - Si on remplace le sous-arbre de racine v_1 par le sous-arbre de racine v_2, on obtient $uwy \in L$
    - On peut plutôt remplacer le sous-arbre de racine v_2 par une nouvelle instance de sous-arbre de racine v_1, on obtient $uv^2 wx^2 y \in L$
    - On peut répéter l'opération précédente avec l'arbre obtenu et on obtient que $\forall i : uv^i wx^i y \in L$

Lemme de pompage pour les CFG





Application

Montrer que pour $\Sigma = \{a, b, c\}$, $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ n'est pas algébrique. On procède donc par l'absurde en supposant le contraire.

- Soit k l'entier dont le lemme de pompage garantit l'existence. Considérons le mot $t = a^k b^k c^k$.
- Il doit donc exister une factorisation $t = uvwxy$ pour laquelle on a (i) $|vx| \geq 1$, (i) $|vwx| \geq k$ et (iii) $\forall i \geq 0, uv^i wx^i y \in L$.
- La propriété (ii) assure que le facteur vwx ne peut pas contenir simultanément les lettres a et c : en effet, tout facteur de t contenant un a et un c doit aussi avoir le facteur b^k, et donc être de longueur $\geq k + 2$.
- Supposons que vwx ne contienne pas la lettre c (l'autre cas étant complètement analogue) : en particulier ni v ni x ne le contient. Donc le mot $uv^i wx^i y \in L$ d'après (iii) a le même nombre de c que le mot t initial ; mais comme son nombre de a ou bien de b est différent (d'après (i)), d'où la contradiction.

Opérations de clôture

Théorèmes

1. L_1 et L_2 algébrique $\Rightarrow L_1 \cup L_2$, $L_1.L_2$ et L^* algébriques
2. L algébrique $\Rightarrow L^R$ aussi
3. Si L et M sont algébriques $\Rightarrow L \cap M$ peut ne pas être algébrique.
4. L est algébrique et Ms est régulier $\Rightarrow L \cap M$ est algébrique.
5. L est algébrique $\Rightarrow \underline{L}$ ne peut être algébrique.
6. Si L et M sont algébriques $\Rightarrow L \setminus M$ peut ne pas être algébrique.

La démonstration de chacun des items précédents devra être fait en exercice

Indécidabilité dans les langages algébriques

Quelques propriétés connues

- Il n'existe pas d'algorithme pour décider si une grammaire est ambiguë ou si elle est ambiguë de façon inhérente
- Il n'existe pas d'algorithme pour décider si l'intersection de deux langages algébriques est vide
- Il n'existe pas d'algorithme pour décider si le langage algébrique $L = \Sigma^*$
- Il n'existe pas d'algorithme pour décider si deux grammaires sont équivalentes (rappelons : la preuve du résultat (positif) obtenu pour les langages rationnels utilisait la clôture par intersection de ces langages)
- Il n'existe pas d'algorithme pour décider si $L(G_1) \subseteq L(G_2)$.
- Il n'existe pas d'algorithme pour décider si une CFG engendre en fait un langage régulier