

Les expressions régulières

Définitions

Introduction

Comment décrire un langage ??

Étant donné un mot, appartient il à un langage donné ??

Nous allons parler de la théorie des langages, en particulier nous décrivons les expressions régulières, et par conséquent les langages réguliers

Les langages

Définitions

On appelle **alphabet** un ensemble fini non vide A de symboles (lettres de 1 ou plusieurs caractères).

On appelle **mot** toute séquence finie d'éléments de A .

On note ϵ le **mot vide**.

On note A^* l'ensemble infini contenant tous les mots possibles sur A .

On note A^+ l'ensemble des mots non vides que l'on peut former sur A , c'est à dire $A^+ = A^* - \{\epsilon\}$

On note $|m|$ la longueur du mot m , c'est à dire le nombre de symboles de A composant le mot.

On note A^n l'ensemble des mots de A^* de longueur n . Remarque : $A^* = \bigcup_{n=0}^{\infty} A^n$

Exemples

Soit l'alphabet $A = \{a, b, c\}$. $aaba$, $bbbacbb$, c et ϵ sont des mots de A^* , de longueurs respectives 4, 7, 1 et 0.

Soit l'alphabet $A = \{aa, b, c\}$. aba n'est pas un mot de A^* . $baab$, caa , bc , $aaaa$ sont des mots de A^* de longueurs respectives 3, 2, 2 et 2.

Notation

On note \cdot l'opérateur de concaténation de deux mots : si $u = u_1 \dots u_n$ (avec $u_i \in A$) et $v = v_1 \dots v_p$ (avec $v_i \in A$), alors la concaténation de u et v est le mot $u.v = u_1 \dots u_n v_1 \dots v_p$

Remarque : un mot de n lettres est en fait la concaténation de n mots d'une seule lettre.

Les langages

Propriété

$$|u.v| = |u| + |v|$$

$$(u.v).w = u.(v.w) \text{ (associativité)}$$

ε est l'élément neutre pour \cdot : $u.\varepsilon = \varepsilon.u = u$

Remarque : nous écrirons désormais uv pour $u.v$

Définition

On appelle langage sur un alphabet A tout sous-ensemble de A^* .

Exemples

Soit l'alphabet $A = \{a, b, c\}$

Soit L_1 l'ensemble des mots de A^* ayant autant de a que de b . L_1 est le langage infini $\{\varepsilon, c, ccc, \dots, ab, ba, \dots, abccc, acbcc, acbcb, \dots, aabb, abab, abba, baab, \dots, acbcbcbccccc, \dots, bbbcccaacbbcabcccaac, \dots\}$

Soit L_2 l'ensemble de tous les mots de A^* ayant exactement 4 a . L_2 est le langage infini $\{aaaa, aaaac, aaaca, \dots, aabaa, \dots, caaaba, \dots, abcabbbaacc, \dots\}$

Operation sur les langages

Opérations sur les langages

union : $L_1 \cup L_2 = \{w \text{ tq } w \in L_1 \text{ ou } w \in L_2\}$

intersection : $L_1 \cap L_2 = \{w \text{ tq } w \in L_1 \text{ et } w \in L_2\}$

concaténation : $L_1 L_2 = \{w = w_1 w_2 \text{ tq } w_1 \in L_1 \text{ et } w_2 \in L_2\}$

puissance : $L^n = \{w = w_1 \dots w_n \text{ tq } w_i \in L \text{ pour tout } i \in \{1, \dots, n\}\}$

étoile : $L^* = \cup_{n \geq 0} L^n$

Les langages réguliers

Problème

étant donné un langage, comment décrire tous les mots acceptables ? Comment décrire un langage ?

Il existe plusieurs types de langage (classification), certains étant plus facile à décrire que d'autres. On s'intéresse ici aux **langages réguliers**.

Définitions

Un langage régulier L sur un alphabet A est défini récursivement de la manière suivante :

- $\{\epsilon\}$ est un langage régulier sur A
- Si a est une lettre de A , $\{a\}$ est un langage régulier sur A
- Si R est un langage régulier sur A , alors R^n et R^* sont des langages réguliers sur A
- Si R_1 et R_2 sont des langages réguliers sur A , alors $R_1 \cup R_2$ et $R_1 R_2$ sont des langages réguliers

Les langages réguliers se décrivent très facilement par une **expression régulière**.

Les langages réguliers

Définitions

Les expressions régulières (E.R.) sur un alphabet A et les langages qu'elles décrivent sont définis récursivement de la manière suivante :

- ε est une E.R. qui décrit le langage $\{\varepsilon\}$
- Si $a \in A$, alors a est une E.R. qui décrit $\{a\}$
- Si r est une E.R. qui décrit le langage R , alors $(r)^*$ est une E.R. décrivant R^*
- Si r est une E.R. qui décrit le langage R , alors $(r)^+$ est une E.R. décrivant R^+
- Si r et s sont des E.R. qui décrivent respectivement les langages R et S , alors $(r)|(s)$ est une E.R. décrivant $R \cup S$
- Si r et s sont des E.R. qui décrivent respectivement les langages R et S , alors $(r)(s)$ est une E.R. décrivant RS
- Il n'y a pas d'autres expressions régulières

Remarques

on conviendra des priorités décroissantes suivantes : $*$, concaténation, $|$ C'est à dire par exemple que $ab^*|c = ((a)((b)^*))|(c)$

En outre, la concaténation est distributive par rapport à $|$: $r(s|t) = rs|rt$ et $(s|t)r = sr|tr$.

Les langages réguliers

Exemples

- $(a|b)^* = (b|a)^*$ dénote l'ensemble de tous les mots formés de a et de b , ou le mot vide.
- $(a)|((b)^*(c)) = a|b^*c$ est soit le mot a , soit les mots formés de 0 ou plusieurs b suivi d'un c . C'est à dire $\{a, c, bc, bbc, bbbc, bbbbc, \dots\}$
- $(a^*|b^*)^* = (a|b)^* = ((\varepsilon|a)b^*)^*$ décrit tous les mots sur $A = \{a, b\}$ ou encore A^*
- $(a|b)^*abb(a|b)^*$ dénote l'ensemble des mots sur $\{a, b\}$ ayant le facteur abb
- $b^*ab^*ab^*ab^*$ dénote l'ensemble des mots sur $\{a, b\}$ ayant exactement 3 a
- $(abbc|baba)^+ao(cc|bb)^* = \{abbcua, \dots, babaabbababaaa, \dots, abbcabbcacccbbbb, \dots\}$

Remarques

$(a|b)^*a(a|b)^*$, qui décrit les mots sur $\{a, b\}$ ayant au moins un a est **ambiguë**. Car, par exemple, le mot $abaab$ "colle" à l'expression régulière de plusieurs manières :

$$abaab = \varepsilon . a . baab \text{ avec } \varepsilon \in (a|b)^*, \text{ et } baab \in (a|b)^*$$

$$abaab = ab . a . ab \text{ avec } ab \in (a|b)^*, \text{ et } ab \in (a|b)^*$$

$$abaab = aba . a . b \text{ avec } aba \in (a|b)^*, \text{ et } b \in (a|b)^*$$

Par contre, l'e.r. $b^*a(a|b)^*$ décrit le même langage et n'est **pas** ambiguë.

$$abaab = \varepsilon . a . baab \text{ avec } \varepsilon \in b^*, \text{ et } baab \in (a|b)^*$$

Les automates à états finis

Problématique

Problème

Le problème qui se pose est de pouvoir reconnaître si un mot donné appartient à un langage donné. Un **reconnaisseur** pour un langage est un programme qui prend en entrée une chaîne x et répond oui si x est une phrase (un mot) du langage et non sinon.

Théorème

Les automates à états finis (A.E.F.) sont des reconnaisseurs pour les langages réguliers.

Automates à états finis

Définitions

Un automate à états finis (AEF) est défini par

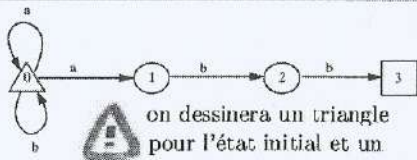
- un ensemble fini E d'états
- un état $e_0 \in E$ distingué comme étant l'état initial
- un ensemble fini T (T inclus dans E) d'états distingués comme états finaux (ou états terminaux)
- un alphabet Σ des symboles d'entrée
- une fonction de transitions Δ qui à tout couple formé d'un état et d'un symbole de Σ fait correspondre un ensemble (éventuellement vide) d'états : $\Delta(e_i, a) = \{e_{i_1}, \dots, e_{i_n}\}$

Exemple

$\Sigma = \{a, b\}$, $E = \{0, 1, 2, 3\}$, $e_0 = 0$, $T = \{3\}$

$\Delta(0, a) = \{0, 1\}$, $\Delta(0, b) = \{0\}$, $\Delta(1, b) = \{2\}$, $\Delta(2, b) = \{3\}$ (et $\Delta(e, l) = \emptyset$ sinon)

Représentation graphique



Représentation par une table de transition

état	a	b
0	0,1	0
1	-	2
2	-	3
3	-	-

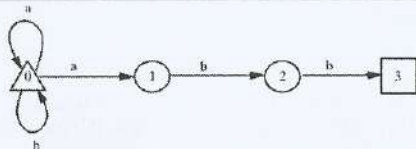
$e_0 = 0$ et $T = \{3\}$

Automates à états finis

Définition

Le langage reconnu par un automate est l'ensemble des chaînes qui permettent de passer de l'état initial à un état terminal.

Exemple



L'automate de l'exemple précédent accepte le langage régulier (l'expression régulière)

Remarque

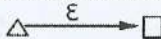
Un automate peut très facilement être simulé par un algorithme (et donc on peut écrire un programme simulant un AEF). C'est encore plus facile si l'automate est **déterministe**, c'est à dire lorsqu'il n'y a pas à choisir entre 2 transitions). Ce que signifie donc le théorème 3.1 c'est que l'on peut écrire un programme reconnaissant tout mot (toute phrase) de tout langage régulier. Ainsi, si l'on veut faire l'analyse lexicale d'un langage régulier, il suffit d'écrire un programme simulant l'automate qui lui est associé.

Construction d'un AFN à partir d'une E.R.

Définitions

On appelle ϵ -transition, une transition par le symbole ϵ entre deux états.

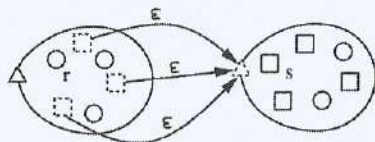
Pour une expression régulière s , on note $A(s)$ un automate reconnaissant cette expression.

- automate acceptant la chaîne vide 

- automate acceptant la lettre a 

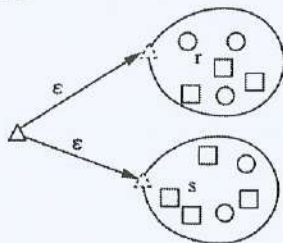
- automate acceptant $(r)(s)$

1. mettre une ϵ -transition de chaque état terminal de $A(r)$ vers l'état initial de $A(s)$
2. les états terminaux de $A(r)$ ne sont plus terminaux
3. le nouvel état initial est celui de $A(r)$
4. (l'ancien état initial de $A(s)$ n'est plus état initial)



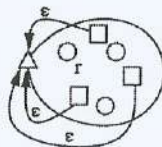
- automate reconnaissant $r|s$

1. créer un nouvel état initial q
2. mettre une ϵ -transition de q vers les états initiaux de $A(r)$ et $A(s)$
3. (les états initiaux de $A(r)$ et $A(s)$ ne sont plus états initiaux)



- automate reconnaissant r^+

mettre des ϵ -transition de chaque état terminal de $A(r)$ vers son état initial



Construction d'un AFN à partir d'une E.R.

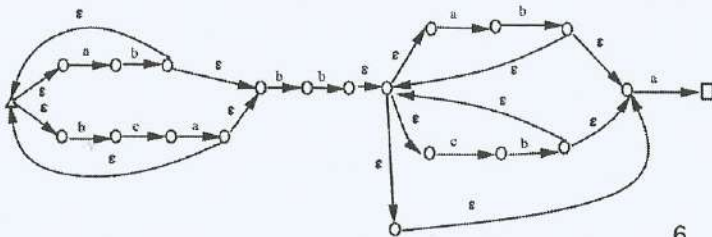
Exemple1

Donner un AFN des expressions régulières suivantes :

- a^*
- a^+
- $a|b$
- $a^*|b$
- $a^*|b^+$
- $a^+|b^+$
- $(a|b)^+$
- $(a|b)^*$

Exemple1

Donner une ER que reconnaît l'automate suivant :



Automates finis déterministes (AFD)

Définitions

On appelle ϵ -transition, une transition par le symbole ϵ entre deux états.

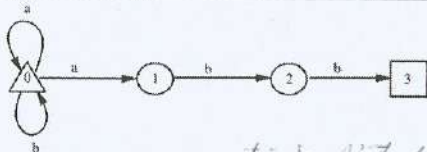
Un automate fini est dit **déterministe** lorsqu'il ne possède pas de ϵ -transition et lorsque pour chaque état e et pour chaque symbole a , il y a au plus un arc étiqueté a qui quitte e .

Remarques

L'automate donné en exemple précédemment qui reconnaît $(a|b)^*abb$ n'est pas déterministe, puisque de l'état 0, avec la lettre a , on peut aller soit dans l'état 0 soit dans l'état 1.

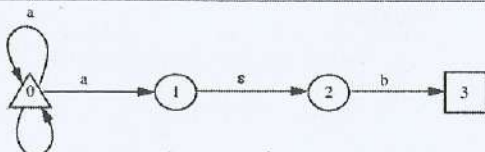
Les AFD sont plus faciles à simuler (pas de choix dans les transitions, donc jamais de retours en arrière à faire). Il existe des algorithmes permettant de **déterminiser** un automate non déterministe (c'est à dire de construire un AFD qui reconnaît le même langage que l'AFN¹ donné). L'AFD obtenu comporte en général plus d'états que l'AFN, donc le programme le simulant occupe plus de mémoire.

Exemple1



AFN car à partir de l'état 0
il y a deux arcs étiquetés 'a'.

Exemple2



car il existe une ϵ -transition.

Déterminisation d'un AFN ne contenant pas de ϵ -transition

Algorithme

1. Partir de l'état initial : $E = \{e_0\}$
2. Construire $E^{(1)}$ l'ensemble des états obtenus à partir de E par la transition a : $E^{(1)} = \Delta(E, a)$
3. Recommencer 2 pour toutes les transitions possibles et pour chaque nouvel ensemble d'état $E^{(i)}$
4. Tous les ensemble d'états $E^{(i)}$ contenant au moins un état terminal deviennent terminaux
5. Renommer alors les ensemble d'états en tant que simples états .

Exemple

état	a	b
0	0,2	1
1	3	0,2
2	3,4	2
3	2	1
4	-	3

$e_0 = 0$ et $T = \{2, 3\}$

Déterminisation d'un AFN ne contenant pas de ϵ transition

Solution

Définition

On appelle ε -fermeture de l'ensemble d'états $T = \{e_1, \dots, e_n\}$ l'ensemble des états accessibles depuis un état e_i de T par des ε -transitions

$$\varepsilon\text{-fermeture}(\{e_1, \dots, e_n\}) = \{e_1, \dots, e_n\} \cup \{e \text{ tq } \exists e_i \text{ avec } i = 1, 2, \dots, n \text{ tq } e_i \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} e\}$$

Calcul de ε transition

Mettre tous les états de T dans une pile P

Initialiser $\varepsilon\text{-fermeture}(T)$ à T

Tant que P est non vide faire

 Soit p l'état en sommet de P

 dépiler P

 Pour chaque état e tel qu'il y a une ε -transition entre p et e faire

 Si e n'est pas déjà dans $\varepsilon\text{-fermeture}(T)$

 ajouter e à $\varepsilon\text{-fermeture}(T)$

 empiler e dans P

 fini

finpour

fin tantque

Exemple

état	a	b	c	ε
0	2	-	0	1
1	3	4	-	-
2	-	-	1,4	0
3	-	1	-	-
4	-	-	3	2

$$e_0 = 0 \text{ et } T = \{4\}$$

Déterminisation d'un AFN contenant de ϵ -transition

Algorithme

1. Partir de l' ϵ -fermeture de l'état initial
2. Rajouter dans la table de transition toutes les ϵ -fermetures des nouveaux "états" produits, avec leurs transitions
3. Recommencer 2 jusqu'à ce qu'il n'y ait plus de nouvel "état"
4. Tous les "états" contenant au moins un état terminal deviennent terminaux
5. Renommer alors les états.

Exemple

état	a	b	c	ϵ
0	2	-	0	1
1	3	4	-	-
2	-	-	1,4	0
3	-	1	-	-
4	-	-	3	2

$e_0 = 0$ et $T = \{4\}$

Minimisation d'un AFN

Objectif

obtenir un automate ayant le minimum d'états possible.

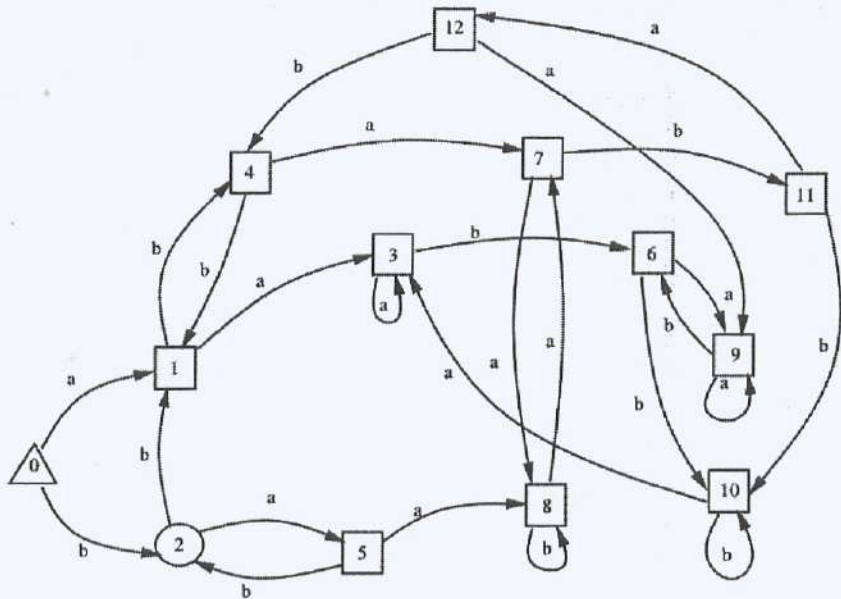
on définit des classes d'équivalence d'états par raffinements successifs. Chaque classe d'équivalence obtenue forme un seul même état du nouvel automate.

Algorithme

- 1 - Faire deux classes : A contenant les états terminaux et B contenant les états non terminaux.
- 2 - S'il existe un symbole a et deux états e_1 et e_2 d'une même classe tels que $\Delta(e_1, a)$ et $\Delta(e_2, a)$ n'appartiennent pas à la même classe, alors créer une nouvelle classe et séparer e_1 et e_2 . On laisse dans la même classe tous les états qui donnent un état d'arrivée dans la même classe.
- 3 - Recommencer 2 jusqu'à ce qu'il n'y ait plus de classes à séparer.
- 4 - Chaque classe restante forme un état du nouvel automate

Exemple (voir diapo 13)

Exemple



Calcul d'une ER à partir d'un AEF

Définition

On appelle L_i le langage que reconnaîtrait l'automate si e_i était son état initial. On peut alors écrire un système d'équations liant tous les L_i :

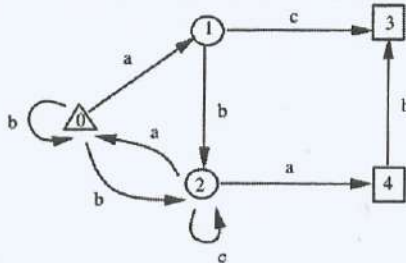
- chaque transition $\Delta(e_i, a) = e_j$ permet d'écrire l'équation $L_i = aL_j$
- pour chaque $e_i \in T$ on a l'équation $L_i = \epsilon$
- les équations $L_i = \alpha$ et $L_i = \beta$ se regroupent en $L_i = \alpha|\beta$

On résout ensuite le système (on cherche à calculer L_0) en remarquant juste que

Propriété

si $L = \alpha L|\beta$ alors $L = \alpha^*\beta$

Exemple (voir Slide 13)



Déterminisation d'un AFN (cas général)

Définition

On appelle ϵ -fermeture de l'ensemble d'états $T = \{e_1, \dots, e_n\}$ l'ensemble des états accessibles depuis un état e_i de T par des ϵ -transitions

$$\epsilon\text{-fermeture}(\{e_1, \dots, e_n\}) = \{e_1, \dots, e_n\} \cup \{e \text{ tq } \exists e_i \text{ avec } i = 1, 2, \dots, n \text{ tq } e_i \xrightarrow{\epsilon} \xrightarrow{\epsilon} \xrightarrow{\epsilon} \dots \xrightarrow{\epsilon} e\}$$

Exemple

Analyseur syntaxique

Problématique

Préambule

Tout langage de programmation possède des règles qui indiquent la structure syntaxique d'un programme bien formé. Par exemple, en Pascal, un programme bien formé est composé de blocs, un bloc est formé d'instructions, une instruction de ...

La **syntaxe** d'un langage peut être décrite par une **grammaire**. Cette grammaire décrit comment les unités lexicales doivent être agencées.

L'analyseur syntaxique reçoit une suite d'unités lexicales de la part de l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage.

Le problème est donc

- étant donnée une grammaire G
 - étant donné un mot m (un programme)
- \implies est ce que m appartient au langage généré par G ?

Le principe est d'essayer de construire un **arbre de dérivation**. Il existe deux méthodes pour cette construction : méthode (analyse) descendante et méthode (analyse) ascendante.

Grammaires et arbres de dérivation

La grammaire par les exemples

Exemples (définitions informelles de grammaires) :

dans le langage naturel, une phrase est composée d'un sujet suivi d'un verbe suivi d'un complément (pour simplifier ...).

Par exemple : L'étudiant subit un cours

On dira donc :

phrase = *sujet* *verbe* *complément*

Ensuite, il faut expliquer ce qu'est un *sujet*, un *verbe*, un *complément*. Par exemple :

sujet = *article* *adjectif* *nom*

| *article* *nom* *adjectif*

| *article* *nom*

article = *le* | *la* | *un* | *des* | *l'*

adjectif = *malin* | *stupide* | *couleur*

couleur = *vert* | *rouge* | *jaune*

ainsi de suite ...

une expression conditionnelle en C est : *if* (*expression*) *instruction*

Par exemple : *if* (*$x < 10$*) *$a = a + b$*

Il faut encore définir ce qu'est une *expression* et ce qu'est une *instruction* ...

On distingue les

- **symboles terminaux** : les lettres du langage (*le*, *la*, *if* ... dans les exemples)
- **symboles non-terminaux** : les symboles qu'il faut encore définir (ceux en italique dans les exemples précédents)

Grammaires et arbres de dérivation

Définition 1

Une grammaire est la donnée de $G = (V_T, V_N, S, P)$ où

- V_T est un ensemble non vide de symboles terminaux (alphabet terminal)
- V_N est un ensemble de symboles non-terminaux, avec $V_T \cap V_N = \emptyset$
- S est un symbole initial $\in V_N$ appelé axiome
- P est un ensemble de règles de productions (règles de réécritures)

Définition 2

Une règle de production $\alpha \rightarrow \beta$ précise que la séquence de symboles α ($\alpha \in (V_T \cup V_N)^+$) peut être remplacée par la séquence de symboles β ($\beta \in (V_T \cup V_N)^*$).
 α est appelée partie gauche de la production, et β partie droite de la production.

Exemple 1

symboles terminaux (alphabet) : $V_T = \{a, b\}$

symboles non-terminaux : $V_N = \{S\}$

axiome : S

règles de production :

$$\begin{cases} S \rightarrow \epsilon \\ S \rightarrow aSb \end{cases} \text{ qui se résument en } S \rightarrow \epsilon \mid aSb$$

Exemple 2

$G = \langle V_T, V_N, S, P \rangle$ avec

$V_T = \{ \text{il, elle, parle, est, devient, court, reste, sympa, vite} \}$

$V_N = \{ \text{PHRASE, PRONOM, VERBE, COMPLEMENT, VERBETAT, VERBACTION} \}$

$S = \text{PHRASE}$

$P = \{ \text{PHRASE} \rightarrow \text{PRONOM VERBE COMPLEMENT}$

$\text{PRONOM} \rightarrow \text{il} \mid \text{elle}$

$\text{VERBE} \rightarrow \text{VERBETAT} \mid \text{VERBACTION}$

$\text{VERBETAT} \rightarrow \text{est} \mid \text{devient} \mid \text{reste}$

$\text{VERBACTION} \rightarrow \text{parle} \mid \text{court}$

$\text{COMPLEMENT} \rightarrow \text{sympa} \mid \text{vite}$

Arbre de dérivation

Définition 1

On appelle **dérivation** l'application d'une ou plusieurs règles à partir d'un mot de $(V_T \cup V_N)^+$.

On notera \rightarrow une dérivation obtenue par application d'une seule règle de production, $\xrightarrow{*}$ une dérivation obtenue par l'application de n règles de production, où $n \geq 0$ et $\xrightarrow{+}$ une dérivation obtenue par l'application de n règles de production, où $n > 0$

Exemple 1

sur la grammaire de l'exemple 1

$$S \rightarrow \epsilon$$

$$S \rightarrow aSb$$

$$aSb \rightarrow aaSbb$$

$$S \xrightarrow{*} ab$$

$$S \xrightarrow{*} aaabbb$$

$$S \xrightarrow{*} aaSbb$$

Exemple 2

sur la grammaire de l'exemple 2

PHRASE \rightarrow PRONOM VERBE COMPLEMENT $\xrightarrow{*}$ elle VERBETAT sympa

PHRASE $\xrightarrow{*}$ il parle vite

PHRASE $\xrightarrow{*}$ elle court sympa

Arbre de dérivation

Définition 2

Etant donnée une grammaire G , on note $L(G)$ le langage généré par G et défini par

$\{w \in (V_T)^* \mid \text{tq } S \rightarrow w\}$

(c'est à dire tous les mots composés uniquement de symboles terminaux (de lettres de l'alphabet) que l'on peut former à partir de S).

L'exemple 1 nous donne $L(G) = \{a^n b^n, n \geq 0\}$

Définition 3

On appelle **arbre de dérivation** (ou **arbre syntaxique**) tout arbre tel que

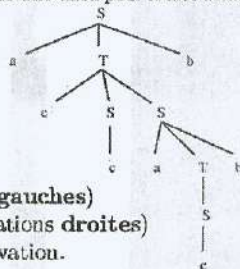
- la racine est l'axiome
- les feuilles sont des symboles terminaux
- les noeuds sont des symboles non-terminaux
- les fils d'un noeud X sont β_0, \dots, β_n si et seulement si $X \rightarrow \beta_0 \dots \beta_n$ est une production (avec $\beta_i \in V_T \cup V_N$)

Exemple

Soit la grammaire ayant S pour axiome et pour règles de production

$$P = \begin{cases} S \rightarrow aTb \mid c \\ T \rightarrow cSS \mid S \end{cases}$$

Un arbre de dérivation pour le mot $accacbb$ est



$S \rightarrow aTb \rightarrow acSSb \rightarrow accSb \rightarrow accaTbb \rightarrow accaSbb \rightarrow accacbb$ (dérivations gauches)

ou $S \rightarrow aTb \rightarrow acSSb \rightarrow acSaTbb \rightarrow acSaSbb \rightarrow acSacbb \rightarrow accacbb$ (dérivations droites)

Ces deux suites **différentes** de dérivations donnent le **même** arbre de dérivation.

Arbre de dérivation

Définition 3

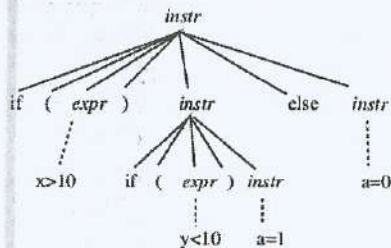
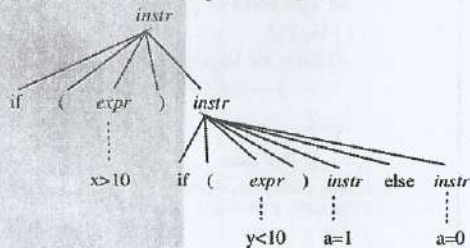
Une grammaire est dite ambiguë s'il existe un mot de $L(G)$ ayant plusieurs arbres syntaxiques.

Remarque : la grammaire précédente n'est pas ambiguë.

Exemple : Soit G donnée par

$$\begin{cases} instr \rightarrow \text{if } (expr) \text{ instr else instr} \\ instr \rightarrow \text{if } (expr) \text{ instr} \\ instr \rightarrow \dots \\ expr \rightarrow \dots \end{cases}$$

Cette grammaire est ambiguë car le mot $m = \text{if } (x > 10) \text{ if } (y < 0) a = 1 \text{ else } a = 0$ possède deux arbres syntaxiques différents :



car il y a deux interprétations syntaxiques possibles :

Mise en œuvre d'un analyseur syntaxique

Analyseur syntaxique

L'analyseur syntaxique reçoit une suite d'unités lexicales (de symboles terminaux) de la part de l'analyseur lexical. Il doit dire si cette suite (ce mot) est syntaxiquement correct, c'est à dire si c'est un mot du langage généré par la grammaire qu'il possède. Il doit donc essayer de construire l'arbre de dérivation de ce mot. S'il y arrive, alors le mot est syntaxiquement correct, sinon il est incorrect.

Il existe deux approches (deux méthodes) pour construire cet arbre de dérivation : une méthode descendante et une méthode ascendante.

Analyse descendante

Principe

construire l'arbre de dérivation du haut (la racine, c'est à dire l'axiome de départ) vers le bas (les feuilles, c'est à dire les unités lexicales).

On se retrouve avec



$w = \text{acb}$

Exemple 1

$$\begin{cases} S \rightarrow aAb \\ A \rightarrow cd|c \end{cases} \quad \text{avec le mot } w = acb$$

En lisant le c , on ne sait pas s'il faut prendre la règle $A \rightarrow cd$ ou la règle $A \rightarrow c$. Pour le savoir, il faut lire aussi la lettre suivante (b). Ou alors, il faut se donner la possibilité de faire des **retour en arrière** : on essaye la 1ère règle ($A \rightarrow cd$), on aboutit à un échec, alors on retourne en arrière et on essaye la deuxième règle et là ça marche.

Conclusion

ce qui serait pratique, ça serait d'avoir une table qui nous dit : quand je lis tel caractère et que j'en suis à dériver tel symbole non-terminal, alors j'applique telle règle et je ne me pose pas de questions. Ça existe, et ça s'appelle une **table d'analyse**.

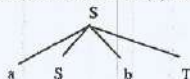
Analyse descendante

Principe

construire l'arbre de dérivation du haut (la racine, c'est à dire l'axiome de départ) vers le bas (les feuilles, c'est à dire les unités lexicales).

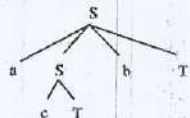
On part avec l'arbre contenant le seul sommet S

La lecture de la première lettre du mot (a) nous permet d'avancer la construction



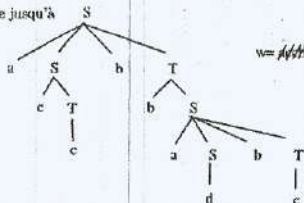
$w = \cancel{a}cbbadbc$

puis la deuxième lettre nous amène à



$w = \cancel{ac}bbadbc$

et ainsi de suite jusqu'à



$w = \cancel{accbbadbc}$

On a trouvé un arbre de dérivation, donc le mot appartient au langage.

Exemple 2

$$\begin{cases} S \rightarrow aSbT|cT|d \\ T \rightarrow aT|bS|c \end{cases}$$

avec le mot $w = accbbadbc$

Table d'analyse LL1

Définition

Pour construire une table d'analyse, on a besoin des ensembles PREMIER et SUIVANT

Calcul de Premier

Pour toute chaîne α composée de symboles terminaux et non-terminaux, on cherche $\text{PREMIER}(\alpha)$: l'ensemble de tous les terminaux (y compris ϵ) qui peuvent commencer une chaîne qui se dérive de α

C'est à dire que l'on cherche toutes les lettres a telles qu'il existe une dérivation $\alpha \xrightarrow{*} a\beta$ (β étant une chaîne quelconque composée de symboles terminaux et non-terminaux). Cas particulier : $\epsilon \in \text{PREMIER}(\alpha)$ si et seulement si il existe une dérivation $\alpha \xrightarrow{*} \epsilon$

Exemple

$$\begin{cases} S \rightarrow Ba \\ B \rightarrow cP | bP | P | \epsilon \\ P \rightarrow dS \end{cases}$$

$S \xrightarrow{*} a$, donc $a \in \text{PREMIER}(S)$ $S \xrightarrow{*} cPa$, donc $c \in \text{PREMIER}(S)$

$S \xrightarrow{*} bPa$, donc $b \in \text{PREMIER}(S)$ $S \xrightarrow{*} dSa$, donc $d \in \text{PREMIER}(S)$

Il n'y a pas de dérivation $S \xrightarrow{*} \epsilon$

Donc $\text{PREMIER}(S) = \{a, b, c, d\}$

$B \xrightarrow{*} dS$, donc $d \in \text{PREMIER}(B)$

$aB \xrightarrow{*} aa$, donc $\text{PREMIER}(aB) = \{a\}$

$BSb \xrightarrow{*} ab$, donc $a \in \text{PREMIER}(BS)$

Table d'analyse LL1 – calcul de Premier –

Algorithme de construction des ensembles Premier(X) pour $X \in (V_T \cup V_N)$

1. Si X est un non-terminal et $X \rightarrow Y_1 Y_2 \dots Y_n$ est une production de la grammaire (avec Y_i symbole terminal ou non-terminal) alors
 - ajouter les éléments de $\text{PREMIER}(Y_1)$ sauf ϵ dans $\text{PREMIER}(X)$
 - s'il existe j ($j \in \{2, \dots, n\}$) tel que pour tout $i = 1, \dots, j - 1$ on a $\epsilon \in \text{PREMIER}(Y_i)$, alors ajouter les éléments de $\text{PREMIER}(Y_j)$ sauf ϵ dans $\text{PREMIER}(X)$
 - si pour tout $i = 1, \dots, n$ $\epsilon \in \text{PREMIER}(Y_i)$, alors ajouter ϵ dans $\text{PREMIER}(X)$
 2. Si X est un non-terminal et $X \rightarrow \epsilon$ est une production, ajouter ϵ dans $\text{PREMIER}(X)$
 3. Si X est un terminal, $\text{PREMIER}(X) = \{X\}$.
- Recommencer jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles PREMIER .

Table d'analyse LLI - calcul de Premier -

Algorithme de construction des ensembles $\text{Premier}(\alpha)$ pour $\alpha \in (V_T \cup V_N)^*$

On a $\alpha = Y_1 Y_2 \dots Y_n$ avec $Y_i \in V_T$ ou $Y_i \in V_N$

si Y_1 est un symbole terminal alors ajouter Y_1 aux **PREMIER**(α)

SINOI

// Y_1 est un symbole non terminal

ajouter les éléments de $\text{PREMIER}(Y_1)$ sauf ε dans $\text{PREMIER}(\alpha)$

si $\epsilon \in \text{PREMIER}(Y_1)$ alors

// faire la même chose avec Y_2

si Y_2 est un symbole terminal alors ajouter Y_2 aux PREMIER(α)

sinon

// Y_2 est un symbole non terminal

ajouter les éléments de $\text{PREMIER}(Y_2)$ sauf ε dans $\text{PREMIER}(\alpha)$

si $c \in \text{PREMIER}(Y_2)$ alors

// faire la même chose avec Y_3

...

// faire la même chose avec Y_n

si $Y_n \in V_T$ alors ajouter Y_n aux $\text{PREMIERS}(\alpha)$

sincron

// Y_n est un symbole non terminal

ajouter les $\text{PREMIER}(Y_n)$ sauf ε aux $\text{PREMIER}(\alpha)$

si $\epsilon \in \text{PREMIER}(Y_n)$ alors

ajouter ε aux PREMIER(α)

finis

finsi

2

finsi

નિમ્ન

finisi

firsi

Table d'analyse LLI - calcul de Premier -

Exemple 1

On considère la grammaire suivante, calculer les premiers de chaque symbole

Non terminal

$$\begin{cases} E \rightarrow TE' \\ E' \rightarrow +TE' \mid -TE' \mid \varepsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid /FT' \mid \varepsilon \\ F \rightarrow (E) \mid \text{nb} \end{cases}$$

Exemple 2

On considère la grammaire suivante, calculer les premiers de chaque symbole

Non terminal

$$\begin{cases} S \rightarrow ABCe \\ A \rightarrow aA \mid \varepsilon \\ B \rightarrow bB \mid cB \mid \varepsilon \\ C \rightarrow de \mid da \mid dA \end{cases}$$

Table d'analyse LL1 – calcul de Suivant –

Définition

Pour tout non-terminal A , on cherche $\text{SUIVANT}(A)$: l'ensemble de tous les symboles terminaux a qui peuvent apparaître immédiatement à droite de A dans une dérivation : $S \xrightarrow{*} \alpha A a \beta$

Exemple

$$\begin{cases} S \rightarrow Sc|Ba \\ B \rightarrow BP a|bPb|P|\varepsilon \\ P \rightarrow dS \end{cases} \quad \begin{array}{l} a, b, c, d \in \text{SUIVANT}(S) \text{ car il y a les dérivations } S \xrightarrow{*} Sc, S \xrightarrow{*} dSa, S \xrightarrow{*} bdSba \text{ et } S \xrightarrow{*} dSdSaa \\ d \in \text{SUIVANT}(B) \text{ car } S \xrightarrow{*} BdSaa \end{array}$$

Algorithme de construction des ensembles Suivant

1. Ajouter un marqueur de fin de chaîne (symbole \$ par exemple) à $\text{SUIVANT}(S)$
(où S est l'axiome de départ de la grammaire)
 2. Pour chaque production $A \rightarrow \alpha B \beta$ où B est un non-terminal, alors ajouter le contenu de $\text{PREMIER}(\beta)$ à $\text{SUIVANT}(B)$, **sauf** ε
 3. Pour chaque production $A \rightarrow \alpha B$, alors ajouter $\text{SUIVANT}(A)$ à $\text{SUIVANT}(B)$
 4. Pour chaque production $A \rightarrow \alpha B \beta$ avec $\varepsilon \in \text{PREMIER}(\beta)$, ajouter $\text{SUIVANT}(A)$ à $\text{SUIVANT}(B)$
- Recommencer à partir de l'étape 3 jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles SUIVANT .

Table d'analyse LL1 – calcul de Suivant –

Exemple 1

On considère la grammaire suivante, calculer les suivants de chaque symbole

Non terminal

$$\begin{cases} E \rightarrow TE' \\ E' \rightarrow +TE' \mid -TE' \mid \varepsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid /FT' \mid \varepsilon \\ F \rightarrow (E) \mid \text{nb} \end{cases}$$

Exemple 2

On considère la grammaire suivante, calculer les suivants de chaque symbole

Non terminal

$$\begin{cases} S \rightarrow aSb \mid cd \mid SAe \\ A \rightarrow aAdB \mid \varepsilon \\ B \rightarrow bb \end{cases}$$

Table d'analyse – Construction de la table LL1 –

Définition

Une table d'analyse est un tableau M à deux dimensions qui indique pour chaque symbole non-terminal A et chaque symbole terminal a ou symbole $\$$ la règle de production à appliquer.

Algorithme de construction de la table d'analyse

- Pour chaque production $A \rightarrow \alpha$ faire
 1. pour tout $a \in \text{PREMIER}(\alpha)$ (et $a \neq \epsilon$), rajouter la production $A \rightarrow \alpha$ dans la case $M[A, a]$
 2. si $\epsilon \in \text{PREMIER}(\alpha)$, alors pour chaque $b \in \text{SUIVANT}(A)$ ajouter $A \rightarrow \alpha$ dans $M[A, b]$
- Chaque case $M[A, a]$ vide est une erreur de syntaxe

Exemple : Compléter la table suivante

$$\begin{cases} E \rightarrow TE' \\ E' \rightarrow +TE' \mid -TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid /FT' \mid \epsilon \\ F \rightarrow (E) \mid \text{nb} \end{cases}$$

	nb	+	-	*	/	()	\$
E	$E \rightarrow TE'$							
E'								
T	$T \rightarrow FT'$							
T'								
F	$F \rightarrow \text{nb}$							