Cours d'automates et langages et applications

E. Alata

INSA Toulouse - 4ième IR

Introduction

Présentation du langage `Python`

Les langages

Rappels sur les expressions régulières

Les grammaires

Les grammaires de type 1

Les grammaires de type 2

Les grammaires de type 3

`Lex`

`Yacc`

`Compilation`

Part I

Notations 1/3

- Alphabet : Σ
- Alphabet particulier : Σ_x
- Lettre de l'alphabet : a
- Lettre de l'alphabet, indicée : a_i
- Mot sur un alphabet : ω, x, y
- Mot vide : ϵ
- Longueur d'un mot sur un alphabet : $|\omega|$, $|x|$, $|y|$
- Nombre d'occurence d'un caractère dans un mot : $|\omega|_a$, $|x|_a$, $|y|_a$
- Longueur d'un mot sur une partie d'un alphabet : $|\omega|_{\Sigma_p}$, $|x|_{\Sigma_p}$, $|y|_{\Sigma_p}$
- i-ième caractère d'un mot : ω_i, x_i, y_i
- Sous-mot : $\omega_{[i;j]}$
- Mot miroir : $\tilde{\omega}$
- Répétition d'un mot : ω^k
- Langage : L
- Langage particulier : L_x
- Préfixes d'un mot : $Pref(\omega)$
- Préfixes communs : $Pc(x, y)$
- Plus long préfixe commun : $Plpc(x, y)$

Notations *2/3*

- Relation de préfixe : $\preceq_p$
- Relation d'ordre quelconque sur les mots : $\preceq_x$
- Relation d'ordre quelconque sur les caractères : $\preceq_x$
- Relation d'ordre lexicographique sur les mots : $\preceq_l$
- Relation d'ordre alphabétique sur les mots : $\preceq_a$
- Distance de préfixe : $d_p(x, y)$
- Distance d'édition : $d_e(x, y)$
- Ensemble des mots : Σ^*
- Ensemble des mots sans le mot vide : $\Sigma^+ = \Sigma^*/\epsilon$
- Fermeture de Kleene d'un langage : L^*, L^+
- Union de langages : $L_1 \cup L_2$
- Intersection de langages : $L_1 \cap L_2$
- Produit de langages : $L_1 L_2$
- Complément d'un langage : $\overline{L}$
- Différence de langages : L_1/L_2
- Ensemble des préfixes d'un langage : $Pref(L)$
- Ensemble des suffixes d'un langage : $Suff(L)$
- Ensemble des facteurs d'un langage : $Fact(L)$

Notations *3/3*

- Expression régulière : $\mathcal{r}$, $\mathcal{s}$
- Union d'expressions régulières : $\mathcal{r}|\mathcal{s}$
- Concaténation d'expressions régulières : $\mathcal{r}\mathcal{s}$
- Répétition d'expressions régulières : $\mathcal{r}^*$
- Langage représenté par une expression régulière : $L(\mathcal{r})$

Introduction

Présentation du langage Python

Les langages

Rappels sur les expressions régulières

Les langages naturels 1/1

- Le langage naturel est 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[?]

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, par exemple en aérospatial

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

  - *AeroSpace and Defence Industries Association of Europe (ASD)*
  - Réduction de l'ambiguïté
  - Constitué de règles de construction des phrases et d'un dictionnaire
  - Employé pour la définition des exigences
    RBE - Requirement Based Engineering

Simplified Technical English – Extrait des règles *1/1*

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 1/1

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.

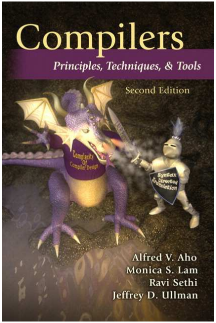
Les langages formels 1/1

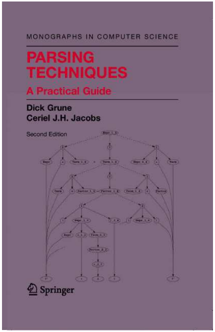
- Intuitivement
  - Ensemble de règles permettant de communiquer sans ambiguïté
  - Ensemble de mots admissibles
- Grammaire $\rightarrow$ formalisme
- Exemples d'utilisation des langages formels
  - Vérification de syntaxe
    Est-ce que mon programme C est syntaxiquement correct ?
  - Analyse de fichiers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

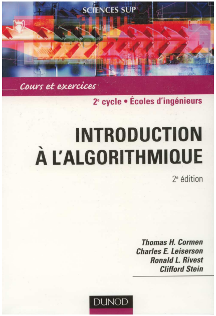
Pourquoi ce cours ? *1/1*

- Automates : dans le langage `UML`, les machines à café, ...
- Langages : dans les fichiers `RFC`, fichiers de configuration, ...
- *Parsers* : traduction liée à `XML`, `google traduction`, ...
- Compilation : fonctionnement de `gcc`, de `javac`, ...
- Décompilation : fonctionnement de `AspectJ`, analyses `WCET`, ...

Lectures conseillées 1/1







- D. Grune and C.J.H. Jacobs, *Parsing techniques - a practical guide*, Ellis Horwood, 1991
- A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman, *Compilers: Principles, techniques, and tools (2nd edition)*, 2 ed., Addison Wesley, 2006
- T. Cormen, C. Leiserson, and R. Rivest, *Introduction à l'algorithmique*, Dunod, 2004

Introduction

Présentation du langage Python

Les langages

Rappels sur les expressions régulières

Présentation générale *1/1*

- Site officiel : `http://python.org`
- Licence : `Python Software Foundation` compatible GPL
- Le langage `Python` est interprété, multi-paradigme et multi-plateforme

- L'accent a été mis sur la programmation impérative et oritentée objet
- Les interpréteurs `python` ont de nombreux points communs avec les interpréteurs de `bytecode` `Java` : ramasse miette, `bytecode` pour la gestion des objets, ...
- La suite de ce chapitre présente la syntaxe de ce langage tout en la comparant au langage `Java`

Syntaxe générale *1/1*

- Symbole pour les commentaires : #
- Blocs délimités par l'indentation

> *Tout token qui n'est pas un espace à la gauche d'un tel token sur la ligne précédente est pris comme le début d'une nouvelle déclaration*[**?**]

- Syntaxe `Python` concise ($\neq$ `Java` qui est verbeux)
- Typage dynamique fort
- Structures de contrôle : `if`, `else`, `while`, ...
- Mots-clés : `pass`, `del`, `break`, `continue`, ...
- Tests : `==`, `and`, `or`, ...

Les variables et les types 1/1

- Déclaration et utilisation des types simples

Listing 1: 02-python/pTypesSimples.py

```python
s = "Hello World"
i = 123
j = None
print(type(s))
print(type(i))
print(type(j))
i = s
print(type(i))
i = i + 2
```

Listing 2: 02-python/pTypesSimples.log

```
<class 'str'>
<class 'int'>
<class 'NoneType'>
<class 'str'>
Traceback (most recent call last):
  File "pTypesSimples.py", line 9, in <module>
    i = i + 2
TypeError: Can't convert 'int' object to str
      implicitly
```

Listing 3: 02-python/jTypesSimples.java

```java
class jTypesSimples {
  public static void main (String[] p){
    int a = 5;
    String b = "coucou";
    a = b;
    a = a + 1;
  }
}
```

Listing 4: 02-python/jTypesSimples.log

```
jTypesSimples.java:5: incompatible types
found    : java.lang.String
required: int
    a = b;
        ^
1 error
```

Les collections *1/4*

- Chaînes de caractères, dictionnaires, listes et tuples
- Contenu : objets hétérogènes
- Séquences d'objets → itérables
- Chaînes de caractères et tuples → *immuables*
- Dictionnaires et listes → *modifiables*

Les collections *2/4*

- Utilisation des listes

Listing 5: 02-python/pList.py

```python
l = ['chaine', 3, 5.0]
print(l[0], len(l))
l.append("truc")
print(l)
del l[0]
print(l)
```

Listing 6: 02-python/pList.log

```
chaine 3
['chaine', 3, 5.0, 'truc']
[3, 5.0, 'truc']
```

Listing 7: 02-python/jList.java

```java
import java.util.*;
class jList {
  public static void main (String[] p){
    ArrayList<Object> l = new ArrayList<Object>(
           Arrays.asList(new Object[]{"chaine", 3,
            5.0}));
    System.out.println(l.get(0) + "" + l.size());
    l.add("truc");
    System.out.println(l);
    l.remove(0);
    System.out.println(l);
  }
}
```

Listing 8: 02-python/jList.log

```
chaine3
[chaine, 3, 5.0, truc]
[3, 5.0, truc]
```

Les collections *3/4*

- Utilisation des listes ../..
  - Autre type similaire : le tuple $\approx$ liste / suppression
  - Fonction `map(f, l)` – application de la fonction `f` sur chaque élément de la liste `l` et retour de la liste des résultats
  - Fonction `any(f, l)` – retourne vrai si l'application de la fonction `f` sur un des éléments de la liste `l` retourne vrai, sinon retourne faux
  - `l[i:j]` retourne une nouvelle liste contenant les éléments de l'indice i à $j-1$
    - `i` vaut 0 par défaut
    - `j` vaut `len(l)` par défaut
    - `l[:]` construit un clone de la liste
    - Si `j` est négatif, alors l'indice est considéré à partir de la fin de la liste
  - `l * n` retourne une nouvelle liste contenant n fois les éléments de la liste `l`
  - `l + k` retourne une nouvelle liste contenant les éléments de `l` et de `k`
- Exercice : qu'affiche ce programme ?

Listing 9: `02-python/ex-list.py`

```
l = ['chaine', 3, 5.0]
m = l[1:]
print(m)
del m[0]
print(m)
print(l)
print(l * 2 + m)
```

Les collections 4/4

- Utilisation des dictionnaires

Listing 10: 02-python/pDictionnary.py

```python
d = {'andre':1978, 'simon':1982, 'vincent':2001, 2:'deux', 3:'trois'}
print(d)
print(d['andre'])
print(d.keys())
print(list(d.keys()))
print(d.values())
print(list(d.values()))
del d['andre']
print(d)
```

Listing 11: 02-python/pDictionnary.log

```
{3: 'trois', 'andre': 1978, 'vincent': 2001, 'simon': 1982, 2: 'deux'}
1978
dict_keys([3, 'andre', 'vincent', 'simon', 2])
[3, 'andre', 'vincent', 'simon', 2]
dict_values(['trois', 1978, 2001, 1982, 'deux'])
['trois', 1978, 2001, 1982, 'deux']
{3: 'trois', 'vincent': 2001, 'simon': 1982, 2: 'deux'}
```

Listing 12: 02-python/jDictionnary.java

```java
import java.util.*;
class jDictionnary {
  public static void print(HashMap d) {
    Iterator i = d.entrySet().iterator();
    while (i.hasNext()) {
      Map.Entry e = (Map.Entry)i.next();
      System.out.print(e.getKey()+":");
      System.out.print(e.getValue()+"/");
    }
    System.out.println();
  }
  public static void main(String p[]) {
    HashMap<Object, Object> d = new HashMap<Object, Object>();
    d.put("andre", 1978);
    d.put("simon", 1982);
    d.put("vincent", 2001);
    d.put(2, "deux");
    d.put(3, "trois");
    print(d);
    d.remove("simon");
    print(d);
  }
}
```

Structures de contrôle 1/3

- Structures de contrôle principales

Listing 13: 02-python/pControl.py

```python
# Condition 'si'
valeur = int(input('Age ? '))
if valeur >= 18:
  print('adulte')
elif valeur > 12:
  print('adolescent')
else:
  print('enfant')

# Boucle 'tant-que'
i = 0
while i != 10 :
  i = i + 1
  print("i=%d " % (i), end='')
print()

# Boucle 'pour'
for i in range(10, 20):
  print("i=%d " % i, end='')
print()
```

Listing 14: 02-python/pControl.log

```
Age ? enfant
i=1 i=2 i=3 i=4 i=5 i=6 i=7 i=8 i=9 i=10
i=10 i=11 i=12 i=13 i=14 i=15 i=16 i=17 i=18 i=19
```

Structures de contrôle 2/3

- Structures de contrôle particulières

Listing 15: `02-python/pControlComp.py`

```
# Utilisation de la fonction map sur une liste
l = [1, 5, 2, 7, 3, 16]
p = map(lambda x: x % 2 == 1, l)
print(p)
print(list(p))

# Definition d'une liste en comprehension
p = [x % 2 == 1 for x in l]
print(p)

# Iteration sur les clefs et valeurs d'un dictionnaire
d = {'truc':4, 'bidule':3, 'machin':6, 'chouette':9}
for k, v in d.items():
  print(k, ':␣', v, end="␣;␣")
print()
```

Listing 16: `02-python/pControlComp.log`

```
<map object at 0x1006c49d0>
[True, True, False, True, True, False]
[True, True, False, True, True, False]
machin :  6 ; bidule :  3 ; truc :  4 ; chouette :  9 ;
```

Structures de contrôle 3/3

- Structures de contrôle et modificateurs de déroulement
  - Il est également possible de modifier le déroulement d'une boucle en utilisant les mots-cléfs `break` et `continue`
  - `break` permet de sortir d'une boucle
  - `continue` permet un retour en début de boucle
  - Faites attention lors de la modification des itérateurs ou lors de l'utilisation des instructions `break` et `continue`
- Exercice : qu'affiche ce programme ?

Listing 17: `02-python/ex-loop.py`

```python
l = [1, 4, 3, 7, 9]
for i in l:
  print(i, end=" ")
  del l[0]
print()

l = [1, 4, 3, 7, 9]
for i in l[:]:
  print(i, end=" ")
  del l[0]
print()
```

Modules 1/1

- Les interpréteurs `Python` sont installés avec de nombreux modules permettant d'étendre les fonctionnalités
- L'utilisation d'un module se fait via la déclaration : `import nom_du_module`
- Quelques modules utiles :
  - `re` – `Regular expression operations`
  - `string` – `Common string operations`
  - `math` – `Mathematical functions`
  - `random` – `Generate pseudo-random numbers`
  - `itertools` – `Functions creating iterators for efficient looping`
  - `operator` – `Standard operators as functions`
  - `os.path` – `Common pathname manipulations`
  - `pickle` – `Python object serialization`
  - `os` – `Miscellaneous operating system interfaces`
  - `io` – `Core tools for working with streams`

Exceptions *1/1*

- Déclanchement et capture d'une exception

Listing 18: 02-python/pException.py

```
try:
  print(int("n"))
except Exception as e:
  raise Exception("zut")
```

Listing 19: 02-python/pException.log

```
Traceback (most recent call last):
  File "pException.py", line 2, in <module>
    print(int("n"))
ValueError: invalid literal for int() with base 10: 'n'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "pException.py", line 4, in <module>
    raise Exception("zut")
Exception: zut
```

Listing 20: 02-python/jException.java

```
public class jException {
  public static void main(String args[]) throws Exception {
    try {
      System.out.println(Integer.parseInt("n"));
    } catch (Exception e) {
      throw new Exception("zut");
    }
  }
}
```

Listing 21: 02-python/jException.log

```
Exception in thread "main" java.lang.Exception: zut
        at jException.main(jException.java:6)
```

Déclaration de fonctions 1/1

- En Python, il existe "uniquement" des fonctions

Listing 22: 02-python/pFunction.py

```python
import sys

def fib(n, fn1=1, fn=0):
    if (n == 0):
        return fn
    else:
        return fib(n - 1, fn, fn + fn1)

if __name__ == "__main__":
  n = int(sys.argv[1])
  print(fib(n))
```

Listing 23: 02-python/jFunction.java

```java
public class jFunction {
  public static long fib(int n) {
    if (n <= 1) {
      return n;
    } else {
      return fib(n - 1) + fib(n - 2);
    }
  }
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    System.out.println(fib(n));
  }
}
```

Bytecode 1/2

- Aperçu du bytecode Java

Listing 24: 02-python/jFunction.dis

```
Compiled from "jFunction.java"
public class jFunction extends java.lang.Object{
public jFunction();
  Code:
   0:   aload_0
   1:   invokespecial    #1; //Method java/lang/Object."<init>":()V
   4:   return

public static long fib(int);
  Code:
   0:   iload_0
   1:   iconst_1
   2:   if_icmpgt        8
   5:   iload_0
   6:   i2l
   7:   lreturn
   8:   iload_0
   9:   iconst_1
   10:  isub
   11:  invokestatic     #2; //Method fib:(I)J
   14:  iload_0
   15:  iconst_2
   16:  isub
   17:  invokestatic     #2; //Method fib:(I)J
   20:  ladd
   21:  lreturn

public static void main(java.lang.String[]);
  Code:
   0:   aload_0
   1:   iconst_0
   2:   aaload
   3:   invokestatic     #3; //Method java/lang/Integer.parseInt:(Ljava/lang/String;)I
   6:   istore_1
```

Bytecode 2/2

- Aperçu du bytecode Python

Listing 25: 02-python/pFunction.dis

```
  4           0 LOAD_FAST                0 (n)
              3 LOAD_CONST               1 (0)
              6 COMPARE_OP               2 (==)
              9 POP_JUMP_IF_FALSE       16

  5          12 LOAD_FAST                2 (fn)
             15 RETURN_VALUE

  7     >>   16 LOAD_GLOBAL              0 (fib)
             19 LOAD_FAST                0 (n)
             22 LOAD_CONST               2 (1)
             25 BINARY_SUBTRACT
             26 LOAD_FAST                2 (fn)
             29 LOAD_FAST                2 (fn)
             32 LOAD_FAST                1 (fn1)
             35 BINARY_ADD
             36 CALL_FUNCTION            3
             39 RETURN_VALUE
             40 LOAD_CONST               0 (None)
             43 RETURN_VALUE
```

Pour finir *1/1*

- Peut-on traduire du `bytecode` `Java` en `bytecode` `Python` ? Et l'inverse ?

Introduction

Présentation du langage Python

Les langages

Rappels sur les expressions régulières

Alphabet, mot et langage 1/1

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 $\quad$ 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^* $\quad$ $L \subset \Sigma^*$

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

Opérations sur les mots 1/11

- 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 $\quad |\epsilon| = 0$
  - $|\omega|_a$ = 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 $^{2/11}$

Préfixe

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

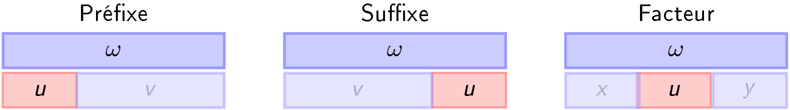
- Ensemble des préfixes : $Pref(\omega) = \{u | u \in \Sigma^*, \exists v \in \Sigma^*, uv = \omega\}$
- Ensemble des préfixes communs : $Pc(x, y) = Pref(x) \cap Pref(y)$
- Plus long préfixe commun : $Plpc(x, y) = \arg \max\limits_{u \in Pc(x,y)} |u|$

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 $xuy = \omega$

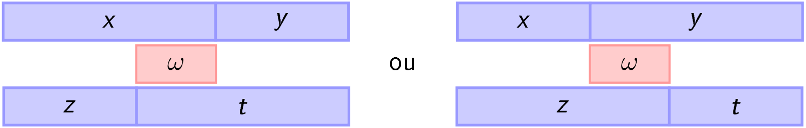


Opérations sur les mots 3/11

Lemme de Levy

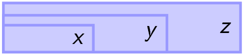
Soient x, y, z et t des mots de Σ^*

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



Opérations sur les mots $^{4/11}$

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

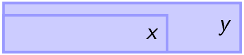


  - Cette relation est antisymétrique car, si x est un préfixe de y et y est un préfixe de x, alors x égale y $x \preceq_p y \wedge y \preceq_p x \Rightarrow x = y$
  - L'ordre défini par cette relation est partiel. Par exemple, le mot *abcd* n'est pas le préfixe de *efgh* et inversement. Cette particularité vient du fait que quelque soit $a_1a_2 \in \Sigma^2$, a_1 et a_2 ne sont pas reliés par la notion de préfixe

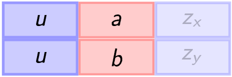
Opérations sur les mots $^{5/11}$

- 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 $\preceq_l$

$$x \preceq_l y \Rightarrow \begin{cases} x \preceq_p y, \text{ ou} \\ \exists (u, z_x, z_y) \in (\Sigma^*)^3, (a, b) \in \Sigma^2 \quad \text{t.q.} \\ \quad x = uaz_x \wedge y = ubz_y \wedge a \preceq_x b \wedge b \not\preceq_x a \end{cases}$$



ou



$a \preceq_x b \wedge b \not\preceq_x a$

  - 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 *6/11*

- 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 7/11

- Distances entre mots – distance de Levenstein
  - Un script d'édition (en anglais *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 :
    - $insert(i, a)$ permet d'insérer le caractère a à l'indice i
    - $delete(i)$ permet de supprimer le caractère à l'indice i
    - $replace(i, a)$ permet de 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`[?] de `Gnu`
  - Peut-on employer cette distance pour créer un système de plagiat ?

Opérations sur les mots 8/11

- 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 *9/11*

- Distances entre mots – application ../..
  - Comment, à partir de la matrice d_e, obtenir le script d'édition ?
    Il suffit de traverser cette matrice en suivant la "diagonale" qui part du bas à droite en 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 ?

Listing 26: `03-langages/pEditScript.py`

```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)
# 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
```

Opérations sur les mots 10/11

- 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 *11/11*

- Distances entre mots – application ../..
  - Utilisation en littérature pour identifier l'appartenance de textes (Molière ou Corneille ?)[**?**]
  - Amélioration possible : *Four Russian*[**?**]
  - 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

		a	a
	0	1	2
a	1	0	1
a	2	1	0

		a	b
	0	1	2
a	1	0	1
a	2	1	1

		a	b
	0	1	2
a	1	0	1
b	2	1	0

		b	b
	0	1	2
a	1	1	2
b	2	1	1

...

Opérations sur les langages *1/1*

- 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 les langages L_1 et L_2
  - $L_1 \subset \Sigma^*$ et $L_2 \subset \Sigma^*$
  - 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^*) $\quad L_1^* = \bigcup_{i \geq 0} L_1^i \qquad L_1^+ = \bigcup_{i \geq 1} L_1^i$
- 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 1/1

- 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 1/1

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

Présentation du langage Python

Les langages

Rappels sur les expressions régulières

Construction des expressions régulières *1/1*

- Les expressions régulières sont construites à partir d'expressions régulières atomiques (noté `e.r.a.`)
  - L'ensemble vide $\emptyset$ est une `e.r.a.`
  - Le symbole ϵ est une `e.r.a.`
  - Chaque symbole de l'alphabet Σ est une `e.r.a.`
- Par assemblage de ces `e.r.a.` 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 1/1

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

Propriétés des fonctions 1/2

- 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 2/2

- 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 *1/1*

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 1/1

- 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

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

Applications – syntaxe de grep et bash 1/1

- Equivalences entre les notations
  - Cette liste est non exhaustive
  - Pour les `e.r.`, 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 `e.r.`, 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 `e.r.`, la notation $a_i|\ldots$ est à remplacer par tous les caractères de la table `ASCII` du caractère a_i jusqu'à la fin

$\mathfrak{r}$	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]`

Part II

Les grammaires

Les grammaires de type 1

Les grammaires de type 2

Les grammaires de type 3

Formalisme 1/3

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 2/3

- 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 3/3

- 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 aux 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 1/2

- Liste de prénoms
  - Grammaire $G = (T, N, R, S)$ associée

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

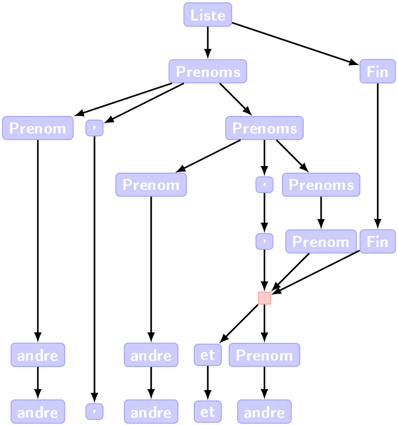
$$
R=\left\{\begin{array}{lrcl}
R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} \\
R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms Fin} \\
R_3 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom} \\
R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom , Prenoms} \\
R_5 & \textbf{, Prenom Fin} & \rightarrow & \textbf{et Prenom} \\
R_6 & \textbf{Prenom} & \rightarrow & \textbf{andre} \\
R_7 & \textbf{Prenom} & \rightarrow & \textbf{liam} \\
R_8 & \textbf{Prenom} & \rightarrow & \textbf{phil}
\end{array}\right\}
$$

  - La règle R_2 signifie que **Liste** peut être remplacé par **Prenoms** puis **Fin**
  - Remarque : 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 $\rightarrow$ *nécessité des parsers*

Exemple de grammaire 2/2

- Liste de prénoms .../...
  - Exemple de dérivation

Liste	$\Rightarrow (R_2)$	Prenoms Fin
	$\Rightarrow (R_4)$	Prenom , Prenoms Fin
	$\Rightarrow (R_4)$	Prenom , Prenom , Prenoms Fin
	$\Rightarrow (R_3)$	Prenom , Prenom , Prenom Fin
	$\Rightarrow (R_5)$	Prenom , Prenom et Prenom
	$\Rightarrow (R_6)$	andre , Prenom et Prenom
	$\Rightarrow (R_6)$	andre , andre et Prenom
	$\Rightarrow (R_6)$	andre , andre et andre



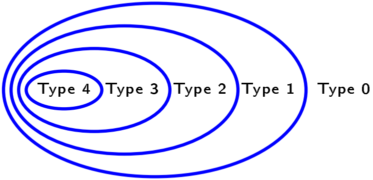
Programme de génération de mots d'une grammaire 1/1

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 $\leftarrow (\text{liste} \setminus \{\alpha\}) \cup \{\beta \in (N \cup T)^* : \alpha \Rightarrow \beta\}$
    return mots
end function

Classification des grammaires 1/1

- 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 *1/1*

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 $\beta \in (N \cup T)^*$

→ Il n'y a aucune restriction sur les règles de production

- Ces grammaires sont difficiles à manipuler

Structures de données 1/1

- 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, etc.)

Type de grammaire	Nom de la grammaire	Structure de données	Automate
Type 0/1	PS – *phrase structure*	Graphe acyclique orienté	Machine de Turing
Type 2	CF – *context free*	Arbre	Automate à pile
Type 3	FS – *finite state*	Liste	Automate fini

Les grammaires

Les grammaires de type 1

Les grammaires de type 2

Les grammaires de type 3

Deux types de grammaire de type 1 *1/1*

- Il existe deux définitions **équivalentes** des grammaires de type 1
  - Grammaire *monotone*
  - Grammaire *contextuelle*

Grammaire *monotone* 1/1

Grammaire *monotone*

Une grammaire est *monotone* si aucune règle possède une partie gauche avec plus de symboles que la partie droite

- $\forall(\alpha, \beta) \in R, |\alpha| \leq |\beta|$
- Par exemple, la règle **, Prenom Fin → et Prenom** devient interdite
- La mot vide ϵ ne peut pas être engendré par une telle grammaire

Grammaire *contextuelle* 1/1

Grammaire *contextuelle*

Une grammaire est *contextuelle* si toutes ses règles sont sensibles au contexte ; Une règle est sensible au contexte si seulement un symbole de la partie gauche est remplacé par d'autres symboles dans la partie droite et tous les autres symboles de la partie gauche se retrouvent dans le même ordre et inchangés dans la partie droite

- Toutes les règles de production sont de la forme $\gamma A \delta \rightarrow \gamma \beta \delta$ avec $\gamma, \beta, \delta \in (T \cup N)^*, A \in N$ et $\beta \neq \epsilon$
- (γ, δ) est le contexte de la règle
- Chaque règle de production ne change qu'un seul symbole non-terminal à la fois (A est transformé en β)
- A représentant un seul symbole non-terminal et β représentant au moins un symbole ($\beta \neq \epsilon$), alors une grammaire *contextuelle* est *monotone*

Grammaire *monotone*, *bornée à droite* 1/3

Lemme

Si G est une grammaire *monotone*, alors il existe une grammaire G' également *monotone*, qui engendre le même langage que G, telle que toutes les parties droites de ses règles de production ont une taille inférieure ou égale à deux et aucun terminal n'apparaît dans les parties gauches

- Démonstration du Lemme

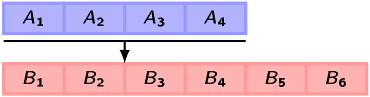
$\rightarrow$ Il suffit de trouver une méthode pour passer de G à G'

Grammaire *monotone*, *bornée à droite* 2/3

- Toutes les règles de production de G sont de la forme $\alpha_1 \dots \alpha_m \rightarrow \beta_1 \dots \beta_n$ avec $\forall i \in [1; m], \alpha_i \in (T \cup N)$, $\forall i \in [1; n], \beta_i \in (T \cup N)$ et $m \leq n$
- G' est obtenu à partir de G en quatre étapes
  1. Remplacer chaque terminal a qui apparaît à droite dans une règle par un non-terminal X_a et ajouter la règle de production $X_a \rightarrow a$
  2. Remplacer chaque règle de la forme $A_1 \dots A_m \rightarrow B_1 \dots B_n$ avec $2 \leq m < n$, par
     $$\left\{\begin{array}{rcl} A_1 \dots A_m & \rightarrow & B_1 \dots B_{m-1} X_m \\ X_m & \rightarrow & B_m \dots B_n \end{array}\right.$$
  3. Remplacer chaque règle de la forme $A_1 \rightarrow B_1 \dots B_n$ avec $2 < n$, par
     $$\left\{\begin{array}{rrcl} & A_1 & \rightarrow & B_1 X_1 \\ \forall i \in [2; n-2], & X_{i-1} & \rightarrow & B_i X_i \\ & X_{n-2} & \rightarrow & B_{n-1} B_n \end{array}\right.$$
  4. Remplacer chaque règle de la forme $A_1 \dots A_n \rightarrow B_1 \dots B_n$ avec $2 < n$, par
     $$\left\{\begin{array}{rrcl} & A_1 A_2 & \rightarrow & B_1 X_1 \\ \forall i \in [2; n-2], & X_{i-1} A_{i+1} & \rightarrow & B_i X_i \\ & X_{n-2} A_n & \rightarrow & B_{n-1} B_n \end{array}\right.$$
- *NB : 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*

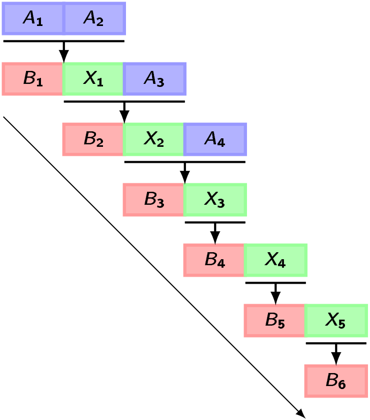
Grammaire *monotone*, *bornée à droite* 3/3

- Les X_i ajoutés ne permettent pas d'engendrer de nouveaux mots
- La seule manière de *consommer* complètement un non-terminal X_i est de poursuivre en dérivant tous les non-terminaux $(X_j)_{j \geq i}$, dans l'ordre
- Représentation graphique de la méthode



$$A_1 A_2 A_3 A_4 \rightarrow B_1 B_2 B_3 B_4 B_5 B_6$$

$$\Rightarrow \left\{ \begin{array}{rcl} A_1 A_2 & \rightarrow & B_1 X_1 \\ X_1 A_3 & \rightarrow & B_2 X_2 \\ X_2 A_4 & \rightarrow & B_3 X_3 \\ X_3 & \rightarrow & B_4 X_4 \\ X_4 & \rightarrow & B_5 X_5 \\ X_5 & \rightarrow & B_6 \end{array} \right.$$



Grammaire *monotone*, *bornée à droite* et *contextuelle* 1/9

Lemme

Si G est une grammaire de type 1 ne contenant que des règles de production de la forme $AB \rightarrow CD$, alors il existe une grammaire G' *contextuelle*, qui engendre le même langage que G

- Démonstration du Lemme

→ Il suffit de trouver une méthode pour passer de G à G'

Grammaire *monotone*, *bornée à droite* et *contextuelle* 2/9

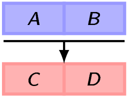
- Par définition, toutes les règles de production de G sont de la forme $AB \rightarrow CD$
- G' est obtenu à partir de G en trois étapes
  1. Conserver les règles de la forme $A \rightarrow \beta$
  2. Conserver les règles de la forme $AB \rightarrow CD$, avec $A = C$ ou $B = D$
  3. Remplacer les règles de la forme $AB \rightarrow CD$, avec $A \neq C$ et $B \neq D$, par

$$\left\{ \begin{array}{rcl} AB & \rightarrow & X_1 B \\ X_1 B & \rightarrow & X_1 X_2 \\ X_1 X_2 & \rightarrow & C X_2 \\ C X_2 & \rightarrow & CD \end{array} \right.$$

- *NB : 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*

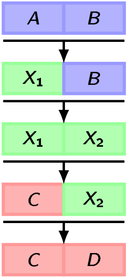
Grammaire *monotone*, *bornée à droite* et *contextuelle* 3/9

- Les X_i ajoutés ne permettent pas d'engendrer de nouveaux mots
- La seule manière de *consommer* complètement un non-terminal X_i est de poursuivre en dérivant tous les non-terminaux $(X_j)_{j \geq i}$, dans l'ordre
- Représentation graphique de la méthode



$$AB \rightarrow CD$$

$$\Rightarrow \left\{ \begin{array}{rcl} AB & \rightarrow & X_1 B \\ X_1 B & \rightarrow & X_1 X_2 \\ X_1 X_2 & \rightarrow & C X_2 \\ C X_2 & \rightarrow & CD \end{array} \right.$$



Grammaire *monotone*, *bornée à droite* et *contextuelle* 4/9

- Deux nouveaux symboles non-terminaux sont utilisés

1. Conserver les règles de la forme $A \to \beta$
2. Conserver les règles de la forme $AB \to CD$, avec $A = C$ ou $B = D$
3. Remplacer les règles de la forme $AB \to CD$, avec $A \neq C$ et $B \neq D$, par

$$\left\{ \begin{array}{rcl} AB & \to & X_1 B \\ X_1 B & \to & X_1 X_2 \\ X_1 X_2 & \to & C X_2 \\ C X_2 & \to & CD \end{array} \right.$$

- Peut-on optimiser cette méthode ?

Grammaire *monotone*, *bornée à droite* et *contextuelle* 5/9

- Utilisation de moins de nouveaux symboles non-terminaux ?
  1. Remplacer les règles de la forme $AB \rightarrow CD$, avec $A \neq C$ et $B \neq D$, par
  $$\left\{ \begin{array}{rcl} AB & \rightarrow & X_1 B \\ X_1 B & \rightarrow & X_1 D \\ X_1 D & \rightarrow & CD \end{array} \right.$$
- Ces règles sont effectivement contextuelles
- Cette nouvelle méthode utilise uniquement un seul nouveau symbole non-terminal
- Il faut s'assurer que les mots engendrés par l'ancienne grammaire peuvent l'être par la nouvelle
- Il faut s'assurer que de nouveaux mots ne peuvent pas être engendrés par la nouvelle grammaire

Grammaire *monotone*, *bornée à droite* et *contextuelle* 6/9

- Exemple avec la grammaire pour le langage {`c b b`}

S=**S** T={**c**, **b**} N={**S**, **A**, **B**, **C**, **D**}

$$R=\left\{\begin{array}{lrcl} R_1 & \mathbf{S} & \rightarrow & \mathbf{A\ B} \\ R_2 & \mathbf{A\ B} & \rightarrow & \mathbf{C\ D} \\ R_3 & \mathbf{D} & \rightarrow & \mathbf{B\ B} \\ R_4 & \mathbf{C} & \rightarrow & \mathbf{c} \\ R_5 & \mathbf{B} & \rightarrow & \mathbf{b} \end{array}\right\}$$

- Liste des mots engendrés

`c b b`

- Cette grammaire est *monotone*

$$\begin{array}{lrcll} R_1 & \mathbf{S} & \rightarrow & \mathbf{A\ B} & \checkmark \\ R_2 & \mathbf{A\ B} & \rightarrow & \mathbf{C\ D} & \checkmark \\ R_3 & \mathbf{D} & \rightarrow & \mathbf{B\ B} & \checkmark \\ R_4 & \mathbf{C} & \rightarrow & \mathbf{c} & \checkmark \\ R_5 & \mathbf{B} & \rightarrow & \mathbf{b} & \checkmark \end{array}$$

Grammaire *monotone*, *bornée à droite* et *contextuelle* 7/9

- Exemple avec la grammaire pour le langage {`c b b`}

S=**S** T={**c**, **b**} N={**S**, **A**, **B**, **C**, **D**}

$$R=\left\{\begin{array}{llcl} R_1 & \mathbf{S} & \rightarrow & \mathbf{A\ B} \\ R_2 & \mathbf{A\ B} & \rightarrow & \mathbf{C\ D} \\ R_3 & \mathbf{D} & \rightarrow & \mathbf{B\ B} \\ R_4 & \mathbf{C} & \rightarrow & \mathbf{c} \\ R_5 & \mathbf{B} & \rightarrow & \mathbf{b} \end{array}\right\}$$

- Cette grammaire n'est pas *contextuelle*

R_1	**S**	→	**A B**	✓
R_2	**A B**	→	**C D**	✗
R_3	**D**	→	**B B**	✓
R_4	**C**	→	**c**	✓
R_5	**B**	→	**b**	✓

⇒ Tentative de création d'une grammaire *contextuelle* avec la méthode précédente

Grammaire *monotone*, *bornée à droite* et *contextuelle* 8/9

- Exemple avec la grammaire pour le langage {`c b b`}

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

$$R=\left\{\begin{array}{lrcl} R_1 & \mathbf{S} & \rightarrow & \mathbf{A}\ \mathbf{B} \\ R_2 & \mathbf{A}\ \mathbf{B} & \rightarrow & \mathbf{C}\ \mathbf{D} \\ R_3 & \mathbf{D} & \rightarrow & \mathbf{B}\ \mathbf{B} \\ R_4 & \mathbf{C} & \rightarrow & \mathbf{c} \\ R_5 & \mathbf{B} & \rightarrow & \mathbf{b} \end{array}\right\}$$

- Grammaire résultant de la transformation

$$S=\mathbf{S}\ T=\{\mathbf{c},\mathbf{b}\}\ N=\{\mathbf{S},\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D},\mathbf{Y}_0\}$$

$$R=\left\{\begin{array}{lrcl} R_1 & \mathbf{S} & \rightarrow & \mathbf{A}\ \mathbf{B} \\ R_2 & \mathbf{A}\ \mathbf{B} & \rightarrow & \mathbf{Y}_0\ \mathbf{B} \\ R_3 & \mathbf{Y}_0\ \mathbf{B} & \rightarrow & \mathbf{Y}_0\ \mathbf{D} \\ R_4 & \mathbf{Y}_0\ \mathbf{D} & \rightarrow & \mathbf{C}\ \mathbf{D} \\ R_5 & \mathbf{D} & \rightarrow & \mathbf{B}\ \mathbf{B} \\ R_6 & \mathbf{C} & \rightarrow & \mathbf{c} \\ R_7 & \mathbf{B} & \rightarrow & \mathbf{b} \end{array}\right\}$$

Grammaire *monotone*, *bornée à droite* et *contextuelle* 9/9

- Cette nouvelle grammaire est *contextuelle*

R_1	**S**	→	**A B**	✓
R_2	**A B**	→	**$\mathbf{Y}_0$ B**	✓
R_3	**$\mathbf{Y}_0$ B**	→	**$\mathbf{Y}_0$ D**	✓
R_4	**$\mathbf{Y}_0$ D**	→	**C D**	✓
R_5	**D**	→	**B B**	✓
R_6	**C**	→	**c**	✓
R_7	**B**	→	**b**	✓

- Liste des mots engendrés (non exhaustive)

  c b b

  c b b b

  c b b b b

  c b b b b b

- Pire pour **AB** ⇒ **AD** ⇒ **CD** si il existe déjà une règle $\mathbf{D}\alpha \rightarrow \mathbf{B}\beta$!

Grammaire *montone* vers *contextuelle* $^{1/1}$

Théorème

Si G est une grammaire *monotone* alors il existe une grammaire *contextuelle* G' qui engendre le même langage

- Grammaire *monotone* $\rightarrow$ Grammaire *contextuelle*
  - Démonstration $\rightarrow$ il suffit de trouver une méthode pour passer de G à G'
  - Application successivement des deux méthodes précédentes

Exemple $n^o 1$ *1/7*

- Grammaire pour le langage $\{a^n b^n c^n\}_{n>0}$

S=**S** T={**a**, **b**, **c**} N={**S**, **Q**}

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

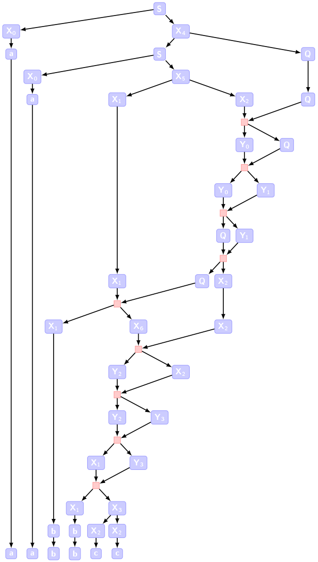
Exemple $n^o 1$ $^{2/7}$

- Grammaire pour le langage $\{a^n b^n c^n\}_{n>0}$ . . . / . . .

  S=**S** T={**a**, **b**, **c**} N={**S**, **Q**}

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

- Principe de génération
  - *Les* **a** *sont générés à la bonne place, à gauche*
  - *Pour chaque* **a** *généré, un* **Q** *est génère à droite*
  - *Dès le bon nombre de* **a** *atteint, le mot est :* **aaa. . . aabcQQ. . . Q**
  - *Les* **Q** *sont ramenés au centre (entre un* **b** *et un* **c***), à tour de rôle*
  - *Un* **Q** *au centre est remplacé par* **bc** *permettant ainsi à un autre* **Q** *de s'insérer à son tour entre ce* **b** *et ce* **c**



Exemple $n^o 1$ 3/7

- Grammaire pour le langage $\{a^n b^n c^n\}_{n>0}$ .../...

S=**S** T={**a**, **b**, **c**} N={**S**, **Q**}

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

- Cette grammaire n'est pas *contextuelle* $\rightarrow$ la règle R_3 change la place de **c**

$$\begin{array}{lllll} R_1 & \mathbf{S} & \rightarrow & \mathbf{a\ S\ Q} & ✓ \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{a\ b\ c} & ✓ \\ R_3 & \mathbf{c\ Q} & \rightarrow & \mathbf{Q\ c} & ✗ \\ R_4 & \mathbf{b\ Q\ c} & \rightarrow & \mathbf{b\ b\ c\ c} & ✓ \end{array}$$

- Cette grammaire est *monotone*

$$\begin{array}{lllll} R_1 & \mathbf{S} & \rightarrow & \mathbf{a\ S\ Q} & ✓ \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{a\ b\ c} & ✓ \\ R_3 & \mathbf{c\ Q} & \rightarrow & \mathbf{Q\ c} & ✓ \\ R_4 & \mathbf{b\ Q\ c} & \rightarrow & \mathbf{b\ b\ c\ c} & ✓ \end{array}$$

$\Rightarrow$ Construction d'une grammaire *contextuelle* qui reconnaît ce langage

Exemple $n^o 1$ $^{4/7}$

- Passage à une grammaire *monotone*, *bornée à droite*

Étape 1

R_1	S	→	a S Q		X_0	→	a	R_1
					S	→	X_0 S Q	R_2
R_2	S	→	a b c		X_1	→	b	R_3
					X_2	→	c	R_4
					S	→	X_0 X_1 X_2	R_5
R_3	c Q	→	Q c		X_2 Q	→	Q X_2	R_6
R_4	b Q c	→	b b c c		X_1 Q X_2	→	X_1 X_1 X_2 X_2	R_7

Étape 2

R_1	X_0	→	a		X_0	→	a	R_1
R_2	S	→	X_0 S Q		S	→	X_0 S Q	R_2
R_3	X_1	→	b		X_1	→	b	R_3
R_4	X_2	→	c		X_2	→	c	R_4
R_5	S	→	X_0 X_1 X_2		S	→	X_0 X_1 X_2	R_5
R_6	X_2 Q	→	Q X_2		X_2 Q	→	Q X_2	R_6
R_7	X_1 Q X_2	→	X_1 X_1 X_2 X_2		X_1 Q X_2	→	X_1 X_1 X_3	R_7
					X_3	→	X_2 X_2	R_8

Exemple $n^o 1$ $^{5/7}$

- Passage à une grammaire *monotone*, *bornée à droite* .../...

Étape 2

R_1	X_0	$\rightarrow$	a	X_0	$\rightarrow$	a	R_1
R_2	S	$\rightarrow$	X_0 S Q	S	$\rightarrow$	X_0 S Q	R_2
R_3	X_1	$\rightarrow$	b	X_1	$\rightarrow$	b	R_3
R_4	X_2	$\rightarrow$	c	X_2	$\rightarrow$	c	R_4
R_5	S	$\rightarrow$	X_0 X_1 X_2	S	$\rightarrow$	X_0 X_1 X_2	R_5
R_6	X_2 Q	$\rightarrow$	Q X_2	X_2 Q	$\rightarrow$	Q X_2	R_6
R_7	X_1 Q X_2	$\rightarrow$	X_1 X_1 X_2 X_2	X_1 Q X_2	$\rightarrow$	X_1 X_1 X_3	R_7
				X_3	$\rightarrow$	X_2 X_2	R_8

Étape 3 et 4

R_1	X_0	$\rightarrow$	a	X_0	$\rightarrow$	a	R_1
R_2	S	$\rightarrow$	X_0 S Q	S	$\rightarrow$	X_0 X_4	R_2
				X_4	$\rightarrow$	S Q	R_3
R_3	X_1	$\rightarrow$	b	X_1	$\rightarrow$	b	R_4
R_4	X_2	$\rightarrow$	c	X_2	$\rightarrow$	c	R_5
R_5	S	$\rightarrow$	X_0 X_1 X_2	S	$\rightarrow$	X_0 X_5	R_6
				X_5	$\rightarrow$	X_1 X_2	R_7
R_6	X_2 Q	$\rightarrow$	Q X_2	X_2 Q	$\rightarrow$	Q X_2	R_8
R_7	X_1 Q X_2	$\rightarrow$	X_1 X_1 X_3	X_1 Q	$\rightarrow$	X_1 X_6	R_9
				X_6 X_2	$\rightarrow$	X_1 X_3	R_{10}
R_8	X_3	$\rightarrow$	X_2 X_2	X_3	$\rightarrow$	X_2 X_2	R_{11}

Exemple $n^o 1$ $^{6/7}$

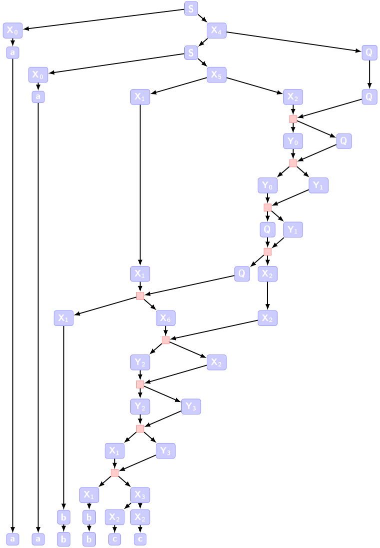
- Passage à une grammaire *monotone*, *bornée à droite* .../...

R_1	X_0	→	a	X_0	→	a	R_1
R_2	S	→	X_0 X_4	S	→	X_0 X_4	R_2
R_3	X_4	→	S Q	X_4	→	S Q	R_3
R_4	X_1	→	b	X_1	→	b	R_4
R_5	X_2	→	c	X_2	→	c	R_5
R_6	S	→	X_0 X_5	S	→	X_0 X_5	R_6
R_7	X_5	→	X_1 X_2	X_5	→	X_1 X_2	R_7
R_8	X_2 Q	→	Q X_2	X_2 Q	→	Y_0 Q	R_8
				Y_0 Q	→	Y_0 Y_1	R_9
				Y_0 Y_1	→	Q Y_1	R_{10}
				Q Y_1	→	Q X_2	R_{11}
R_9	X_1 Q	→	X_1 X_6	X_1 Q	→	X_1 X_6	R_{12}
R_{10}	X_6 X_2	→	X_1 X_3	X_6 X_2	→	Y_2 X_2	R_{13}
				Y_2 X_2	→	Y_2 Y_3	R_{14}
				Y_2 Y_3	→	X_1 Y_3	R_{15}
				X_1 Y_3	→	X_1 X_3	R_{16}
R_{11}	X_3	→	X_2 X_2	X_3	→	X_2 X_2	R_{17}

Exemple $n^o 1$ $^{7/7}$

- Passage à une grammaire *monotone*, *bornée à droite* . . . / . . .

$$
\begin{array}{lll}
S & \Rightarrow (R_2) & X_0\ X_4 \\
 & \Rightarrow (R_1) & a\ X_4 \\
 & \Rightarrow (R_3) & a\ S\ Q \\
 & \Rightarrow (R_6) & a\ X_0\ X_5\ Q \\
 & \Rightarrow (R_1) & a\ a\ X_5\ Q \\
 & \Rightarrow (R_7) & a\ a\ X_1\ X_2\ Q \\
 & \Rightarrow (R_8) & a\ a\ X_1\ Y_0\ Q \\
 & \Rightarrow (R_9) & a\ a\ X_1\ Y_0\ Y_1 \\
 & \Rightarrow (R_{10}) & a\ a\ X_1\ Q\ Y_1 \\
 & \Rightarrow (R_{11}) & a\ a\ X_1\ Q\ X_2 \\
 & \Rightarrow (R_{12}) & a\ a\ X_1\ X_6\ X_2 \\
 & \Rightarrow (R_{13}) & a\ a\ X_1\ Y_2\ X_2 \\
 & \Rightarrow (R_{14}) & a\ a\ X_1\ Y_2\ Y_3 \\
 & \Rightarrow (R_{15}) & a\ a\ X_1\ X_1\ Y_3 \\
 & \Rightarrow (R_{16}) & a\ a\ X_1\ X_1\ X_3 \\
 & \Rightarrow (R_{17}) & a\ a\ X_1\ X_1\ X_2\ X_2 \\
 & \Rightarrow (R_4) & a\ a\ b\ X_1\ X_2\ X_2 \\
 & \Rightarrow (R_4) & a\ a\ b\ b\ X_2\ X_2 \\
 & \Rightarrow (R_5) & a\ a\ b\ b\ c\ X_2 \\
 & \Rightarrow (R_5) & a\ a\ b\ b\ c\ c
\end{array}
$$



Exemple $n^o 2$ *1/6*

- Grammaire pour la liste des prénoms

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

$$R=\left\{\begin{array}{lrcl} R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} \\ R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms Fin} \\ R_3 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom} \\ R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom , Prenoms} \\ R_5 & \textbf{, Prenom Fin} & \rightarrow & \textbf{et Prenom} \\ R_6 & \textbf{Prenom} & \rightarrow & \textbf{andre} \\ R_7 & \textbf{Prenom} & \rightarrow & \textbf{liam} \\ R_8 & \textbf{Prenom} & \rightarrow & \textbf{phil} \end{array}\right\}$$

Exemple $n^o 2$ $^{2/6}$

- Grammaire pour la liste des prénoms .../...

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

$$R=\left\{\begin{array}{llcl} R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} \\ R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms Fin} \\ R_3 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom} \\ R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom , Prenoms} \\ R_5 & \textbf{, Prenom Fin} & \rightarrow & \textbf{et Prenom} \\ R_6 & \textbf{Prenom} & \rightarrow & \textbf{andre} \\ R_7 & \textbf{Prenom} & \rightarrow & \textbf{liam} \\ R_8 & \textbf{Prenom} & \rightarrow & \textbf{phil} \end{array}\right\}$$

- Cette grammaire n'est pas *contextuelle*
  $\rightarrow$ la règle R_5 change plusieurs symboles

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**	✓

Exemple $n^o 2$ $^{3/6}$

- Grammaire pour la liste des prénoms …/…

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

$$R=\left\{\begin{array}{lrcl} R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} \\ R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms Fin} \\ R_3 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom} \\ R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom , Prenoms} \\ R_5 & \textbf{, Prenom Fin} & \rightarrow & \textbf{et Prenom} \\ R_6 & \textbf{Prenom} & \rightarrow & \textbf{andre} \\ R_7 & \textbf{Prenom} & \rightarrow & \textbf{liam} \\ R_8 & \textbf{Prenom} & \rightarrow & \textbf{phil} \end{array}\right\}$$

- Cette grammaire n'est pas *monotone*

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**	✓

- Quel est le type de cette grammaire ?

⇒ Existe-t-il une grammaire *contextuelle* qui reconnaît ce langage ?

Exemple $n^o 2$ $^{4/6}$

- Grammaire pour la liste des prénoms .../...
  - Le problème provient de la règle R_5 contenant plus de symboles à gauche qu'à droite et qui contient un contexte variable
  - Solution : créer de nouveaux non-terminaux
  - Nouvelle grammaire

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

$$R=\left\{\begin{array}{lrcl}
R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} \\
R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms} \\
R_3 & \textbf{Prenoms} & \rightarrow & \textbf{PrenomFin} \\
R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom V Prenoms} \\
R_5 & \textbf{V PrenomFin} & \rightarrow & \textbf{et PrenomFin} \\
R_6 & \textbf{et PrenomFin} & \rightarrow & \textbf{et Prenom} \\
R_7 & \textbf{V Prenom} & \rightarrow & \textbf{, Prenom} \\
R_8 & \textbf{Prenom} & \rightarrow & \textbf{andre} \\
R_9 & \textbf{Prenom} & \rightarrow & \textbf{liam} \\
R_{10} & \textbf{Prenom} & \rightarrow & \textbf{phil}
\end{array}\right\}$$

Exemple $n^o 2$ $^{5/6}$

- Grammaire pour la liste des prénoms .../...

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

$$R=\left\{\begin{array}{lrcl} R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} \\ R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms} \\ R_3 & \textbf{Prenoms} & \rightarrow & \textbf{PrenomFin} \\ R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom V Prenoms} \\ R_5 & \textbf{V PrenomFin} & \rightarrow & \textbf{et PrenomFin} \\ R_6 & \textbf{et PrenomFin} & \rightarrow & \textbf{et Prenom} \\ R_7 & \textbf{V Prenom} & \rightarrow & \textbf{, Prenom} \\ R_8 & \textbf{Prenom} & \rightarrow & \textbf{andre} \\ R_9 & \textbf{Prenom} & \rightarrow & \textbf{liam} \\ R_{10} & \textbf{Prenom} & \rightarrow & \textbf{phil} \end{array}\right\}$$

- Cette grammaire est *monotone*

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

Exemple $n^o 2$ $^{6/6}$

- Grammaire pour la liste des prénoms .../...

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

$$R=\left\{\begin{array}{lrcl}
R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} \\
R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms} \\
R_3 & \textbf{Prenoms} & \rightarrow & \textbf{PrenomFin} \\
R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom V Prenoms} \\
R_5 & \textbf{V PrenomFin} & \rightarrow & \textbf{et PrenomFin} \\
R_6 & \textbf{et PrenomFin} & \rightarrow & \textbf{et Prenom} \\
R_7 & \textbf{V Prenom} & \rightarrow & \textbf{, Prenom} \\
R_8 & \textbf{Prenom} & \rightarrow & \textbf{andre} \\
R_9 & \textbf{Prenom} & \rightarrow & \textbf{liam} \\
R_{10} & \textbf{Prenom} & \rightarrow & \textbf{phil}
\end{array}\right\}$$

- Cette grammaire est *contextuelle*

$$\begin{array}{lrcll}
R_1 & \textbf{Liste} & \rightarrow & \textbf{Prenom} & \checkmark \\
R_2 & \textbf{Liste} & \rightarrow & \textbf{Prenoms} & \checkmark \\
R_3 & \textbf{Prenoms} & \rightarrow & \textbf{PrenomFin} & \checkmark \\
R_4 & \textbf{Prenoms} & \rightarrow & \textbf{Prenom V Prenoms} & \checkmark \\
R_5 & \textbf{V PrenomFin} & \rightarrow & \textbf{et PrenomFin} & \checkmark \\
R_6 & \textbf{et PrenomFin} & \rightarrow & \textbf{et Prenom} & \checkmark \\
R_7 & \textbf{V Prenom} & \rightarrow & \textbf{, Prenom} & \checkmark \\
R_8 & \textbf{Prenom} & \rightarrow & \textbf{andre} & \checkmark \\
R_9 & \textbf{Prenom} & \rightarrow & \textbf{liam} & \checkmark \\
R_{10} & \textbf{Prenom} & \rightarrow & \textbf{phil} & \checkmark
\end{array}$$

Programme permettant d'identifier si un mot appartient au langage 1/3

$\rightarrow$ Identifier si un mot α peut être engendré par cette grammaire

- Il suffit d'engendrer tous les mots et d'attendre que α soit engendré
- A un instant donné, comment savoir si le mot aurait déjà du être engendré ? (*important pour ne pas attendre indéfiniment*)
- Les grammaire de type 1 sont monotones

$\rightarrow$ Quelque soit la dérivation $\alpha \Rightarrow \beta$, la taille du mot β est forcément supérieure ou égale à la taille du mot α

- Il existe un algorithme qui affiche les mots par ordre croissant de taille

Programme permettant d'identifier si un mot appartient au langage 2/3

Figure: Algorithme

function TestSiMotGénéré(G, α) ▷ $G = (T, N, R, S), \alpha \in (N \cup T)^*$
    liste $\leftarrow \{S\}$
    while $|\text{liste}| > 0$ **do**
        $\beta \leftarrow \underset{\gamma}{\text{argmin}}\{|\gamma| : \gamma \in \text{liste}\}$
        if $\beta = \alpha$ **then**
            return `True`
        if $|\beta| > |\alpha|$ **then**
            return `False`
        liste $\leftarrow (\text{liste} \setminus \{\beta\}) \cup \{\gamma \in (N \cup T)^* : \beta \Rightarrow \gamma\}$
    return `False`
end function

Programme permettant de lister les mots d'un langage 1/1

Listing 27: `programs/lib/TestWord.py`

```
G = {"R": [ (("B",), ("(", "C", ")")),
            (("C",), ("(", ")")),       (("A",), ("(", "B", ")", "A")),
            (("A",), ("(", ")")),       (("A",), ("(", "B", ")")),
            (("B",), ("(", ")")),       (("B",), ("(", "C", ")", "B")),
  ],   "S": "A",   "N": ["A", "B", "C"],   "T": ["(", ")"] }

def ps_word_match_rule(word, rule, offset):
  for i in range(offset, len(word) - len(rule[0]) + 1):
    if word[i:i + len(rule[0])] == list(rule[0]):
      return i
  return -1

def mn_test_word(grammar, alpha):
  alpha = [a for a in alpha]
  stack = [[grammar["S"]]]
  while len(stack) > 0:
    i = min(range(len(stack)), key=lambda x:len(stack[x]))
    beta = stack[i]
    del stack[i]
    if beta == alpha:
      return True
    if len(beta) > len(alpha):
      return False
    for rule in grammar["R"]:
      offset = ps_word_match_rule(beta, rule, 0)
      while offset >= 0:
        stack.append(beta[:offset] + list(rule[1]) + beta[offset + len(rule[0]):])
        offset = ps_word_match_rule(beta, rule, offset + 1)
  return False

print(mn_test_word(G, "(())(())"))
```

Les grammaires

Les grammaires de type 1

Les grammaires de type 2

Les grammaires de type 3

Grammaire de type 2 $^{1/1}$

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 $A \in N$
- 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 aux grammaires *hors-contexte* : elles peuvent engendrer le mot vide
- Grammaire *hors-contexte* $\equiv$ grammaire algébrique

Exercice *1/1*

- 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 1/1

- La grammaire des prénoms est-elle une grammaire *hors-contexte* ?

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

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

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**	✓

Dérivation d'un mot *1/8*

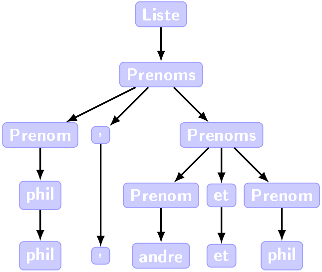
- La dérivation d'un symbole non-terminal se fait indépendamment du contexte
- Les symboles non-terminaux peuvent être dérivés dans n'importe quel ordre
- Il peut exister plusieurs dérivations pour un même mot
- Ces dérivations sont équivalentes
- Une dérivation d'un mot d'une grammaire *hors-contexte* peut également être représentée par un graphe de dérivation
- 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

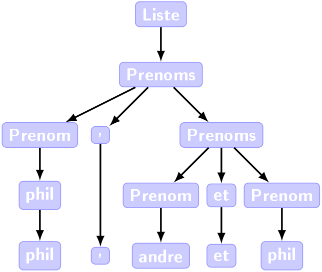
$\Rightarrow$ Il s'agit d'un arbre de dérivation

Dérivation d'un mot 2/8

Liste	$\Rightarrow (R_2)$	Prenoms
	$\Rightarrow (R_4)$	Prenom , Prenoms
	$\Rightarrow (R_7)$	phil , Prenoms
	$\Rightarrow (R_3)$	phil , Prenom et Prenom
	$\Rightarrow (R_7)$	phil , Prenom et phil
	$\Rightarrow (R_5)$	phil , andre et phil

Liste	$\Rightarrow (R_2)$	Prenoms
	$\Rightarrow (R_4)$	Prenom , Prenoms
	$\Rightarrow (R_3)$	Prenom , Prenom et Prenom
	$\Rightarrow (R_7)$	phil , Prenom et Prenom
	$\Rightarrow (R_5)$	phil , andre et Prenom
	$\Rightarrow (R_7)$	phil , andre et phil





Dérivation d'un mot 3/8

- 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 $^{4/8}$

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 5/8

- Que se passe-t-il si la grammaire contient la règle A→A ?

→ 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

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

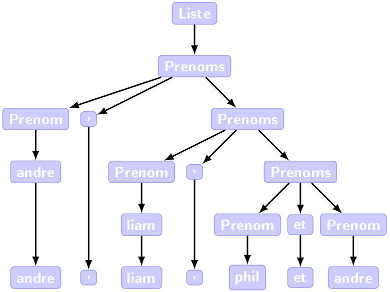
$$\delta_1 = \alpha_1 A_1 \beta_1 \Rightarrow^D \alpha_1 \gamma_1 \beta_1 = \delta_2 = \alpha_2 A_2 \beta_2 \Rightarrow^D \alpha_2 \gamma_2 \beta_2 \ldots \delta_n = \alpha_n A_n \beta_n \Rightarrow^D \alpha_n \gamma_n \beta_n = \delta_{n+1}, \quad \forall i \ \beta_i \in T^* \text{ et } \forall i \in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

Dérivation d'un mot 6/8

- Exemple de dérivation à gauche avec la grammaire des prénoms

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

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



$$
\begin{array}{lll}
\text{Liste} & \Rightarrow (R_2) & \text{Prenoms} \\
 & \Rightarrow (R_4) & \text{Prenom , Prenoms} \\
 & \Rightarrow (R_5) & \text{andre , Prenoms} \\
 & \Rightarrow (R_4) & \text{andre , Prenom , Prenoms} \\
 & \Rightarrow (R_6) & \text{andre , liam , Prenoms} \\
 & \Rightarrow (R_3) & \text{andre , liam , Prenom et Prenom} \\
 & \Rightarrow (R_7) & \text{andre , liam , phil et Prenom} \\
 & \Rightarrow (R_5) & \text{andre , liam , phil et andre}
\end{array}
$$

Dérivation d'un mot $^{7/8}$

Théorème

Quelque soit le mot engendré par une grammaire $G = (T, N, R, S)$ *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 forcément 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 $^{8/8}$

- 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 $\{\mathbf{abc}\}$

S=**S** T={**a**, **b**, **c**} N={**S**, **A**, **B**, **C**}

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

  - 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 1/2

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 $^{2/2}$

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 1/26

- 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 2/26

- 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 3/26

- 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 $^{4/26}$

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

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

$$R=\left\{\begin{array}{lccl}
R_1 & \mathbf{S} & \rightarrow & \\
R_2 & \mathbf{S} & \rightarrow & \mathbf{S\ (\ S\ )} \\
R_3 & \mathbf{S} & \rightarrow & \mathbf{D\ S\ (\ B\ )} \\
R_4 & \mathbf{A} & \rightarrow & \mathbf{S} \\
R_5 & \mathbf{A} & \rightarrow & \mathbf{B} \\
R_6 & \mathbf{A} & \rightarrow & \mathbf{a} \\
R_7 & \mathbf{B} & \rightarrow & \mathbf{S} \\
R_8 & \mathbf{C} & \rightarrow & \mathbf{a\ D} \\
R_9 & \mathbf{D} & \rightarrow & \mathbf{a\ D} \\
R_{10} & \mathbf{D} & \rightarrow & \mathbf{a\ C}
\end{array}\right\}$$

    - Suppression des symboles inutiles

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

$$R=\left\{\begin{array}{lccl}
R_1 & \mathbf{S} & \rightarrow & \\
R_2 & \mathbf{S} & \rightarrow & \mathbf{S\ (\ S\ )} \\
R_3 & \mathbf{A} & \rightarrow & \mathbf{S} \\
R_4 & \mathbf{A} & \rightarrow & \mathbf{B} \\
R_5 & \mathbf{A} & \rightarrow & \mathbf{a} \\
R_6 & \mathbf{B} & \rightarrow & \mathbf{S}
\end{array}\right\}$$

Transformation d'une grammaire 5/26

- 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 6/26

- 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 7/26

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

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

$$R=\left\{\begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{S}\ (\ \mathbf{S}\ ) \\ R_3 & \mathbf{A} & \rightarrow & \mathbf{S} \\ R_4 & \mathbf{A} & \rightarrow & \mathbf{B} \\ R_5 & \mathbf{A} & \rightarrow & \mathbf{a} \\ R_6 & \mathbf{B} & \rightarrow & \mathbf{S} \end{array}\right\}$$

    - Suppression des symboles inaccessibles

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

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

Transformation d'une grammaire 8/26

- Suppression des ϵ-productions
  - Si un symbole A produit le mot ϵ, alors cette production est supprimée et de nouvelles règles sont produites de manière à explorer toutes les possibilités de production vide ou non via 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

Figure: Algorithme

function Combinaison$((B, \beta), A, i)$ $\triangleright$ $|\beta| = n$ et $\beta = \beta_0 \ldots \beta_n$
  if $n == i$ **then**
    return $\{(B, \beta)\}$
  if $\beta_i == A$ **then**
    $\beta' = \beta_0 \ldots \beta_{i-1}\beta_{i+1} \ldots \beta_n$
    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) $\quad \triangleright\ G = (T, N, R, S)$
 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**
  $\mathrm{R} \leftarrow \bigcup_{r \in R} \{(a, b) \in \text{Combinaison}(r, A, 0) : |b| > 0\}$
 $R \leftarrow R \cup \{(T_e, S)\}$
 $N \leftarrow N \cup \{T_e\}$
 return $G' = (T, N, R, T_e)$
end function

Transformation d'une grammaire *10/26*

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

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

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

    - Suppression des ϵ-productions

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

      $$R=\left\{\begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{(\ )} \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{(\ S\ )} \\ R_3 & \mathbf{S} & \rightarrow & \mathbf{S\ (\ )} \\ R_4 & \mathbf{S} & \rightarrow & \mathbf{S\ (\ S\ )} \\ R_5 & \mathbf{Te} & \rightarrow & \mathbf{S} \\ R_6 & \mathbf{Te} & \rightarrow & \end{array}\right\}$$

Transformation d'une grammaire 11/26

- 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 $A \to B$, $B \to \alpha$ et $B \to \beta$, alors la règle unitaire $A \to B$ est remplacée par les règles $A \to \alpha$ et $A \to \beta$
  - Une précaution consiste à éviter les boucles infinies (si $A \to B$ et $B \to A$)

Transformation d'une grammaire

- Suppression des règles unitaires .../...

Figure: Algorithme

```
function SuppressionReglesUnitaires(G)              ▷ G = (T, N, R, S)
    deleted ← {}
    marks ← {(A, β) ∈ R : β ∈ N}
    while |marks| > 0 do
        Q ← {r ∈ R : r ∉ marks}
        R ← Q
        for (C, δ) ∈ Q do
            for (A, β) ∈ marks do
                if β = C ∧ (A, δ) ∉ deleted then
                    R ← R ∪ {(A, δ)}
        deleted ← deleted ∪ marks
        marks ← {(A, β) ∈ R : β ∈ 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 inacessibles et sans ϵ-productions

$$S=\textbf{Te} \ T=\{\textbf{(}, \textbf{)}\} \ N=\{\textbf{S}, \textbf{Te}\}$$

$$R=\left\{\begin{array}{llll} R_1 & \textbf{S} & \rightarrow & \textbf{( )} \\ R_2 & \textbf{S} & \rightarrow & \textbf{(S)} \\ R_3 & \textbf{S} & \rightarrow & \textbf{S ( )} \\ R_4 & \textbf{S} & \rightarrow & \textbf{S (S)} \\ R_5 & \textbf{Te} & \rightarrow & \textbf{S} \\ R_6 & \textbf{Te} & \rightarrow & \end{array}\right\}$$

    - Suppression règles unitaires

$$S=\textbf{Te} \ T=\{\textbf{(}, \textbf{)}\} \ N=\{\textbf{S}, \textbf{Te}\}$$

$$R=\left\{\begin{array}{llll} R_1 & \textbf{S} & \rightarrow & \textbf{( )} \\ R_2 & \textbf{S} & \rightarrow & \textbf{(S)} \\ R_3 & \textbf{S} & \rightarrow & \textbf{S ( )} \\ R_4 & \textbf{S} & \rightarrow & \textbf{S (S)} \\ R_5 & \textbf{Te} & \rightarrow & \\ R_6 & \textbf{Te} & \rightarrow & \textbf{( )} \\ R_7 & \textbf{Te} & \rightarrow & \textbf{(S)} \\ R_8 & \textbf{Te} & \rightarrow & \textbf{S ( )} \\ R_9 & \textbf{Te} & \rightarrow & \textbf{S (S)} \end{array}\right\}$$

Transformation d'une grammaire *14/26*

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 15/26

- Toutes les règles de production de G sont de la forme $A \rightarrow \alpha_1 \dots \alpha_n$, 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 $A \rightarrow \alpha_1 \dots \alpha_n$ 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
     $$\left\{ \begin{array}{rcl} A & \rightarrow & \alpha_1 X_{\alpha_2} \\ X_{\alpha_2} & \rightarrow & \alpha_2 \end{array} \right.$$
- *NB : 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 *16/26*

- Exemple de transformation en forme normale de Chomsky

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

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

$$R=\left\{\begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{(\ )} \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{(\ S\ )} \\ R_3 & \mathbf{S} & \rightarrow & \mathbf{S\ (\ )} \\ R_4 & \mathbf{S} & \rightarrow & \mathbf{S\ (\ S\ )} \\ R_5 & \mathbf{Te} & \rightarrow & \\ R_6 & \mathbf{Te} & \rightarrow & \mathbf{(\ )} \\ R_7 & \mathbf{Te} & \rightarrow & \mathbf{(\ S\ )} \\ R_8 & \mathbf{Te} & \rightarrow & \mathbf{S\ (\ )} \\ R_9 & \mathbf{Te} & \rightarrow & \mathbf{S\ (\ S\ )} \end{array}\right\}$$

Forme normale de Chomsky associée

S=**Te** T={**(**, **)**} N={**S**, **Te**, $\mathbf{X_0}$, $\mathbf{X_1}$, $\mathbf{X_2}$, $\mathbf{X_3}$, $\mathbf{X_4}$, $\mathbf{X_5}$, $\mathbf{X_6}$, $\mathbf{X_7}$, $\mathbf{X_8}$, $\mathbf{X_9}$}

$$R=\left\{\begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{X_8\ X_9} \\ R_2 & \mathbf{S} & \rightarrow & \mathbf{X_8\ X_0} \\ R_3 & \mathbf{X_0} & \rightarrow & \mathbf{S\ X_9} \\ R_4 & \mathbf{S} & \rightarrow & \mathbf{S\ X_1} \\ R_5 & \mathbf{X_1} & \rightarrow & \mathbf{X_8\ X_9} \\ R_6 & \mathbf{S} & \rightarrow & \mathbf{S\ X_2} \\ R_7 & \mathbf{X_2} & \rightarrow & \mathbf{S\ X_3} \\ R_8 & \mathbf{X_3} & \rightarrow & \mathbf{S\ X_9} \\ R_9 & \mathbf{Te} & \rightarrow & \\ R_{10} & \mathbf{Te} & \rightarrow & \mathbf{X_8\ X_9} \\ R_{11} & \mathbf{Te} & \rightarrow & \mathbf{X_8\ X_4} \\ R_{12} & \mathbf{X_4} & \rightarrow & \mathbf{S\ X_9} \\ R_{13} & \mathbf{Te} & \rightarrow & \mathbf{S\ X_5} \\ R_{14} & \mathbf{X_5} & \rightarrow & \mathbf{X_8\ X_9} \\ R_{15} & \mathbf{Te} & \rightarrow & \mathbf{S\ X_6} \\ R_{16} & \mathbf{X_6} & \rightarrow & \mathbf{S\ X_7} \\ R_{17} & \mathbf{X_7} & \rightarrow & \mathbf{S\ X_9} \\ R_{18} & \mathbf{X_8} & \rightarrow & \mathbf{(} \\ R_{19} & \mathbf{X_9} & \rightarrow & \mathbf{)} \end{array}\right\}$$

Transformation d'une grammaire $^{17/26}$

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 18/26

- 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 suivis 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

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

Transformation d'une grammaire 19/26

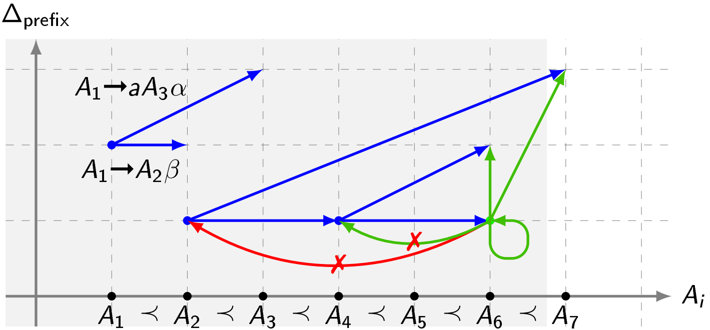
- Suppression de la récursivité gauche indirecte
  - La récursivité indirecte correspond – au niveau de l'arbre de dérivation – à un sous-arbre ayant le symbole non-terminal A pour racine et contenant ce même symbole sur l'un des nœuds de la branche la plus à gauche de ce sous-arbre
  - De manière simplifiée : $A \Rightarrow^* B\beta \Rightarrow^* A\alpha$
  - 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 en forme normale de Chomsky où les règles sont du type $A \rightarrow BC$ ou $A \rightarrow a$
  - Au fur et à mesure de la transformation, des règles d'une nouveau type vont apparaître : $A \rightarrow a\beta$ avec $\beta \in (N \cup T)^*$
  - Nous supposerons que dès de départ, ces trois types sont représentés
  - Les symboles non-terminaux sont ordonnés en A_i avec $\forall i < j, \quad ord(A_i) \prec ord(A_j)$

Transformation d'une grammaire 20/26

- Suppression de la récursivité gauche indirecte .../...
  - Supposons que toutes les règles ayant A_k pour partie gauche pour $k \leq i$ soient en forme normale de *presque* Greibach
  - La prise en compte des règles ayant A_{i+1} pour partie gauche peut remettre en cause la forme des règles ayant A_k pour partie gauche avec $k \leq i$
  - En particulier, il peut exister une règle $A_{i+1} \rightarrow A_k\alpha$ avec $k \leq i$
  - Cette règle doit être remplacée pour supprimer la récursivité qu'elle apporte, 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 $A_k \rightarrow \beta_j$
  - Exemple avec $A \rightarrow E$ et $A \rightarrow A + E$ avec $A \prec E$
    $A \Rightarrow A + E \Rightarrow A + E + E \ldots \equiv A \Rightarrow E + A \Rightarrow E + E + A \ldots$
    $\equiv A \rightarrow E$, $A \rightarrow EA'$, $A' \rightarrow +E$, $A' \rightarrow +EA'$

Transformation d'une grammaire $^{21/26}$

- 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**
          $R \leftarrow R \cup \{(A_i, \delta\beta)\}$
    $R_{A_\alpha} \leftarrow \{(\gamma, \delta) \in R : \gamma = A_i \wedge \delta = A_i\zeta\}$
    $R_{A_\beta} \leftarrow \{(\gamma, \delta) \in R : \gamma = A_i \wedge \forall\zeta \in (N \cup T)^*\ \delta \neq A_i\zeta\}$
    $(R, N) \leftarrow (R \setminus R_{A_\alpha}, N \cup \{A_i'\})$
    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 23/26

- Exemple de suppression de la récursivité gauche indirecte

Forme normale de Chomsky

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

$$R=\left\{\begin{array}{llll} R_1 & \mathbf{E} & \rightarrow & \mathbf{E} + \mathbf{T} \\ R_2 & \mathbf{E} & \rightarrow & \mathbf{T} \\ R_3 & \mathbf{T} & \rightarrow & \mathbf{T} * \mathbf{F} \\ R_4 & \mathbf{T} & \rightarrow & \mathbf{F} \\ R_5 & \mathbf{F} & \rightarrow & (\ \mathbf{E}\ ) \\ R_6 & \mathbf{F} & \rightarrow & \mathbf{a} \end{array}\right\}$$

Grammaire sans récursivité gauche

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

$$R=\left\{\begin{array}{llll} R_1 & \mathbf{E} & \rightarrow & \mathbf{T} \\ R_2 & \mathbf{T} & \rightarrow & \mathbf{F} \\ R_3 & \mathbf{F} & \rightarrow & (\ \mathbf{E}\ ) \\ R_4 & \mathbf{F} & \rightarrow & \mathbf{a} \\ R_5 & \mathbf{E} & \rightarrow & \mathbf{T}\ \mathbf{E}' \\ R_6 & \mathbf{E}' & \rightarrow & + \mathbf{T} \\ R_7 & \mathbf{E}' & \rightarrow & + \mathbf{T}\ \mathbf{E}' \\ R_8 & \mathbf{T} & \rightarrow & \mathbf{F}\ \mathbf{T}' \\ R_9 & \mathbf{T}' & \rightarrow & * \mathbf{F} \\ R_{10} & \mathbf{T}' & \rightarrow & * \mathbf{F}\ \mathbf{T}' \end{array}\right\}$$

Transformation d'une grammaire 24/26

- 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 25/26

- Factorisation à gauche .../...

Figure: Algorithme

function FactorisationGauche(G) $\triangleright\ G = (T, N, R, S)$
    $W \leftarrow N$
    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**
                $\delta \leftarrow \gamma$
                $\delta' \leftarrow \delta\zeta : R'_1 = (A, \delta\zeta\beta) \wedge \zeta \in (N \cup T)$ $\triangleright\ R'_1$: la 1ère règle de R'
                while $\forall(\alpha, \beta) \in R', \exists\zeta : \beta = \delta'\zeta$ **do** $\triangleright$ 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 26/26

- Exemple de factorisation à gauche

Grammaire sans récursivité gauche

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

$$R=\left\{\begin{array}{llll} R_1 & \mathbf{E} & \rightarrow & \mathbf{T} \\ R_2 & \mathbf{T} & \rightarrow & \mathbf{F} \\ R_3 & \mathbf{F} & \rightarrow & \mathbf{(\ E\ )} \\ R_4 & \mathbf{F} & \rightarrow & \mathbf{a} \\ R_5 & \mathbf{E} & \rightarrow & \mathbf{T\ E'} \\ R_6 & \mathbf{E'} & \rightarrow & \mathbf{+\ T} \\ R_7 & \mathbf{E'} & \rightarrow & \mathbf{+\ T\ E'} \\ R_8 & \mathbf{T} & \rightarrow & \mathbf{F\ T'} \\ R_9 & \mathbf{T'} & \rightarrow & \mathbf{*\ F} \\ R_{10} & \mathbf{T'} & \rightarrow & \mathbf{*\ F\ T'} \end{array}\right\}$$

Grammaire factorisée à gauche

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

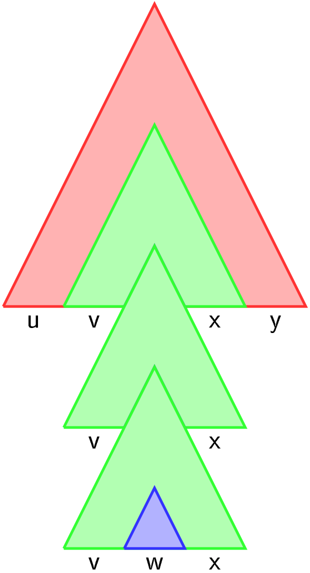
$$R=\left\{\begin{array}{llll} R_1 & \mathbf{F} & \rightarrow & \mathbf{(\ E\ )} \\ R_2 & \mathbf{F} & \rightarrow & \mathbf{a} \\ R_3 & \mathbf{E'} & \rightarrow & \mathbf{+\ T} \\ R_4 & \mathbf{E'} & \rightarrow & \mathbf{+\ T\ E'} \\ R_5 & \mathbf{T'} & \rightarrow & \mathbf{*\ F} \\ R_6 & \mathbf{T'} & \rightarrow & \mathbf{*\ F\ T'} \\ R_7 & \mathbf{E} & \rightarrow & \mathbf{T\ P_0} \\ R_8 & \mathbf{P_0} & \rightarrow & \\ R_9 & \mathbf{P_0} & \rightarrow & \mathbf{E'} \\ R_{10} & \mathbf{T} & \rightarrow & \mathbf{F\ P_1} \\ R_{11} & \mathbf{P_1} & \rightarrow & \\ R_{12} & \mathbf{P_1} & \rightarrow & \mathbf{T'} \end{array}\right\}$$

Lemme de l'étoile pour les langages algébriques 1/5

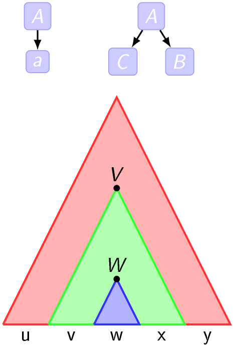
Lemme

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 2/5



- 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)

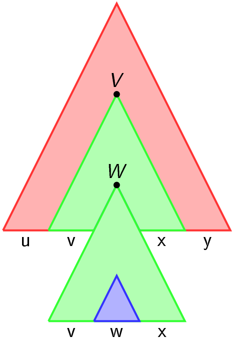
$\rightarrow$ En ignorant les feuilles, nous avons un arbre binaire

- 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 3/5

- 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, $A \Rightarrow^* v^iAx^i \Rightarrow^* v^iwx^i$
- 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 4/5

- Il existe une version plus forte de ce lemme ⇒ 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 5/5

- 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 1/1

- 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

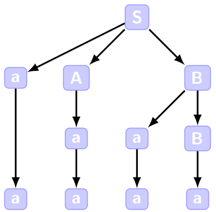
Ambiguïté *1/3*

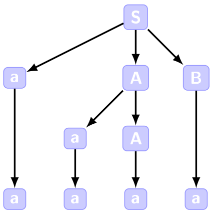
- Une grammaire est considérée ambiguë s'il existe deux dérivations à gauches différentes pour un même mot $\rightarrow$ les deux arbres de dérivation sont donc différents
- Exemple avec la grammaire

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

  $$R = \left\{ \begin{array}{llll} R_1 & \mathbf{S} & \rightarrow & \mathbf{a\ A\ B} \\ R_2 & \mathbf{A} & \rightarrow & \mathbf{a} \\ R_3 & \mathbf{A} & \rightarrow & \mathbf{a\ A} \\ R_4 & \mathbf{B} & \rightarrow & \mathbf{a} \\ R_5 & \mathbf{B} & \rightarrow & \mathbf{a\ B} \end{array} \right\}$$

- Donnez deux dérivations à gauche différentes pour le mot *aaaa*





Ambiguïté 2/3

- 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

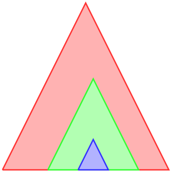
  - La dérivation à gauche impose un ordre sur les symboles non-terminaux à dériver mais elle ne va pas jusqu'à imposer un ordre sur les règles à appliquer

- 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 3/3

- 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 $n!/k$ fois le lemme de l'étoile

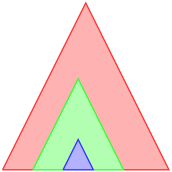
$$\begin{array}{rlllll}
\omega & = a^{n+n!} b^{n-k-1} & b^k & bc & c^k & c^{n-k-1} \\
& = u & v & w & x & y \\
\omega' & = u & v^{n!/k+1} & w & x^{n!/k+1} & y \\
& = a^{n+n!} b^{n-k-1} & b^{k\times(n!/k+1)} & bc & c^{k\times(n!/k+1)} & c^{n-k-1} \\
& = a^{n+n!} b^{n-k-1} & b^{n!+k} & bc & c^{n!+k} &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



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

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

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



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

Notation *1/4*

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

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

- Notation BNF associée

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

Notation 2/4

- 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` → `*A`
  - `S =   A / A S` → `+A`
  - `S = A A / A S` → `3*A`
- Exemple avec la grammaire

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

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

- Notation ABNF associée

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

Notation 3/4

- 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 4/4

- 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) ]
```

Les grammaires

Les grammaires de type 1

Les grammaires de type 2

Les grammaires de type 3

Grammaire de type 3 1/3

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 $S \Rightarrow a_0A_0 \Rightarrow a_0a_1A_1 \Rightarrow a_0a_1a_2A_2 \Rightarrow a_0a_1a_2a_3A_3 \Rightarrow \dots$
- 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 2/3

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 *3/3*

- Exemple avec la grammaire des prénoms

S=**L** T={**andre**, **liam**, **phil**, **et**, **,**} N={**L**, **N**, **M**, **F**}

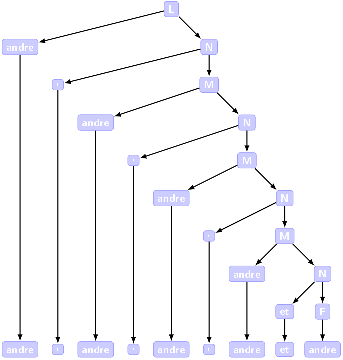
$$R=\left\{\begin{array}{cccl} R_1 & \mathbf{L} & \rightarrow & \mathbf{andre} \\ R_2 & \mathbf{L} & \rightarrow & \mathbf{liam} \\ R_3 & \mathbf{L} & \rightarrow & \mathbf{phil} \\ R_4 & \mathbf{L} & \rightarrow & \mathbf{andre\ N} \\ R_5 & \mathbf{L} & \rightarrow & \mathbf{liam\ N} \\ R_6 & \mathbf{L} & \rightarrow & \mathbf{phil\ N} \\ R_7 & \mathbf{N} & \rightarrow & \mathbf{,\ M} \\ R_8 & \mathbf{N} & \rightarrow & \mathbf{et\ F} \\ R_9 & \mathbf{M} & \rightarrow & \mathbf{andre\ N} \\ R_{10} & \mathbf{M} & \rightarrow & \mathbf{phil\ N} \\ R_{11} & \mathbf{M} & \rightarrow & \mathbf{liam\ N} \\ R_{12} & \mathbf{F} & \rightarrow & \mathbf{andre} \\ R_{13} & \mathbf{F} & \rightarrow & \mathbf{liam} \\ R_{14} & \mathbf{F} & \rightarrow & \mathbf{phil} \end{array}\right\}$$

- Notation ABNF associée

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

Graphe de production 1/1

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



Opérations sur les langages réguliers *1/1*

- 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 1/1

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. $\forall i > 0, vw^ix \in L$

- 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

Part III

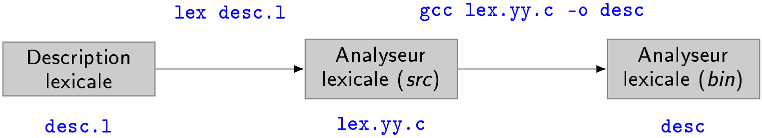
Lex

Yacc

Compilation

Introduction à Lex 1/1

- 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 *1/1*

```
%{
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 1/2

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

Expressions régulières de Lex 2/2

- 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 *1/1*

`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 $^{1/2}$

- 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 2/2

- 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 1/1

- 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

Listing 28: `09-lex/syntaxe.l`

```
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]);}
```

Lex

Yacc

Compilation

Introduction à Yacc 1/1

- `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 *1/1*

- 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 *1/1*

- 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 1/3

- 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 :

Listing 29: `10-yacc/ligne.y`

```
%{
#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");}
```

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

Zone contenant les règles 2/3

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

Listing 30: `10-yacc/ligne2.y`

```
%{
#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");}
```

Zone contenant les règles 3/3

- Exemple avec la grammaire

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

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

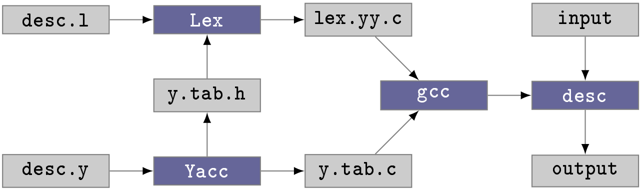
- Fichier Yacc associé

Listing 31: 10-yacc/prenoms.y

```
%{
#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 ;
```

Combinaison de Lex et Yacc 1/4

- 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 2/4

- Fichier Lex associé à la liste des prénoms

Listing 32: 10-yacc/prenoms.l

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

%%

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

Listing 33: 10-yacc/prenoms.y

```
%{
#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 ;
```

Combinaison de Lex et Yacc 3/4

- Contenu du fichier `y.tab.h`

Listing 34: 10-yacc/prenoms-y.tab.h.extrait

```
/* 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;
```

Combinaison de Lex et Yacc 4/4

- 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

Listing 35: 10-yacc/erreur.l

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

Listing 36: 10-yacc/erreur.y

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

- Exécution

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

Exemple *1/1*

- 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

Listing 37: `10-yacc/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 38: `10-yacc/html.y`

```
%{
#include <stdio.h>
%}
%token tDHTML tFHTML tDIMG
%token tFIMG tDBODY tFBODY
%token tTEXTE tNL
%start Document
%%
Document :
    tDHTML Corps tFHTML
    {printf("OK\n");} ;
Corps :
    /* Vide */
  | Element Corps ;
Element : tTEXTE
  | tNL
  | tDIMG Corps tFIMG
  | tDBODY Corps tFBODY
  | tDHTML Corps tFHTML ;
```

Attributs des symboles 1/4

- 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

→ Manque de souplesse – Que faire ?

Attributs des symboles 2/4

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

Listing 39: 10-yacc/html2.y

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

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

Attributs des symboles 3/4

- 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 4/4

- 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 *1/2*

- Construire une calculatrice

Listing 40: 10-yacc/calc.l

```
%{
#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; }
```

Exemple 2/2

- Construire une calculatrice . . . / . . .

Listing 41: 10-yacc/calc.y

```
%{ /* 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;
}
```

Conflits *1/3*

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

Conflits 2/3

- Conflit *reduce/ reduce*

Listing 42: 10-yacc/conflit1.l

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

Listing 43: 10-yacc/conflit1a.y

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

Listing 44: 10-yacc/conflit1b.y

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

```
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 3/3

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

Listing 45: 10-yacc/shift.y

```
%{
#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;
}
```

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

Exercice *1/1*

- Ecrivez un parser pour la grammaire $a^n b^n c^n$

Lex

Yacc

Compilation

Introduction à la compilation 1/3

- Un programme informatique est une séquence d'instructions d'un processeur qui indique les étapes à suivre pour résoudre un problème
- Pour le dévoloppement de ces programmes, nous disposons du jeu d'instructions du processeur (*il peut différer d'un processeur à l'autre*)

intel

Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 2A: Instruction Set Reference, A-M

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of five volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-M*, Order Number 253666; *Instruction Set Reference N-Z*, Order Number 253667; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669. Refer to all five volumes when evaluating your design needs.

Order Number: 253666-037US
January 2011

CONTENTS

PAGE

3.2 INSTRUCTIONS (A-M) 3-22
AAA—ASCII Adjust After Addition 3-23
AAD—ASCII Adjust AX Before Division 3-25
AAM—ASCII Adjust AX After Multiply 3-27
AAS—ASCII Adjust AL After Subtraction 3-29
ADC—Add with Carry 3-31
ADD—Add 3-35
ADDPD—Add Packed Double-Precision Floating-Point Values 3-38
ADDPS—Add Packed Single-Precision Floating-Point Values 3-40
ADDSD—Add Scalar Double-Precision Floating-Point Values 3-42
ADDSS—Add Scalar Single-Precision Floating-Point Values 3-44
ADDSUBPD—Packed Double-FP Add/Subtract 3-46
ADDSUBPS—Packed Single-FP Add/Subtract 3-49
AESDEC—Perform One Round of an AES Decryption Flow 3-52
AESDECLAST—Perform Last Round of an AES Decryption Flow 3-54
AESENC—Perform One Round of an AES Encryption Flow 3-56
AESENCLAST—Perform Last Round of an AES Encryption Flow 3-58
AESIMC—Perform the AES InvMixColumn Transformation 3-60
AESKEYGENASSIST—AES Round Key Generation Assist 3-62
AND—Logical AND 3-64
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values 3-67
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values 3-69
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values . 3-71
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values 3-73
ARPL—Adjust RPL Field of Segment Selector 3-75
BLENDPD — Blend Packed Double Precision Floating-Point Values 3-78
BLENDPS — Blend Packed Single Precision Floating-Point Values 3-80
BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values 3-83
BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values 3-86
BOUND—Check Array Index Against Bounds 3-89
VBROADCAST—Load with Broadcast 3-92
BSF—Bit Scan Forward 3-96
BSR—Bit Scan Reverse 3-99
BSWAP—Byte Swap 3-102
BT—Bit Test 3-104
BTC—Bit Test and Complement 3-107
BTR—Bit Test and Reset 3-110
BTS—Bit Test and Set 3-113
CALL—Call Procedure 3-116
CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword 3-135
CLC—Clear Carry Flag 3-137
CLD—Clear Direction Flag 3-138
CLFLUSH—Flush Cache Line 3-139
CLI — Clear Interrupt Flag 3-141

CLC—Clear Carry Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F8	CLC	A	Valid	Valid	Clear CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA

Description

Clears the CF flag in the EFLAGS register. Operation is the same in all non-64-bit modes and 64-bit mode.

Operation

CF ← 0;

Flags Affected

The CF flag is set to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

Introduction à la compilation 2/3

- Exercice n°1 : développez une programme permettant de calculer $n!$ en assembleur x86

Listing 46: 11-compil/factorial.asm

```
FACTORIAL:
  // Sauvegarde des registres utilises
  push ebx
  // Recuperation du parametre "n" dans ebx
  mov ebx, dword ptr [esp + 8]
  // Test si "n==1"
  cmp ebx, 1
  jg RECURSE
  // "n==1" donc, "fact(1)=1"
  mov eax, 1
  jmp STOP
  // "n!=1" donc, "fact(n()=fact(n-1)*n"
RECURSE:
  dec ebx
  call FACTORIAL
  inc ebx
  mul ebx
STOP:
  pop ebx
  // Le resultat est dans eax
  ret
```

- Exercice n°2 : développez un programme de traitement de textes en assembleur x86

Introduction à la compilation *3/3*

- Qu'est-ce qu'un compilateur ?
  Un outil permettant de traduire un programme d'un langage vers un autre
- Habituellement, cette traduction est réalisée à partir d'un langage de haut niveau vers un langage de bas niveau

`latex`	:	`tex`	$\rightarrow$	`postscript`
`xpdf`	:	`pdf`	$\rightarrow$	`Pixels`
`gcc`	:	`C`	$\rightarrow$	`asm x86`
`firefox`	:	`html`	$\rightarrow$	`Widgets`
`graphviz`	:	`dot`	$\rightarrow$	`png`

- Le compilateur met également à profit les spécificités de l'architecture de destination (jeu d'instructions enrichi, organisation de la mémoire, etc)
- Il guide l'utilisateur en affichant des messages d'erreurs et des *warnings*

Exemple C → asm x86 1/1

Listing 47: 11-compil/simple.c

```c
/* Headers */
#include <stdio.h>

/* Fonction inutile */
void f(int* a) {
  *a = 12;
}
/* Programme principal */
int main(void) {
  int i;
  i = 0;
  i = 8;
  printf("%d\n", i);
  f(&i);
  printf("%d\n", i);
  return 0;
}
```

Listing 48: 11-compil/simple.asm

```
$ objdump -d simple
...
080483e0 <main>:
 80483e0: 8d 4c 24 04          lea  ecx,[esp+0x4]
 80483e4: 83 e4 f0             and  esp,0xfffffff0
 80483e7: ff 71 fc             push DWORD PTR [ecx-0x4]
 80483ea: 55                   push ebp
 80483eb: 89 e5                mov  ebp,esp
 80483ed: 51                   push ecx
 80483ee: 83 ec 14             sub  esp,0x14
 80483f1: c7 44 24 04 08 00 00 mov  DWORD PTR [esp+0x4],0x8
 80483f8: 00
 80483f9: c7 04 24 f4 84 04 08 mov  DWORD PTR [esp],0x80484f4
 8048400: e8 ef fe ff ff       call 80482f4 <printf@plt>
 8048405: c7 44 24 04 0c 00 00 mov  DWORD PTR [esp+0x4],0xc
 804840c: 00
...
```

Listing 49: 11-compil/simple.symbols

```
$ objdump -t a.out
...
08048440 g     F .text  0000005a              __libc_csu_init
00000000       F *UND*  00000036              printf@@GLIBC_2.0
08049668 g       *ABS*  00000000              __bss_start
08049670 g       *ABS*  00000000              _end
080483d0 g     F .text  0000000e              f
08049668 g       *ABS*  00000000              _edata
0804849a g     F .text  00000000              .hidden __i686.get_pc_thunk.bx
080483e0 g     F .text  00000042              main
08048294 g     F .init  00000000              _init
```

Les difficultés *1/1*

- Un compilateur doit pouvoir traduire tous les programmes valides
- Il doit traduire les programmes valides de manière cohérente
- Il doit détecter les erreurs de syntaxe
- Il doit traduire chaque instruction du programme initial
- Il doit gérer efficacement les ressources à sa disposition

$\rightarrow$ Besoin d'un méthodologie

Comment faire ? *1/1*

- Structuration du travail en plusieurs étapes
- Pour les compilateurs habituels

Analyse lexicale

Analyse syntaxique

Analyse sémantique

Génération de code intermédiaire

Optimisation du code intermédiaire

Génération du code objet

Analyse lexicale *1/1*

- L'analyse lexicale permet de préparer la séquence de caractères pour l'analyse syntaxique
  - Suppression des éléments qui n'entrent pas dans le cadre de l'analyse syntaxique (commentaires, espaces, etc)
- Les caractères sont regroupés en élements $\rightarrow$ les tokens
  - Chaque mot clé du langage est associé à un token
  - Chaque symbole de ponctuation du langage est associé à un token
- De plus, l'analyse syntaxique devient indépendante de la source de la séquence de caractères

Analyse syntaxique *1/1*

- L'analyse syntaxique permet de vérifier que la séquence de tokens est cohérente vis-à-vis d'une grammaire
- Les actions peuvent être associées aux règles
- Habituellement, ces actions permettent de construire un arbre syntaxique abstrait (AST)
- Cet arbre permet plus facilement de mener les opérations d'optimisation (il est dépouillé des nœuds correspondant aux symboles non-terminaux dans l'arbre de dérivation)
- Dans notre cas, nous allons directement produire le code machine

Quelques pistes pour la suite *1/1*

- Gestion des déclarations dans les blocs des structures `if-else` et `while` $\rightarrow$ modification de la table des symboles en pile
  - A l'entrée dans un bloc : mémorisation du sommet de la table des symboles
  - A la sortie : restauration du sommet de la table des symboles
- Gestion des fonctions : autant de tables des symboles que de fonctions
- Invocation des fonctions : utilisation de l'adressage indirect + utilisation des instructions `call` et `ret`
- Gestion des pointeurs : utilisation de l'adressage indirect

Table des symboles 1/1

- La table des symboles contient l'ensemble des symboles déclarés lors des instructions précédentes
- Elle permet de mémoriser le type de chaque variable
- Elle permet de vérifier qu'un symbole rencontré dans une instruction a bien été déclaré
- Elle permet également de vérifier qu'un symbole a été initialisé avant d'être utilisé en lecture
- Elle est renseignée au fur et à mesure de la déclaration des variables
- Peut-elle contenir la valeur associé à chaque symbole ?

```
1   int main(void) {
2     int i;
3     float k = 8.0;
4     k = i;
4     return 0;
5   }
```

Nom du symbole	Type du symbole	Initialisé ?
`main`	`int(*)(void)`	non
`i`	`int`	non
`k`	`float`	oui

Projet système informatique *1/1*

- Démarche proposée
  1. Construction de l'analyseur lexicale reconnaissant tous les tokens
  2. Construction de l'analyseur syntaxique reconnaissant les programmes valides
     $\rightarrow$ les actions contiennent uniquement des `printf`
  3. Création de la table des symboles
  4. Implémentation des actions pour les déclarations
  5. Implémentation des actions pour le `print`
  6. Implémentation des actions pour les expressions arithmétiques
  7. Implémentation de la structure `if-else`
  8. Implémentation de la structure `while`
  9. Ajout des fonctions
  10. Ajout des pointeurs
  11. Ajout des chaînes de caractères