Automates et langages et applications

Eric Alata eric.alata@laas.fr

INSA Toulouse - 4ième IR

18 janvier 2015

Introduction

Les langages

Les grammaires

Les langages naturels

- Employé par les êtres humains pour communiquer
- Ambiguïté – plusieurs sens possibles

j'accompagne les étudiants du département au secrétariat

- Incohérence – déductions nécessaires pour comprendre l'enchaînement des idées[WG06]

Coherent example text

a. The weather at the rocket launch site in Kourou was good yesterday.
b. Therefore, the launch of the new Ariane rocket could take place as scheduled.
c. The rocket carried two test satellites into orbit.

Incoherent example text

a. A new communications satellite was launched.
b. Therefore, Mary likes spinach.
c. John stayed home in bed.

- Problème pour communiquer précisément, ex : aérospatial

⇒ *Simplified Technical English (STE)* – ASD-STE100[Aer05]

  - *AeroSpace and Defence Industries Association of Europe*
  - Objectif : réduction de l'ambiguïté
  - Règles de construction des phrases et dictionnaire
  - Employé pour la définition des exigences
    Requirement Based Engineering (RBE)

Simplified Technical English – Extrait des règles

List of Writing Rules

SECTION 1 - WORDS

RULE: 1.1 Choose the words from:
- Approved words in the Dictionary (Part 2)
- Words that qualify as Technical Names (Refer to Rule 1.5)
- Words that qualify as Technical Verbs (Refer to Rule 1.10).

RULE: 1.2 Use approved words from the Dictionary only as the part of speech given.

RULE: 1.3 Keep to the approved meaning of a word in the Dictionary. Do not use the word with any other meaning.

RULE: 1.4 Only use those forms of verbs and adjectives shown in the Dictionary.

RULE: 1.5 You can use words that are Technical Names.

RULE: 1.6 Use a Technical Name only as a noun or an adjective, not as a verb.

RULE: 1.6A Some unapproved words are used to complete Technical Names. Do not use these unapproved words unless they are part of a Technical Name.

RULE: 1.7 Use the official name (shortened if necessary).

RULE: 1.8 Do not use different Technical Names for the same thing.

RULE: 1.9 If you have a choice, use the shortest and simplest name.

RULE: 1.10 You can use words that are Technical Verbs.

RULE: 1.11 Use Technical Verbs only as verbs, not as nouns (unless the noun form qualifies as a Technical Name). You can use the past participle of the verb as an adjective (refer to Section 3).

RULE: 1.12 Once you choose the words to describe something, continue to use these same words (particularly Technical Names).

RULE: 1.13 Make your instructions as specific as possible.

RULE: 1.14 Use consistent spelling.

Simplified Technical English – Extrait du dictionnaire

ASD-STE100

Keyword (part of speech)	Assigned Meaning/ USE	APPROVED EXAMPLE	Not Acceptable
A (art)	Function word: Indefinite article	A FUEL PUMP IS INSTALLED IN ZONE XXXX.	
abaft (pre)	AFT OF	THE CONTROL UNIT IS INSTALLED AFT OF THE FLIGHT COMPARTMENT	The control unit is installed abaft the flight compartment.
abandon (v)	STOP	STOP THE ENGINE START PROCEDURE.	Abandon engine start.
abate (v)	DECREASE	WHEN THE WIND SPEED DECREASES TO BELOW 30 KNOTS, YOU CAN OPEN THE CARGO DOOR.	When the wind abates to below 30 knots, you can open the cargo door.
ability (n)	CAN (v)	ONE GENERATOR CAN SUPPLY POWER FOR ALL THE SYSTEMS.	One generator has the ability to supply power for all the systems.
able (adj)	CAN (v)	IF YOU CAN START THE ENGINE, DO A BITE TEST.	If you are able to start the engine, do a BITE test.
abnormal (adj)	UNUSUAL, INCORRECT	LISTEN FOR UNUSUAL NOISES.	Check for abnormal noises.
		IF YOU FIND THAT THE QUANTITY OF AIR FROM THE VENT MAST IS INCORRECT, DO A SYSTEM TEST.	If abnormal air escape from the vent mast is noted, do a system test.

Les langages formels

- Intuitivement
  - Ensemble de règles permettant de communiquer sans ambiguïté
  - Ensemble de mots admissibles
- Grammaire $\rightarrow$ formalisme
- Exemples d'utilisation des langages formels
  - Vérification de syntaxe
    Est-ce que mon programme C *est syntaxiquement correct ?*
  - Analyse de fichiers de configuration
    Lecture des fichiers INI, *des signatures de* SNORT
  - Traduction de documents
    XML $\rightarrow$ DOC   C $\rightarrow$ *binaire*
  - Plus généralement, problèmes de décision
    Est-ce que [X] appartient à [Y] ?
  - Reconnaissance de signatures d'attaques
- Suite du cours $\Rightarrow$ exclusivement les langages formels

Introduction

Les langages

Les grammaires

Alphabet, mot et langage

Alphabet

Un alphabet Σ est un ensemble fini non vide de caractères

- Caractères alphabétiques : $\Sigma_a = \{a, b, c, d, \ldots, z\}$
- Alphabet des nombres : $\Sigma_n = \{0, 1, 2, 3, \ldots, 9, \underline{.}\}$

Mot

Un mot ω est une suite de caractères d'un alphabet Σ, ϵ est le mot vide

- Mots sur Σ_a : *aaaa*, *truc*, *qsfaer*
- Mots sur Σ_n : 12, 011102.123, 1234.5678.9

Langage

Un ensemble de mots sur un alphabet Σ définit un langage L
L'ensemble de tous les mots sur un alphabet Σ est noté Σ^* $\quad L \subseteq \Sigma^*$

- Langage sur Σ_n : $L_n = \{12, 1235, 15.4\} \subset \Sigma_n^*$

Opérations sur les mots

- Soient Σ un alphabet, (ω, x, y) des mots sur Σ et Σ_p une partie de Σ
- Longueur
  - $|\omega|$ = longueur du mot ω = nombre de caractères qu'il contient $\qquad |\epsilon| = 0$
  - $|\omega|_a$ = nombre d'occurences du caractère a dans le mot ω
  - $|\omega|_{\Sigma_p}$ = nombre d'occurences des caractères de Σ_p dans ω
- Manipulation des caractères d'un mot
  - Le i-ième caractère d'un mot est noté ω_i
  - Le sous-mot de ω correspondant à $\omega_i\omega_{i+1}\ldots\omega_j$ avec $i \leq j$ est noté $\omega_{[i;j]}$
  - Le mot miroir de $\omega = a_1a_2\ldots a_n$ est $\tilde{\omega} = a_n\ldots a_2a_1$
- Concaténation
  - $x = a_1a_2\ldots a_n$ et $y = b_1b_2\ldots b_m \Rightarrow xy = a_1a_2\ldots a_nb_1b_2\ldots b_m$
  - $|xy| = |x| + |y|$
  - $\epsilon\omega = \omega\epsilon = \omega$
  - $\omega^0 = \epsilon \qquad \omega^k = \omega^{(k-1)}\omega \qquad \omega^2 = \omega\omega \qquad \omega^k = \underbrace{\omega\omega\cdots\omega}_{k \text{ copies de } \omega}$

Opérations sur les mots

Préfixe

u est un préfixe de $\omega \in \Sigma^*$ s'il existe $v \in \Sigma^*$ tel que $uv = \omega$

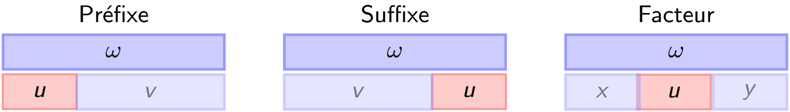
- Ensemble des préfixes : $Pref(\omega) = \{u | u \in \Sigma^*, \exists v \in \Sigma^*, uv = \omega\}$
- Ensemble des préfixes communs : $Pc(x, y) = Pref(x) \cap Pref(y)$
- Plus long préfixe commun : $Plpc(x, y) = \arg \max_{u \in Pc(x,y)} |u|$

Suffixe

u est un suffixe de $\omega \in \Sigma^*$ s'il existe $v \in \Sigma^*$ tel que $vu = \omega$

Facteur

u est un facteur de $\omega \in \Sigma^*$ s'il existe $x \in \Sigma^*$ et $y \in \Sigma^*$ tels que $xuy = \omega$

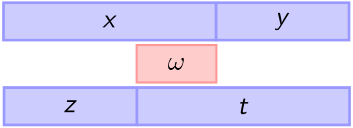


Opérations sur les mots

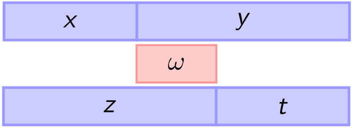
Lemme de Levy

Soient x, y, z et t des mots de Σ^*

$$xy = zt \Leftrightarrow \exists \omega \in \Sigma^* \text{ t.q. } \begin{cases} x\omega = z \quad \wedge \quad y = \omega t, \text{ ou} \\ x = z\omega \quad \wedge \quad \omega y = t \end{cases}$$

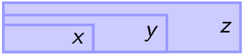


ou



Opérations sur les mots

- Relation d'ordre partiel
  - Préfixe, suffixe et facteur définissent des relations d'ordre sur les mots
  - Prenons l'exemple de la relation de préfixe, notée $\preceq_p$
  - Cette relation est réflexive car ω est le préfixe de lui-même $\qquad \omega \preceq_p \omega$
  - Cette relation est transitive car, si x est un préfixe de y qui, lui-même est un préfixe de z, alors x est un préfixe de z $\qquad x \preceq_p y \preceq_p z \Rightarrow x \preceq_p z$

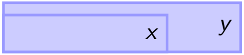


  - Cette relation est antisymétrique car, si x est un préfixe de y et y est un préfixe de x, alors x égale y $\qquad x \preceq_p y \wedge y \preceq_p x \Rightarrow x = y$
  - L'ordre défini par cette relation est partiel. Par exemple, le mot *abcd* n'est pas le préfixe de *efgh* et inversement. Cette particularité vient du fait que quelque soit $a_1 a_2 \in \Sigma^2$, a_1 et a_2 ne sont pas reliés par la notion de préfixe

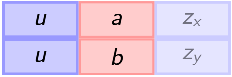
Opérations sur les mots

- Relation d'ordre total
  - Si une relation notée $\preceq_x$ définit un ordre total sur Σ, alors il est possible de définir un ordre total sur Σ^*
  - Premier exemple d'ordre total : l'ordre lexicographique, noté $\preceq_l$

$$x \preceq_l y \Rightarrow \begin{cases} x \preceq_p y, \text{ ou} \\ \exists (u, z_x, z_y) \in (\Sigma^*)^3, (a, b) \in \Sigma^2 \quad \text{t.q.} \\ \quad x = uaz_x \wedge y = ubz_y \wedge a \preceq_x b \wedge b \npreceq_x a \end{cases}$$



ou



$a \preceq_x b \wedge b \npreceq_x a$

  - Cet ordre est celui des dictionnaires
  - *antisymetrique* $\preceq_l$ *truc* et *aaaaaaaaaaaaa* $\preceq_l$ *b* ? !
- Deuxième exemple d'ordre total : l'ordre alphabétique, noté $\preceq_a$

$$x \preceq_a y \Rightarrow \begin{cases} |x| < |y|, \text{ ou} \\ |x| = |y| \wedge x \preceq_l y \end{cases}$$

Opérations sur les mots

- Distances entre mots – distance de préfixe
  - Définissons une distance qui s'appuit sur la notion de préfixe, notée $d_p(x, y)$
  - $d_p(x, y) = |xy| - 2 \times |Plpc(x, y)|$
  - Exemple avec les mots *voile* et *voisin*
    - $Pc(voile, voisin) = \{\epsilon, v, vo, voi\}$
    - $Plpc(voile, voisin) = voi$
    - $d_p(voile, voisin) = |voilevoisin| - 2 \times |voi| = 11 - 2 \times 3 = 5$
  - Intuitivement : *on soustrait le plus long préfixe aux deux mots et le nombre de caractères restants correspond à la distance entre ces deux mots*
  - $d_p(x, y)$ est effectivement une distance
    - $d_p(x, y) \geq 0$
    - $d_p(x, y) = 0 \Rightarrow x = y$
    - $d_p(x, y) \leq d_p(x, \omega) + d_p(\omega, y)$

Opérations sur les mots

- Distances entre mots – distance de Levenstein
  - Un script d'édition (*edit script*) correspond à une succession d'opérations sur les mots permettant de passer d'un mot x à un mot y
  - Les opérations sont :
    - $insert(i, a)$ pour insérer le caractère a à l'indice i
    - $delete(i)$ pour supprimer le caractère à l'indice i
    - $replace(i, a)$ pour remplacer le caractère à l'indice i par le caractère a
  - Notons que $(replace(i, a)) \equiv (delete(i), insert(i, a))$
  - Chaque opération a un coût, par défaut 1
  - Exemple : $avion \rightarrow (delete(1), insert(3, s), insert(4, i)) \rightarrow vision$
  - La longueur du script d'édition correspond à la distance d'édition
  - Commande `diff`[HM76] de GNU
  - Peut-on employer cette distance pour créer un système de plagiat ?

Opérations sur les mots

- Distances entre mots – application
  - Comment trouver le plus petit script d'édition permettant de passer du mot *visionner* au mot *voisin* ?
  - La stratégie consiste à supposer que $d_e(i,j)$ représente la distance d'édition entre les mots $x_{[1;i]}$ et $y_{[1;j]}$ et trouver une récurrence pour $d_e(i,j)$ permettant d'obtenir $d_e(|x|,|y|)$

$$d_e(i,j) = \begin{cases} i+j \text{ si } \quad j=0 \vee i=0 \\ \min(d_e(i-1,j)+1, d_e(i,j-1)+1, d_e(i-1,j-1)+1) \text{ si } x_i \neq y_j \\ \min(d_e(i-1,j)+1, d_e(i,j-1)+1, d_e(i-1,j-1)) \text{ sinon} \end{cases}$$

  - Un script d'édition correspond à un parcourt de la matrice d_e croissant sur les indices i et j

		y_1	$\cdots$	y_n
	$d_e(0,0)$	$d_e(0,1)$	$\cdots$	$d_e(0,n)$
x_1	$d_e(1,0)$	$d_e(1,1)$	$\cdots$	$d_e(1,n)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x_m	$d_e(m,0)$	$d_e(m,1)$	$\cdots$	$d_e(m,n)$

$d_e(i-1,j-1)$	$d_e(i-1,j)$
$d_e(i,j-1)$	$d_e(i,j)$

Opérations sur les mots

- Distances entre mots – application ../..
  - Comment, à partir de la matrice d_e, obtenir le script d'édition ?
  - ⇒ Traverser cette matrice en suivant la "diagonale" qui part du bas à droite et remonte en haut à gauche, en suivant les $d_e(i, j)$ décroissants
  - Pourquoi cette stratégie est-elle viable ?
  - Ce plus petit script d'édition est-il unique ?

```python
a = "visionner"
b = "voisin"
# Construction de la matrice.
m = len(a) + 1
n = len(b) + 1
d = [[0] * n for i in range(m)]
for i in range(0, n):
  d[0][i] = i
for i in range(0, m):
  d[i][0] = i
for i in range(1, m):
  for j in range(1, n):
    if a[i - 1] == b[j - 1]:
      d[i][j] = min(d[i-1][j]+1, d[i][j-1]+1, d[i
              -1][j-1])
    else:
      d[i][j] = min(d[i-1][j]+1, d[i][j-1]+1, d[i
              -1][j-1]+1)
```

Listing 1: pEditScript.py

```python
# Recuperation du script (i et j vallent deja m-1
        et n-1).
s = []
while i > 0 and j > 0:
  x = min(d[i-1][j], d[i][j-1], d[i-1][j-1])
  if d[i-1][j-1] == x:
    if d[i-1][j-1] != d[i][j]:
      s = ["replace(%d,%c)" % (j, b[j-1])] + s
    i = i - 1
    j = j - 1
  elif d[i][j-1] == x:
    s = ["insert(%d,%c)" % (j, b[j-1])] + s
    j = j - 1
  elif d[i-1][j] == x:
    s = ["delete(%d)" % (j)] + s
    i = i - 1
```

Listing 2: pEditScript.py

Opérations sur les mots

- Distances entre mots – application ../..

		v	o	i	s	i	n
	0	1	2	3	4	5	6
v	1	0	1	2	3	4	5
i	2	1	1	1	2	3	4
s	3	2	2	2	1	2	3
i	4	3	3	2	2	1	2
o	5	4	3	3	3	2	2
n	6	5	4	4	4	3	2
n	7	6	5	5	5	4	3
e	8	7	6	6	6	5	4
r	9	8	7	7	7	6	5

Script d'édition : (insert(2,o),delete(5),delete(6),delete(6),delete(6))

Opérations sur les mots

- Distances entre mots – application *../..*
  - Utilisation en littérature pour identifier l'appartenance de textes (Molière ou Corneille ?)[LL00]
  - Amélioration possible : *Four Russian*[ADKF70]
  - Construction d'une base complète de blocs de taille donnée
  - Découpage de la matrice d_e en sous-matrices de même taille, avec chevauchement d'une ligne et d'une colonne
  - Résolution de ces sous-matrices en utilisant la base de blocs
  - Le cœur des sous-matrices n'est pas à calculer !
  - Exemple pour l'alphabet $\{a, b\}$ et une taille de 3

$$
\begin{array}{c|c|c|c|}
 & & \text{a} & \text{a} \\ \hline
 & 0 & 1 & 2 \\ \hline
\text{a} & 1 & 0 & 1 \\ \hline
\text{a} & 2 & 1 & 0 \\ \hline
\end{array}
\qquad
\begin{array}{c|c|c|c|}
 & & \text{a} & \text{b} \\ \hline
 & 0 & 1 & 2 \\ \hline
\text{a} & 1 & 0 & 1 \\ \hline
\text{a} & 2 & 1 & 1 \\ \hline
\end{array}
\qquad
\begin{array}{c|c|c|c|}
 & & \text{a} & \text{b} \\ \hline
 & 0 & 1 & 2 \\ \hline
\text{a} & 1 & 0 & 1 \\ \hline
\text{b} & 2 & 1 & 0 \\ \hline
\end{array}
\qquad
\begin{array}{c|c|c|c|}
 & & \text{b} & \text{b} \\ \hline
 & 0 & 1 & 2 \\ \hline
\text{a} & 1 & 1 & 2 \\ \hline
\text{b} & 2 & 1 & 1 \\ \hline
\end{array}
\;\ldots
$$

Opérations sur les langages

- Ensemble de tous les mots – fermeture de Kleene
  - L'ensemble des mots sur Σ est noté Σ^*
  - L'ensemble des mots non vides sur Σ est noté Σ^+ $\quad \Sigma^+ = \Sigma^*/\epsilon$
- Les langages sont des ensembles de mots – soient L_1 et L_2
  - $L_1 \subseteq \Sigma^*$ et $L_2 \subseteq \Sigma^*$
  - Union de langages $\quad L_1 \cup L_2 = \{\omega | \omega \in L_1 \vee \omega \in L_2\}$
  - Intersection de langages $\quad L_1 \cap L_2 = \{\omega | \omega \in L_1 \wedge \omega \in L_2\}$
  - Produit de langages $\quad L_1 L_2 = \{xy | x \in L_1 \wedge y \in L_2\}$
  - Complément d'un langage $\quad \overline{L_1} = \{\omega \in \Sigma^* | \omega \notin L_1\}$
  - Différence de langages $\quad L_1/L_2 = \{\omega \in \Sigma | \omega \in L_1 \wedge \omega \notin L_2\}$
  - Fermeture de Kleene d'un langage (L_1^*) $\quad L_1^* = \bigcup_{i \geq 0} L_1^i \quad L_1^+ = \bigcup_{i \geq 1} L_1^i$
- Extensions des notions de préfixe (et autres) sur les langages
  - Ensemble des préfixes d'un langage $\quad Pref(L) = \bigcup_{\omega \in L} Pref(\omega)$
  - Ensemble des suffixes d'un langage $\quad Suff(L) = \bigcup_{\omega \in L} Suff(\omega)$
  - Ensemble des facteurs d'un langage $\quad Fact(L) = \bigcup_{\omega \in L} Fact(\omega)$

Langages réguliers

- Les langages réguliers sont définis par induction
- Soit Σ un alphabet :
  - $\{\epsilon\}$ et $\emptyset$ sont réguliers
  - $\forall a \in \Sigma, \{a\}$ est régulier
  - Si L_1 et L_2 sont des langages réguliers,
    alors $L_1 \cup L_2$, $L_1 L_2$ et L_1^* sont aussi réguliers

Exercices

1. Soit $\Sigma = \{a, b\}$, que vaut Σ^* ?
2. Soit $\Sigma = \{a, b\}$, combien y-a-t-il de mots dans Σ^* ?
3. Soit Σ un alphabet et ω un mot de Σ^*. Que vaut $|\omega|_\Sigma$?
4. ϵ fait-il partie de l'alphabet Σ ?
5. Soit $\Sigma = \{a, \ldots, z\}$ et $\omega = abcdef$. Que vaut $|\omega|$?
6. Que vaut $|\omega^n|$?
7. Soient $\Sigma = \{0, 1, 2, \ldots, 9\}$, $\Sigma_p = \{0, 2, 4, 6, 8\}$, $\Sigma_i = \Sigma / \Sigma_p$ et un mot sur l'alphabet Σ, $\omega = 02163523$.
   Que vaut $|\omega|_{\Sigma_i}$?
8. Si $\omega = \tilde{\omega}$ alors quelle est la nature de ω ?
9. Donnez la fermeture de Kleene de $\{a, b, c\}$

Introduction

Les langages

Les grammaires

Formalisme

Grammaire

Une grammaire est un *4*-uplet $G = (T, N, R, S)$

- T : ensemble fini de symboles terminaux
  (alphabet terminal)
- N : ensemble fini de symboles non-terminaux
  (alphabet des variables)
- R : ensemble fini de règles de production
  ensemble de paires (α, β)
- S : symbole de départ
  axiome de la grammaire

- $T \cap N = \emptyset$
- $R \subset (T \cup N)^* \times (T \cup N)^*$
- $S \in N$

Formalisme

- Convention
  - Un symbole qui commence par une majuscule est un symbole non-terminal
  - Un symbole qui ne commence pas par une majuscule est un symbole terminal
  - Une lettre grecque est un élément de $(T \cup N)^*$
- Système de réécriture
  - Une grammaire $G = (T, N, R, S)$ est un ensemble de règles de réécriture
  - Les paires $(\alpha, \beta) \in R$ sont notées $\alpha \rightarrow \beta$
  - Le symbole [$\rightarrow$] signifie [*peut être remplacé par*]
  - Une règle $\alpha \rightarrow \beta$ peut être appliquée au mot $\gamma\alpha\delta$ en remplaçant α par β
  - L'application d'une règle $\alpha \rightarrow \beta$ à un mot $\gamma\alpha\delta$ est nommée dérivation et notée $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$
  - Si il existe $\alpha_2, \ldots, \alpha_{n-1} \in (T \cup N)^*$ tels que $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$ alors $\alpha_1 \Rightarrow^* \alpha_n$

Formalisme

- Graphe de dérivation
  - La dérivation d'un mot peut être représentée par un graphe orienté
  - Les nœuds correspondent aux symboles des mots
  - Les arcs connectent les symboles d'un mot participant à la dérivation des symboles, du mot dérivé, issus de cette dérivation
  - Croisements d'arrêtes possibles
  - Le mot du langage correspond à la suite ordonnée de gauche à droite des nœuds sans fils telle que tous les nœuds correspondent à des symboles terminaux
- Langage engendré
  - Le langage engendré par une grammaire $G = (T, N, R, S)$, noté $L_G(S)$, est l'ensemble des mots contenant uniquement des symboles terminaux, qui peuvent être dérivés par les règles R
  - $L_G(S) = \{\alpha \in T^* : S \Rightarrow^* \alpha\}$
- Utilité des grammaires
  - Vérifier qu'une phrase est valide (*parser*)
  - Générer des phrases valides
  - Vérifier des propriétés sur le langage

Exemple de grammaire – liste de prénoms

S=**Liste** T=\{**andre**, **liam**, **phil**, **et**, **,**\} N=\{**Liste**, **Prenom**, **Prenoms**, **Fin**\}

R_1	**Liste**	→	**Prenom**
R_2	**Liste**	→	**Prenoms Fin**
R_3	**Prenoms**	→	**Prenom**
R_4	**Prenoms**	→	**Prenom , Prenoms**
R_5	**, Prenom Fin**	→	**et Prenom**
R_6	**Prenom**	→	**andre**
R_7	**Prenom**	→	**liam**
R_8	**Prenom**	→	**phil**

- La règle R_2 signifie que **Liste** peut être remplacé par **Prenoms Fin**
- Il existe plusieurs façons de remplacer le symbole non-terminal **Prenoms** (cf. règles R_3 et R_4)
  $\Rightarrow$ il peut exister plusieurs dérivations pour un même mot
- Vérification : $T \cap N = \emptyset$
- Intuitivement on se rend compte que **phil liam , et** n'est pas un mot engendré par G, d'où la nécessité des *parsers*

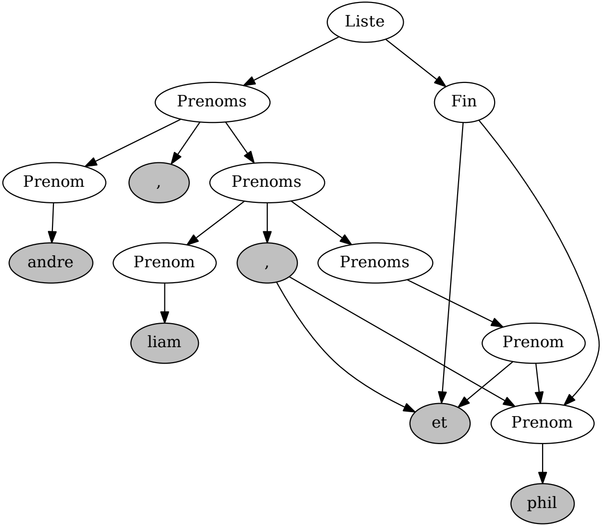
Exemple de grammaire – liste de prénoms

- Exemple de dérivation

$$
\begin{array}{lcl}
\textbf{Liste} & \Rightarrow & \textbf{Prenoms Fin} \\
 & \Rightarrow & \textbf{Prenom , Prenoms Fin} \\
 & \Rightarrow & \textbf{Prenom , Prenom , Prenoms Fin} \\
 & \Rightarrow & \textbf{Prenom , Prenom , Prenom Fin} \\
 & \Rightarrow & \textbf{Prenom , Prenom et Prenom} \\
 & \Rightarrow & \textbf{Prenom , Prenom et phil} \\
 & \Rightarrow & \textbf{Prenom , liam et phil} \\
 & \Rightarrow & \textbf{andre , liam et phil}
\end{array}
$$

Exemple de grammaire – liste de prénoms

- Exemple de graphe de dérivation



Programme de génération de mots d'une grammaire

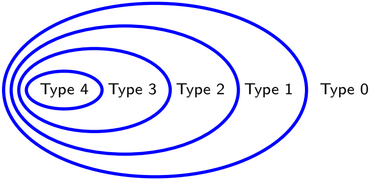
Figure – Algorithme

function GénérationMots(G) $\triangleright$ $G = (T, N, R, S)$
  liste $\leftarrow \{S\}$
  mots $\leftarrow \{\}$
  while $|\text{liste}| > 0$ **do**
    $\alpha \leftarrow \text{liste}_0$
    if $\alpha \in T^*$ **then**
      mots $\leftarrow$ mots $\cup \{\alpha\}$
    else
      liste $\leftarrow (\text{liste} \setminus \{\alpha\}) \cup \{\beta \in (N \cup T)^* : \alpha \Rightarrow_G \beta\}$
  return mots
end function

Complexité ? Terminaison ?

Classification des grammaires

- Classification de Chomsky
  - Quatre types de grammaires : type 0 à type 3
  - Extension à un cinquième type : type 4
    Elle correspond à l'énumération
    $G = (T, N, R, S), \quad T = \{abc, bbc, bca\}, \quad N = \{M\}, \quad S = M,$
    $R = \{(M, \mathbf{abc}), (M, \mathbf{bbc}), (M, \mathbf{bca})\}$
  - Le type i est obtenu en appliquant des restrictions sur le type $i - 1$
  - Les grammaires du type $i - 1$ peuvent exprimer plus de langages différents que les grammaires du type i
  - Le type d'une grammaire correspond au plus petit type auquel elle appartient



Grammaire de type 0

Grammaire de type 0

Une grammaire est de type 0 si toutes les règles sont de la forme $\alpha \rightarrow \beta$, avec $\alpha \in (N \cup T)^* \times N \times (N \cup T)^*$ et $\beta \in (N \cup T)^*$

$\rightarrow$ Il n'y a aucune restriction sur les règles de production

- Ces grammaires sont difficiles à manipuler

Structures de données

- Pour chaque type de grammaire, il existe une structure de données adaptée pour la reprśentation des dérivations
- Chaque type de langage peut être traité par un type particulier d'automate
- Il existe également d'autres types de grammaires qui définissent des restrictions différentes (grammaire d'arbre, grammaire moyennement sensible au contexte, etc.)
- La suite du cours est consacrée aux grammaires suivantes :

Type de grammaire	Nom de la grammaire	Structure de données	Automate
Type 1	Sensible au contexte	Graphe acyclique orienté	Machine de Turing
Type 2	Algébrique ou Hors-contexte	Arbre	Automate à pile
Type 3	Régulière	Liste	Automate fini

Exercices

1. Donnez une grammaire pour le langage des palindromes
2. Donnez une grammaire pour le langage des expressions arithmétiques
3. Donnez une grammaire pour le langage copie
4. Donnez une grammaire pour le langage copie *multiples*
5. Donnez une grammaire pour le langage XML
6. Donnez une grammaire pour le langage des paquets TCP-IP

Langages de type 3

Rappels sur les expressions régulières

Les grammaires de type 3

Les automates à états finis

`Lex`

Langages de type 3

Rappels sur les expressions régulières

Les grammaires de type 3

Les automates à états finis

Lex

Construction des expressions régulières

- Les expressions régulières (notées ER) sont construites à partir d'expressions régulières atomiques (notées ERA)
  - L'ensemble vide $\emptyset$ est une ERA
  - Le symbole ϵ est une ERA
  - Chaque symbole de l'alphabet Σ est une ERA
- Par assemblage de ces ERA par différentes fonctions, nous obtenons des expressions régulières plus complexes
- Soit r et s deux expressions régulières
  - Concaténation : rs est également une expression régulière
  - Union : $r|s$ est également une expression régulière
  - Répétition : r^* est également une expression régulière
- Cette définition est proche de celle des langages réguliers (*nous verrons pouquoi plus tard*)

Signification des expressions régulières

- Les expressions régulières permettent de représenter des langages
- Le langage représenté par l'expression régulière r est noté $L(r)$
- Langages associés aux ERA et aux fonctions
  - $L(\emptyset) = \emptyset$
    langage contenant aucun mot
  - $L(\epsilon) = \{\epsilon\}$
    langage contenant uniquement le mot vide
  - $\forall a \in \Sigma \quad L(a) = \{a\}$
    langage contenant uniquement le mot a
  - $L(rs) = L(r)L(s) = \{xy | x \in L(r) \wedge y \in L(s)\}$
    langage contenant les mots de $L(r)$ concaténés aux mots de $L(s)$
    r^i correspond à $i - 1$ concaténations de r
  - $L(r|s) = L(r) \cup L(s) = \{\omega | \omega \in L(r) \vee \omega \in L(s)\}$
    langage contenant les mots de $L(r)$ et les mots de $L(s)$
  - $L(r^*) = \bigcup_{i \in \mathbb{N}} L(r^i)$

Propriétés des fonctions

- Propriétés algébriques
  - Concaténation
    - $L(\epsilon r) = L(r\epsilon) = L(r)$
    - $L(\emptyset r) = L(r\emptyset) = L(\emptyset)$
    - $L(r(st)) = L((rs)t) = L(rst)$

    $\rightarrow$ La concaténation est associative

    $\rightarrow$ ϵ est l'élément neutre

    $\rightarrow$ $\emptyset$ est l'élément absorbant
  - Union
    - $L(\emptyset|r) = L(r|\emptyset) = L(r)$
    - $L(r|r) = L(r)$
    - $L(r|s) = L(s|r)$
    - $L(r|(s|t)) = L((r|s)|t) = L(r|s|t)$

    $\rightarrow$ L'union est commutative et associative

    $\rightarrow$ $\emptyset$ est l'élément neutre
  - Répétition
    - $L(rr^*) = L(r)L(r^*) = L(r^*r)$
    - $L(\emptyset^*) = L(\emptyset)^* = \{\epsilon\} = L(\epsilon)$

Propriétés des fonctions

- Propriétés algébriques ../..
  - Distributivité de la concaténation sur l'union
    - $L(r(s|t)) = L(r)L(s|t) = L(r)(L(r) \cup L(t)) = L(rs|rt)$
    - $L((r|s)t) = L(r|s)L(t) = (L(r) \cup L(s))L(t) = L(rt|st)$
  - Combinaisons avec des répétitions
    - $L(rr^*|\epsilon) = L(r^*)$
    - $L((r|s)^*) = L((r^*s^*)^*)$
    - $L((rs)^*r) = L(r(sr)^*)$
- Ces propriétés sont fortement utiles pour simplifier les expressions régulières

Exemples d'expressions régulières

r	$L(r)$	Exemple de mots
$a\|b$	$\{a, b\}$	$a, \ b$
ab	$\{ab\}$	ab
ab^*	$\{a\}\{b\}^*$	$abbbbbbbb$
$(ab)^*$	$\{ab\}^*$	$abababababab$
$(a\|b)^*$	$\{a, b\}^*$	$abbaabaaaba$
$(aa\|b)^*$	$\{aa, b\}^*$	$aaaabbbbbbaabbbbaabaab$

Simplification de $(a|b)^*(b^*|a^*)^*$

Extension de la notation

- Certaines notations peuvent être lourdes
  - Adresse mail : $(a|b|\cdots|z)(a|b|\cdots|z)^*(.(a|b|\cdots|z)(a|b|\cdots|z)^*)^*$@ $(a|b|\cdots|z)(a|b|\cdots|z)^*(.(a|b|\cdots|z)(a|b|\cdots|z)^*)\ (.(a|b|\cdots|z)(a|b|\cdots|z)^*)^*$

$\rightarrow$ Utilisation d'une notation abrégée

  - $r^+ = rr^*$
  - $r? = (r|\epsilon)$
  - $[a-z] = (a|b|\cdots|z)$
  - $[abcd] = (a|b|c|d)$
  - $[a-z0-9] = (a|b|\cdots|z|0|1|\cdots|9)$
  - Adresse mail : $[a-z]^+(.[a-z]^+)^*$@$[a-z]^+(.[a-z]^+)^+$

Applications – syntaxe de grep et bash

- Equivalences entre les notations
  - Cette liste est non exhaustive
  - Pour les ER, la notation $\ldots|a_i$ est à remplacer par tous les caractères de la table ASCII du début jusqu'au caractère a_i
  - Pour les ER, la notation $a_1|\ldots|a_2$ est à remplacer par tous les caractères de la table ASCII entre a_1 et a_2
  - Pour les ER, la notation $a_i|\ldots$ est à remplacer par tous les caractères de la table ASCII du caractère a_i jusqu'à la fin

r	`grep`	`bash`
$[abc]$	`[abc]`	`[abc]`
$[\ldots|a|\ldots]$	`.`	`?`
$a[\ldots|a|\ldots]^*$	`^a`	`a*`
$[\ldots|a|\ldots]^*a$	`a$`	`*a`
$[\ldots|a|f|\ldots]$	`[^bcde]`	`[^bcde]`

Question ouverte

Il vous est demandé de développer un outil qui effectue une recherche d'un motif textuelle dans une base de données immense. Quelle bibliothèque utilisez-vous ?

Langages de type 3

Grammaire de type 3

Règle *régulière*

Une règle est *régulière* si sa partie de droite contient un symbole terminal suivi d'au plus un symbole non-terminal, $A \rightarrow a$ ou $A \rightarrow aB$ avec $A, B \in N$ et $a \in T$

Grammaire *régulière*

Une grammaire est *régulière* si toutes ses règles sont régulières

Langage régulier

Un langage est rgulier si il existe une grammaire *régulière* qui l'engendre

- Toutes les dérivations de longueur n produisent n symboles terminaux tous placé en préfixe avec éventuellement un symbole non terminal en fin $S \Rightarrow a_0A_0 \Rightarrow a_0a_1A_1 \Rightarrow a_0a_1a_2A_2 \Rightarrow a_0a_1a_2a_3A_3 \Rightarrow \ldots$
- La dérivation peut prendre fin lorsque le dernier symbole terminal est produit par une règle de la forme $A \rightarrow a$, ce qui élimine la présence du seul et unique symbole non-terminal

Grammaire de type 3

Grammaire *linéaire*

Une grammaire est *linéaire* si toutes ses règles ont une partie de droite qui contient au plus un symbole non-terminal

Grammaire *linéaire à gauche*

Une grammaire est *linéaire à gauche* si toutes ses règles ont une partie de droite qui contient un ou plusieurs symboles terminaux suivi au plus d'un symbole non-terminal

$A \rightarrow \alpha$ ou $A \rightarrow \beta B$ avec $A, B \in N$, $\alpha \in T^+$ et $\beta \in T^*$

- Les grammaires *linéaires à gauche* peuvent être transformées en grammaires régulières
- Les grammaires *linéaires* sont-elles toutes de type 3 ?

Grammaire de type 3

- Exemple avec la grammaire des prénoms
  S=**L** T={**andre**, **liam**, **phil**, **et**, **,**} N={**L**, **N**, **M**, **F**}

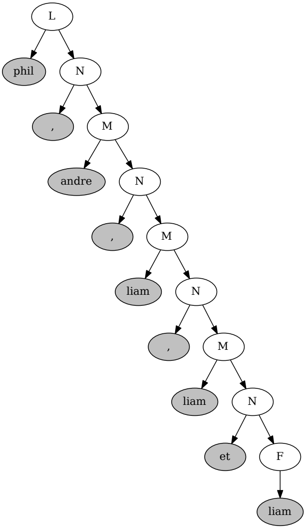
R_1	**L**	→	**andre**
R_2	**L**	→	**liam**
R_3	**L**	→	**phil**
R_4	**L**	→	**andre N**
R_5	**L**	→	**liam N**
R_6	**L**	→	**phil N**
R_7	**N**	→	**, M**
R_8	**N**	→	**et F**
R_9	**M**	→	**andre N**
R_{10}	**M**	→	**liam N**
R_{11}	**M**	→	**phil N**
R_{12}	**F**	→	**andre**
R_{13}	**F**	→	**liam**
R_{14}	**F**	→	**phil**

- Notation ABNF associée

```
Liste = ?(*((andre/liam/phil) ,) et) andre/liam/phil
```

Graphe de production

- Une dérivation issue d'une grammaire régulière peut également être représentée sous la forme d'un graphe $\rightarrow$ chaîne de dérivation



Opérations sur les langages réguliers

- L'union de deux langages réguliers est aussi un langage régulier
  - Soient $G_1 = (T_1, N_1, R_1, S_1)$ et $G_2 = (T_2, N_2, R_2, S_2)$ deux grammaires régulières qui engendrent deux langages réguliers
  - Soit $G' = (T_1 \cup T_2, N_1 \cup N_2, R', S')$ avec $R' = R_1 \cup R_2 \cup \{(S', \alpha) : (S_1, \alpha) \in R_1\} \cup \{(S', \alpha) : (S_2, \alpha) \in R_2\}$
  - La grammaire G' est régulière
- La concaténation de deux langages réguliers est aussi un langage régulier
- L'étoile d'un langage régulier est aussi un langage régulier
- La complémentation d'un langage régulier est aussi un langage régulier
- L'intersection de deux langages réguliers est aussi un langage régulier

Lemme de l'étoile pour les langages réguliers

Lemme de l'étoile pour les langages réguliers

Si un langage L est régulier, alors il existe un entier n dépendant uniquement de L tel que pour tout mot $\omega \in L$ de longueur $|\omega| \geq n$, il existe une factorisation telle que :

1. $\omega = vwx$
2. $w \neq \epsilon$
3. $|vx| \leq n$
4. $\forall i > 0, vw^i x \in L$

- Ce lemme est utilisé pour démontrer qu'un langage n'est pas régulier
- De même, il ne permet pas de démontrer qu'un langage est régulier car il s'agit d'une condition nécessaire mais pas suffisante
- Exemple : langage copie

Langages de type 3

Rappels sur les expressions régulières

Les grammaires de type 3

Les automates à états finis

Lex

Automates finis déterministes

Automate fini déterministe

Un automate fini déterministe est un 5-uplet $AFD = (Q, \Sigma, \delta, q_0, F)$

- Q : ensemble fini des états de l'automate
- Σ : alphabet fini
- δ : fonction de transition $Q \times \Sigma \times Q$
- q_0 : état initial
- F : ensemble des états finaux

- La fonction δ s'applique sur les symboles de l'alphabet
- Elle permet d'obtenir la fonction δ' qui s'applique sur les mots
- $\delta'(q, \epsilon) = q \quad \delta'(q, \omega a) = \delta(\delta'(q, \omega), a)$

Automates finis non-déterministes

Automate fini non-déterministe

Un automate fini non-déterministe est un 5-uplet $AFN = (Q, \Sigma, \delta, q_0, F)$

- Q : ensemble fini des états de l'automate
- Σ : alphabet fini
- δ : fonction de transition $Q \times \Sigma \times 2^Q$
- q_0 : état initial
- F : ensemble des états finaux

- La fonction δ s'applique sur les symboles de l'alphabet
- Elle permet d'obtenir la fonction δ' qui s'applique sur les mots
- $\delta'(q, \epsilon) = \{q\} \quad \delta'(q, \omega a) = \{p : \exists r \in \delta'(q, \omega) \wedge p \in \delta(r, a)\}$

Automates finis non-déterministes avec ϵ-transition

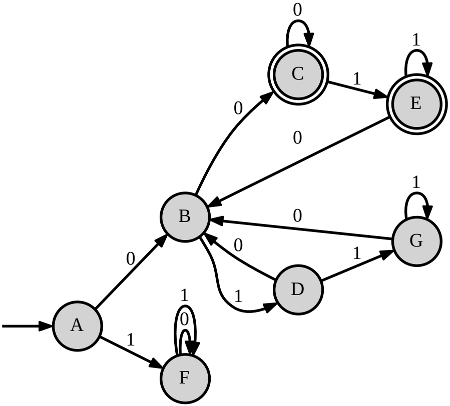
Automate fini non-déterministe avec ϵ-transition

Un automate fini non-déterministe est un 5-uplet $AFN = (Q, \Sigma, \delta, q_0, F)$

- Q : ensemble fini des états de l'automate
- Σ : alphabet fini
- δ : fonction de transition $Q \times (\Sigma \cup \{\epsilon\}) \times 2^Q$
- q_0 : état initial
- F : ensemble des états finaux

- Si un langage est accepté par un AFN avec ϵ-transition alors il est également accepté par un AFN sans ϵ-transition

Représentation graphique d'un *AFD*



Exercice

- Donnez un automate pour l'ensemble des mots sur l'alphabet $\{0, 1\}$ et finissant par la séquence 010
- Cet automate est de quel type ?
- Peut-on construire un automate *AFD* pour ce langage ?

$AFN \rightarrow AFD$

- Un langage accepté par un AFN est également accepté par un AFD
- Pour la conversion, nous introduisons la notion d'ϵ-fermeture qui correspond à l'ensemble des états atteignables depuis un état, en suivant des ϵ-transitions
- L'idée est de partir de l'état initial et de considérer l'ensemble des états atteignables comme de nouveaux états et de recommencer avec les nouveaux états

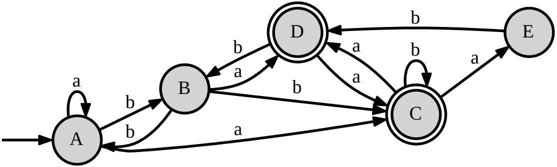
$AFN \rightarrow AFD$

FIGURE – Algorithme

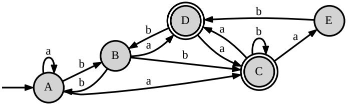
function AFN2AFD(AFN) $\triangleright\ AFN = (Q, \Sigma, \delta, q_0, F)$
  $\delta' \leftarrow \{\}$
  $q'_0 \leftarrow \{q_0\} \cup \epsilon$-fermeture$(q_0)$
  $Q' \leftarrow \{q'_0\}$
  $Todo \leftarrow \{q'_0\}$
  while $Todo \neq \emptyset$ **do**
    $Next \leftarrow \{\}$
    for all $q' \in Todo$ **do**
      for all $a \in \Sigma$ **do**
        $q" \leftarrow \epsilon$-fermeture$(\{y \in Q : \exists x \in q', (x, a, y) \in \delta\})$
        if $q" \notin Q'$ **then**
          $Q' \leftarrow Q' \cup \{q"\}$
          $Next \leftarrow Next \cup \{q"\}$
        $\delta' \leftarrow \delta' \cup \{(q', a, q")\}$
    $Todo \leftarrow Next$
  $F' \leftarrow \{q' \in Q' : q' \cap F \neq \emptyset\}$
  return $AFD = (Q', \Sigma, \delta', q'_0, F')$
end function

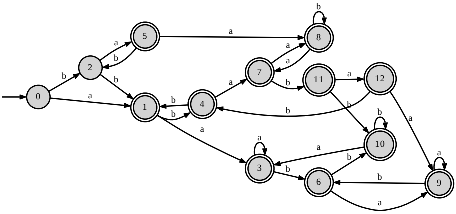
AFN → *AFD*

- Donnez l'*AFD* associé à l'*AFN* suivant



AFN → AFD





	a	b
A	A,C	B
A,C	A,C,E,D	C,B
B	D	A,C
A,C,E,D	A,C,E,D	C,B,D
C,B	E,D	A,C
D	C	B
C,B,D	C,E,D	A,C,B
E,D	C	B,D
C	E,D	C
C,E,D	C,E,D	C,B,D
A,C,B	A,C,E,D	A,C,B
B,D	C,D	A,C,B
C,D	C,E,D	C,B

AFD $\rightarrow$ *AFD* Minimal

- Le nombre d'états d'un *AFD* peut être impotant
- Il peut exister un *AFD* qui reconnait le même langage et qui possède moins d'états $\rightarrow$ algorithme
- L'idée est de considérer que tous les états finaux de l'*AFD* initial doivent être regroupés et de même pour les autres $\rightarrow$ 2 groupes
- Ensuite, il faut identifier les états d'un même groupe qui ne sont pas équivalents $\rightarrow$ pour un symbole, il existe une transition depuis chacun d'eux vers des états qui ne font pas parti du même groupe
- L'identification permet de découper à nouveau les groupes en deux
- Lorsqu'il n'est plus nécessaire de découper, nous avons obtenu l'*AFD* minimal $\rightarrow$ les groupes sont les états

$AFD \rightarrow AFD$ Minimal

Figure – Algorithme

function INITIALISER(AFD) $\triangleright$ $AFD = (Q, \Sigma, \delta, q_0, F)$
  $t \leftarrow \{\}$
  $n_c \leftarrow 1$
  for all $q \in Q$ **do**
    if $q \in F$ **then**
      $t \leftarrow t \cup \{(q, 0)\}$
    else
      $n_c \leftarrow 2$
      $t \leftarrow t \cup \{(q, 1)\}$
  return (t, n_c)
end function
function EQUIVALENT(AFD, p, q, t) $\triangleright$ $AFD = (Q, \Sigma, \delta, q_0, F)$
  $c_p \leftarrow c : (p, c) \in t$
  $c_q \leftarrow c : (q, c) \in t$
  if $c_p \neq c_q$ **then**
    return `False`
  for all $a \in \Sigma$ **do**
    $c_{pa} \leftarrow c : \exists (p, a, t) \in \delta \wedge (p, c) \in t$
    $c_{qa} \leftarrow c : \exists (q, a, t) \in \delta \wedge (q, c) \in t$
    if $c_{pa} \neq c_{qa}$ **then**
      return `False`
  return `True`
end function

AFD → *AFD* Minimal

FIGURE – Algorithme

function AFFINER(AFD, t) ▷ $AFD = (Q, \Sigma, \delta, q_0, F)$
    $t' \leftarrow \{\}$
    $n_c \leftarrow 0$
    $A \leftarrow Liste(Q)$ ▷ $Q_0, Q_1, Q_2, \ldots, Q_n$
    $i \leftarrow 0$
    while $i \neq |A|$ **do**
        $p \leftarrow Q_i$
        $newc \leftarrow$ `True`
        $j \leftarrow 0$
        while $newc \wedge j < i$ **do**
            $q \leftarrow Q_j$
            $newc \leftarrow \neg Equivalent(AFD, p, q, t)$
            if $\neg newc$ **then**
                $t' \leftarrow t' \cup \{(p, c) : (q, c) \in t'\}$
            $j \leftarrow j + 1$
        if $newc$ **then**
            $t' \leftarrow t' \cup \{(p, n_c)\}$
            $n_c \leftarrow n_c + 1$
        $i \leftarrow i + 1$
    return (t', n_c)
end function

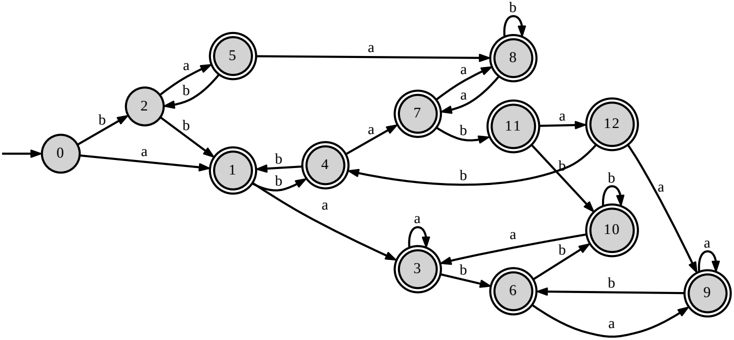
$AFD \rightarrow AFD$ Minimal

FIGURE – Algorithme

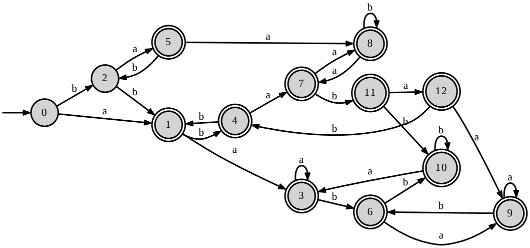
function MINAFD(AFD) ▷ $AFD = (Q, \Sigma, \delta, q_0, F)$
    $(t, n_c) \leftarrow Initialiser(AFD)$
    $Fini \leftarrow$ `False`
    while $\neq Fini$ **do**
        $(t, n'_c) \leftarrow Affiner(AFD, t)$
        $Fini \leftarrow n_c = n'_c$
        $n_c \leftarrow n'_c$
    return (t, n_c)
end function

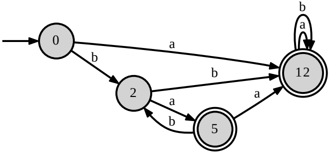
$AFD \rightarrow AFD$ Minimal

- Donnez l'*AFD* minimal associé à l'*AFD* suivant



$AFD \rightarrow AFD$ Minimal





	t_1	t_2	t_3
0	0	0	0
1	1	1	1
2	0	2	2
3	1	1	1
4	1	1	1
5	1	3	3
6	1	1	1
7	1	1	1
8	1	1	1
9	1	1	1
10	1	1	1
11	1	1	1
12	1	1	1

Langage reconnaissable

- Langage reconnaissable = ensemble des mots acceptés par l'*AFD*
- Un mot est accepté si il existe un chemin de l'état initial q_0 vers un des états finaux de F étiquetté avec les lettres du mot, dans l'ordre
- Joueur d'automate

FIGURE – Algorithme

function JOUEURAUTOMATE(AFD, s, w) $\triangleright$ $AFD = (Q, \Sigma, \delta, q_0, F), s \in Q$
    if $|w| = 0$ **then**
        return $s \in F$
    $n \leftarrow y : (s, w_0, y) \in \delta$
    return JoueurAutomate($AFD, n, w_{[1;|w|]}$)
end function

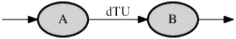
$ER \rightarrow AFN$

- Tout langage régulier est reconnaissable par un *AFN* avec ϵ-transitions
- Algorithme de Thompson : chaque opération sur les langages réguliers est traduite en une structure sur les automates

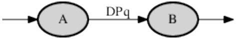
Expression régulirère — *AFN*

$L(\emptyset) = \emptyset$

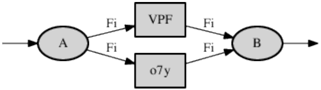
$L(\epsilon) = \{\epsilon\}$



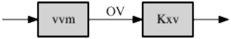
$L(a) = \{a\}$



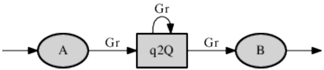
$L(e_1|e_2) = L(e_1) \cup L(e_2)$



$L(e_1.e_2) = L(e_1).L(e_2)$



$L(e^*) = L(e)^*$



$ER \rightarrow AFN$

- Construire l'automate correspondant à l'expression régulière $a|(b|c)^*c$

$AFN \rightarrow ER$

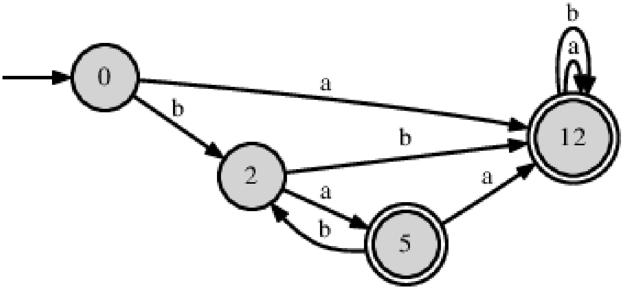
- Il est également possible de passer d'un automate à états finis à une expression régulière
- Chaque état de l'automate e_i est associé à une variable L_i, elle-même associée à une équation dont les variables sont les variables associées aux autres états
- L'équation associée à un état correspond au langage reconnu depuis cet état
- L'ensemble des équations forme un système dont la solution pour l'état initial correspond à l'expression régulière reconnaissant le même langage que l'automate

$AFN \rightarrow ER$

- Règles de transformation
  - $(e_i, a, e_j) \in \delta \Rightarrow L_i = aL_j$
  - $e_i \in T \Rightarrow L_i = \epsilon$
  - $L_i = r_1 \wedge L_i = r_2 \Rightarrow L_i = r_1|r_2$
  - Notons que l'expression régulière a^*b est solution de l'équation $L_i = aL_i|b$
  - Si, après simplification de l'écriture avec la notation $|$, les équations correspondant à L_i sont toutes de la forme $L_i = rL_i$ alors l'état correspondant est un état puit de l'automate et cette équation ne possède pas de solution $L_i = rL_i \not\Rightarrow L_i = r^*$

$AFN \rightarrow ER$

- Traduire en expression régulière l'automate suivant



- $L_0 = a(a|b)^*|bb(a|b)^*|ba(ba)^*(\epsilon|a(a|b)^*|bb(a|b)^*)$
- $L_0 = (ba)^*(a|bb)(a|b)^*|(ba)^+$

$AFN \rightarrow ER$

- Cette transformation peut également être traitée par programmation dynamique
- Soit $r_{i,j}^{0}$ l'expression régulière représentant l'ensemble des symboles permettant, depuis l'état q_i, d'atteindre l'état q_j
- Soit $r_{i,j}^{k}$ l'expression régulière représentant l'ensemble des séquences de symboles permettant, depuis l'état q_i, d'atteindre l'état q_j en traversant uniquement des états q_l tels que $l \leq k$
- $r_{i,j}^{k} = r_{i,j}^{k-1} | r_{i,k}^{k-1} (r_{k,k}^{k-1})^{*} r_{k,j}^{k-1}$
- Si n est le nombre d'états, s est l'indice de l'état initial et $(f_0, f_1, \ldots, f_m)$ est l'ensemble des indices des états finaux de F, alors l'expression régulière rechercheée est $r_{s,f_0}^{n} | r_{s,f_1}^{n} | r_{s,f_2}^{n} | \ldots | r_{s,f_m}^{n}$

Analyse lexicale

- L'objectif de l'analyse lexicale est d'identifier l'expression régulière – parmi un ensemble – reconnaissant le plus long préfixe d'une séquence de caractères
- Comment construire cet analyseur ?
- Comment passer d'un ensemble d'expressions régulières à un analyseur ?
- Que faire en cas d'erreur ?

Analyse lexicale

FIGURE – Algorithme

function ANALYSEURLEXICAL(R, C)

Require: R : un ensemble d'expressions régulières

Require: C : la chaîne à analyser

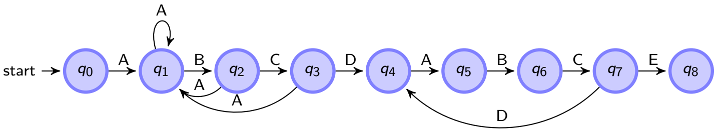
$afds \leftarrow \{AFN2AFD(ER2AFN(r)) : r \in R\}$
$indices \leftarrow []$
$candidats \leftarrow []$
$states \leftarrow []$
for all $afd = (Q, \Sigma, \delta, q_0, F) \in afds$ **do**
  $indices[afd] \leftarrow -1$
  $states[afd] \leftarrow q_0$
  $candidats[afd] \leftarrow$ `True`
$i \leftarrow 0$
while $i < |C| \wedge any(candidats)$ **do**
  for all $afd = (Q, \Sigma, \delta, q_0, F) \in afds : candidats[afd]$ **do**
    $states[afd] \leftarrow c : (states[afd], C[i], c) \in \delta$
    $candidats[afd] \leftarrow states[afd] \notin Puits(afd)$
    if $states[afd] \in F$ **then**
      $indices[afd] \leftarrow i$
  $i \leftarrow i + 1$
$fsm \leftarrow \arg\max_{fsm \in fsms} indices[fsm]$
return $(fsm, indices[fsm])$
end function

Analyse lexicale

- Cet algorithme n'est pas optimisé – proposition
- Pour chaque automate, marquer ses états par l'indice de cet automate
- Créer un nouvel automate, avec un état initial marqué par les indices de tous les automates et connecté par des ϵ-transitions à tous les états initiaux des automtes
- Passer de cet automate à un automate *AFD* en mettant à jour le marquage
- Minimiser le nombre détat en mettant à jour le marquage

Reconnaissance d'une chaîne dans un texte

- Construction de l'automate reconnaissant le motif 'ABCDABCE'



	T	E	G	A	B	C	D
0	0	0	0	1	0	0	0
1	0	0	0	1	2	0	0
2	0	0	0	1	0	3	0
3	0	0	0	1	0	0	4
4	0	0	0	5	0	0	0
5	0	0	0	0	6	0	0
6	0	0	0	0	0	7	0
7	0	8	0	0	0	0	4
8	0	0	0	0	0	0	0

Langages de type 3

Rappels sur les expressions régulières

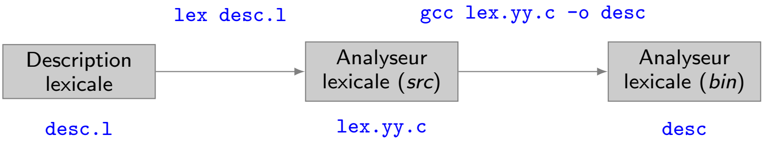
Les grammaires de type 3

Les automates à états finis

`Lex`

Introduction à Lex

- `Lex` est un `parser` permettant de construire un analyseur lexical à partir d'une description lexicale
- L'analyseur lexicale consomme les informations sur l'entrée standard et produit en sortie une liste de `tokens`
- Les informations sont lues caractère par caractère
- Dès que la séquence de caractères lue `match` avec une des descriptions lexicales, l'action associée est exécutée
- Lorsque, pour une séquence donnée, il est impossible d'identifier une description lexicale correspondante, la séquence est affichée en sortie de l'analyseur lexical
- Si plusieurs descriptions lexicales peuvent `matcher` une séquence de caractères, la description choisie est celle permettant de `matcher` la plus longue séquence



Structure d'un fichier Lex

```
%{
int i;
%}
```

Inclusions, déclarations et définissions.
Copié tel quel dans le fichier `lex.yy.c`.

```
NB [0-9]+
...
%%
```

Déclaration d'expressions régulières.
Utilisé pour faciliter l'écriture des règles.

```
{NB} {printf("Un nombre\n");}
[a-z]+ {printf("Un mot\n");}
...
%%
```

Règles. Chacune est composée d'un motif et d'une action.

```
void main(void) {
  yylex();
}
...
```

Implémentation des fonctions.

Expressions régulières de Lex

<table>
<tr><th>Méta-caractère</th><th>Signification</th></tr>
<tr><td>.</td><td>Tout sauf un retour à la ligne</td></tr>
<tr><td>\n</td><td>Retour à la ligne (aussi : \r, \t, etc)</td></tr>
<tr><td>*</td><td>Zéro ou plusieurs copies de l’expression qui précède</td></tr>
<tr><td>+</td><td>Une ou plusieurs copies de l’expression qui précède</td></tr>
<tr><td>?</td><td>Zéro ou une copie de l’expression qui précède</td></tr>
<tr><td>{n}</td><td>n copies de l’expression qui précède</td></tr>
<tr><td>^</td><td>Début de ligne</td></tr>
<tr><td>$</td><td>Fin de ligne</td></tr>
<tr><td>|</td><td>Alternatives</td></tr>
<tr><td>"</td><td>Chaîne de caractères (les caractères + et autres perdent leur signification)</td></tr>
<tr><td>[ab]</td><td>Alternatives entre caractères</td></tr>
<tr><td>[^ab]</td><td>Alternatives entre caractères autres que ceux mentionnés</td></tr>
<tr><td>a/b</td><td>a si suivi de b</td></tr>
</table>

Expressions régulières de Lex

- Exemple d'expressions

```
separateur    [ \t\n]
espaces       {separateur}+
lettre        [A-Za-z]
chiffre       [0-9]
identifiant   {lettre}({lettre}|{chiffre})*
nombre        {chiffre}+(\.{chiffre}+)?(E[+\-]?{chiffre}+)?
```

- Exemple de règles

```
{nombre}        {printf("nb: %f\n", atof(yytext));}
{identifiant}   {printf("id: %s\n", yytext);}
[a-z]+          {printf("mn: %s\n", yytext); /* !! */}
```

Variables et fonctions prédéfinies

`yyin`	Variable représentant le fichier de lecture
`yyout`	Variable représentant le fichier d'écriture
`yytext`	Variable représentant la chaîne de caractères reconnue
`yyleng`	Variable représentant la longueur de `yytext`
`yylex()`	Invocation de `Lex`
`yyless(k)`	Retire les k premiers caractères de `yytext`
	Les autres seront les premiers analysés pour la prochaine reconnaissan
`yymore()`	Concaténation du texte reconnue à la prochaine valeur de `yytext`

Les états de `Lex`

- Certains motifs peuvent avoir différentes significations
- Par exemples, gestion de noms
  - `autres="toi moi eux"`
  - `who="test toto ici" autres`
- Une possibilité est de reconnaitre une chaîne entre guillemets et la découper ensuite ?
- Plus simple utilisation des états de `Lex`
  - L'état initial est 0
  - La déclaration des états se fait avec `%start`
  - Le changement d'état se fait avec la macro `BEGIN`

Les états de Lex

- Exemple

```
%start normal identifiant

%%

<normal>^[a-za-z]+          {printf("definition : %s\n", yytext);}
<normal>=                   {printf("separateur\n");}
<normal>[a-za-z]+           {printf("lien : %s\n", yytext);}
<normal>\"                  {BEGIN identifiant;}
<identifiant>[a-za-z]+      {printf("chaine : %s\n", yytext);}
<identifiant>\"             {BEGIN normal;}
<normal,identifiant>.       { /* aucune action */ }
<normal,identifiant>\n      { /* aucune action */ }

%%

int main(void) {
  BEGIN normal;
  return yylex();
}
```

Exercice

- Lex est utilisé pour identifier des `tokens`
- Il peut également être utilisé pour effectuer des corrections sur le fichier en entrée

⇒ Les valeurs correctes sont recopiées sur la sortie
les motifs permettent de détecter les erreurs et leurs actions les corrigent

- Exercice : créer un outil de correction de composition typographique pour des textes français
- Fichier Lex

```
modea  [,.]
modeb  [;:!?]

%%

"␣""␣"+         {yyless(1);}

"␣)"            {yyless(1);}
")"/[^ ]        {printf(")␣");}

[^ ]"("         {printf("%c␣", yytext[0]); yyless(1);}
"(""␣"          {printf("(");}

"␣"{modea}      {yyless(1);}
{modea}/[^ ]    {printf("%c␣", yytext[0]);}

[^ ]{modeb}     {printf("%c␣", yytext[0]); yyless(1);}
{modeb}/[^ ]    {printf("%c␣", yytext[0]);}
```

Listing 3: syntaxe.l

Langages de type 2

Les grammaires de type 2

Les automates à pile

Parsers

`Yacc`

Langages de type 2

Grammaire de type 2

Grammaire hors-contexte

Une grammaire est hors-contexte si toutes ses règles sont sensibles au contexte avec un contexte de droite et un contexte de gauche vide

- Autre façon de définir : les grammaires hors-contexte sont des grammaires contextuelles avec un contexte vide
- Toutes les règles de production sont de la forme $A \rightarrow \alpha$ avec $\alpha \in (T \cup N)^+$ et $\mathbf{A} \in N$
- Le contexte étant vide, la dérivation se fait indépendamment des symboles à droite ou à gauche du symbole A dans le mot
- Une souplesse est accordée à certaines grammaires hors-contexte : elles peuvent engendrer le mot vide
- Grammaire hors-contexte $\equiv$ grammaire algébrique

Exercice

- Donnez une grammaire pour les langages suivants
  - $\{a^n b^n : n \in \mathbb{N}\}$
  - Expressions arithmétiques correctement parenthésées
  - Mots contenants un nombre différent de a et de b
    a, b, aab, $baaab$...

Exemple

- La grammaire des prénoms n'est pas hors-contexte.

$S=$**Liste** $T=\{$**andre**, **liam**, **phil**, **et**, **,**$\}$ $N=\{$**Liste**, **Prenom**, **Prenoms**, **PrenomFin**, **V**$\}$

R_1	**Liste**	$\rightarrow$	**Prenom**
R_2	**Liste**	$\rightarrow$	**Prenoms**
R_3	**Prenoms**	$\rightarrow$	**PrenomFin**
R_4	**Prenoms**	$\rightarrow$	**Prenom V Prenoms**
R_5	**V PrenomFin**	$\rightarrow$	**et PrenomFin**
R_6	**et PrenomFin**	$\rightarrow$	**et Prenom**
R_7	**V Prenom**	$\rightarrow$	**, Prenom**
R_8	**Prenom**	$\rightarrow$	**andre**
R_9	**Prenom**	$\rightarrow$	**liam**
R_{10}	**Prenom**	$\rightarrow$	**phil**

- Existe-t-il une grammaire hors-contexte pour ce langage ?

Exemple

- Grammaire hors-contexte pour la liste des prénoms

S=**Liste** T={**andre**, **liam**, **phil**, **et**, **,**} N={**Liste**, **Prenom**, **Prenoms**}

R_1	**Liste**	$\rightarrow$	**Prenom**
R_2	**Liste**	$\rightarrow$	**Prenoms**
R_3	**Prenoms**	$\rightarrow$	**Prenom et Prenom**
R_4	**Prenoms**	$\rightarrow$	**Prenom , Prenoms**
R_5	**Prenom**	$\rightarrow$	**andre**
R_6	**Prenom**	$\rightarrow$	**liam**
R_7	**Prenom**	$\rightarrow$	**phil**

Dérivation d'un mot

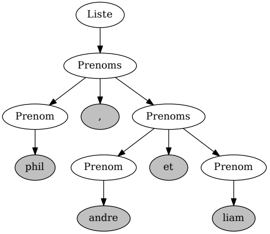
- La dérivation d'un symbole non-terminal se fait indépendamment du contexte
- Les symboles non-terminaux peuvent être dérivés dans n'importe quel ordre
- Il peut exister plusieurs dérivations pour un même mot
- Ces dérivations sont équivalentes
- Une dérivation d'un mot d'une grammaire *hors-contexte* peut également être représentée par un graphe de dération
- Etant donné que le contexte de gauche et de droite est vide, il n'y a plus de *croisements* dans le graphe, contrairement aux graphes pour les grammaires de type 1

$\Rightarrow$ Il s'agit d'un arbre de dérivation

Dérivation d'un mot – exemple

- Exemple de dérivation

Liste	$\Rightarrow$	**Prenoms**
	$\Rightarrow$	**Prenom , Prenoms**
	$\Rightarrow$	**Prenom , Prenom et Prenom**
	$\Rightarrow$	**Prenom , Prenom et liam**
	$\Rightarrow$	**Prenom , andre et liam**
	$\Rightarrow$	**phil , andre et liam**



- L'arbre correspond à un ensemble de dérivations
- Dans cet exemple, nous aurions pu dériver le symbole **Prenom** du mot intermédiaire **Prenom , Prenoms** en tout premier lieu, pour obtenir phil , Prenoms et poursuivre ensuite jusquà l'obtention de **phil , andre et liam**

Dérivation d'un mot

- Plusieurs dérivations pour un même mot $\Rightarrow$ difficulté pour les algorithmes
- Plusieurs dérivations pour un même mot $\Rightarrow$ même arbre de dérivation
- Par convention, il est possible d'imposer un ordre dans les dérivations

$\Rightarrow$ À un arbre de dérivation ne doit correspondre qu'une seule dérivation

- La contrainte sur l'ordre ne doit pas limiter les mots engendrés

$\Rightarrow$ Dérivation à gauche et dérivation à droite

Dérivation d'un mot

Dérivation à gauche (resp. droite)

Une dérivation à gauche (resp. droite) d'une grammaire hors-contexte (notée $\Rightarrow^G$, resp. $\Rightarrow^D$) est une dérivation telle que chaque étape est obtenue par réécriture du symbole non-terminal le plus à gauche (resp. droite), dans chaque mot intermédiaire

$\alpha_1 A_1 \beta_1 \Rightarrow^G \alpha_1 \gamma_1 \beta_1 = \alpha_2 A_2 \beta_2 \Rightarrow^G \alpha_2 \gamma_2 \beta_2 \ldots \alpha_n A_n \beta_n \Rightarrow^G \alpha_n \gamma_n \beta_n$,
$\forall i \ \alpha_i \in T^*$

$\alpha_1 A_1 \beta_1 \Rightarrow^D \alpha_1 \gamma_1 \beta_1 = \alpha_2 A_2 \beta_2 \Rightarrow^D \alpha_2 \gamma_2 \beta_2 \ldots \alpha_n A_n \beta_n \Rightarrow^D \alpha_n \gamma_n \beta_n$,
$\forall i \ \beta_i \in T^*$

- La dérivation à gauche impose un ordre sur les symboles non-terminaux à dériver mais elle ne va pas jusqu'à imposer un ordre sur les règles à appliquer

Dérivation d'un mot

- Que se passe-t-il si la grammaire contient la règle $A \rightarrow A$?

$\rightarrow$ Il peut exister des dérivations avec une infinité de mots intermédiaires qui ne changent pas

Dérivation à gauche (resp. droite) minimale

Une dérivation à gauche (resp. droite) minimale d'une grammaire hors-contexte est une dérivation telle que chaque étape est obtenue par réécriture du symbole non-terminal le plus à gauche (resp. droite), dans chaque mot intermédiaire et telle que chaque mot intermédiaire obtenu est différent du précédent

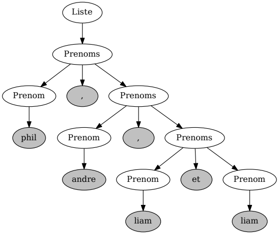
$\delta_1 = \alpha_1 A_1 \beta_1 \Rightarrow^G \alpha_1 \gamma_1 \beta_1 = \delta_2 = \alpha_2 A_2 \beta_2 \Rightarrow^G \alpha_2 \gamma_2 \beta_2 \ldots \delta_n = \alpha_n A_n \beta_n \Rightarrow^G \alpha_n \gamma_n \beta_n = \delta_{n+1}, \quad \forall i \; \alpha_i \in T^*$ et $\forall i \in [1; n] \; \delta_i \neq \delta_{i+1}$

$\delta_1 = \alpha_1 A_1 \beta_1 \Rightarrow^D \alpha_1 \gamma_1 \beta_1 = \delta_2 = \alpha_2 A_2 \beta_2 \Rightarrow^D \alpha_2 \gamma_2 \beta_2 \ldots \delta_n = \alpha_n A_n \beta_n \Rightarrow^D \alpha_n \gamma_n \beta_n = \delta_{n+1}, \quad \forall i \; \beta_i \in T^*$ et $\forall i \in [1; n] \; \delta_i \neq \delta_{i+1}$

Dérivation d'un mot – exemple avec les prénoms

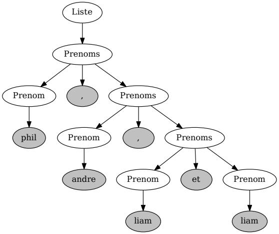
- Dérivation gauche

Liste	$\Rightarrow$	**Prenoms**
	$\Rightarrow$	**Prenom , Prenoms**
	$\Rightarrow$	**phil , Prenoms**
	$\Rightarrow$	**phil , Prenom , Prenoms**
	$\Rightarrow$	**phil , andre , Prenoms**
	$\Rightarrow$	**phil , andre , Prenom et Prenom**
	$\Rightarrow$	**phil , andre , liam et Prenom**
	$\Rightarrow$	**phil , andre , liam et liam**



- Dérivation droite

Liste	$\Rightarrow$	**Prenoms**
	$\Rightarrow$	**Prenom , Prenoms**
	$\Rightarrow$	**Prenom , Prenom , Prenoms**
	$\Rightarrow$	**Prenom , Prenom , Prenom et Prenom**
	$\Rightarrow$	**Prenom , Prenom , Prenom et liam**
	$\Rightarrow$	**Prenom , Prenom , liam et liam**
	$\Rightarrow$	**Prenom , andre , liam et liam**
	$\Rightarrow$	**phil , andre , liam et liam**



Dérivation d'un mot

Théorème

Quelque soit le mot engendré par une grammaire G hors-contexte, il existe une dérivation à gauche (resp. droite) qui engendre ce mot
$\forall u \in T^*, (S \Rightarrow^* u) \Leftrightarrow (S \Rightarrow^G u)$ et $\forall u \in T^*, (S \Rightarrow^* u) \Leftrightarrow (S \Rightarrow^D u)$

- Pour $\Rightarrow$, une dérivation à gauche correspond à un cas particulier de dérivation et $\forall u \in T^*, (S \Rightarrow^G u) \Rightarrow (S \Rightarrow^* u)$ est évident (de même pour la dérivation droite)
- Pour $\Leftarrow$, nous profitons du fait que le contexte des règles est vide
  - Soit une dérivation non-gauche qui permet d'engendrer le mot ω

  $\rightarrow$ Lors d'une des étapes, au moins deux non-terminaux étaient en concurrence pour être dérivés et ce n'est pas celui le plus à gauche qui a été dérivé
  - $S \Rightarrow^* \omega_1 A \omega_2 B \omega_3 \Rightarrow \omega_1 A \omega_2 \beta \omega_3 \Rightarrow^* \omega_1 A \omega_4 \Rightarrow \omega_1 \alpha \omega_4 \Rightarrow^* \omega$, avec $\omega_1 \in T^*$
  - Il suffit d'inverser l'ordre de dérivation – deux non-terminaux en concurrence peuvent être dérivés dans n'importe quel ordre – et de traiter toutes les étapes de la même manière
  - $S \Rightarrow^* \omega_1 A \omega_2 B \omega_3 \Rightarrow \omega_1 \alpha \omega_2 B \omega_3 \Rightarrow^* \omega_1 \alpha \omega_2 \beta \omega_3 \Rightarrow^* \omega$, avec $\omega_1 \in T^*$

Dérivation d'un mot

- Autant il est vrai que tous les mots d'un langage peuvent être générés par des dérivation à gauche autant ce n'est pas forcément vrai pour les mots intermédiaires
- Exemple avec la grammaire qui engendre le langage **abc**

$$S{=}\mathbf{S}\ \ T{=}\{\mathbf{a},\ \mathbf{b},\ \mathbf{c}\}\ \ N{=}\{\mathbf{S},\ \mathbf{A},\ \mathbf{B},\ \mathbf{C}\}$$

$$\begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{A\ B\ C} \\ R_2 & \mathbf{A} & \rightarrow & \mathbf{a} \\ R_3 & \mathbf{B} & \rightarrow & \mathbf{b} \\ R_4 & \mathbf{C} & \rightarrow & \mathbf{c} \end{array}$$

  - Donnez la liste des mots intermédiaires et engendrés par cette grammaire, quelque soit le type de dérivation considéré
  - De même en considérant uniquement les dérivations à gauche
  - Le mot **AbC** peut-il être généré par une dérivation à gauche ?

Caractéristiques d'une grammaire

Symbole inutile

Un symbole est inutile si il est impossible de dériver un mot composé uniquement de symboles terminaux à partir de ce symbole
$X \in N$ est inutile $\Leftrightarrow \nexists w \in T^* : X \Rightarrow^* w$

Symbole inaccessible

Un symbole est inaccessible si il est impossible de dériver un mot composé de ce symbole à partir de l'axiome S
$X \in N$ est inaccessible $\Leftrightarrow \nexists \alpha X \beta \in (N \cup T)^* : S \Rightarrow^* \alpha X \beta$

ϵ-production

Une ϵ-production est une règle qui produit le mot vide ϵ : $A \rightarrow \epsilon$

Règle unitaire

Une règle unitaire est une règle de la forme $A \rightarrow B$, avec $A \in N$ et $B \in N$

Caractéristiques d'une grammaire

Symbole non-terminal récursif

Un symbole non-terminal A est dit récursif si il existe une dérivation de la forme $A \Rightarrow^* \alpha A \beta$ avec $\alpha, \beta \in (N \cup T)^*$
De plus, si $\alpha = \epsilon$ (*resp.* $\beta = \epsilon$) alors A est dit récursif à gauche (*resp. droite*)

Récursivité directe

Une récursivité gauche (*resp. droite*) directe est telle qu'il existe une règle $A \rightarrow A\alpha$ (*resp.* $A \rightarrow \beta A$)

Factorisation à gauche

Une grammaire est dite factorisée à gauche si, pour toutes les règles ayant la même partie de droite, il n'existe pas de règles commencant par les mêmes symboles dans les parties de gauche (préfixe commun)
$\forall A \in N, \nexists (\alpha \neq \epsilon, \beta \neq \epsilon, \gamma \neq \epsilon) : (A, \alpha\beta) \in R \wedge (A, \alpha\gamma) \in R$

Transformation d'une grammaire

- Les grammaires peuvent être inutilement complexes
- Les transformations permettent d'obtenir des grammaires équivalentes plus simples
- Certains algorithmes fonctionnent seulement si certaines des caractéristiques précédentes sont vérifiées ou non sur la grammaire utilisée

$\Rightarrow$ Transformation des grammaires

Transformation d'une grammaire

- Suppression des symboles inutiles
  - L'idée est d'identifier les symboles utiles et ensuite supprimer les autres
  - Les symboles terminaux sont utiles
  - Un symbole non-terminal qui produit uniquement des symboles utiles dans une même règle est utile
  - L'algorithme initialise une liste avec uniquement les symboles terminaux et rajoute au fur et à mesure des symboles utiles à cette liste jusqu'à ce que ce ne soit plus possible
  - Les symboles qui n'ont pas été ajouté à la liste sont inutiles et peuvent être supprimés

Transformation d'une grammaire

- Suppression des symboles inutiles

FIGURE – Algorithme

function SUPPRESSIONDESSYMBOLESINUTILES(G) $\triangleright\ G = (T, N, R, S)$
    marks $\leftarrow \{\}$
    remain $\leftarrow N$
    continue $\leftarrow$ `True`
    while continue **do**
        continue $\leftarrow$ `False`
        for $A \in$ remain **do**
            if $\exists (B, \beta) \in R : B = A \wedge \beta \in (\text{mark} \cup T)^*$ **then**
                marks $\leftarrow$ marks $\cup \{A\}$
                remain $\leftarrow$ remain $\setminus \{A\}$
                continue $\leftarrow$ `True`
    R $\leftarrow \{(A, \beta) \in \text{R} : A \in \text{marks} \wedge \beta \in (\text{marks} \cup T)^*\}$
    N $\leftarrow$ marks
    return $G' = (T, N, R, S)$
end function

Transformation d'une grammaire

- Suppression des symboles inutiles
  - Exemple de transformation
    - Grammaire de départ

S=**S** T={**a**, **(**, **)**} N={**S**, **A**, **B**, **C**, **D**}

R_1	**S**	→	
R_2	**S**	→	**S (S)**
R_3	**S**	→	**D S (B)**
R_4	**A**	→	**S**
R_5	**A**	→	**B**
R_6	**A**	→	**a**
R_7	**B**	→	**S**
R_8	**C**	→	**a D**
R_9	**D**	→	**a D**
R_{10}	**D**	→	**a C**

    - Suppression des symboles inutiles

S=**S** T={**a**, **(**, **)**} N={**B**, **S**, **A**}

R_1	**S**	→	
R_2	**S**	→	**S (S)**
R_3	**A**	→	**S**
R_4	**A**	→	**B**
R_5	**A**	→	**a**
R_6	**B**	→	**S**

Transformation d'une grammaire

- Suppression des symboles inaccessibles
  - L'idée est d'identifier les symboles accessibles et ensuite supprimer les autres
  - L'axiome **S** est accessible
  - Un symbole produit par un symbole accessible est accessible
  - L'algorithme initialise une liste avec uniquement l'axiome **S** et rajoute au fur et à mesure des symboles accessibles à cette liste jusqu'à ce que ce ne soit plus possible
  - Les symboles qui n'ont pas été ajoutés à la liste sont inaccessibles et peuvent être supprimés
  - Les symboles terminaux peuvent également être supprimés

Transformation d'une grammaire

- Suppression des symboles inaccessibles

FIGURE – Algorithme

function SUPPRESSIONDESSYMBOLESINACCESSIBLES(G) $\triangleright$ $G = (T, N, R, S)$
    marks $\leftarrow \{S\}$
    remain $\leftarrow (N \cup T) \setminus$ marks
    continue $\leftarrow$ `True`
    while continue **do**
        continue $\leftarrow$ `False`
        for $C \in$ remain **do**
            if $\exists(A, \beta) \in R : A \in$ marks $\wedge\ C \in \beta$ **then**
                marks $\leftarrow$ marks $\cup \{C\}$
                remain $\leftarrow$ remain $\setminus \{C\}$
                continue $\leftarrow$ `True`
    R $\leftarrow \{(A, \beta) \in$ R $: A \in$ marks$\}$
    N $\leftarrow$ N $\setminus$ remain
    T $\leftarrow$ T $\setminus$ remain
    return $G' = (T, N, R, S)$
end function

Transformation d'une grammaire

- Suppression des symboles inaccessibles
  - Exemple de transformation
    - Grammaire sans symboles inutiles

$$S=\mathbf{S}\ \ T=\{\mathbf{a},\ \mathbf{(},\ \mathbf{)}\}\ \ N=\{\mathbf{B},\ \mathbf{S},\ \mathbf{A}\}$$

R_1	**S**	→	
R_2	**S**	→	**S (S)**
R_3	**A**	→	**S**
R_4	**A**	→	**B**
R_5	**A**	→	**a**
R_6	**B**	→	**S**

    - Suppression des symboles inaccessibles

$$S=\mathbf{S}\ \ T=\{\mathbf{(},\ \mathbf{)}\}\ \ N=\{\mathbf{S}\}$$

R_1	**S**	→	
R_2	**S**	→	**S (S)**

Transformation d'une grammaire

- Suppression des ϵ-productions
  - Si un symbole **A** produit le mot ϵ, alors la règle correspondante est supprimée et de nouvelles règles sont produites de manière à explorer toutes les possibilités de dérivation directe vide ou non de A
  - Par exemple, si $A \rightarrow \epsilon$ et $B \rightarrow \alpha A \beta A \gamma$, alors la règle $A \rightarrow \epsilon$ est supprimée et les nouvelles règles suivantes sont ajoutées :
    $B \rightarrow \alpha A \beta A \gamma$ $B \rightarrow \alpha A \beta \gamma$ $B \rightarrow \alpha \beta A \gamma$ $B \rightarrow \alpha \beta \gamma$
  - Il est nécessaire de disposer d'un algorithme qui génère toutes les possibilitées de suppression de **A** dans une règle
  - Ajout d'un nouvel axiome **S'** à la grammaire (avec $S' \rightarrow S$)
  - Si $\epsilon \in L(G)$ alors ajout d'une règle qui engendre ϵ depuis **S'**

FIGURE – Algorithme

function COMBINAISON($(B, \beta), A, i$) ▷ $|\beta| = n$ et $\beta = \beta_0 \dots \beta_n$
    if $n == i$ **then**
        return $\{(B, \beta)\}$
    if $\beta_i == A$ **then**
        $\beta' = \beta_0 \dots \beta_{i-1} \beta_{i+1} \dots \beta_n$
        return Combinaison($(B, \beta), A, i+1$) $\cup$ Combinaison($(B, \beta'), A, i$)
    return Combinaison($(B, \beta), A, i+1$)
end function

Transformation d'une grammaire

- Suppression des ϵ-productions

FIGURE – Algorithme

function SUPPRESSIONEPSILONPRODUCTIONS(G) $\triangleright$ $G = (T, N, R, S)$
    marks $\leftarrow \{A \in N : \exists(A, \alpha) \in R \wedge |\alpha| = 0\}$
    remain $\leftarrow N \setminus$ marks
    continue $\leftarrow$ `True`
    while continue **do**
        continue $\leftarrow$ `False`
        for $C \in$ remain **do**
            if $\exists(A, \beta) \in R : A = C \wedge \beta \in \text{marks}^*$ **then**
                marks $\leftarrow$ marks $\cup \{C\}$
                remain $\leftarrow$ remain $\setminus \{C\}$
                continue $\leftarrow$ `True`
    for $A \in$ marks **do**
        $\text{R} \leftarrow \bigcup_{r \in R} \{(a, b) \in \text{Combinaison}(r, A, 0) : |b| > 0\}$
    $R \leftarrow R \cup \{(T_e, S\}$ $\triangleright$ A ajouter ici : traitement si $\epsilon \in L(G)$
    $N \leftarrow N \cup \{T_e\}$
    return $G' = (T, N, R, T_e)$
end function

Transformation d'une grammaire

- Suppression des ϵ-productions
  - Exemple de transformation
    - Grammaire sans symboles inutiles et sans symboles inacessibles

$$
\begin{array}{llll}
\multicolumn{4}{l}{S{=}\mathbf{S}\ \ T{=}\{\mathbf{(},\ \mathbf{)}\}\ \ N{=}\{\mathbf{S}\}} \\
R_1 & \mathbf{S} & \rightarrow & \\
R_2 & \mathbf{S} & \rightarrow & \mathbf{S}\ \mathbf{(}\ \mathbf{S}\ \mathbf{)}
\end{array}
$$

    - Suppression des ϵ-productions

$$
\begin{array}{llll}
\multicolumn{4}{l}{S{=}\mathbf{N0}\ \ T{=}\{\mathbf{(},\ \mathbf{)}\}\ \ N{=}\{\mathbf{N0},\ \mathbf{S}\}} \\
R_1 & \mathbf{N0} & \rightarrow & \mathbf{S} \\
R_2 & \mathbf{S} & \rightarrow & \mathbf{S}\ \mathbf{(}\ \mathbf{S}\ \mathbf{)} \\
R_3 & \mathbf{S} & \rightarrow & \mathbf{S}\ \mathbf{(}\ \mathbf{)} \\
R_4 & \mathbf{S} & \rightarrow & \mathbf{(}\ \mathbf{S}\ \mathbf{)} \\
R_5 & \mathbf{S} & \rightarrow & \mathbf{(}\ \mathbf{)} \\
R_6 & \mathbf{N0} & \rightarrow &
\end{array}
$$

Transformation d'une grammaire

- Suppression des règles unitaires
  - L'idée est de remplacer chaque règle unitaire par des règles qui produisent directement ce que le symbole produit par la règle unitaire peut produire
  - Par exemple, si $A \rightarrow B$, $B \rightarrow \alpha$ et $B \rightarrow \beta$, alors la règle unitaire $A \rightarrow B$ est remplacée par les règles $A \rightarrow \alpha$ et $A \rightarrow \beta$
  - Une précaution consiste à éviter les boucles infinies (si $A \rightarrow B$ et $B \rightarrow A$)

Transformation d'une grammaire

- Suppression des règles unitaires

FIGURE – Algorithme

function SUPPRESSIONREGLESUNITAIRES(G) $\triangleright$ $G = (T, N, R, S)$
  deleted $\leftarrow \{\}$
  marks $\leftarrow \{(A, \beta) \in R : \beta \in N\}$
  while $|\text{marks}| > 0$ **do**
    $Q \leftarrow \{r \in R : r \notin \text{marks}\}$
    $R \leftarrow Q$
    for $(C, \delta) \in Q$ **do**
      for $(A, \beta) \in \text{marks}$ **do**
        if $\beta = C \wedge (A, \delta) \notin \text{deleted}$ **then**
          $R \leftarrow R \cup \{(A, \delta)\}$
    deleted $\leftarrow$ deleted $\cup$ marks
    marks $\leftarrow \{(A, \beta) \in R : \beta \in N\}$
  return $G' = (T, N, R, S)$
end function

Transformation d'une grammaire

- Suppression des règles unitaires
  - Exemple de transformation
    - Grammaire sans symboles inutiles, sans symboles inaccessibles et sans ϵ-productions

S=**N0** $T=\{$**(**, **)**$\}$ $N=\{$**N0**, **S**$\}$

R_1	**N0**	→	**S**
R_2	**S**	→	**S (S)**
R_3	**S**	→	**S ( )**
R_4	**S**	→	**(S)**
R_5	**S**	→	**( )**
R_6	**N0**	→	

    - Suppression des règles unitaires

S=**N0** $T=\{$**(**, **)**$\}$ $N=\{$**N0**, **S**$\}$

R_1	**S**	→	**S (S)**
R_2	**S**	→	**S ( )**
R_3	**S**	→	**(S)**
R_4	**S**	→	**( )**
R_5	**N0**	→	
R_6	**N0**	→	**S (S)**
R_7	**N0**	→	**S ( )**
R_8	**N0**	→	**(S)**
R_9	**N0**	→	**( )**

Transformation d'une grammaire

Forme normale de Chomsky

Une grammaire est en forme normale de Chomsky si toutes les règles sont de la forme $A \rightarrow BC$ ou $A \rightarrow a$ avec $a \in T$
La règle $S \rightarrow \epsilon$ est autorisée si le symbole **S** est l'axiome et si ce symbole n'apparaît pas dans la partie de droite de toutes les règles

- La contrainte sur la production du mot vide peut être satisfaite en transformant la grammaire avec les algorithmes précédents
- La démarche est similaire à celle utilisée pour obtenir une grammaire monotone bornée à droite à partir d'une grammaire monotone

Transformation d'une grammaire

- Toutes les règles de production de G sont de la forme $A \rightarrow \alpha_1 \ldots \alpha_n$, avec $\forall i \in [1; n] \quad \alpha_i \in (T \cup N)$
- G' est obtenue à partir de G en trois étapes
  1. Remplacer chaque règle de la forme $A \rightarrow \alpha_1 \ldots \alpha_n$ avec $\forall i \in [1; n] \quad \alpha_i \in (T \cup N)$ et $n \geq 3$, par
  $$\left\{ \begin{array}{rrcl} & A & \rightarrow & \alpha_1 X_1 \\ \forall i \in [1; n-3] & X_i & \rightarrow & \alpha_{i+1} X_{i+1} \\ & X_{n-2} & \rightarrow & \alpha_{n-1} \alpha_n \end{array} \right.$$
  2. Remplacer chaque règle de la forme $A \rightarrow \alpha_1 \alpha_2$ avec $\alpha_1 \in T$, par
  $$\left\{ \begin{array}{rcl} A & \rightarrow & X_{\alpha_1} \alpha_2 \\ X_{\alpha_1} & \rightarrow & \alpha_1 \end{array} \right.$$
  3. Remplacer chaque règle de la forme $A \rightarrow \alpha_1 \alpha_2$ avec $\alpha_2 \in T$, par
  $$\left\{ \begin{array}{rcl} A & \rightarrow & \alpha_1 X_{\alpha_2} \\ X_{\alpha_2} & \rightarrow & \alpha_2 \end{array} \right.$$
- Les X_i sont ajoutés à l'ensemble des non-terminaux de G' et ne sont pas les mêmes d'une règle de G traitée à l'autre

Transformation d'une grammaire

- Exemple de transformation en forme normale de Chomsky

Grammaire sans symboles inutiles, sans symboles inacessibles, sans ϵ-productions et sans règles unitaires

S=**N0** T={**(**, **)**} N={**N0**, **S**}

R_1	**S**	→	**S (S)**
R_2	**S**	→	**S ( )**
R_3	**S**	→	**(S)**
R_4	**S**	→	**( )**
R_5	**N0**	→	
R_6	**N0**	→	**S (S)**
R_7	**N0**	→	**S ( )**
R_8	**N0**	→	**(S)**
R_9	**N0**	→	**( )**

Forme normale de Chomsky associée

S=**N0** T={**(**, **)**} N={**N8**, **N7**, **N6**, **N5**, **N4**, **N3**, **N2**, **N1**, **N0**, **S**, **N1**, **N2**}

R_1	**S**	→	**S N1**
R_2	**N1**	→	**N1 N2**
R_3	**N2**	→	**S N2**
R_4	**S**	→	**S N3**
R_5	**N3**	→	**N1 N2**
R_6	**S**	→	**N1 N4**
R_7	**N4**	→	**S N2**
R_8	**S**	→	**N1 N2**
R_9	**N0**	→	
R_{10}	**N0**	→	**S N5**
R_{11}	**N5**	→	**N1 N6**
R_{12}	**N6**	→	**S N2**
R_{13}	**N0**	→	**S N7**
R_{14}	**N7**	→	**N1 N2**
R_{15}	**N0**	→	**N1 N8**
R_{16}	**N8**	→	**S N2**
R_{17}	**N0**	→	**N1 N2**
R_{18}	**N1**	→	**(**
R_{19}	**N2**	→	**)**

Transformation d'une grammaire

Forme normale de Greibach

Une grammaire est en forme normale de Greibach si toutes les règles sont de la forme $A \rightarrow a\beta$ avec $\beta \in N^*$
La règle $S \rightarrow \epsilon$ est autorisée si le symbole S est l'axiome et si ce symbole n'apparaît pas dans la partie de droite de toutes les règles

- Par construction, une grammaire en forme normale de Greibach n'est pas récursive à gauche
- Tout langage engendré par une grammaire hors-contexte peut être engendré par une grammaire en forme normale de Greibach
- Pour supprimer la récursivité gauche d'une grammaire, il suffit de la transformer en forme normale de Greibach
- Dans la suite, nous nous intéressons à la forme normale presque Greibach dans laquelle nous choisissons un ordre sur les symboles non-terminaux et nous autorisons les règles de la forme $A \rightarrow B\alpha$ si et seulement si $ord(A) \prec ord(B)$
- Cette forme n'est pas récursive à gauche car sinon il faudrait une règle de la forme $B \rightarrow A\alpha$ avec $ord(A) \prec ord(B)$

Transformation d'une grammaire

- Suppression de la récursivité gauche directe
  - L'idée est de remplacer la récursivité gauche par la récursivité droite
  - Exemple avec les règles $A \rightarrow A\alpha$ et $A \rightarrow \beta$, $\alpha, \beta \in ((N \cup T) \setminus \{A\})^*$
  - $\rightarrow$ Génèration de mots composés de plusieurs α à droite avec un β à gauche
  - Ces deux règles peuvent être transformées en $A \rightarrow \beta$, $A \rightarrow \beta A'$, $A' \rightarrow \alpha A'$ et $A' \rightarrow \alpha$
  - $\rightarrow$ Génèration de mots composés d'un β à gauche suivi de plusieurs α à droite
  - Pour chacun des symboles non-terminaux, remplacer les règles ayant ce symbole à gauche et qui sont de la forme $\forall i \in [1;m], A \rightarrow A\alpha_i$ et $\forall i \in [1;n], A \rightarrow \beta_i$, par

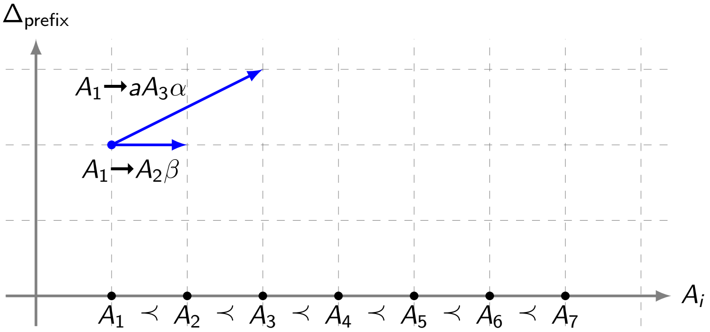
$$\left\{ \begin{array}{llcl} \forall i \in [1;n] & A & \rightarrow & \beta_i \\ \forall i \in [1;n] & A & \rightarrow & \beta_i A' \\ \forall i \in [1;m] & A' & \rightarrow & \alpha_i \\ \forall i \in [1;m] & A' & \rightarrow & \alpha_i A' \end{array} \right.$$

Transformation d'une grammaire

- Suppression de la récursivité gauche indirecte
  - La récursivité indirecte correspond à des dérivations de la forme $A \Rightarrow^* B\beta \Rightarrow^* A\alpha$
  - La mise en place d'un ordre total sur les symboles non-terminaux autorise éventuellement $A \Rightarrow^* B$ si $ord(A) \prec ord(B)$, mais dans ce cas, il nous permet d'identifier les situations telles que $B \Rightarrow^* A$ qui sont alors remplacées pour ainsi supprimer la récursivité gauche indirecte
  - Il est préférable de partir d'une grammaire sans ϵ-productions et sans cycles
  - Les symboles non-terminaux sont ordonnés en A_i avec $\forall i < j, \quad ord(A_i) \prec ord(A_j)$
  - La suppression débute par le premier non terminal
  - Supposons que $ord(A_k) \prec ord(A_i)$
  - Une règle de la forme $A_i \rightarrow A_k\alpha$ entraîne de la récursivité
  - Cette règle doit être remplacée avant de prendre en compte les symboles non-terminaux suivants
  - Elle peut être remplacée par autant de règles $A_{i+1} \rightarrow \beta_j\alpha$ qu'il y a de règles $A_k \rightarrow \beta_j$

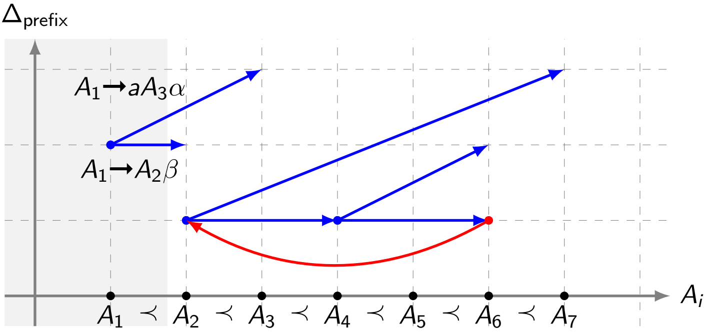
Transformation d'une grammaire

- Suppression de la récursivité gauche indirecte
  - Pour l'illustration de la méthode, nous définissons la notion d'évolution minimale de la taille du préfixe, notée $\Delta_{\text{prefix}}(r)$ pour $r \in R$
  - Une règle de la forme $A \rightarrow A\alpha$ ne change pas le préfixe du mot : $\Delta_{\text{prefix}}(A \rightarrow A\alpha) = 0$
  - Une règle de la forme $A \rightarrow aB$ change le préfixe du mot en lui ajoutant le terminal a : $\Delta_{\text{prefix}}(A \rightarrow aB) = 1$



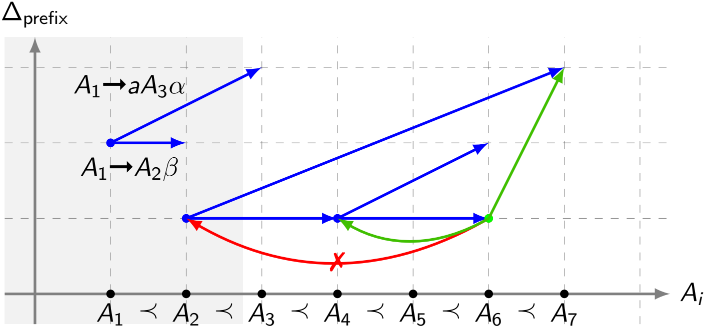
Transformation d'une grammaire

- Suppression de la récursivité gauche indirecte
  - Pour l'illustration de la méthode, nous définissons la notion d'évolution minimale de la taille du préfixe, notée $\Delta_{\text{prefix}}(r)$ pour $r \in R$
  - Une règle de la forme $A \rightarrow A\alpha$ ne change pas le préfixe du mot : $\Delta_{\text{prefix}}(A \rightarrow A\alpha) = 0$
  - Une règle de la forme $A \rightarrow aB$ change le préfixe du mot en lui ajoutant le terminal a : $\Delta_{\text{prefix}}(A \rightarrow aB) = 1$



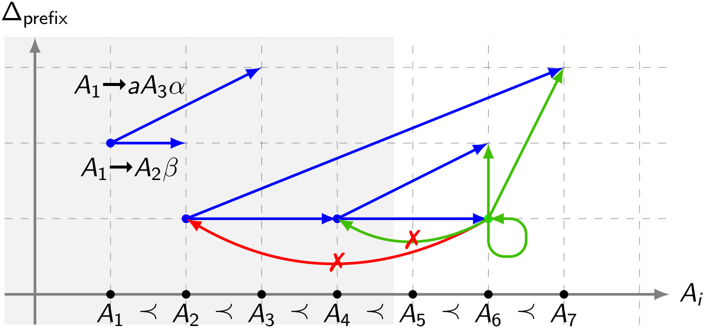
Transformation d'une grammaire

- Suppression de la récursivité gauche indirecte
  - Pour l'illustration de la méthode, nous définissons la notion d'évolution minimale de la taille du préfixe, notée $\Delta_{\text{prefix}}(r)$ pour $r \in R$
  - Une règle de la forme $A \rightarrow A\alpha$ ne change pas le préfixe du mot : $\Delta_{\text{prefix}}(A \rightarrow A\alpha) = 0$
  - Une règle de la forme $A \rightarrow aB$ change le préfixe du mot en lui ajoutant le terminal a : $\Delta_{\text{prefix}}(A \rightarrow aB) = 1$



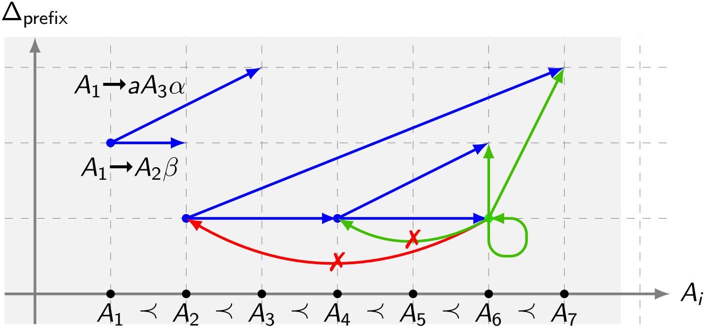
Transformation d'une grammaire

- Suppression de la récursivité gauche indirecte
  - Pour l'illustration de la méthode, nous définissons la notion d'évolution minimale de la taille du préfixe, notée $\Delta_{\text{prefix}}(r)$ pour $r \in R$
  - Une règle de la forme $A \rightarrow A\alpha$ ne change pas le préfixe du mot : $\Delta_{\text{prefix}}(A \rightarrow A\alpha) = 0$
  - Une règle de la forme $A \rightarrow aB$ change le préfixe du mot en lui ajoutant le terminal a : $\Delta_{\text{prefix}}(A \rightarrow aB) = 1$



Transformation d'une grammaire

- Suppression de la récursivité gauche indirecte
  - Pour l'illustration de la méthode, nous définissons la notion d'évolution minimale de la taille du préfixe, notée $\Delta_{\text{prefix}}(r)$ pour $r \in R$
  - Une règle de la forme $A \rightarrow A\alpha$ ne change pas le préfixe du mot : $\Delta_{\text{prefix}}(A \rightarrow A\alpha) = 0$
  - Une règle de la forme $A \rightarrow aB$ change le préfixe du mot en lui ajoutant le terminal a : $\Delta_{\text{prefix}}(A \rightarrow aB) = 1$



Transformation d'une grammaire

- Suppression de la récursivité gauche indirecte

Figure – Algorithme

function SuppressionRecursiviteGaucheIndirecte(G) $\triangleright$ $G = (T, N, R, S)$
A $\leftarrow$ Ordonner(N)
for $i \in [1; n]$ **do**
for $j \in [1; i-1]$ **do**
for $(\alpha, \beta) \in R : \alpha = A_i \wedge \beta = A_j\gamma$ **do**
$R \leftarrow R \setminus \{(A_i, \beta)\}$
for $(\gamma, \delta) \in R : \gamma = A_j$ **do**
$R \leftarrow R \cup \{(A_i, \delta\beta)\}$
$R_{A_\alpha} \leftarrow \{(\gamma, \delta) \in R : \gamma = A_i \wedge \delta = A_i\zeta\}$ $\triangleright$ Suppression de la récursivité
$R_{A_\beta} \leftarrow \{(\gamma, \delta) \in R : \gamma = A_i \wedge \forall\zeta \in (N \cup T)^* \ \delta \neq A_i\zeta\}$ $\triangleright$ directe de A_i
$(R, N) \leftarrow (R \setminus R_{A_\alpha}, N \cup \{A_i'\})$
for $(\gamma, \beta) \in R_{A_\beta}$ **do**
$R \leftarrow R \cup \{(A_i, \beta A_i')\}$
for $(\gamma, \delta) \in R_{A_\alpha} : \delta = A_i\alpha$ **do**
$R \leftarrow R \cup \{(A_i, \alpha), (A_i', \alpha A_i')\}$
return $G' = (T, N, R, S)$
end function

Transformation d'une grammaire

- Exemple de suppression de la récursivité gauche indirecte

Grammaire initiale

S=**E** T={**a**, **(**, **)**, *****, **+**} N={**E**, **T**, **F**}

R_1	**E**	→	**E + T**
R_2	**E**	→	**T**
R_3	**T**	→	**T * F**
R_4	**T**	→	**F**
R_5	**F**	→	**(E)**
R_6	**F**	→	**a**

Grammaire sans récursivité gauche

S=**E** T={**a**, **(**, **)**, *****, **+**} N={**N2**, **N1**, **N0**, **E**, **T**, **F**, **N0**, **N1**, **N2**, **N3**, **N4**}

R_1	**E**	→	**E N0**
R_2	**N0**	→	**N4 T**
R_3	**E**	→	**T**
R_4	**T**	→	**T N1**
R_5	**N1**	→	**N3 F**
R_6	**T**	→	**F**
R_7	**F**	→	**N1 N2**
R_8	**N2**	→	**E N2**
R_9	**F**	→	**a**
R_{10}	**N0**	→	**a**
R_{11}	**N1**	→	**(**
R_{12}	**N2**	→	**)**
R_{13}	**N3**	→	*****
R_{14}	**N4**	→	**+**

Transformation d'une grammaire

- Factorisation à gauche
  - Si une grammaire G contient deux règles de la forme $A \rightarrow \alpha\beta_1$ et $A \rightarrow \alpha\beta_2$ avec $\alpha \neq \epsilon$, $\beta_1 \neq \epsilon$ et $\beta_2 \neq \epsilon$, alors elle n'est pas factorisée à gauche

Transformation d'une grammaire

- Factorisation à gauche

FIGURE – Algorithme

function FACTORISATIONGAUCHE(G) ▷ $G = (T, N, R, S)$
  $W \leftarrow N$
  while $W \neq \emptyset$ **do**
    $V \leftarrow \emptyset$
    for all $A \in W$, $\gamma \in (N \cup T)$ **do**
      $R' \leftarrow \{(\alpha, \beta) \in R : \alpha = A, \beta = \gamma\delta\}$
      if $|R'| > 1$ **then**
        $\delta \leftarrow \gamma$
        $\delta' \leftarrow \delta\zeta : R'_1 = (A, \delta\zeta\beta) \wedge \zeta \in (N \cup T)$ ▷ R'_1 : la 1ère règle de R'
        while $\exists \zeta : \forall(\alpha, \beta) \in R', \beta = \delta'\zeta$ **do** ▷ Recherche de $Plpc(R')$
          $\delta \leftarrow \delta'$
          $\delta' \leftarrow \delta\zeta : R'_1 = (A, \delta\zeta\beta) \wedge \zeta \in (N \cup T)$
        $V \leftarrow V \cup \{X_A\}$
        $R \leftarrow (R \setminus R') \cup \{(A, \delta X_A)\}$
        for all $(A, \delta\beta) \in R'$ **do**
          $R \leftarrow R \cup \{(X_A, \beta)\}$
    $W \leftarrow V$
    $N \leftarrow N \cup V$
  return $G' = (T, N, R, S)$
end function

Transformation d'une grammaire

- Exemple de factorisation à gauche

Grammaire sans récursivité gauche

$$S{=}\mathbf{E}\ \ T{=}\{\mathbf{a},\ \mathbf{*},\ +\}\ \ N{=}\{\mathbf{E}\}$$

R_1	**E**	→	**E + E**
R_2	**E**	→	**E * E**
R_3	**E**	→	**a**

Grammaire factorisée à gauche

$$S{=}\mathbf{E}\ \ T{=}\{\mathbf{a},\ \mathbf{*},\ +\}\ \ N{=}\{\mathbf{N0},\ \mathbf{E}\}$$

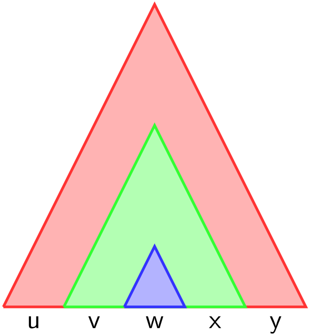
R_1	**E**	→	**a**
R_2	**E**	→	**E N0**
R_3	**N0**	→	*** E**
R_4	**N0**	→	**+ E**

Lemme de l'étoile pour les langages algébriques

Lemme

Si un langage L est algébrique, alors il existe un entier n dépendant uniquement de L tel que pour tout mot $\omega \in L$ de longueur $|\omega| \geq n$, il existe une factorisation telle que :

1. $\omega = uvwxy$
2. $vx \neq \epsilon$
3. $w \neq \epsilon$
4. $|vwx| \leq n$
5. $\forall i > 0, uv^iwx^iy \in L$

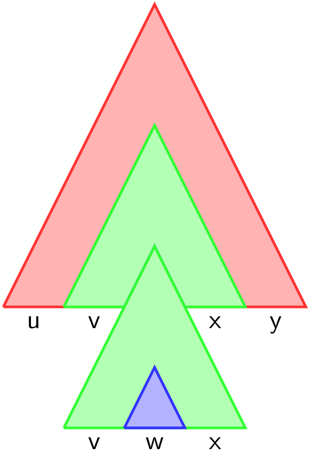


Lemme de l'étoile pour les langages algébriques

Lemme

Si un langage L est algébrique, alors il existe un entier n dépendant uniquement de L tel que pour tout mot $\omega \in L$ de longueur $|\omega| \geq n$, il existe une factorisation telle que :

1. $\omega = uvwxy$
2. $vx \neq \epsilon$
3. $w \neq \epsilon$
4. $|vwx| \leq n$
5. $\forall i > 0, uv^iwx^iy \in L$

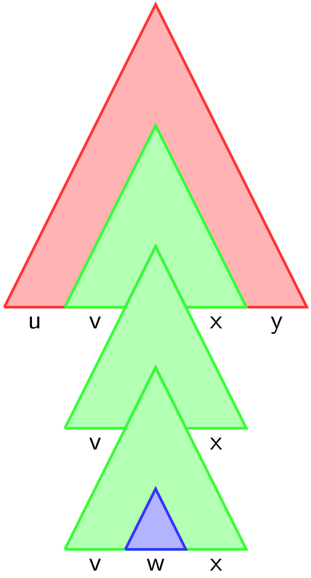


Lemme de l'étoile pour les langages algébriques

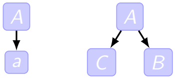
Lemme

Si un langage L est algébrique, alors il existe un entier n dépendant uniquement de L tel que pour tout mot $\omega \in L$ de longueur $|\omega| \geq n$, il existe une factorisation telle que :

1. $\omega = uvwxy$
2. $vx \neq \epsilon$
3. $w \neq \epsilon$
4. $|vwx| \leq n$
5. $\forall i > 0, uv^i wx^i y \in L$



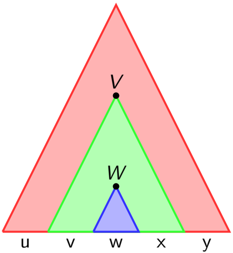
Lemme de l'étoile pour les langages algébriques



- Soit une grammaire en forme normale de Chomsky qui engendre ce langage – les règles sont de la forme $A \rightarrow BC$ ou $A \rightarrow a$
- Les arbres de dérivation associés ont des feuilles issues des règles $A \rightarrow a$ et des nœuds issus des règles $A \rightarrow BC$ (binaires)

$\rightarrow$ En ignorant les feuilles, nous avons un arbre binaire

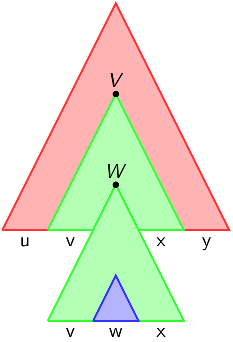
- Un mot de longueur au moins $k = 2^{|N|+1}$ est associé à un arbre de dérivation de hauteur au moins $|N| + 1$
- Il existe donc un chemin de l'axiome **S**, racine de l'arbre, jusqu'à une des feuille, de longueur $|N| + 1$



- Ce chemin ayant une longueur supérieure au nombre de symboles non-terminaux, un des symboles non-terminaux, noté **A**, se retrouve à deux reprises sur ce chemin, en V et W

Lemme de l'étoile pour les langages algébriques

- Le sous-arbre V engendre le mot vwx
- Le sous-arbre W engendre le mot w
- $v \neq \epsilon$ ou $x \neq \epsilon$ car la règle $A \rightarrow BC$ employée au niveau du nœud V permet de deriver v et wx (avec $v \neq \epsilon$) ou de deriver vw et x (avec $x \neq \epsilon$) (*NB. la grammaire est en forme normale de Chomsky et aucune règle autre que l'axiome ne permet de dériver* ϵ)
- Nous avons donc les dérivations $A \Rightarrow^* vAx$ et $A \Rightarrow^* w$ donc, $A \Rightarrow^* v^i A x^i \Rightarrow^* v^i w x^i$
- La grammaire étant *hors-contexte*, le sous-arbre en V peut être *recopié* tel quel en W



Lemme de l'étoile pour les langages algébriques

- Il existe une version plus forte de ce lemme $\Rightarrow$ le lemme d'Ogden

Lemme d'Ogden

Si un langage L est algébrique, alors il existe un entier n dépendant uniquement de L tel que pour tout mot ω de longueur $|\omega| \geq n$ et pour tout choix de positions distinguées dans ω, il existe une factorisation telle que :

1. $\omega = uvwxy$
2. v ou x contient au moins une position distinguée
3. vwx contient au plus n position distinguées
4. $\forall i > 0, uv^iwx^iy \in L$

- Ces lemmes permettent de montrer si un langage est non-algébrique
- Ils ne permettent pas de montrer qu'un langage est algébrique car il s'agit d'une condition nécessaire mais pas suffisante

Lemme de l'étoile pour les langages algébriques

- Exemple d'utilisation avec le langage $L = \{a^j b^j c^j, j > 0\}$
  - Supposons la constante n connue
  - Alors, si L est algébrique, quelque soit le mot ω de longueur supérieure à n, il existe une factorisation $\omega = uvwxy$ telle que $vx \neq \epsilon$, $w \neq \epsilon$, $|vwx| \leq n$ et $\forall i > 0, uv^i wx^i y \in L$
  - Considérons le mot $\omega = a^n b^n c^n \in L$ (sa longueur est donc $> n$)
  - La longueur de vwx étant limitée, vwx ne peut pas contenir à la fois un a et un c – de plus, v ne peut pas contenir plusieurs symboles différents, sinon nous obtenons par exemple $v = a^p b^q$, $v^2 = a^p b^q a^p b^q$ qui ne peut en aucun cas être un facteur de L (*de même pour* x)
  - Considérons $i = 2$ ainsi que toutes les factorisations possibles de ω ($\equiv$ *toutes les positions de vwx dans le mot* $a^n b^n c^n$)
    - $x = a^k \Rightarrow uv^2wx^2y = a^l b^n c^n$ avec $l > n$
    - $v = a^k$, $x = b^l \Rightarrow uv^2wx^2y = a^o b^p c^n$ avec $p > n$
    - $v = b^k$, $x = c^l \Rightarrow uv^2wx^2y = a^n b^o c^p$ avec $p > n$
    - $x = c^k \Rightarrow uv^2wx^2y = a^n b^n c^l$ avec $l > n$
  - Pour ce mot, il n'existe pas de factorisation telle que le lemme est vérifié

$\Rightarrow$ Ce langage n'est pas algébrique

Clôture des langages algébriques

- Union de langages algébriques
  - Si L_1 et L_2 sont des langages algébriques alors il existe deux grammaires *hors-contexte* $G_1 = (T_1, N_1, R_1, S_1)$ et $G_2 = (T_2, N_2, R_2, S_2)$ qui engendrent respectivement L_1 et L_2
  - L'union de ces deux langages est reconnue par la grammaire $G_3 = (T_1 \cup T_2, N_1 \cup N_2, R_1 \cup R_2 \cup \{(S_3, S_1), (S_3, S_2)\}, S_3)$, en renommant si nécessaire les symboles pour que $N_1 \cap N_2 = \emptyset$ et $S_3 \notin (N_1 \cup C_2)$
  - G_3 étant *hors-contexte*, l'union de deux langages algébriques est algébrique
- Démarche similaire pour la concaténation et l'étoile de Kleene
- Intersection de langages algébriques
  - Le langage $L = \{a^j b^j c^j, j > 0\}$ n'est pas algébrique
  - Le langage $L_1 = \{a^j b^j c^k, j > 0, k > 0\}$ est algébrique
  - Le langage $L_2 = \{a^k b^j c^j, j > 0, k > 0\}$ est algébrique
  - $L = L_1 \cap L_2$

  $\Rightarrow$ L'intersection ne conserve pas l'algébricité des langages

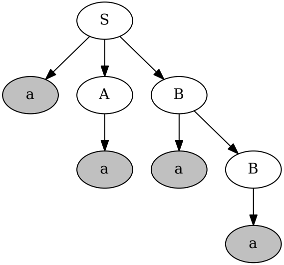
Ambiguïté

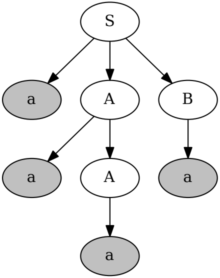
- Une grammaire est considérée ambiguë s'il existe deux dérivations à gauches différentes pour un même mot $\rightarrow$ les deux arbres de dérivation sont donc différents
- Exemple avec la grammaire

S=**S** T={**a**} N={**S**, **A**, **B**}

R_1	**S**	→	**a A B**
R_2	**A**	→	**a**
R_3	**A**	→	**a A**
R_4	**B**	→	**a**
R_5	**B**	→	**a B**

- Deux dérivations à gauche différentes pour le mot *aaaa*





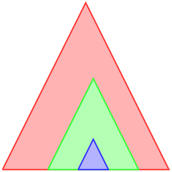
Ambiguïté

- Pour un même langage, il peut exister une grammaire non-ambiguë et une grammaire ambiguë
- L'élimination de l'ambiguïté peut être réalisée en transformant les règles ou en imposant une dérivation à gauche, mais ce n'est pas forcément suffisant
  - La dérivation à gauche impose un ordre sur les symboles non-terminaux à dériver mais elle ne va pas jusqu'à imposer un ordre sur les règles à appliquer
- Un ordre sur les symboles peut être imposé $\Rightarrow$ ordre sur les règles $A \rightarrow B * C \prec A \rightarrow B + C$
- Un langage est inhéremment ambigu si toutes les grammaires qui l'engendre sont ambiguës
- Une manière de prouver qu'un langage est inhéremment ambigu est de s'appuyer sur ces lemmes et de prouver qu'il existe deux dérivations différentes qui reconnaissent le même mot avec des sous-arbres différents

Ambiguïté

- Exemple avec le langage $L = \{a^i b^i c^j, i > 0, j > 0\} \cup \{a^j b^i c^i, i > 0, j > 0\}$
  - $a^{n+n!} b^n c^n \Rightarrow a^{n+n!} b^{n+n!} c^{n+n!}$ en appliquant $(n! + k)/k$ fois le lemme de l'étoile

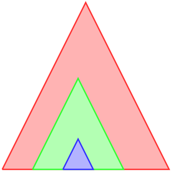
$$\begin{array}{lllllll}
\omega & = a^{n+n!} b^{n-k-1} & b^k & bc & c^k & c^{n-k-1} \\
 & = u & v & w & x & y \\
\omega' & = u & v^{(n!+k)/k} & w & x^{(n!+k)/k} & y \\
 & = a^{n+n!} b^{n-k-1} & b^{k\times((n!+k)/k)} & bc & c^{k\times((n!+k)/k)} & c^{n-k-1} \\
 & = a^{n+n!} b^{n-k-1} & b^{n!+k} & bc & c^{n!+k} & c^{n-k-1}
\end{array}$$



Le sous-arbre correspondant à la dérivation
$vwx = b^k bcc^k \Rightarrow^* v^{(n!+k)/k} wx^{(n!+k)/k} = b^{n!+k+1} c^{n!+k+1}$
contient plus de la majorité des b et des c du mot final sans aucun a

  - $a^n b^n c^{n+n!} \Rightarrow a^{n+n!} b^{n+n!} c^{n+n!}$ en appliquant $(n! + k)/k$ fois le lemme de l'étoile

$$\begin{array}{lllllll}
\omega & = a^{n-k-1} & a^k & ab & b^k & b^{n-k-1} c^{n+n!} \\
 & = u & v & w & x & y \\
\omega' & = u & v^{(n!+k)/k} & w & x^{(n!+k)/k} & y \\
 & = a^{n-k-1} & a^{k\times((n!+k)/k)} & ab & b^{k\times((n!+k)/k)} & b^{n-k-1} c^{n+n!} \\
 & = a^{n-k-1} & a^{n!+k} & ab & b^{n!+k} & b^{n-k-1} c^{n+n!}
\end{array}$$



Le sous-arbre correspondant à la dérivation
$vwx = a^k abb^k \Rightarrow^* v^{(n!+k)/k} wx^{(n!+k)/k} = a^{n!+k+1} b^{n!+k+1}$
contient plus de la majorité des a et des b du mot final sans aucun c

Notation

- Il existe différentes notations pour l'écriture des grammaires *hors-contexte*
- Elles ne permettent pas d'augmenter l'expressivité des grammaires
- Elles sont toutes équivalentes
- Forme de Backus-Naur (BNF), forme de Backus-Naur étendue (EBNF), Augmented Backus-Naur form (ABNF), etc.
- Exemple avec la grammaire

$$S=\textbf{Liste}\ T=\{\textbf{andre}, \textbf{liam}, \textbf{phil}, \textbf{et}, \textbf{,}\}\ N=\{\textbf{Liste}, \textbf{Prenom}, \textbf{Prenoms}\}$$

$$\begin{array}{lrcl}
R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} \\
R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms} \\
R_3 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom et Prenom} \\
R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom , Prenoms} \\
R_5 & \textbf{Prenom} & \rightarrow & \textbf{andre} \\
R_6 & \textbf{Prenom} & \rightarrow & \textbf{liam} \\
R_7 & \textbf{Prenom} & \rightarrow & \textbf{phil}
\end{array}$$

- Notation BNF associée

```
<Liste>   ::= <Prenom> | <Prenoms>
<Prenoms> ::= <Prenom> et <Prenom>
<Prenoms> ::= <Prenom> , <Prenoms>
<Prenom>  ::= andre | liam | phil
```

Notation

- Des raccourcis ont été introduit dans la notation ABNF pour réduite l'empreinte de la notation `http://tools.ietf.org/html/rfc5234`
  - `S =     / A S` $\rightarrow$ `*A`
  - `S =   A / A S` $\rightarrow$ `+A`
  - `S = A A / A S` $\rightarrow$ `3*A`
- Exemple avec la grammaire

S=**Liste** T={**andre**, **liam**, **phil**, **et**, **,**} N={**Liste**, **Prenom**, **Prenoms**}

R_1	**Liste**	→	**Prenom**
R_2	**Liste**	→	**Prenoms**
R_3	**Prenoms**	→	**Prenom et Prenom**
R_4	**Prenoms**	→	**Prenom , Prenoms**
R_5	**Prenom**	→	**andre**
R_6	**Prenom**	→	**liam**
R_7	**Prenom**	→	**phil**

- Notation ABNF associée

```
Liste  = ?(*(Prenom ",") Prenom "et") Prenom
Prenom = "andre" / "liam" / "phil"
```

Notation

- Notation ABNF pour la notation ABNF

```
rulelist       =  1*( rule / (*c-wsp c-nl) )
rule           =  rulename defined-as elements c-nl
                       ; continues if next line starts
                       ;  with white space
rulename       =  ALPHA *(ALPHA / DIGIT / "-")
defined-as     =  *c-wsp ("=" / "=/") *c-wsp
                       ; basic rules definition and
                       ;  incremental alternatives
elements       =  alternation *c-wsp
c-wsp          =  WSP / (c-nl WSP)
c-nl           =  comment / CRLF
                       ; comment or newline
comment        =  ";" *(WSP / VCHAR) CRLF
alternation    =  concatenation
                  *(*c-wsp "/" *c-wsp concatenation)
concatenation  =  repetition *(1*c-wsp repetition)
repetition     =  [repeat] element
repeat         =  1*DIGIT / (*DIGIT "*" *DIGIT)
```

Notation

- Notation ABNF pour la notation ABNF

```
element        =  rulename / group / option /
                  char-val / num-val / prose-val
group          =  "(" *c-wsp alternation *c-wsp ")"
option         =  "[" *c-wsp alternation *c-wsp "]"
char-val       =  DQUOTE *(%x20-21 / %x23-7E) DQUOTE
                       ; quoted string of SP and VCHAR
                       ;  without DQUOTE
num-val        =  "%" (bin-val / dec-val / hex-val)
bin-val        =  "b" 1*BIT
                  [ 1*("." 1*BIT) / ("-" 1*BIT) ]
                       ; series of concatenated bit values
                       ;  or single ONEOF range
dec-val        =  "d" 1*DIGIT
                  [ 1*("." 1*DIGIT) / ("-" 1*DIGIT) ]
hex-val        =  "x" 1*HEXDIG
                  [ 1*("." 1*HEXDIG) / ("-" 1*HEXDIG) ]
```

Sujets d'approfondissement

- Autres types de grammaires : grammaire linéaires à droite, par exemple
- Les problèmes décidables
  - A titre d'exemple, le problème de déterminer si une grammaire algébrique engendre tous les mots sur un alphabet est indécidable[Lan64]
- Les langages de Dyck
- Le théorème de Chomsky-Schützenberger[Boo76]

Théorème de Chomsky-Schützenberger

Un langage est algégrique si il existe un langage de Dyck D, un langage rationnel K et un morphisme h tels que $L = h(D \cap K)$.

- Dans la suite, nous étudions les grammaires de type 3

Langages de type 2

Les grammaires de type 2

Les automates à pile

Parsers

`Yacc`

Limite des automates finis

- Les automates finis n'ont pas d'autre mémoire que leurs états
- Arrivé à un état, on a tout oublié du chemin qui nous y a amené
- Leur mémoire est donc bornée et limitée par leur nombre d'états
- Dépasser cette limite $\rightarrow$ ajout d'une autre mémoire, une pile

Des des automates finis aux automates à pile

- Automate reconnaissant le langage a^*b^*
- Le nombre de a est indépendant du nombre de b
- Pour obtenir un modèle reconnaissant le langage a^nb^n, il faut un moyen de contrôler les transitions
- L'ajout d'une pile à côté et le conditionner les transitions sur l'état de la pile permet d'atteindre un modèle reconnaissant le langage a^nb^n

Automates à pile

Automate à pile non-déterministe

Un automate à pile non-déterministe est un 7-uplet $APN = (Q, \Sigma, \delta, q_0, F, \Gamma, Z)$

- Q : ensemble fini des états de l'automate
- Σ : alphabet fini
- q_0 : état initial
- F : ensemble des états finaux
- Γ : alphabet fini de pile
- Z : symbole initial de pile
- δ : fonction de transition $Q \times (\Gamma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \times Q \times (\Gamma \cup \{\epsilon\})$

Transition

- Une transition de la forme (p, γ, a, q, χ), ou $(p, \gamma, a) \rightarrow (q, \chi)$ signifie que :
  - lorsque l'on est à l'état p,
  - qu'on lit le préfixe a et
  - que la chaîne γ se trouve au sommet de la pile,
- alors :
  - on peut passer à l'état q,
  - en consommant γ et
  - en remplacant γ au sommet de la pile par χ.

Mode de reconnaissance et langage reconnu

- Différents modes de reconnaissance : état acceptant, état acceptant et pile vide, pile vide
- Ces modes de reconnaissance sont équivalents
- Une chaîne ω est reconnue par état acceptant et pile vide s'il y a une suite de transitions partant de l'état initial avec la pile vide, dont la suite des étiquettes est ω, et aboutissant à un état acceptant où la pile est vide
- Le langage reconnu par un automate à pile est l'ensemble des chaînes reconnues

Remarque

- Notion d'automate à pile déterministe
- Tout langage algébrique peut être reconnu par un automate à pile
- Un langage algébrique n'est pas forcément reconnu par un automate à pile déterministe

Exemple d'automate à pile

- $APN = (Q, \Sigma, \delta, q_0, F, \Gamma, Z)$
- $\Sigma = \{a, b\}$
- $Q = \{q_0, q_1\}$
- $q_0 = q_0$
- $Z = Z$
- $\Gamma = \{A, Z\}$
- $F = \{\}$
- $\delta = \left\{ \begin{array}{c} (q_0, \epsilon, Z, q_0, \epsilon), \\ (q_0, a, Z, q_0, A), \\ (q_0, a, A, q_0, AA), \\ (q_0, b, A, q_1, \epsilon), \\ (q_1, b, A, q_1, \epsilon) \end{array} \right\}$
- Mode de reconnaissance par pile vide
- Langage reconnu par cet automate à pile : $\{a^n b^n, n \geq 0\}$

Exemple d'automate à pile

- $APN = (Q, \Sigma, \delta, q_0, F, \Gamma, Z)$
- $\Sigma = \{a, b\}$
- $Q = \{q_0, q_1, q_2\}$
- $q_0 = q_0$
- $Z = Z$
- $\Gamma = \{A, Z\}$
- $F = \{q_2\}$
- $\delta = \left\{ \begin{array}{c} (q_0, \epsilon, Z, q_2, Z), \\ (q_0, a, Z, q_0, ZA), \\ (q_0, a, A, q_0, AA), \\ (q_0, b, A, q_1, \epsilon), \\ (q_1, b, A, q_1, \epsilon), \\ (q_1, \epsilon, Z, q_2, Z) \end{array} \right\}$
- Mode de reconnaissance par état final
- Langage reconnu par cet automate à pile : $\{a^n b^n, n \geq 0\}$

Exercice

- Constuire un automate à pile qui reconnait le langage $\{a^n b^n, n \geq 0\}$ avec le mode de reconnaissance par état final et par pile vide

Grammaire hors-contexte $\rightarrow$ *APN*

- Soit $G = (T, N, R, S)$ une grammaire algébrique sous forme normale de Greibach qui engendre le langage hors-contexte L – L'automate à pile $APN = (Q, \Sigma, \delta, q_0, F, \Gamma, Z)$ reconnaissant le langage L peut être construit de la manière suivante :
  - $Q = \{q\}$
  - $\Sigma = T$
  - $\Gamma = N$
  - $q_0 = q$
  - $F = \emptyset$
  - $\delta = \begin{cases} \forall(A, a\alpha) \in R, & (q, a, A) \rightarrow (q, \alpha) \quad \text{Pour chaque règle } A \rightarrow a\alpha \\ \forall(A, a) \in R, & (q, a, A) \rightarrow (q, \epsilon) \quad \text{Pour chaque règle } A \rightarrow a \end{cases}$
- La pile contient les symboles non-terminaux des mots intermédiaires, qu'il faut donc encore dériver

Grammaire hors-contexte $\rightarrow$ *APN*

- Transformez la grammaire suivante en *APN*

$$S = \mathbf{S} \;\; T = \{\mathbf{a}, \mathbf{b}\} \;\; N = \{\mathbf{S} \; \mathbf{A} \; \mathbf{B}\}$$

$$R = \left\{ \begin{array}{cccl} R_1 & \mathbf{S} & \rightarrow & \mathbf{a} \; \mathbf{B} \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{b} \; \mathbf{A} \\ R_3 & \mathbf{A} & \rightarrow & \mathbf{a} \\ R_4 & \mathbf{A} & \rightarrow & \mathbf{a} \; \mathbf{S} \\ R_5 & \mathbf{A} & \rightarrow & \mathbf{b} \; \mathbf{A} \; \mathbf{A} \\ R_6 & \mathbf{B} & \rightarrow & \mathbf{b} \\ R_7 & \mathbf{B} & \rightarrow & \mathbf{b} \; \mathbf{S} \\ R_8 & \mathbf{B} & \rightarrow & \mathbf{a} \; \mathbf{B} \; \mathbf{B} \end{array} \right\}$$

$APN \rightarrow$ Grammaire hors-contexte

- Soit $APN = (Q, \Sigma, \delta, q_0, F, \Gamma, Z)$ un automate à pile s'arrêtant sur une pile vide et reconnaissant le langage L – La grammaire hors-contexte $G = (T, N, R, S)$ reconnaissant le langage L peut être construit de la manière suivante :
  - $T = \Sigma$
  - $N = \{S\} \cup \{<p, A, q> \in Q \times \Gamma \times Q\}$
  - $R = \left\{ \begin{array}{rcl} S & \rightarrow & <q_0, Z, q> \\ & & \forall q \in Q \\ <q, A, q_n> & \rightarrow & u<p, B_1, q_1><q_1, B_2, q_2>\ldots<q_{n-1}, B_n, q_n> \\ & & \forall \delta(q, u, A) = (p, B_n \ldots B_2 B_1) \text{ et} \\ & & \forall (q_1, q_2, \ldots, q_n) \in Q^n \\ <p, A, q> & \rightarrow & u \\ & & \forall \delta(p, u, A) = (q, \epsilon) \end{array} \right.$
- G engendre ω depuis $<q, A, p>$ si et seulement si, sur l'APN, $(q, \omega, A) \Rightarrow^* (p, \epsilon, \epsilon)$: une dérivation gauche du mot ω est la simulation du fonctionnement de l'APN sur l'entrée ω

$APN \rightarrow$ Grammaire hors-contexte

- $\Sigma = \{a, b\}$
- $Q = \{q_0, q_1\}$
- $q_0 = q_0$
- $Z = Z$
- $\Gamma = \{Z, A\}$
- $F = \{\}$
- $\delta = \left\{ \begin{array}{c} (q_0, a, Z, q_0, A), \\ (q_0, a, A, q_0, AA), \\ (q_0, b, A, q_1, \epsilon), \\ (q_1, b, A, q_1, \epsilon) \end{array} \right\}$

$S = \mathbf{S}$ $T = \{\mathbf{a}, \mathbf{b}\}$ $N = \{\mathbf{S} <q_0, Z, q_0> <q_0, Z, q_1> <q_0, A, q_0> <q_0, A, q_1> <q_1, A, q_1> <q_1, A, q_0>\}$

$$R = \left\{ \begin{array}{llcl} R_1 & \mathbf{S} & \rightarrow & <q_0, Z, q_0> \\ R_2 & \mathbf{S} & \rightarrow & <q_0, Z, q_1> \\ R_3 & <q_0, Z, q_0> & \rightarrow & \mathbf{a} <q_0, A, q_0> \\ R_4 & <q_0, Z, q_1> & \rightarrow & \mathbf{a} <q_0, Z, q_1> \\ R_5 & <q_0, A, q_0> & \rightarrow & \mathbf{a} <q_0, A, q_0> <q_0, A, q_0> \\ R_6 & <q_0, A, q_0> & \rightarrow & \mathbf{a} <q_0, A, q_1> <q_1, A, q_0> \\ R_7 & <q_0, A, q_1> & \rightarrow & \mathbf{a} <q_0, A, q_0> <q_0, A, q_1> \\ R_8 & <q_0, A, q_1> & \rightarrow & \mathbf{a} <q_0, A, q_1> <q_1, A, q_1> \\ R_9 & <q_0, A, q_1> & \rightarrow & \mathbf{b} \\ R_{10} & <q_1, A, q_1> & \rightarrow & \mathbf{b} \end{array} \right\}$$

Langages de type 2

Les grammaires de type 2

Les automates à pile

Parsers

`Yacc`

Analyse syntaxique

- L'analyse syntaxique est employée pour déterminer si une suite de tokens correspond à une structure correcte $\rightarrow$ si le mot appartient au langage
- Les mots sont identifiés par une grammaire
- La structure correspondant à une suite de tokens est l'arbre de dérivation ou arbre syntaxique
- Si la grammaire n'est pas ambiguë, alors il n'existe qu'un seul arbre de dérivation pour chacun mot
- L'arbre syntaxique peut être obtenu de deux façons différentes
    - Par analyse descendante, en partant de l'axiome et en dérivant les symboles non-terminaux jusqu'à l'obtention du mot analysé
    - Par analyse ascendante, en partant du mot et en appliquant les règles à l'envers jusqu'à l'obtention de l'axiome

Analyse descendante – implémentation

- Stratégie pour implémenter un *parser* de ce type
  - Associer une fonction à chaque symbole non-terminal
  - Associer une fonction à chaque partie de droite de chaque règle

$$S = \mathbf{S} \;\; T = \{\mathbf{a}, \mathbf{b}\} \;\; N = \{\mathbf{S}\}$$
$$R = \left\{ \begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{a}\ \mathbf{S}\ \mathbf{b} \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{a}\ \mathbf{b} \end{array} \right\}$$

Analyse descendante – implémentation

$$S = \mathbf{S} \;\; T = \{\mathbf{a}, \mathbf{b}\} \;\; N = \{\mathbf{S}\}$$

$$R = \left\{ \begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{a}\ \mathbf{S}\ \mathbf{b} \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{a}\ \mathbf{b} \end{array} \right\}$$

```c
int parser_S(int i, char* s, int n);
int parser_ab(int i, char* s, int n);
int parser_aSb(int i, char* s, int n);

int parser_S(int i, char* s, int n) {
  int next_i = parser_aSb(i, s, n);
  next_i = (next_i == -1) ? parser_ab(i, s, n) : next_i;
  return next_i;
}

int parser_ab(int i, char* s, int n) {
  return (i + 1 < n && s[i] == 'a' && s[i + 1] == 'b') ? i + 2 : -1;
}

int parser_aSb(int i, char* s, int n) {
  if (i < n && s[i] == 'a') {
    int next_i = parser_S(i + 1, s, n);
    if (next_i != -1 && next_i < n && s[next_i] == 'b') {
      return next_i + 1;
    }
  }
  return -1;
}

int main(int argc, char** argv) {
  if (argc > 1) {
    int i = parser_S(0, argv[1], strlen(argv[1]));
    printf("%d\n", i);
  }
  return 0;
}
```

Listing 4: Analyse descendante

Analyse descendante – implémentation

- Cette stratégie souffre de défaut – donnez un exemple

Analyse descendante – implémentation

$$S = \mathbf{S}\ \ T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}\ \ N = \{\mathbf{S}\ \mathbf{R}\}$$

$$R = \left\{ \begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{R}\ \mathbf{c} \\ R_2 & \mathbf{R} & \rightarrow & \mathbf{a}\ \mathbf{b}\ \mathbf{c} \\ R_3 & \mathbf{R} & \rightarrow & \mathbf{a}\ \mathbf{b} \end{array} \right\}$$

```c
int parser_S(int i, char* s, int n);
int parser_Rc(int i, char* s, int n);
int parser_R(int i, char* s, int n);
int parser_abc(int i, char* s, int n);
int parser_ab(int i, char* s, int n);

int parser_S(int i, char* s, int n) {
  return parser_Rc(i, s, n);
}

int parser_Rc(int i, char* s, int n) {
  int next_i = parser_R(i, s, n);
  if (next_i != -1 && next_i < n && s[next_i] == 'c') {
    return next_i + 1;
  }
  return -1;
}

int parser_R(int i, char* s, int n) {
  int next_i = parser_abc(i, s, n);
  next_i = (next_i == -1) ? parser_ab(i, s, n) : next_i;
  next_i = (next_i == -1) ? parser_ba(i, s, n) : next_i;
  return next_i;
}

int parser_abc(int i, char* s, int n) {
  return (i + 2 < n && s[i] == 'a' && s[i + 1] == 'b' && s[i + 2] == 'c') ? i + 3 :
        -1;
}
```

Listing 5: Analyse descendante

Analyse descendante – implémentation

$$S = \mathbf{S} \quad T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \quad N = \{\mathbf{S}\ \mathbf{R}\}$$

$$R = \left\{ \begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{R}\ \mathbf{c} \\ R_2 & \mathbf{R} & \rightarrow & \mathbf{a}\ \mathbf{b}\ \mathbf{c} \\ R_3 & \mathbf{R} & \rightarrow & \mathbf{a}\ \mathbf{b} \end{array} \right\}$$

```c
int parser_ba(int i, char* s, int n) {
  return (i + 1 < n && s[i] == 'b' && s[i + 1] == 'a') ? i + 2 : -1;
}

int parser_ab(int i, char* s, int n) {
  return (i + 1 < n && s[i] == 'a' && s[i + 1] == 'b') ? i + 2 : -1;
}

int main(int argc, char** argv) {
  if (argc > 1) {
    int i = parser_S(0, argv[1], strlen(argv[1]));
    printf("%d\n", i);
  }
  return 0;
}
```

Listing 6: Analyse descendante

- Dans la fonction `parser_S`, si l'invocation de `parser_R` réussie en dérivant *abc*, mais que le caractère suivant n'est pas *c*, alors on devrait revenir en arrière et essayer les autres alternatives pour `parser_R` $\rightarrow$ cette implémentation ne le permet pas

Analyse descendante – implémentation

- Plus dénéralement, que va-t-il se passer lors de la traduction en `C` de la règle $A \rightarrow B|C|D$?
- Pour résoudre ce problème, soit nous mémorisons le chemin emprunté pour pouvoir revenir en arrière, soit nous modifions la grammaire pour qu'elle se présente sous une forme plus facile à manipuler
- Si la production à appliquer peut être déterminée sans ambiguité, étant donné le caractère courant sur la chaîne d'entrée, alors la grammaire est dite prédictive $\rightarrow$ l'analyse ne nécessite pas de retours en arrière
- Certaines grammaires non-ambiguës ne sont pas prédictives, comme dans l'exemple précédent, mais parfois elles peuvent être transformées en grammaires prédictives
- Savoir si une grammaire est prédictive est un problème indécidable
- Pour rendre prédictive, si possible, il faut mener une factorisation à gauche et éliminer la récursivité gauche

Grammaires prédictives

- Soient deux fonctions *PREMIER* et *SUIVANT*
- $PREMIER(\alpha)$ contient l'ensemble des symboles terminaux qui commencent une dérivation de $\alpha \in (\Sigma \cup N)^*$
  $PREMIER(\alpha) = \{a \in \Sigma : \alpha \Rightarrow^* a\beta\} \cup \{\epsilon : \alpha \Rightarrow^* \epsilon\}$
- $SUIVANT(A)$ contient l'ensemble des symboles terminaux qui peuvent suivre $A \in N$ dans une dérivation
  $SUIVANT(A) = \{a \in \Sigma : S \Rightarrow^* \alpha A a \beta\}$
- Ces deux éléments permettent de calculer une table d'analyse prédictive notée $T[A][a]$, qui indique la règle á appliquer lorsque le symbole non-terminal A doit être dérivé alors que le prochain symbole est a
- Quelle est l'utilité de *SUIVANT* ?

Algorithme pour le calcul de *PREMIER*

1. $\forall a \in T, PREMIER(a) = \{a\}$
2. $PREMIER(\epsilon) = \{\epsilon\}$
3. Pour toutes les règles $(A, \alpha) \in R$:
    1. Soit le plus grand $i \in [0; |\alpha|]$ tel que $\forall j \leq i, \epsilon \in PREMIER(\alpha_j)$
    2. Si $i < |\alpha|$, ajouter $PREMIER(\alpha_j)/\{\epsilon\}$ à $PERMIER(A)$, $\forall j \leq i + 1$
    3. Sinon, ajouter $\{\epsilon\}$ et $PREMIER(\alpha_j)$ à $PERMIER(A)$, $\forall j \leq |\alpha|$
4. Répéter le point précédent tant que des éléments sont ajoutés à *PREMIER*
5. Cette définition peut être étendue pour une séquence de symboles α :
    1. Soit le plus grand $i \in [0; |\alpha|]$ tel que $\forall j \leq i, \epsilon \in PREMIER(\alpha_j)$
    2. Si $i < |\alpha|$, ajouter $PREMIER(\alpha_j)/\{\epsilon\}$ à $PERMIER(\alpha)$, $\forall j \leq i + 1$
    3. Sinon, ajouter $\{\epsilon\}$ et $PREMIER(\alpha_j)$ à $PERMIER(\alpha)$, $\forall j \leq |\alpha|$

Algorithme pour le calcul de *SUIVANT*

1. $SUIVANT(S) = \{EOF\}$
2. Pour toutes les régles de la forme $(A, \alpha B \beta) \in R$:
    1. Ajouter $PREMIER(\beta)/\{\epsilon\}$ à $SUIVANT(B)$
    2. Si $\epsilon \in PREMIER(\beta)$ ou $\beta = \epsilon$, ajouter $SUIVANT(A)$ à $SUIVANT(B)$
3. Répéter le point précédent tant que des éléments sont ajoutés à *SUIVANT*

Application

S=**E** T={**(**, **)**, **n**, *****, $+$} N={**E**, **T**, **F**, **E'**, **T'**}

R_1	**E**	→	**T**
R_2	**T**	→	**F**
R_3	**F**	→	**(E)**
R_4	**F**	→	**n**
R_5	**E**	→	**T E'**
R_6	**E'**	→	$+$ **T**
R_7	**E'**	→	$+$ **T E'**
R_8	**T**	→	**F T'**
R_9	**T'**	→	*** F**
R_{10}	**T'**	→	*** F T'**

Construction de la table

1. Pour chaque couple $(A, a) \in N \times T$, $T[A][a] = \emptyset$
2. Pour toutes les règles $(A, \alpha) \in R$:
    1. Pour chaque $a \in PREMIER(\alpha)$, ajouter α à $T[A][a]$
    2. Si $\epsilon \in PREMIER(\alpha)$:
        1. Pour chaque $a \in SUIVANT(A)$, ajouter α à $T[A][a]$

- Si une cellule de T contient plus d'un élément, alors la grammaire n'est pas LL(1)

Grammaires prédictives

FIGURE – Algorithme

function ANALYSEDESCENDANTE(T)
    $Stack \leftarrow \{Z\}$
    $a \leftarrow next\_token()$
    while des symboles doivent être lus **do**
        $A \leftarrow Stack.pop()$
        if $A \in N \wedge T[A][a] = A \rightarrow \alpha_1\alpha_2\ldots\alpha_n$ **then**
            for all $\alpha \in (\alpha_n\ldots\alpha_2\alpha_1)$ **do**
                $Stack.push(\alpha)$
        if $A \in T \wedge A = a$ **then**
            $a \leftarrow next\_token()$
        if $A \in T \wedge A \neq a$ **then**
            $error()$
end function

Unger – Approche

- Cette grammaire engendre-t-elle le mot *aba* ?

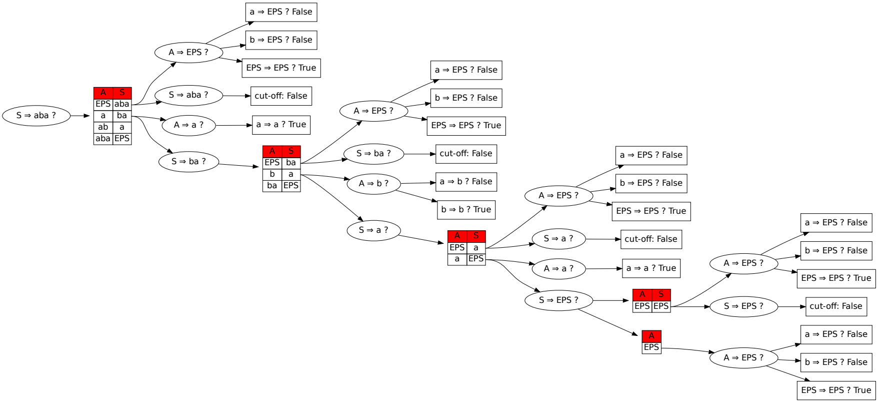
```
S:  S
N:  S A B
T:  'a' 'b'
R:  S -> A S
    S -> A
    A -> 'a'
    A -> 'b'
    A ->
```

- Peut-il être dérivé de l'axiome ?
- Est-il engendré par la règle $S \rightarrow A$? $\quad S \rightarrow A \Rightarrow aba$?
- Est-il engendré par la règle $S \rightarrow AS$? $\quad S \rightarrow AS \Rightarrow aba$?
  - Quelle est la partie de *aba* que pourrait engendrer A ?
    ϵ, *a*, *ab*, *aba* ?
  - Quelle est la partie de *aba* que pourrait engendrer S ?
    aba, *ba*, *a*, ϵ ?

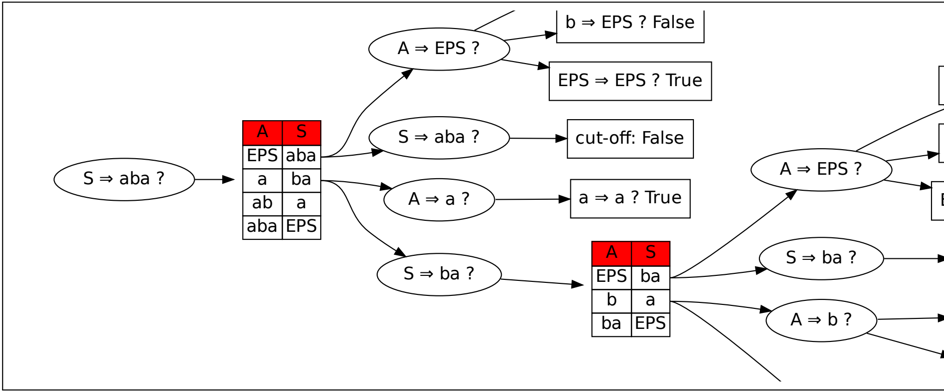
Unger – Approche

- Pour vérifier si *aba* peut être dérivé à partir du symbole non terminal S, il faut trouver une règle avec S pour partie gauche telle qu'il existe une partition de *aba* en radicaux pour laquelle chaque radical peut être dérivé par le symbole correspondant de la partir droite de la règle
  $\Rightarrow$ Non-Directionnel
- La vérification continue récursivement, en considérant chaque radical avec le symbole correspondant dans la partie droite de la règle
  $\Rightarrow$ *depth-first search*
- L'algorithme débute avec l'axiome
  $\Rightarrow$ *Top-Down*

Illustration



Illustration



- Le chemin depuis la racine au point de vérification doit être mémorisé : si la vérification courante a déjà été rencontrée précédemment, elle stoppe en indiquant *False*

Implémentation

```
import itertools

grammar = {
  "S": "S",     "N": ["S", "A", "B"],     "T": ["a", "b"],
  "R": [  ["S",    ["A", "S"]],   ["S",    ["A"]],
          ["A",    ["a"]],         ["A",    ["b"]],   ["A",    []],  ],
}
string = [ "a", "b", "a" ]

def unger(path, string, symbol, grammar):
  if symbol in grammar["T"]:
    return len(string) == 1 and string[0] == symbol
  if not [symbol, string] in path:
    for rule in grammar["R"]:
      if rule[0] == symbol:
        if all([i in grammar["T"] for i in rule[1]]):
          if string == rule[1]:
            return True
        else:
          indices = range(len(string) + 1)
          nb_cut = len(rule[1]) - 1
          for partition in itertools.combinations_with_replacement(indices, nb_cut):
            sub_strings = zip([0] + list(partition), list(partition) + [None])
            sub_strings = [string[i:j] for i, j in sub_strings]
            match = True
            for i, sub_string in enumerate(sub_strings):
              if not unger(path + [[symbol, string]], sub_string, rule[1][i], grammar):
                  match = False
                  break
            if match:
              return True
  return False

print(unger([], string, grammar["S"], grammar))
```

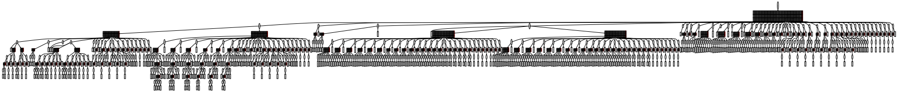
Un autre exemple...

- Soit la grammaire :

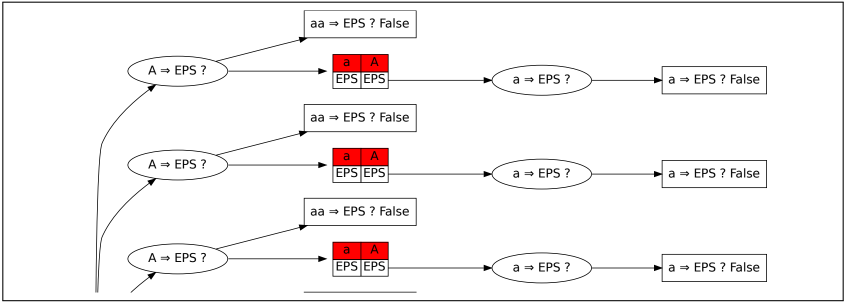
```
S:  S
N:  S A B C
T:  'a' 'b' 'c' 'd'
R:  S -> A B S
    S -> A C S
    S -> 'd'
    A -> 'a' 'a'
    A -> 'a' A
    B -> 'b'
    B -> 'b' B
    C -> 'c'
    C -> 'c' C
```

- Cette grammaire engendre-t-elle le mot *aaabbaaccd* ?

Un autre exemple...

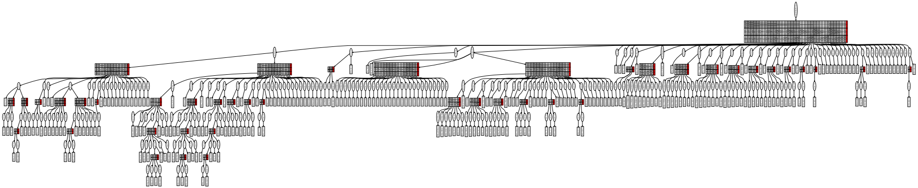


- 1413 “nœuds” !



Piste d'amélioration

- Mémorisation des questions et verdicts rencontrés
- Lors d'une invocation de la fonction récursive, si la question a déjà été rencontrée, retourner le verdict mémorisé
- Les questions et verdicts peuvent également être mémorisés d'une analyse d'une phare à une autre
- Lien avec le parcours en profondeur ?



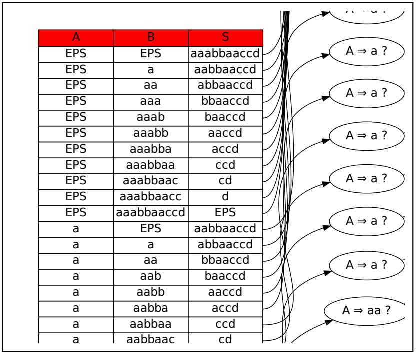
- 659 "nœuds" ! Est-il encore possible de faire mieux ?

Piste d'amélioration

- Rappel de la grammaire :

```
S:  S
N:  S A B C
T:  'a' 'b' 'c' 'd'
R:  S -> A B S
    S -> A C S
    S -> 'd'
    A -> 'a' 'a'
    A -> 'a' A
    B -> 'b'
    B -> 'b' B
    C -> 'c'
    C -> 'c' C
```

- Inutile de tenter de dériver ϵ depuis A !



A	B	S
EPS	EPS	aaabbaaccd
EPS	a	aabbaaccd
EPS	aa	abbaaccd
EPS	aaa	bbaaccd
EPS	aaab	baaccd
EPS	aaabb	aaccd
EPS	aaabba	accd
EPS	aaabbaa	ccd
EPS	aaabbaac	cd
EPS	aaabbaacc	d
EPS	aaabbaaccd	EPS
a	EPS	aabbaaccd
a	a	abbaaccd
a	aa	bbaaccd
a	aab	baaccd
a	aabb	aaccd
a	aabba	accd
a	aabbaa	ccd
a	aabbaac	cd

Piste d'amélioration

- Connaître la taille minimale des phrases engendrées par chaque symbole non-terminal permet d'ignorer les partitions invalides
- Donnez, dans l'exemple précédent, la taille minimale des phrases engendrées par les différents symboles
- La difficulté dans le calcul de cette information provient de la récursivité directe ou indirecte des règles

$\Rightarrow$ Ces règles ne permettent pas d'engendrer des phrases de taille minimale et elles sont supprimées

Piste d'amélioration

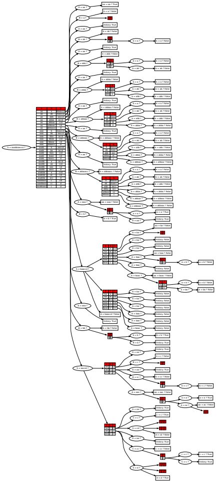
```
def get_len_min(grammar):
  rules = [[r[0], r[1][:]] for r in grammar["R"] if r[0] not in r[1]]
  order = dict(zip(grammar["N"], range(len(grammar["N"]))))
  for i, n in enumerate(grammar["N"]):
    j = 0
    while j < len(rules):
      rule = rules[j]
      if order[rule[0]] > i and n in rule[1]:
        del rules[j]
        k = rule[1].index(n)
        for n_rule in [r for r in rules if r[0] == n]:
          new_rule = [rule[0], rule[1][:k] + n_rule[1] + rule[1][k+1:]]
          if new_rule[0] not in new_rule[1] and new_rule not in rules:
            rules.append(new_rule)
      else:
        j = j + 1
  mins = {}
  for i in range(len(grammar["N"]) -1, -1, -1):
    for rule in [r for r in rules if order[r[0]] == i]:
      rule_len = sum([1 for s in rule[1] if s in grammar["T"]])
      rule_len = rule_len + sum([mins[s] for s in rule[1] if s in grammar["N"]])
      if rule[0] not in mins or mins[rule[0]] > rule_len:
        mins[rule[0]] = rule_len
  for x in grammar["T"]:
    mins[x] = 1
  return mins
```

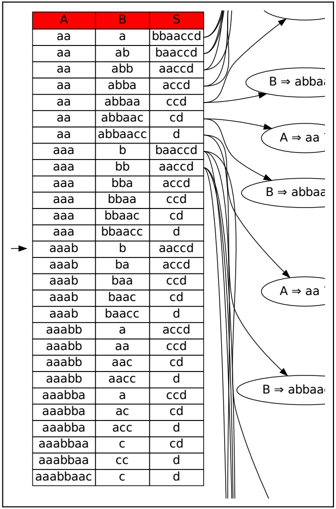
Piste d'amélioration

- Pour l'exemple :

```
S:  S
N:  S A B C
T:  'a' 'b' 'c' 'd'
R:  S -> A B S
    S -> A C S
    S -> 'd'
    A -> 'a' 'a'
    A -> 'a' A
    B -> 'b'
    B -> 'b' B
    C -> 'c'
    C -> 'c' C
Length min:  {'A': 2, 'a': 1, 'C': 1, 'B': 1, 'd': 1, 'c': 1, 'S': 1, 'b': 1}
```

Piste d'amélioration





A	B	S
aa	a	bbaaccd
aa	ab	baaccd
aa	abb	aaccd
aa	abba	accd
aa	abbaa	ccd
aa	abbaac	cd
aa	abbaacc	d
aaa	b	baaccd
aaa	bb	aaccd
aaa	bba	accd
aaa	bbaa	ccd
aaa	bbaac	cd
aaa	bbaacc	d
aaab	b	aaccd
aaab	ba	accd
aaab	baa	ccd
aaab	baac	cd
aaab	baacc	d
aaabb	a	accd
aaabb	aa	ccd
aaabb	aac	cd
aaabb	aacc	d
aaabba	a	ccd
aaabba	ac	cd
aaabba	acc	d
aaabbaa	c	cd
aaabbaa	cc	d
aaabbaac	c	d

- 179 "nœuds" ! Est-il encore possible de faire mieux ?
- B peut-il engendrer une phrase commencant par a ?

Piste d'amélioration

- Pour un symbole non-terminal qui n'engendre pas la phrase vide, connaître l'ensemble des préfixes non nuls des phrases que ce symbole peut engendrer permet d'ignorer les partitions invalides
- Compromis entre la longueur de ces préfixes, leur nombre et la complexité de l'algorithme permettant de les obtenir

Piste d'amélioration

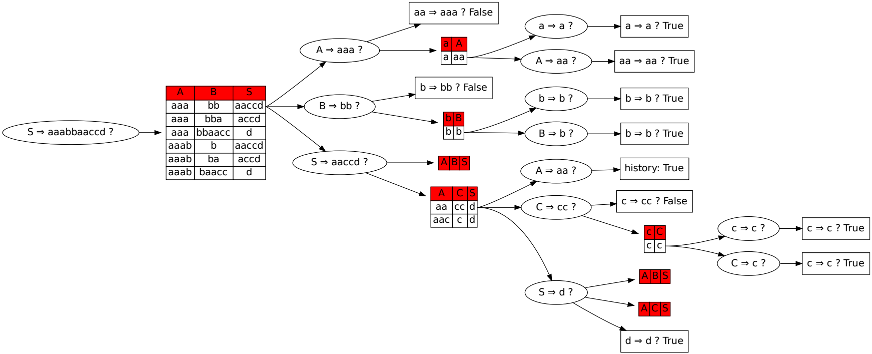
```python
import collections

def get_prefixe(len_min, grammar):
  pa, pb = collections.defaultdict(list), None
  while pb != pa:
    pa, pb = collections.defaultdict(list), pa
    for n in grammar["N"]:
      nrs = [r for r in grammar["R"] if n == r[0]]
      i = 0
      while i < len(nrs):
        if len(nrs[i][1]) == 0:
          if [] not in pa[n]:
            pa[n].append([])
        elif nrs[i][1][0] in grammar["N"]:
          pa[n].extend([j for j in pb[nrs[i][1][0]] if len(j) > 0 and j not in pa[n]])
          if len_min[nrs[i][1][0]] == 0:
            nrs.append([nrs[i][0], nrs[i][1][1:]])
        else:
          j = ([j in grammar["T"] for j in nrs[i][1]] + [False]).index(False)
          if nrs[i][1][:j] not in pa[n]:
            pa[n].append(nrs[i][1][:j])
        i = i + 1
  for k, v in dict(pa).items():
    for i in range(len(v) -1, -1, -1):
      if any([v[i][:len(w)] == w for j, w in enumerate(v) if i != j]):
        del v[i]
  return dict(pa, **dict(zip(grammar["T"], [[[x]] for x in grammar["T"]])))
```

Piste d'amélioration

- Pour l'exemple :

```
S:   S
N:   S A B C
T:   'a' 'b' 'c' 'd'
R:   S -> A B S
     S -> A C S
     S -> 'd'
     A -> 'a' 'a'
     A -> 'a' A
     B -> 'b'
     B -> 'b' B
     C -> 'c'
     C -> 'c' C
 Prefixes : A:['a']  a:['a']  C:['c']  B:['b']
            d:['d']  c:['c']  S:['a', 'd']  b:['b']
```

Piste d'amélioration



- 33 "nœuds" ! Est-il encore possible de faire mieux ?

⇒ Etude de la taille maximale des phrases engendrées par les non-terminaux, des suffixes, des radicaux, etc.

Analyse ascendante

- Principe : on part du texte source (lecture de gauche à droite) et on cherche à remonter jusqu'à l'axiome en procédant par des actions de lecture ou de réduction
- Lecture (*shift*) : consommer et empiler un token dans la pile
- Réduction (*reduce*) : reconnître la partie droite d'une production au sommet de la pile et la transformer avec le non-terminal correspondant
- $\rightarrow$ pile pour mémoriser les tokens et les réductions

Analyse ascendante

- Initialement la pile est vide
- L'algorithme poursuit son exécution tant que des *shift* ou *reduce* sont possibles
- Si plus aucune action ne peut être réalisée alors
  - Si tous les tokens ont été consommé et la pile contient uniquement l'axiome alors le mot est reconnu
  - Sinon le mot n'est pas reconnu
- *Quid* des actions associées aux règles ?

Analyse ascendante

- Soit la grammaire

$$S = \mathbf{S} \;\; T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \;\; N = \{\mathbf{S}\ \mathbf{X}\ \mathbf{Y}\}$$

$$R = \left\{ \begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{X}\ \mathbf{Y} \\ R_2 & \mathbf{X} & \rightarrow & \mathbf{a}\ \mathbf{X} \\ R_3 & \mathbf{X} & \rightarrow & \mathbf{c}\ \mathbf{Y} \\ R_4 & \mathbf{Y} & \rightarrow & \mathbf{b} \end{array} \right\}$$

- Décrire le langage engendré
- Analysez la chaîne *acb*
- Analysez la chaîne *aacb*
- Analysez la chaîne *aacbb*

Langages de type 2

Les grammaires de type 2

Les automates à pile

Parsers

`Yacc`

Introduction à `Yacc`

- `Yacc` est l'acronyme de *Yet another compiler-compiler*
- Il permet de `parser` une séquence de tokens en se basant sur une grammaire *hors-contexte*
- De manière générale, un `parser` vérifie qu'une phrase peut effectivement être générée par une grammaire
- Il existe deux approches
  - Partir de l'axiome et guider les dérivations jusqu'à obtenir la phrase ou s'appercevoir qu'elle ne peut pas être engendrée par la grammaire
  - Partir de la phrase et retrouver les règles qui ont permis de l'engendrer (approche *bottom-up* ou *shift-reduce*)
- `Yacc` utilise une approche *shift-reduce* : il consomme les tokens en entrée au fur et à mesure en les placant dans une pile (*shift*). Lorsque le sommet de la pile (éventuellement plusieurs symboles) correspond à la partie droite d'une règle de la grammaire, il remplace les symboles correspondant par la partie gauche de cette règle (*reduce*).

Structure d'un fichier `Yacc`

- Similaire à la structure d'un fichier `Lex`

```
%{
  Zone contenant la déclarations de variable et
  inclusion de bibliothèques
%}
  Zone contenant les définitions
%%
  Zone contenant les règles
%%
  Zone contenant les fonctions
```

Zone contenant les définitions

- La zone des définitions permet :
  - De définir les tokens du langage `%token tNB`
    $\equiv$ symboles terminaux de la grammaire
    Ils peuvent être définis sur la même ligne `%token tNB tID`
    ou sur des lignes différentes en répétant `%token`
  - De définir l'associativité de certains opérateurs
    Avec `%left tPLUS`, l'expression $a + b + c$ sera évalué $(a + b) + c$
  - De définir l'axiome de départ S
    `%start Input`
- Par convention, nous nommerons les tokens avec, en première lettre, un t miniscule et ensuite uniquement des lettres majuscules

Zone contenant les règles

- La syntaxe utilisée pour l'écriture des règles est semblable à la syntaxe ABNF
  Exemple : `Input: /* Vide */ | Input tNB Ligne ;`
- Par convention, nous nommerons les règles avec, en première lettre, une majuscule et ensuite uniquement des lettres miniscules
- Une action peut être associée à la réduction d'une règle
  Exemple :

```
%{
#include <stdio.h>
%}

%token tNB tFL
%start Input

%%

Input : Ligne Input | Ligne ;
Ligne : tNB Suite {printf("NOMBRE_LIGNE\n");};
Suite : tNB tFL {printf("NOMBRE_SUITE\n");}
```

Listing 7: ligne.y

- Attention à l'ordre d'exécution de ces règles !

Zone contenant les règles

- Les actions peuvent être entrelacées avec les symboles de la règle et elles seront exécutées lors de la réduction de cette règle
  Exemple :

```
%{
#include <stdio.h>
%}

%token tNB tFL
%start Input

%%

Input : Ligne Input | Ligne ;
Ligne : tNB {printf("NOMBRE_LIGNE\n");} Suite;
Suite : tNB tFL {printf("NOMBRE_SUITE\n");}
```

Listing 8: ligne2.y

Zone contenant les règles

- Exemple avec la grammaire S=**Liste** T={**andre**, **liam**, **phil**, **et**, **,**}
  N={**Liste**, **Prenom**, **Prenoms**}

R_1	**Liste**	$\rightarrow$	**Prenom**
R_2	**Liste**	$\rightarrow$	**Prenoms**
R_3	**Prenoms**	$\rightarrow$	**Prenom et Prenom**
R_4	**Prenoms**	$\rightarrow$	**Prenom , Prenoms**
R_5	**Prenom**	$\rightarrow$	**andre**
R_6	**Prenom**	$\rightarrow$	**liam**
R_7	**Prenom**	$\rightarrow$	**phil**

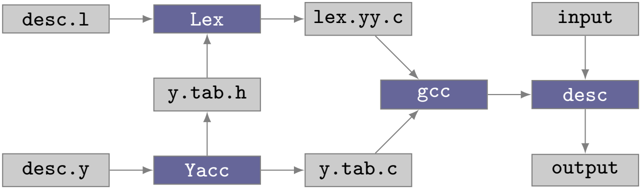
- Fichier `Yacc` associé

```
%{
#include <stdio.h>
%}

%token tLIAM tPHIL tANDRE tET tVIRGULE tFL
%start Start

%%

Start : Liste tFL {printf("OK!\n");} ;
Liste : Prenom | Prenoms ;
Prenoms : Prenom tET Prenom ;
Prenoms : Prenom tVIRGULE Prenoms ;
Prenom : tANDRE | tLIAM | tPHIL ;
```

Listing 9: prenoms.y

Combinaison de Lex et Yacc

- Lex peut être utilisé pour alimenter Yacc avec des tokens : Lex joue le rôle d'analyseur lexical et Yacc d'analyseur syntaxique
- Yacc définit les tokens à utiliser
- La passerelle entre Lex et Yacc est assurée par le fichier y.tab.h – il doit être inclus directement ou indirectement dans le fichier de description de Lex



- Commandes

Yacc :	`yacc -d desc.y`
Lex :	`lex desc.l`
gcc :	`gcc -ll -ly lex.yy.c y.tab.c -o desc`
desc :	`./desc < input > output`

Combinaison de Lex et Yacc

- Fichier Lex associé à la liste des prénoms

```
%{
#include "y.tab.h"
%}

%%

liam     {return tLIAM;}
phil     {return tPHIL;}
andre    {return tANDRE;}
,        {return tVIRGULE;}
et       {return tET;}
\n       {return tFL;}
"␣"      {}
```

Listing 10: prenoms.l

```
%{
#include <stdio.h>
%}

%token tLIAM tPHIL tANDRE tET tVIRGULE tFL
%start Start

%%

Start : Liste tFL {printf("OK!\n");} ;
Liste : Prenom | Prenoms ;
Prenoms : Prenom tET Prenom ;
Prenoms : Prenom tVIRGULE Prenoms ;
Prenom : tANDRE | tLIAM | tPHIL ;
```

Listing 11: prenoms.y

Combinaison de Lex et Yacc

- Contenu du fichier `y.tab.h`

```
/* Tokens.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers
      know about them.  */
   enum yytokentype {
     tLIAM = 258,
     tPHIL = 259,
     tANDRE = 260,
     tET = 261,
     tVIRGULE = 262,
     tFL = 263
   };
#endif
/* Tokens.  */
#define tLIAM 258
#define tPHIL 259
#define tANDRE 260
#define tET 261
#define tVIRGULE 262
#define tFL 263
#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef int YYSTYPE;
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
# define YYSTYPE_IS_TRIVIAL 1
#endif
extern YYSTYPE yylval;
```

Listing 12: prenoms-y.tab.h.extrait

Combinaison de Lex et Yacc

- Attention, vous devez vous assurer de traiter avec Lex tout ce qui se doit. Sinon, les séquences de caractères qui ne sont pas traitées et qui devraient l'être seront affichées sur la sortie et ne seront pas transmises à Yacc

```
%{
#include "y.tab.h"
%}
%%
[0-9]+  {return tNB;}
\n      {return tFL;}
```

Listing 13: erreur.l

```
%{
#include <stdio.h>
%}
%token tNB tFL tEGAL
%start Input
%%
Input : Ligne Input | Ligne ;
Ligne : tNB tEGAL tNB {printf("OK\n");};
```

Listing 14: erreur.y

- Exécution

```
bash-3.2$ ./erreur
45 = 45
syntax error
 = bash-3.2$
```

Exemple

- Construire un parser permettant de vérifier la grammaire d'un document écrit en français
- Construire un parser permettant de vérifier qu'un fichier HTML est bien formé

```
%{
#include "y.tab.h"
%}
%%
"<html>"      {return tDHTML;}
"</html>"     {return tFHTML;}
"<img>"       {return tDIMG;}
"</img>"      {return tFIMG;}
"<body>"      {return tDBODY;}
"</body>"     {return tFBODY;}
[a-zA-Z0-9]*  {return tTEXTE;}
.             {}
```

Listing 15: html.l

```
%{
#include "y.tab.h"
%}
%%
"<html>"      {return tDHTML;}
"</html>"     {return tFHTML;}
"<img>"       {return tDIMG;}
"</img>"      {return tFIMG;}
"<body>"      {return tDBODY;}
"</body>"     {return tFBODY;}
[a-zA-Z0-9]*  {return tTEXTE;}
.             {}
```

Listing 16: html.y

Attributs des symboles

- Certains langages peuvent être très riches et évolutifs
- Par exemple, le langage `HTML` permet l'ajout de nouvelles balises
- Dans l'exemple précédent, nous sommes obligés de modifier notre grammaire pour considérer de nouvelles balises

$\rightarrow$ Manque de souplesse – Que faire ?

Attributs des symboles

- Dans notre exemple, nous pourrions considérer le fichier `Yacc` suivant :

```
%{
#include <stdio.h>
%}
%token tBALISEOUVRANTE
%token tBALISEFERMANTE
%token tTEXTE
%start Document
%%
Document : tBALISEOUVRANTE Corps
       tBALISEFERMANTE
    {printf("OK\n");} ;
Corps :
    /* Vide */
  | tBALISEOUVRANTE Corps tBALISEFERMANTE
  | tTEXTE ;
```

Listing 17: html2.y

- Quels sont les problèmes de cette approche ?
- Le document `<html>texte</machin>` est considéré bien formé
  $\rightarrow$ nous avons changé le langage

Attributs des symboles

- Il faudrait être capable de vérifier que la séquence de caractères à l'origine du token `tBALISEOUVRANTE` est la même que celle à l'origne du token `tBALISEFERMANTE`
- Utilisation d'une variable globale renseignée par `Lex`, pour chaque token ?
- *Comment gérer les règles avec plusieurs tokens du même type ? Ainsi que les règles récursives ? La variable globale risque d'être écrasée ! Peut-être créer plusieurs variables globales ? Combien ?*
- Il est possible d'attribuer des valeurs à des tokens
- De cette manière, `Lex` peut non seulement informer `Yacc` sur la nature des tokens identifiés mais également il peut fournir des informations complémentaires (valeur d'un nombre, chaîne de caractères, etc)

Attributs des symboles

- Comment faire ?
  - Indiquer les différentes informations que `Lex` va pouvoir transmettre à `Yacc`
    `%union {int nb; char *str;}`
  - Indiquer le type d'information associé aux tokens
    `%token <nb> tNB`
  - Les symboles non-terminaux peuvent ègalement être associés à des informations – `Yacc` se charge de cette association
    `%type <str> Ligne`
  - Dans les actions, les valeurs associées aux symboles peuvent être récupérées avec la notation \$1, \$2, etc
  - Dans une action, l'association d'une valeur au symbole non-terminal correspondant à la partie gauche de la règle se fait avec la notation \$\$
- Les espaces mémoires alloués par `Lex` doivent être libérés par `Yacc`

Exemple

- Construire une calculatrice

```
%{
#include <stdlib.h>
#include <stdio.h>
#include "y.tab.h"
%}

%%

[ \t]+  {};
[0-9]+  {
          yylval.nb = atoi(yytext);
          return tNB;
        }
"="     { return tEGAL; }
"-"     { return tSOU; }
"+"     { return tADD; }
"*"     { return tMUL; }
"/"     { return tDIV; }
"("     { return tPO; }
")"     { return tPF; }
[a-z]   {
          yylval.var = yytext[0];
          return tID;
        }
\n      { return tFL; }
.       { return tERROR; }
```

Listing 18: calc.l

Exemple

- Construire une calculatrice

```
%{ /* EXEMPLE DE SYNTAXE A NE PAS SUIVRE ! */
#include <stdlib.h>
#include <stdio.h>
int var[26];
void yyerror(char *s);
%}
%union { int nb; char var; }
%token tFL tEGAL tPO tPF tSOU tADD tDIV tMUL tERROR
%token <nb> tNB
%token <var> tID
%type <nb> Expr DivMul Terme
%start Calculatrice
%%
Calculatrice :    Calcul Calculatrice | Calcul ;
Calcul :          Expr tFL { printf("> %d\n", $1); }
                | tID tEGAL Expr tFL { var[(int)$1] = $3; } ;
Expr :            Expr tADD DivMul { $$ = $1 + $3; }
                | Expr tSOU DivMul { $$ = $1 - $3; }
                | DivMul { $$ = $1; } ;
DivMul :          DivMul tMUL Terme { $$ = $1 * $3; }
                | DivMul tDIV Terme { $$ = $1 / $3; }
                | Terme { $$ = $1; } ;
Terme :           tPO Expr tPF { $$ = $2; }
                | tID { $$ = var[$1]; }
                | tNB { $$ = $1; } ;
%%
void yyerror(char *s) { fprintf(stderr, "%s\n", s); }
int main(void) {
  printf("Calculatrice\n"); // yydebug=1;
  yyparse();
  return 0;
}
```

Listing 19: calc.y

Conflits

- `Yacc` utilise une approche *shift-reduce*
- Deux types de conflits peuvent se produire *reduce*/*reduce* et *shift*/*reduce*

Conflits

- Conflit *reduce/reduce*

```
%{
#include "y.tab.h"
%}
%%
[0-9]+  {return tNB;}
.       {}
```

Listing 20: conflit1.l

```
1  %{
2  #include <stdio.h>
3  %}
4  %token tNB
5  %start Start
6  %%
7  Start : Expr | Expr Start;
8  Expr :  tNB   {printf("2\n");};
9  Expr :  Terme {printf("1\n");};
10 Terme : tNB   {printf("3\n");};
```

Listing 21: conflit1a.y

```
1  %{
2  #include <stdio.h>
3  %}
4  %token tNB
5  %start Start
6  %%
7  Start : Expr | Expr Start;
8  Terme : tNB   {printf("3\n");};
9  Expr :  tNB   {printf("2\n");};
10 Expr :  Terme {printf("1\n");};
```

Listing 22: conflit1b.y

```
bash-3.2$ make
yacc -d conflit1a.y
conflicts: 2 reduce/reduce
conflit1a.y:10.9-30: warning: rule never reduced because of conflicts: Terme: tNB
lex conflit1.l
gcc -ly -ll lex.yy.c y.tab.c -o conflit1a
yacc -d conflit1b.y
conflicts: 2 reduce/reduce
conflit1b.y:9.9-30: warning: rule never reduced because of conflicts: Expr: tNB
lex conflit1.l
gcc -ly -ll lex.yy.c y.tab.c -o conflit1b
```

Conflits

- Conflit *shift*/*reduce* → `Yacc` choisit le *shift*

```
%{
#include <stdio.h>
%}
%token tINSTR tIF tELSE tNL
%%
S :             Instrs S                        | Instrs tNL;
Instrs :        tINSTR                          | Ififelse;
Ififelse :      tIF Instrs { printf("IF\n");}   | tIF Instrs tELSE Instrs { printf("IFELSE\n");};
%%
int yylex() {
  int c = getchar();
  if (c == EOF) {
    return 0;
  } else if (c == 'i') {
    return tIF;
  } else if (c == 'a') {
    return tINSTR;
  } else if (c == 'e') {
    return tELSE;
  } else if (c == '\n') {
    return tNL;
  }
  return c;
}
```

Listing 23: shift.y

```
bash-3.2$ make
yacc -d shift.y
conflicts: 1 shift/reduce
gcc -ly y.tab.c -o shift
```