Cours de compilation MASTER M1

Introduction à la compilation

Pourquoi écrire un compilateur

- La réalisation d'un compilateur nécessite une bonne maîtrise de la programmation (programmation modulaire, récursivité, exemples de grand système complexe…)
- Les techniques de compilation sont nécessaires pour lire, modifier, écrire des données en format interchangeable
  - HTML, XML

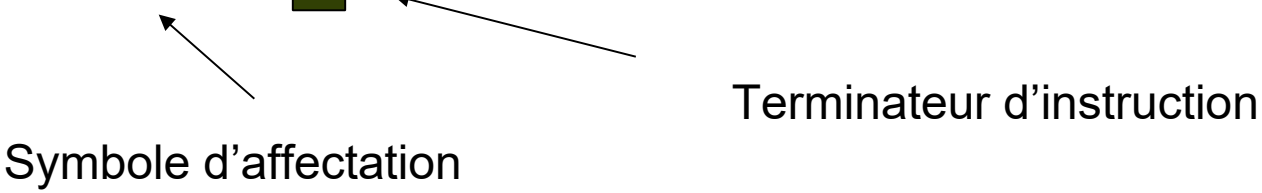
Rôle d'un compilateur

- Le but d'un compilateur est de traduire des programmes écrits dans un langage *source* en un langage *destination*
- Souvent, le *source* est un langage de programmation de haut et la *destination* est en langage machine
- Parfois, traduction de source en source
  - LaTeX en HTML (HEVEA)
  - Pascal en C, FORTRAN en C
  - Manipulation de données en XML
- Le compilateur doit détecter si le *source* est conforme aux règles de programmation

Langages

- Un langage est un ensemble de caractères alpha-numériques que l'on appelle *phrase* du langage
- Les règles définissant la structure des phrases sont définies par une *grammaire*
- Exemple :
  - Une affectation en Java est de la forme

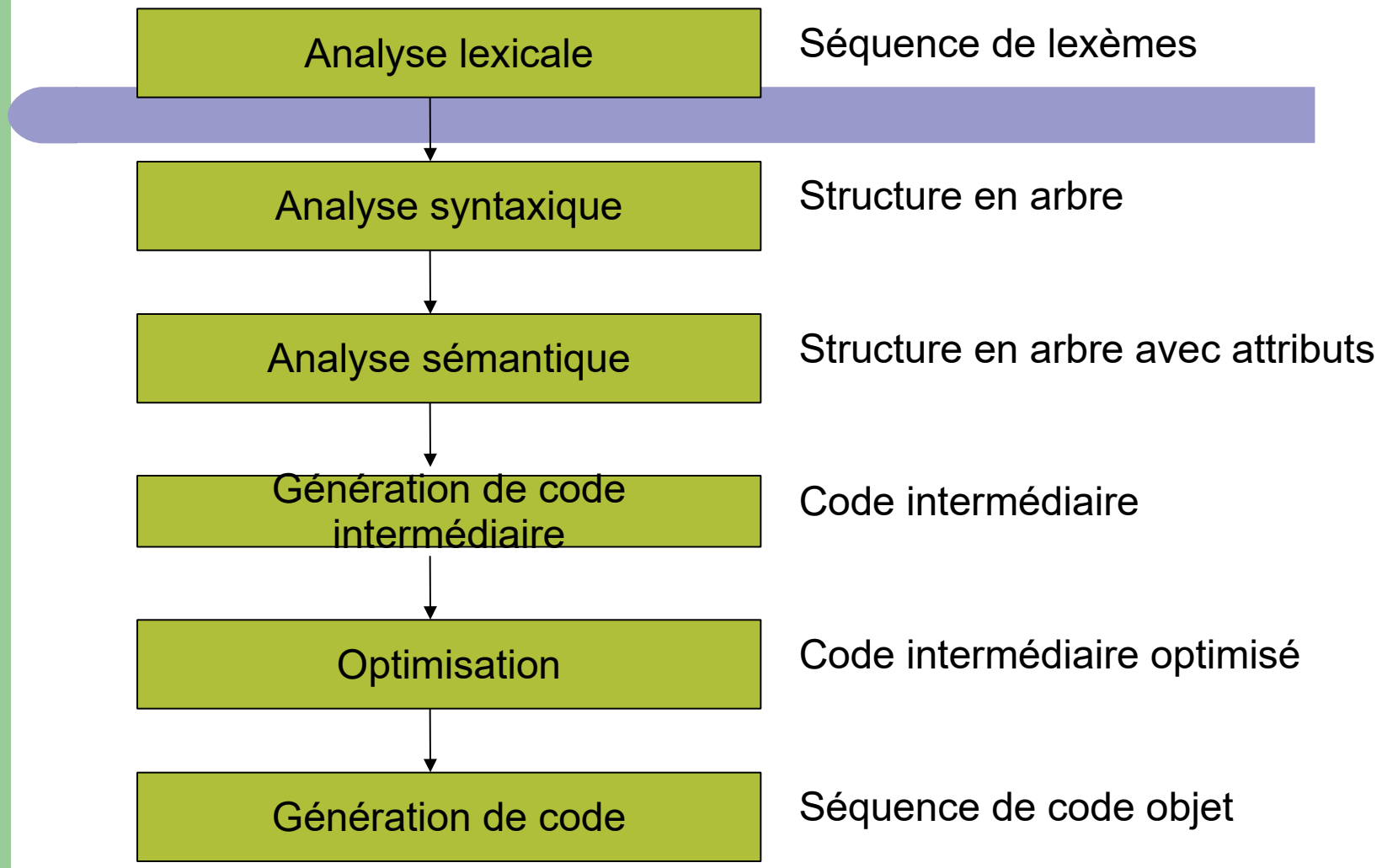
    a = b + 5 ;



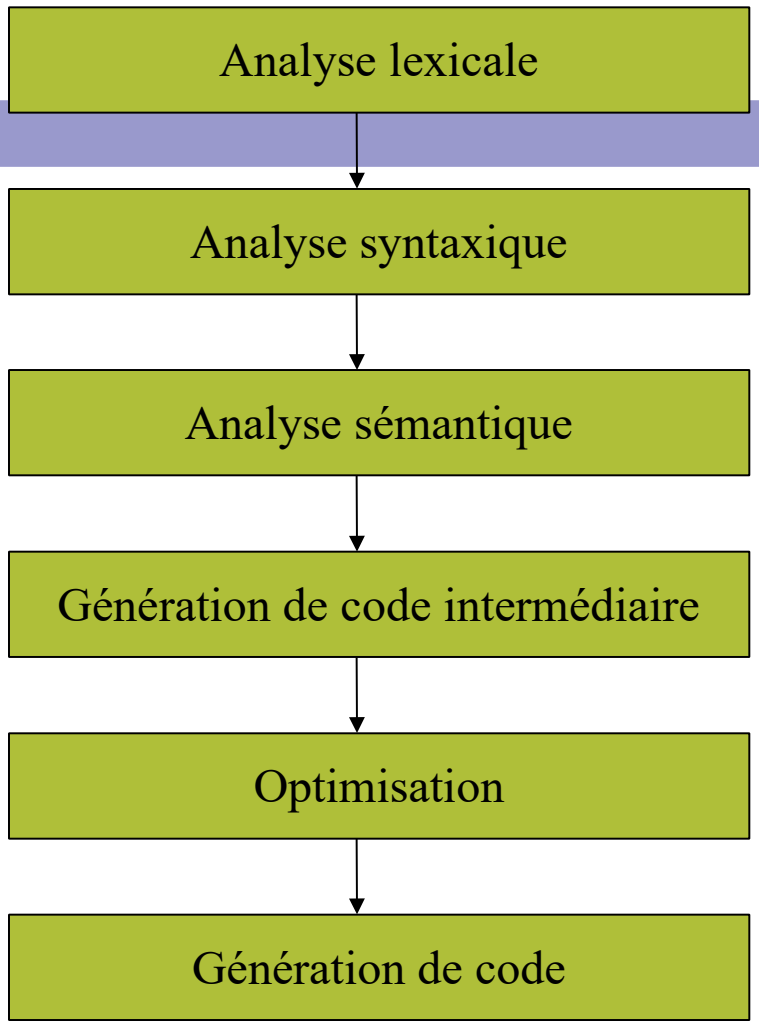
Phases d'un compilateur

- Un compilateur opère généralement en *phases*, chacune d'elles transformant le programme source d'une représentation en une autre
- Les phases ne sont pas toujours clairement définies
- Certaines phases peuvent être absentes

Structure d'un compilateur

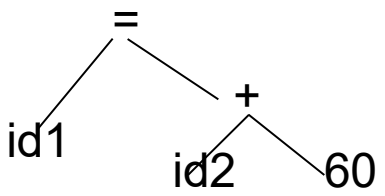


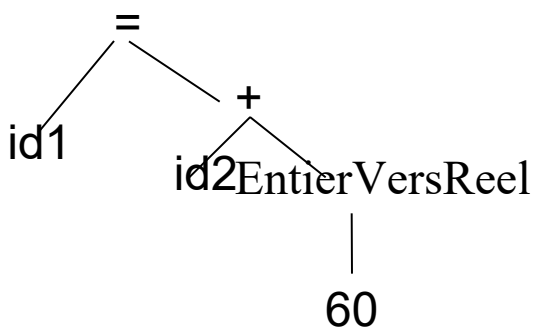
Structure d'un compilateur



a = b + 60;

id1 = id2 + 60 ;





temp1 = EntierVersReel(60)
temp2 = id2 + temp1
id1 = temp2

mov id2, r2
add 60, r2
mov r2, id1

Analyse lexicale

- Parcourt le programme source de gauche à droite, caractère par caractère
- Ignore les caractères superflus (espaces, tab, RC, commentaires)
- Convertit les unités lexicales en tokens ({, id, =, ==, abstract ...)
- Envoie ces informations à l'analyse syntaxique

Exemple :

```
position = initiale + vitesse*60 ;
ID   OPAFFACT ID PLUS ID MULT NB PTVIRG
```

Analyse syntaxique

- Vérifie si la séquence d'unités lexicales fournie par l'analyse lexicale respecte les « règles »syntaxiques du langage

Exemple : règles syntaxiques

EXP → EXP *PLUS* EXP |
EXP *PLUS* EXP

Génération de code

- Produit le code
- Le code généré peut être :
  - Code intermédiaire : facile à produire et à traduire par la suite
  - Code machine translatable

Gestion de la table des symboles

Une table des symboles est une structure de données contenant un enregistrement pour chaque identificateur

Lexème	Catégorie	Type	Paramètres
Position	VAR	REAL	
Somme	FCT	ENTIER	A ENTIER
			B ENTIER

Analyse Sémantique

- Contrôle si le programme est sémantiquement correct (contrôle des types, portée des identificateurs, correspondance entre paramètres formels et paramètres effectifs)
- Collecte des informations pour la production de code (dans la table des symboles)

Cousins des compilateurs

- Les préprocesseurs produisent ce qui sera l'entrée d'un compilateur
  - inclusion de fichiers
  - macro-expansion
- Éditeur de liens (chargement et reliure)
  - le chargement consiste à prendre du code machine translatable et à modifier les adresses translatables
  - le relieur constitue un unique programme à partir de plusieurs fichiers contenant du code machine translatable (compilation séparée ou bibliothèque)

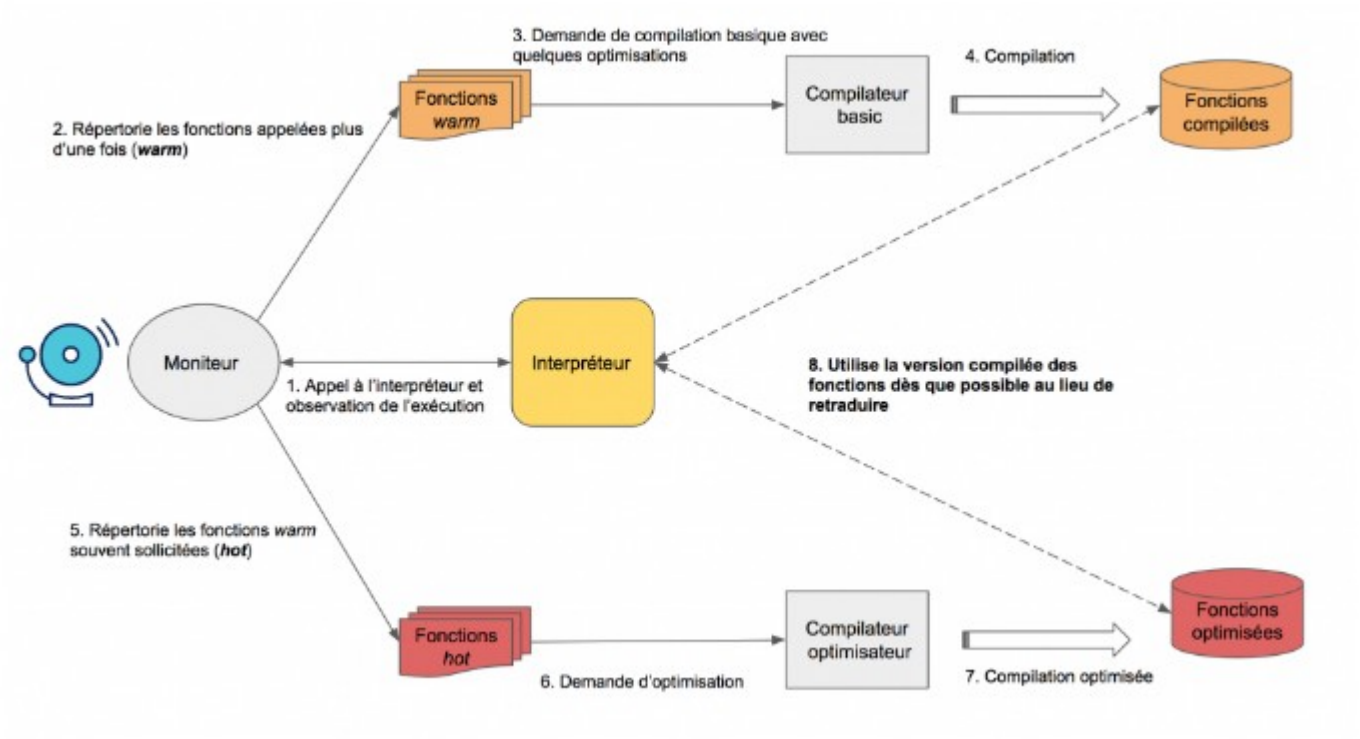
Interpréteur

- Les interpréteurs s'arrêtent à la phase de production de code intermédiaire
- L'exécution du programme *source* consiste alors en une *interprétation* du code intermédiaire par un programme appelé *interpréteur*
- Souvent, il exécute un code intermédiaire (par exemple le code JavaScript)
- Avantages : portabilité et exécution immédiate
- Désavantage : exécution plus lente qu'avec un compilateur complet

Culture : *bytecode*

- Le compilateur ne produit pas de code pour un processeur réel, mais pour un processeur d'une machine virtuelle
- Portabilité : pour une nouvelle architecture, il n'y a pas besoin de modifier le compilateur, il suffit *a priori* de porter le programme qui implémente la machine virtuelle
- **JAVA** : *« Compile once, run everywhere »*
- Un peu exagéré car dans le cas de Java, l'environnement ne se compose pas seulement d'un processeur, mais aussi de nombreuses librairies

Culture JavaScript

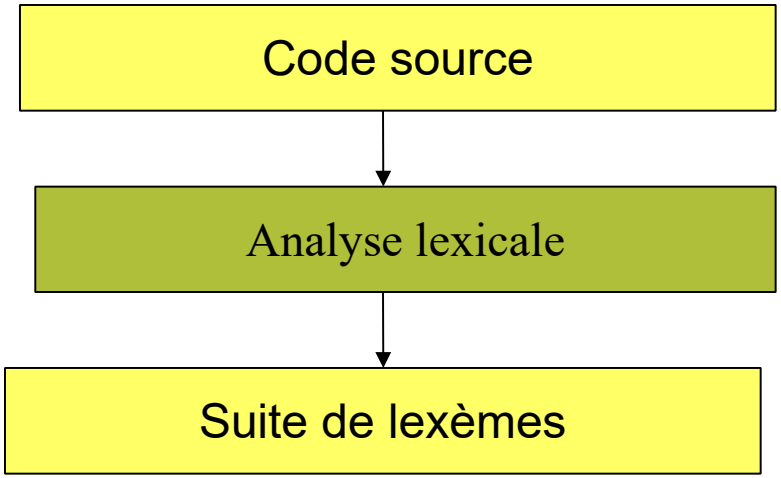


JavaScript (suite)

Javascript execution in V8 is divided into three stages:

- Source to syntax tree: the parser generates an abstract syntax tree (AST) from source
- Syntax tree to bytecode: V8's interpreter Ignition generates bytecode from the syntax tree. Note that this bytecode step was not present before 2017. The pre-2017 V8 is described here.
- Bytecode to machine code: V8's compiler TurboFan generates a graph from bytecode, replacing sections of bytecode with highly-optimized machine code

Analyse lexicale



Introduction

- L'analyse lexicale se trouve au tout début de la chaîne de compilation
- Elle collabore avec l'analyse syntaxique pour analyser la structure du langage
- Elle transforme une suite de caractères en une suite de mots, dit aussi lexèmes *(tokens)*
- L'analyse lexicale utilise essentiellement des automates pour reconnaître des expressions régulières

Principe

- Transforme une suite de caractère en une suite de lexèmes

```
somme = total + 5.25 ;
```

L'analyseur lexical retournera :

```
IDENTIFICATEUR
OPPAFFECT
IDENTIFICATEUR
PLUS
NOMBREREEL
PTVIRG
```

Tâches secondaires

L'analyse lexicale peut également effectuer un certain nombre de tâches :

- Élimination des caractères superflus (espaces, tabulations, RC, commentaires, ...)
- Compter le nombre de RC afin de repérer le numéro de la ligne si une erreur est détectée
- Conversion des lettres en majuscules (pour un langage comme PASCAL par exemple)

Enjeux

- Les analyses lexicales et grammaticales ont un domaine d'application bien plus large que la compilation (analyse de requêtes…)
- Ces deux analyses utilisent principalement les automates et les expressions régulières (utilisées dans de nombreux programmes unix, comme sed, grep, …)

Note

L'étude détaillée des automates et des grammaires formelles pourrait constituer un cours à part entière (cours très théorique au demeurant !!!)

Nous nous contenterons pour ce cours d'une description minimale avec comme but :

- se familiariser avec les expressions régulières
- comprendre les principes de base du fonctionnement des automates

Analyse lexicale et NLP

- L'analyse lexicale est utilisée dans tous les compilateurs et interpréteurs
- Maintenant extrêmement utilisé en NLP (Natural Language Processing) mais
  - Analyseur beaucoup plus complexe

Exemple

"Clairson Internation Corp. said it expects to report a net loss for its second quarter ended March 26 and doesn't expect to meet analysts' profit estimates of $3.9 to $4 million."

- 3 points différents
- 2 apostrophes différentes

M. O'Neill thinks that the boys' stories aren't amusing

- oneill
- O'Neill
- neill

15 décembre 2020

- en général, les espaces séprant des mots. Ici c'est mieux de considérer l'ensemble comme une date

Quelques définitions

Un modèle est une « règle » qui décrit l'ensemble des lexèmes pouvant représenter une unité lexicale

Identificateur	Position Vitesse I1 Somme	Lettre suivie de Lettre et de chiffres
OPAFFACT	=	=
Nombre	-2,50 -9,10 123	Constante numérique : Signe facultatif suivi d'une Suite de chiffres
SI Alors Sinon	Si Alors Sinon	Si Alors Sinon
OPREL	< <= > >= == !=	< ou <= ou > ou >= ou == ou !=

Unités lexicales

Quand différents lexèmes désignent la même unité lexicale, *un attribut* est associé à l'unité lexicale

Exemple :

ID	→ chaîne de caractères « position »
NB	→ valeur : 60
OPREL	→ différentes constantes numériques PPQ, PPE, PGQ, PGE, DIF, EGA

Langages formels

Ensemble fini Σ appelé alphabet, dont les éléments sont appelés caractères

Un mot (sur Σ) est une séquence de caractères (sur Σ)

ε le mot vide

uv la concaténation des mots *u* et *v*

$\Sigma*$ ensemble des mots sur Σ

Un langage sur Σ est un sous ensemble L de $\Sigma*$

Si *U* et *V* sont deux langages sur Σ

- *UV* ensemble concaténation
- *U* (resp U+)* ensemble des mots obtenus par la concaténation arbitraire, éventuellement nul (resp. non nul) de mots de *U*

Exemples

- Σ_1 est l'alphabet français et L_1 l'ensemble des mots du dictionnaire français avec toutes leurs déclinaisons
- Σ_2 est l'ensemble des caractères ASCII, et L_2 est l'ensemble des mots-clefs de JAVA l'ensemble des symboles, l'ensemble des identificateurs…

Formalisme des expressions régulières

a,b, etc... des lettres de Σ, M et N des expressions régulières, $[\![M]\!]$ le langage associé à M

- une lettre de l'alphabet a désigne le langage {a}
- ε désigne le langage $\{\varepsilon\}$
- concaténation : *MN* désigne le langage $[\![M]\!][\![N]\!]$
- M|N désigne le langage $[\![M]\!] \cup [\![N]\!]$
- Répétition : M* désigne le langage $[\![M]\!]^*$

Expression et langage réguliers

Chaque E.R. E définir un langage régulier L(e) de la façon suivante :

si $a \in \Sigma$ $L(a)=\{a\}$

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

et $L(e1|e2)=L(e_1) \cup L(e_2)$

$L(e1e2)=L(e_1)L(e_2)$

$L(e^*)=\cup(L(e))^i$ avec $(L(e))^0=\{\varepsilon\}$

$=\{\varepsilon\} \cup L(e) \cup (L(e))^2 \cup \ldots$

L'opérateur de Kleene * est le plus prioritaire, puis la concaténation, puis l'union

Exemple

Soit $\Sigma=\{a,b,c\}$

$a\|b$	définit le langage {a,b}
$(a\|b)(a\|b)$	{aa,ab,ba,bb}
a^*	$\{\varepsilon,a,a^2,a^3,...\}$
$(a\|b)^*$	$\{\varepsilon,a,b,aa,ab,ba,bb,..\}$
$a^*\|b^*$	$\{\varepsilon,a,a^2,a^3,...,b,b^2,b^3,...\}$ tous les mots formés de a ou b
$a(b\|c)^*$	tous les mots commençant par a suivi de b et de c

Remarques

- Deux E.R. Sont équivalentes si elles définissent le même langage
  Ex : $(a|b)^*=a^*(a|b)^*$
- On utilise parfois la notation e^+ pour ee^*
  $L(e^+) = U_{i=1}(L(e))^i$
- Pour des raisons de lisibilité, il est possible de nommer certaines E.R. Et d'utiliser ces noms dans d'autres E.R.
  - DEC → $(0|1|...|9)^+(\varepsilon|.(0|1|...9|)^+)$
  - Chiffre → $(0|1|...|9)$
  - DEC → $Chiffre^+(\varepsilon|.Chiffre^+)$

Formalisme des expressions régulières

Autres constructions obtenues à partir des précédentes

- *[abc]* pour *(a|b|c)* et *[a1-a2]* pour tout caractère entre a1 et a2 (alphabet supposé ordonné)
- M+ pour MM*
- *[^abc]* désigne le complémentaire de *{a,b,c}* dans Σ
- . pour Σ (tout caractère)
- *a?* est une abréviation de $a|\varepsilon$

Règles de priorité

Problème : il peut y avoir des ambiguïtés avec les expressions régulières

- int peut être comme la déclaration de la variable int a,b,c
- interne peut aussi être reconnu comme un début de déclaration de variables

On choisit par priorité :

- Le lexème le plus long ;
- l'ordre de définition.

Récapitulation

Le rôle d'un analyseur lexical est la reconnaissance des unités lexicales. Une unité lexicale peut (la plupart du temps) être exprimée sous forme d'expression régulière.

Théorème : tout langage régulier est reconnu par un automate fini

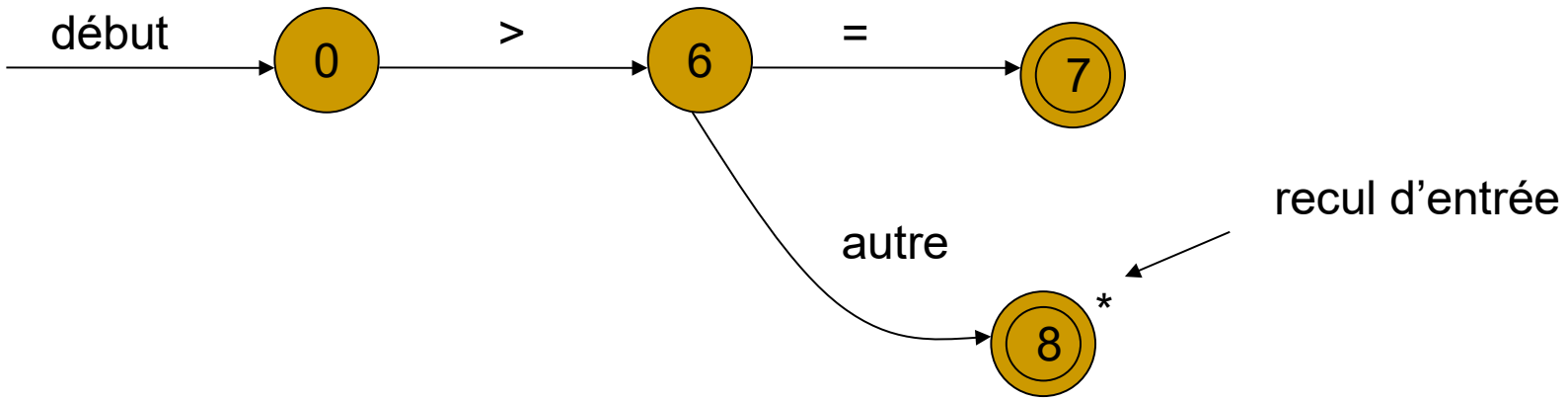
Diagrammes de transition

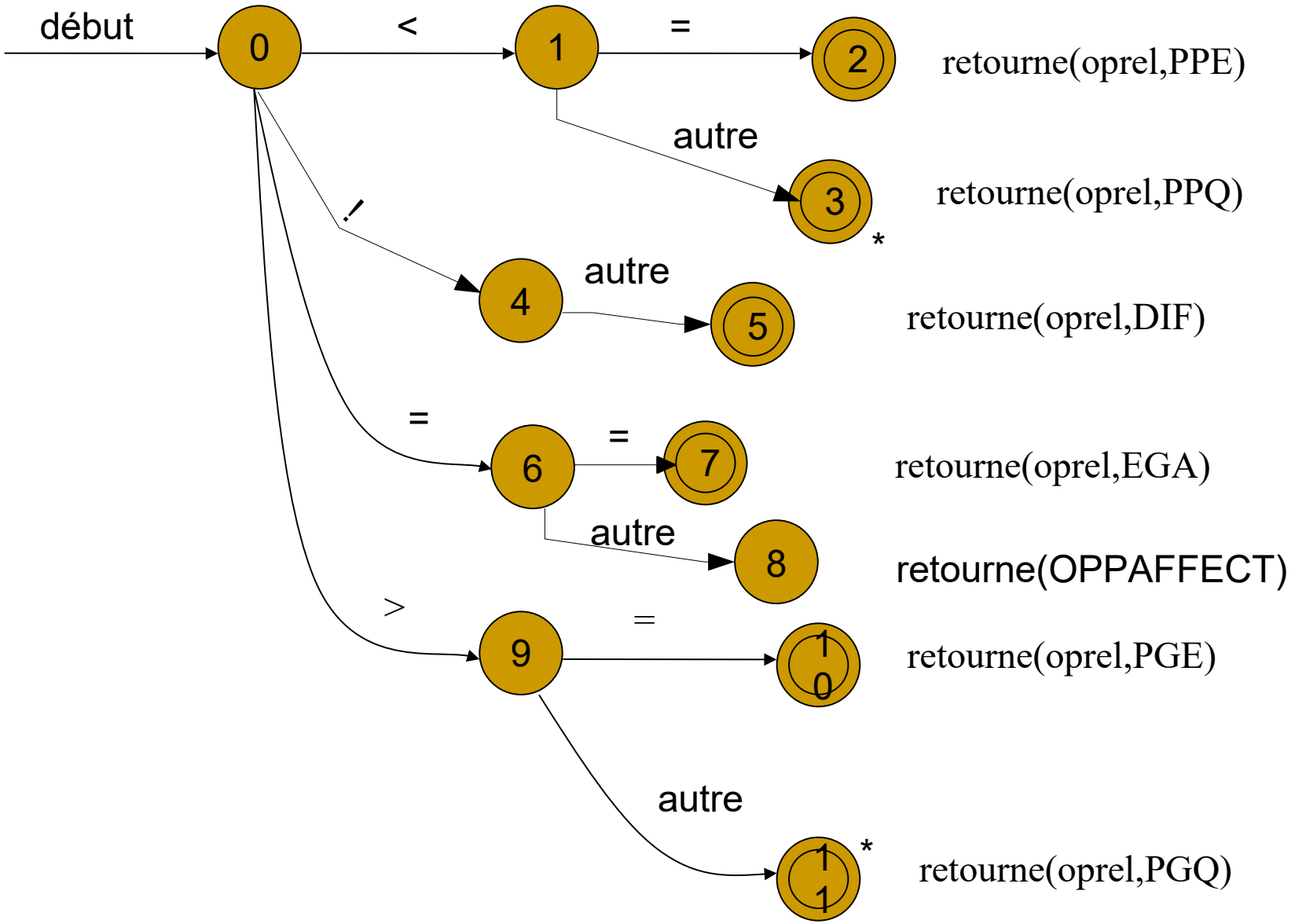
- Les expressions régulières permettent de délimiter les éléments lexicaux du langages
- Comment lire facilement les éléments lexicaux d'un langage (moins facile qu'il n'y paraît !)
- Utilisation des diagrammes de transition

Diagrammes de transition

- Ce sont des automates finis déterministes
- On se déplace de position en position dans le diagramme au fur et à mesure de la lecture des caractères
- Les positions dans l'automate sont appelées *états*
- Les *états* sont reliés par des *arcs* qui ont des étiquettes indiquant les caractères lus
- Les automates sont déterministes (aucun symbole ne peut apparaître comme étiquette de deux arcs quittant un nœud)
- Un *état* appelé *départ* est l'état initial

Exemple




début
0
<
1
=
2
retourne(oprel,PPE)
autre
3
*
retourne(oprel,PPQ)
!
4
autre
5
retourne(oprel,DIF)
=
6
=
7
retourne(oprel,EGA)
autre
8
retourne(OPPAFFECT)
>
9
=
10
retourne(oprel,PGE)
autre
11
*
retourne(oprel,PGQ)

AFD (définition formelle)

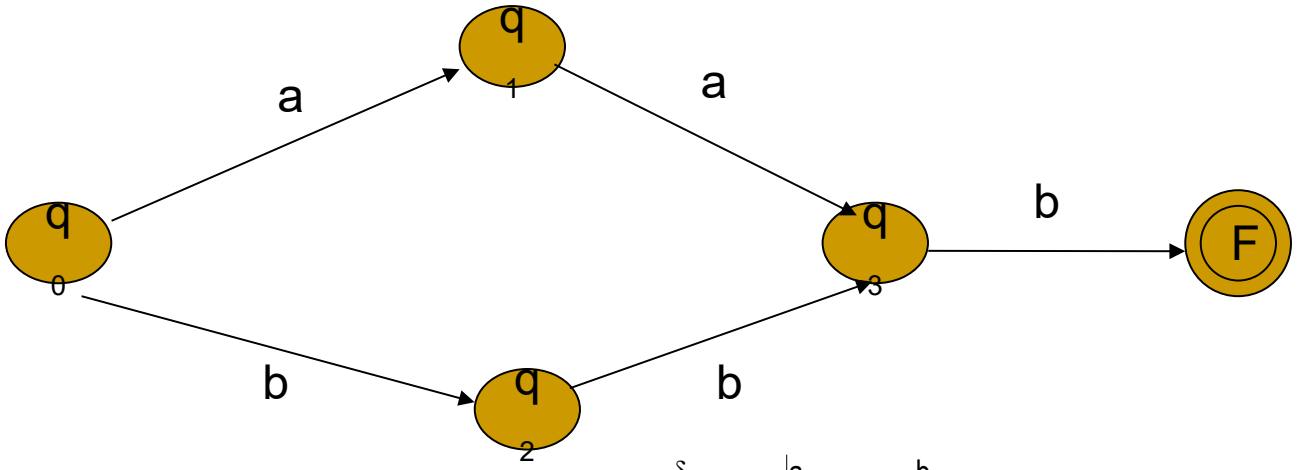
Un automate fini déterministe M est un quintuplet (Σ,Q,δ,q_0,F) où :

- Σ est un alphabet ;
- Q est un ensemble fini d'états ;
- $\delta:Q\times\Sigma \rightarrow Q$ est la fonction de transition ;
- q_0 est l'état initial ;
- F est un ensemble d'états finaux

On peut étendre δ sur $\delta:Q\times \Sigma^* \rightarrow Q$ par $\begin{cases} \delta(q,\varepsilon)=q \\ \delta(q,aw)=\delta(\delta(q,a),w) \end{cases}$

Le langage L(M) reconnu par l'automate M est l'ensemble $\{\omega|\delta(q_0,w)\in F\}$ des mots permettant d'atteindre un état final à partir de l'état initial.

Exemple



δ	a	b
q0	q1	q2
q1	q3	
q2		q3
q3		F

$Q=\{q_0,q_1,q_2,q_3,F\}$

$\Sigma=\{a,b\}$

$F=\{F\}$

L'automate reconnaît le langage $\{aab,bbb\}$

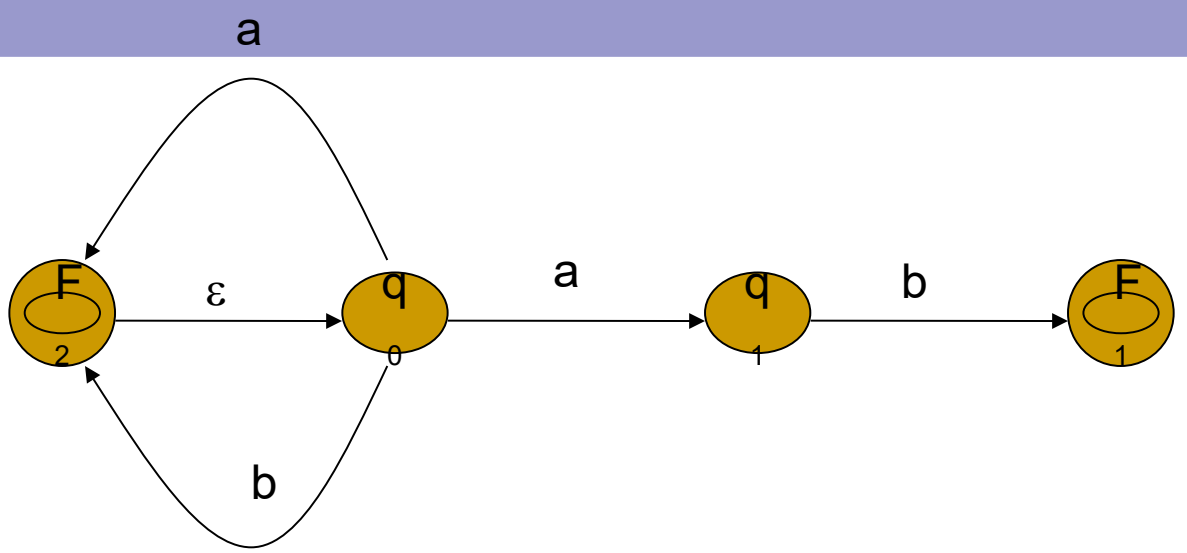
Traduction d'un AFD en code

Etat e = q_0 ;
char c = CarSuiv();
tant que c ≠ EOF faire
    e = Transiter(e,c);
    c = CarSuiv();
fin;
si e ∈ F alors
    retourner ok
sinon retourner non;

Automates finis non-déterministes

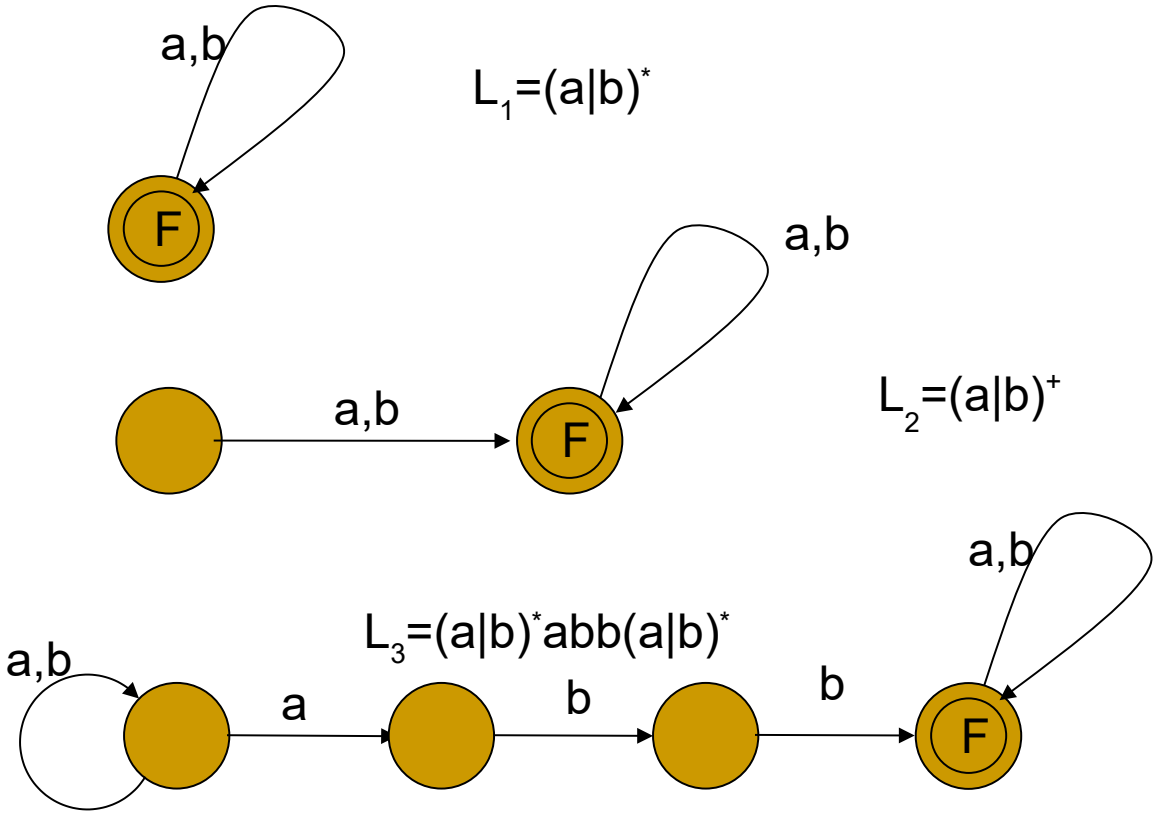
- La définition est la même que celle des automates déterministes, compte tenu des deux détails suivants :
- les transitions sont définies par une relation et non plus par une fonction, c'est-à-dire que plusieurs transitions issues d'un état donné peuvent porter la même étiquette
- il existe des transitions « spontanées » qui portent une étiquette spéciale, classiquement ε

Exemple



L'automate reconnaît le langage des mots d'au moins une lettre formés avec **a** et **b**. On note que le mot **ab** peut être reconnu de deux façons différentes

Exemples



Définition formelle des AFND

La définition est la même que pour les AFD, excepté δ : Qx(ΣU{ε})➔2^Q

On étend δ sur QxΣ* ➔ Q par :

$$\begin{cases} q \in \delta(q,\varepsilon) \\ \delta(q,aw) = \quad q' \in \delta(q,a) \in \delta(q',w) \end{cases}$$

Le langage L(M) reconnu par un automate non déterministe est : $\{w \mid \delta(q_0,w) \cap F \neq 0\}$

Définition formelle - Précisions

- Un AFND reconnaît un mot x ssi il existe un chemin dans le graphe de transitions entre l'état initial et un état d'acceptation tel que les étiquettes le long de ce chemin épellent le mot x.
- Le langage défini par un AFND est l'ensemble des mots qu'il accepte
- Propriété : tout langage reconnu par un AFND peut être décrit par une E.R. et réciproquement

Intérêt des AFND ?

L'intérêt des automates finis non-déterministes est qu'il est facile d'associer un automate (Q,δ,s,F) à une expression régulière M

Construction de *Thompson*

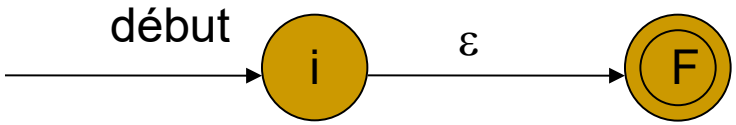
Donnée : une expression régulière *r*

Résultat : une AFND qui reconnaît *L(r)*

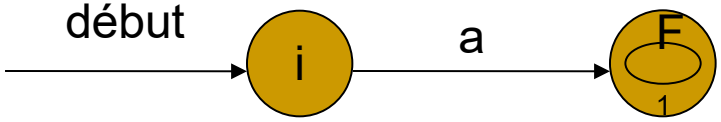
Méthode de Thompson

- Pour ε, construire l'AFND :

cet automate reconnaît ε



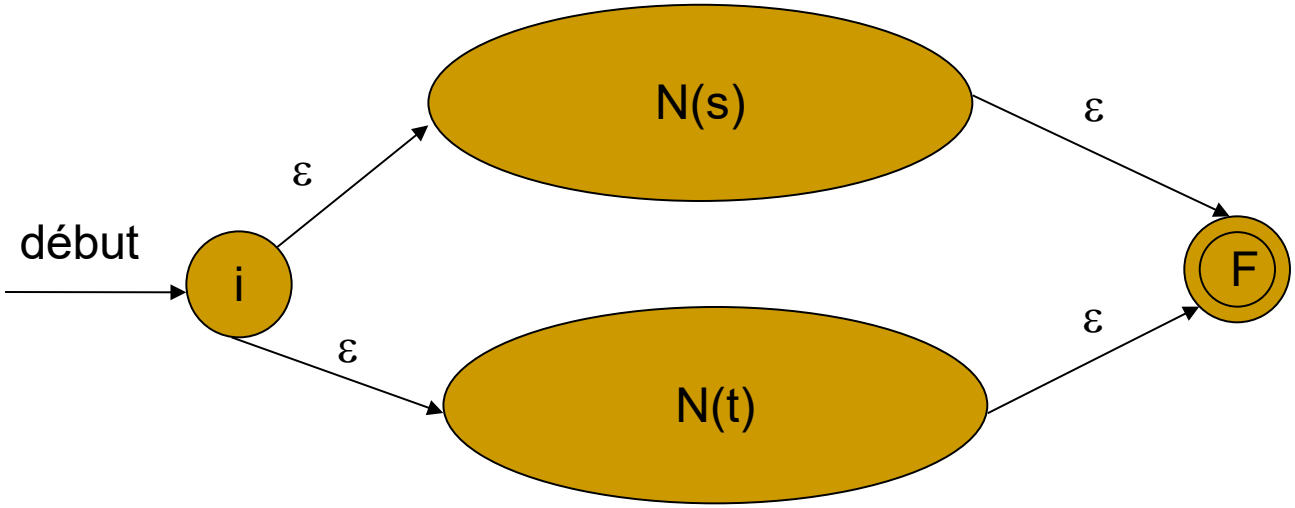
- Pour a appartenant à Σ, construire l'AFND :



cet automate reconnaît {a}

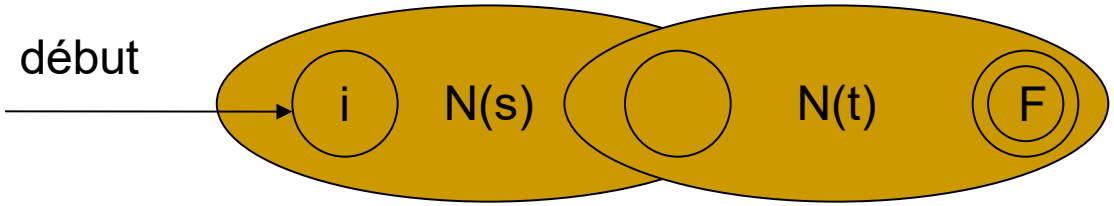
Méthode de Thompson (suite)

- Supposons que N(s) et N(t) soient les AFND pour les expressions régulières s et t, l'automate suivant reconnaît s|t



Méthode de Thompson (suite)

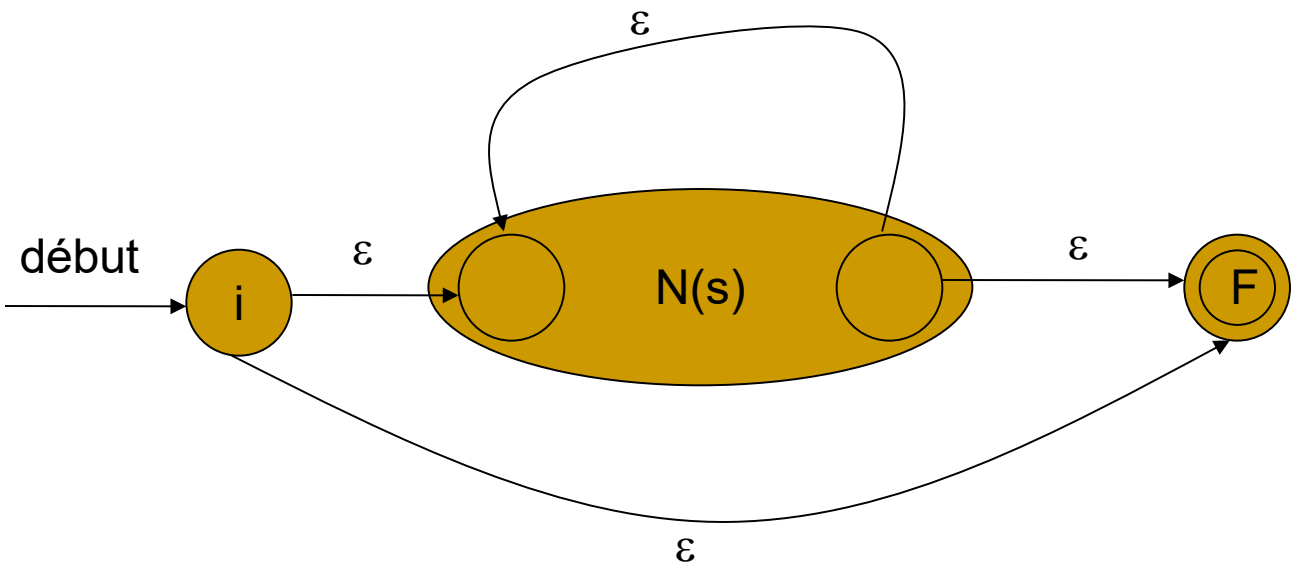
- l'automate suivant reconnaît st



L'état de départ de N(s) devient l'état de départ de l'AFND composé

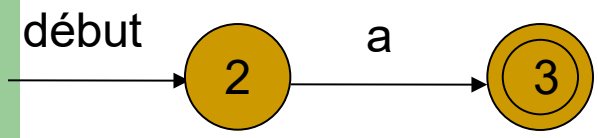
Méthode de Thompson (suite)

- l'automate suivant reconnaît, l'expression régulière s*

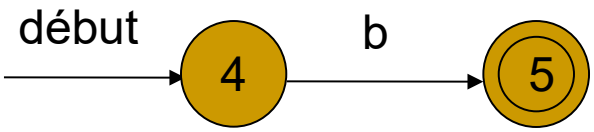


Exemple r=(a|b)*abb

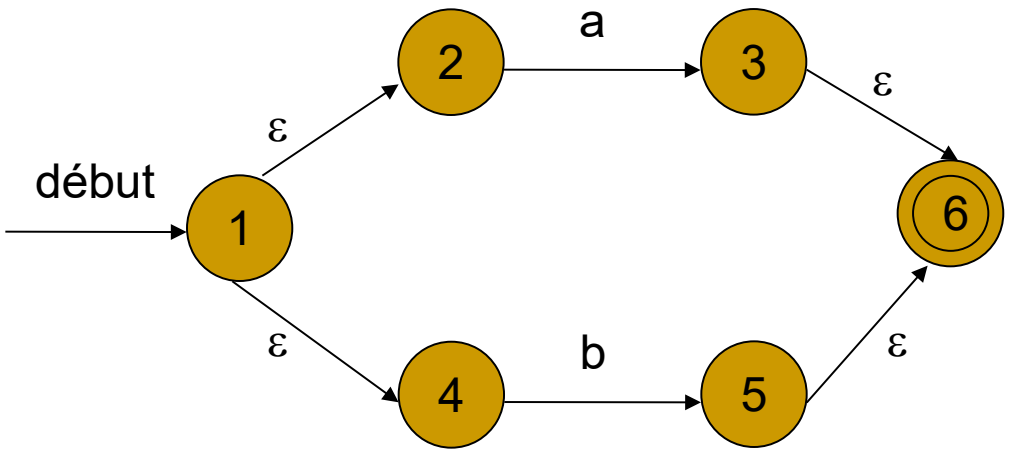
pour le premier a



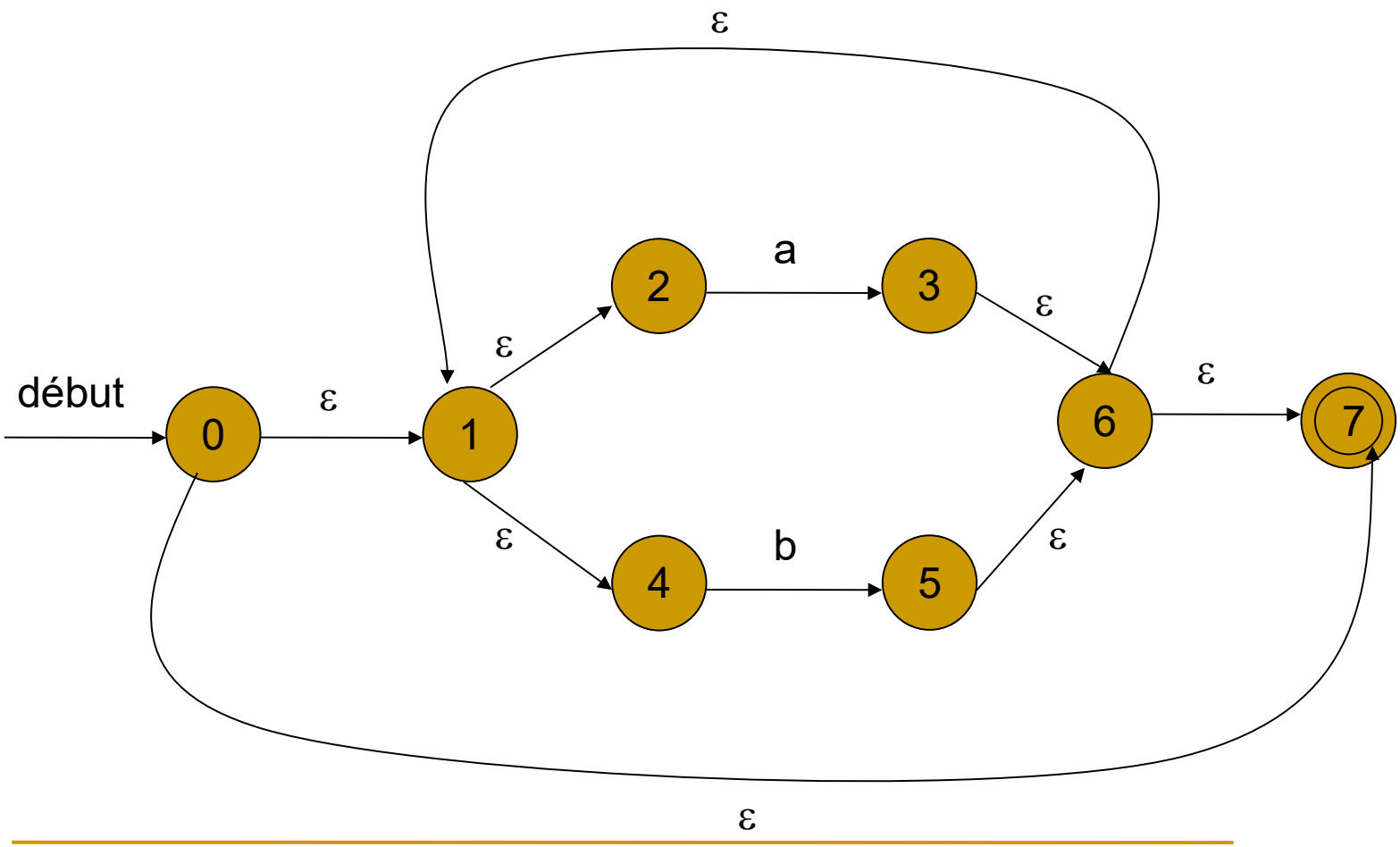
pour b



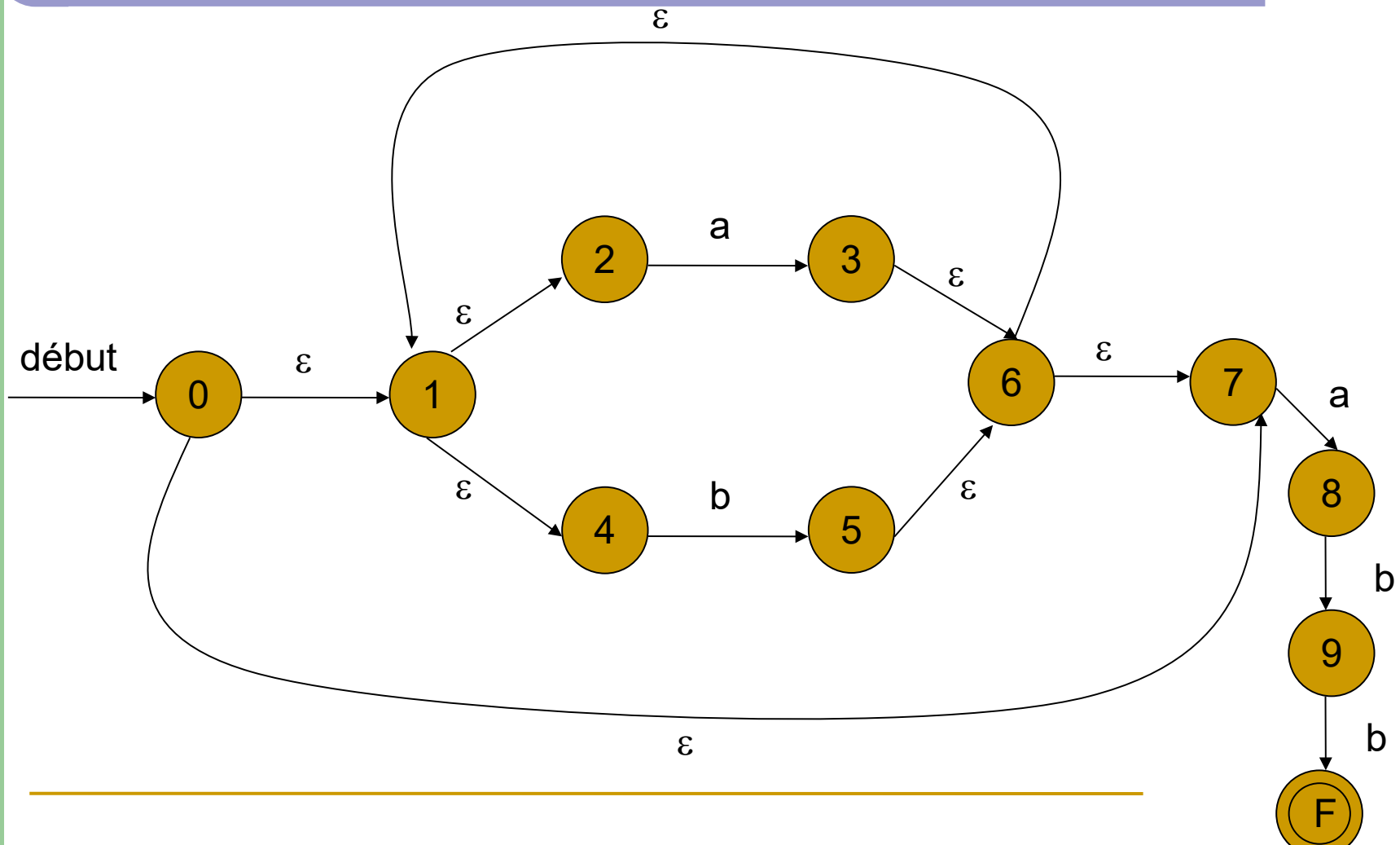
a|b



(a|b)*



(a|b)*abb



Comparaisons AFD - AFND

- AFD : il est très facile de déterminer si un mot est reconnu par un AFD ➔ pas de choix multiples au départ d'un état
- AFND : l'AFD contient plus d'états (ou plus de transitions) que l'AFND correspondant

Un AFD est minimal s'il n'existe pas d'AFD reconnaissant le même langage avec un nombre d'états inférieur.

Résultat : tout langage régulier est reconnu par un AFD minimal qui est unique

Déterminisation d'un automate

L'automate déterministe associé à Q a $2^{|Q|}$ états, mais en général, seulement les états atteignables depuis $\{q_0\}$ nous intéressent.

Note : en pratique, cet ensemble est relativement petit, souvent de l'ordre |Q|

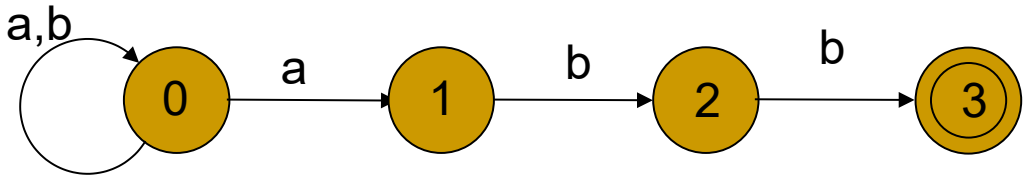
Déterminisation d'un automate

A partir d'un état et en lisant une lettre, la table de transition d'un AFND nous donne l'ensemble des états accessibles.

On identifie cet ensemble à un état de l'AFD associé et on construit ainsi de proche en proche l'AFD.

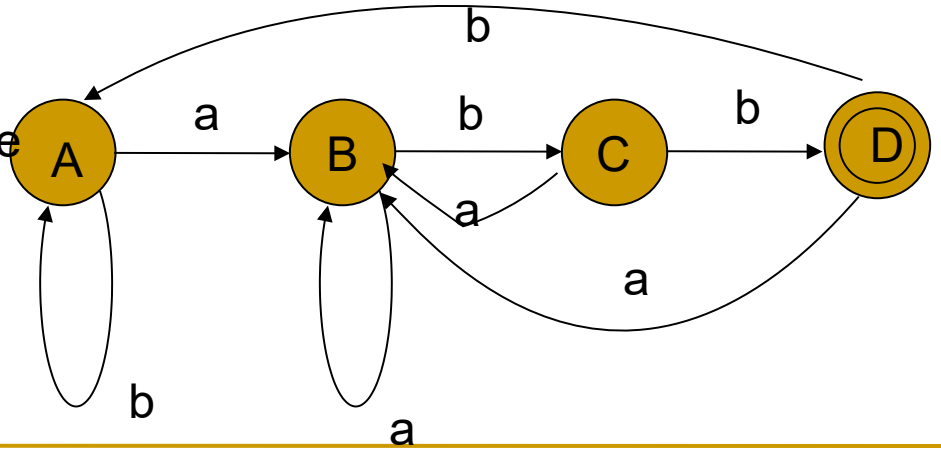
Exemple

L=(a|b)*abb



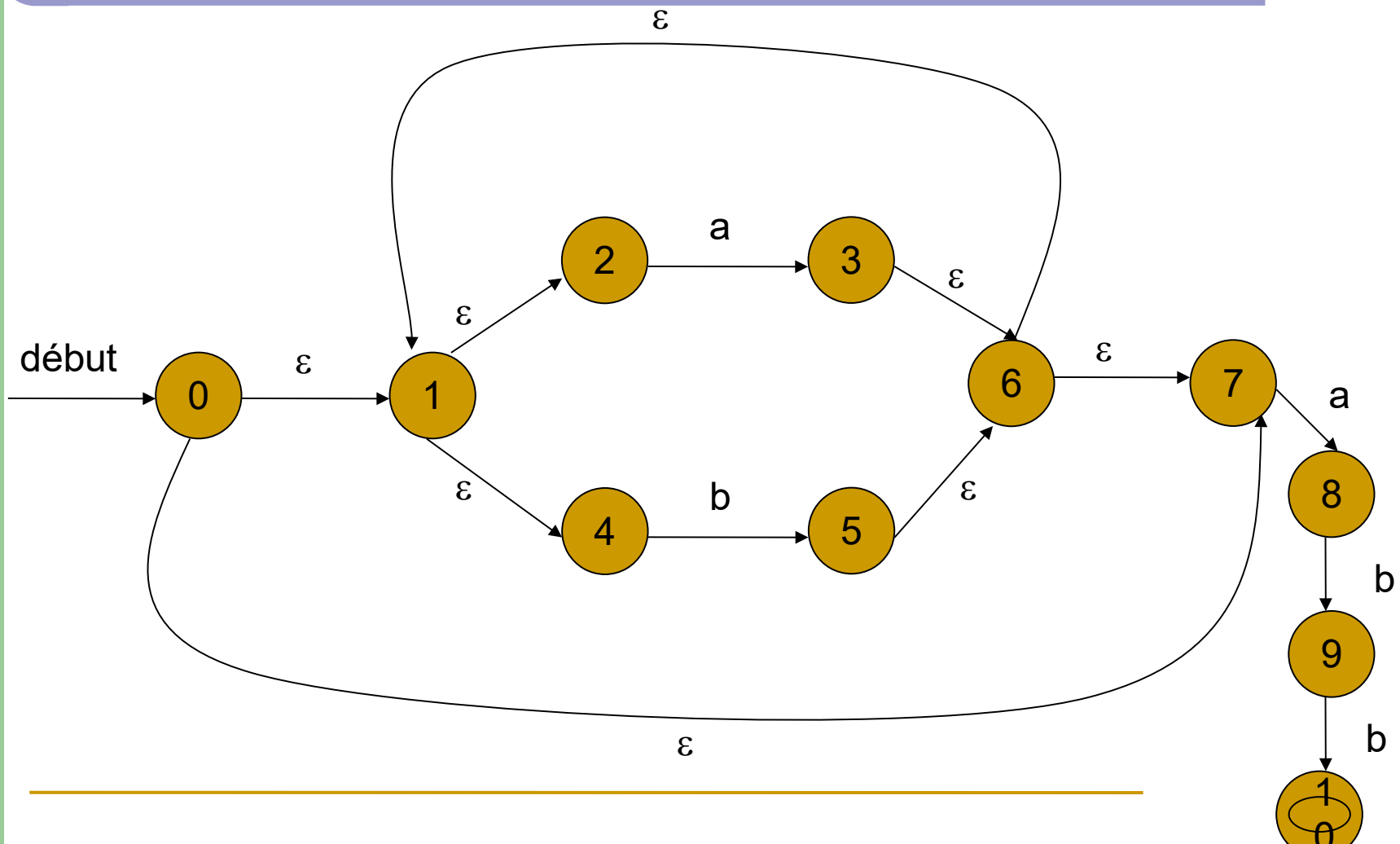
	a	b
A = 0	01	0
B = 01	01	02
C = 02	01	03
D = 03	01	0

L'état initial est l'ensemble des états de départ de l'AFND



Les états finaux sont ceux qui contiennent au moins un état final de l'AFND

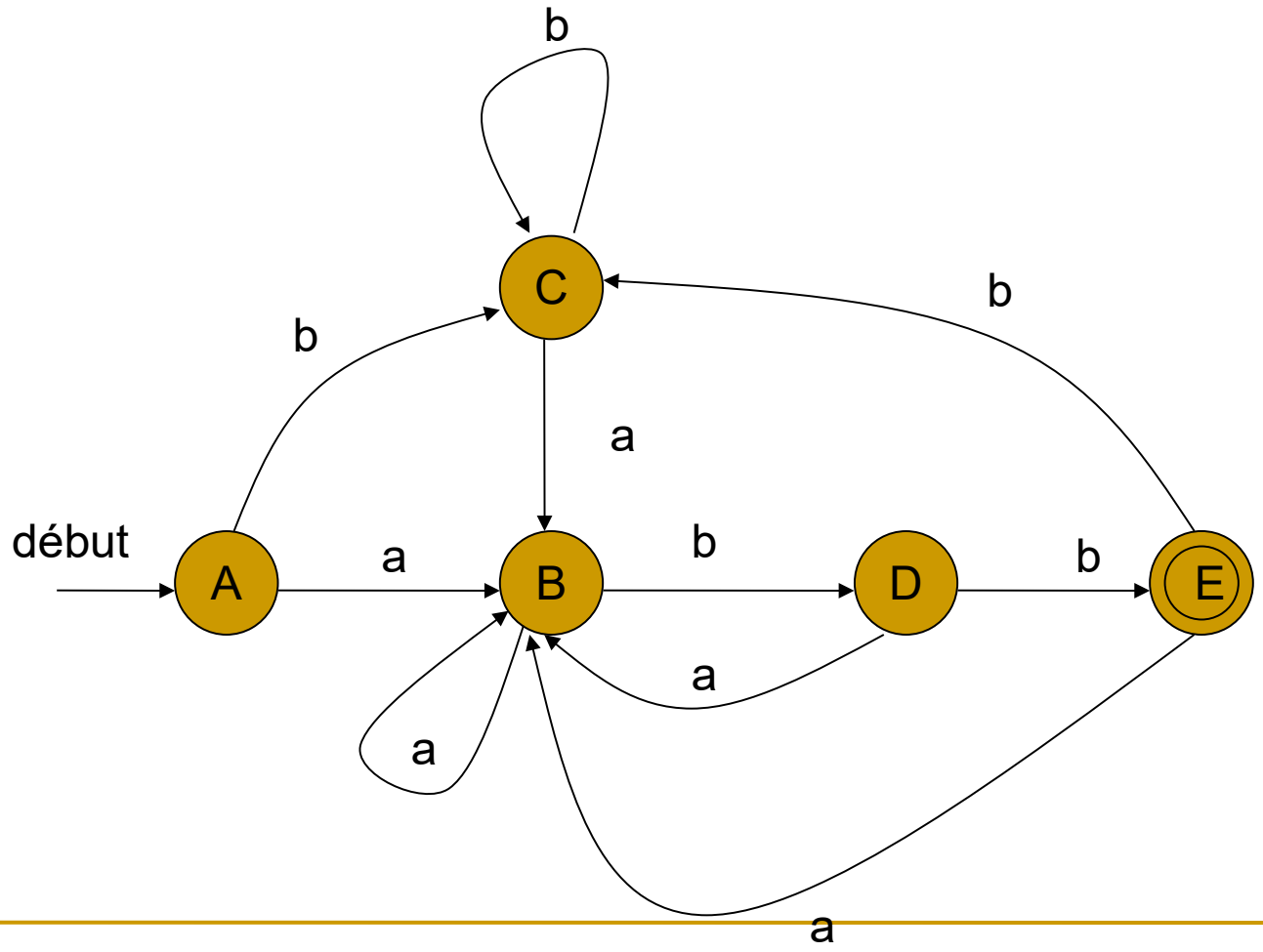
(a|b)*abb



Déterminisation

Etat Initial		a	b
A	01247	3671248	567124
B	1234678	3671248	5671249
C	124567	3671248	567124
D	1245679	3671248	567124 10
E	124567 10	3671248	567124

Exemple



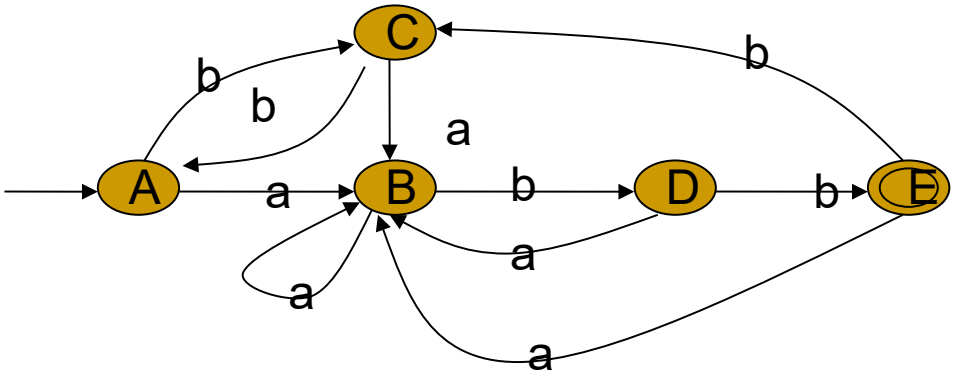
Minimisation d'un AFD

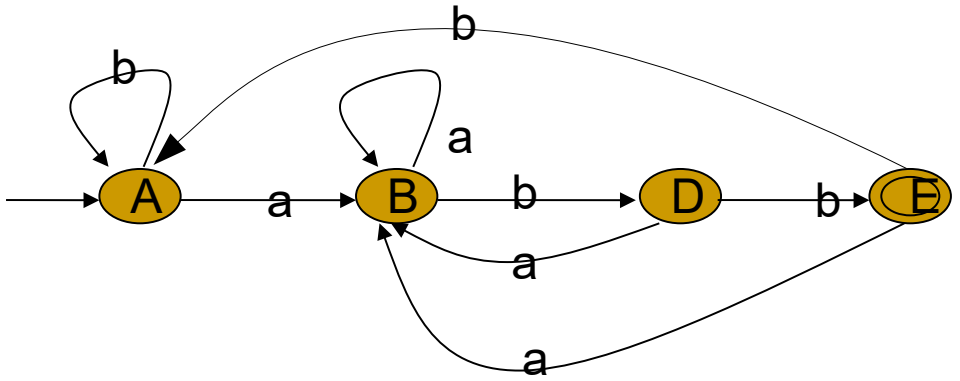
1. Construire une partition initiale π de l'ensemble des états en 2 groupes : les états finaux et les autres
2. Pour chaque groupe G de π : partitionner G en sous-groupes tels que tous les états de ce sous-groupe aient le même comportement (introduire un état « puits » si nécessaire ($\pi_n = \pi_{n-1}$)
3. On choisit un représentant pour chaque sous-groupe qui sera l'état de l'AFD minimal

Minimisation d'un automate

- L'état de départ est le représentant du groupe contenant l'état initial de l'AFD initial
- Les états d'acceptation sont les représentants des groupes formés d'états d'acceptation

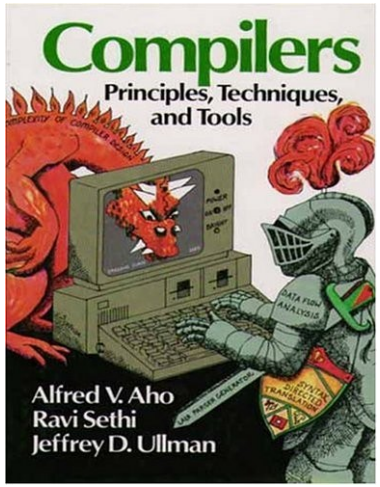
		a	b
NF	A	NF	NF
	B	NF	NF
	C	NF	NF
	D	NF	F
F	E	NF	NF
NF1	A	NF1	NF1
	B	NF1	NF2
	C	NF1	NF1
NF2	D	NF1	F
F	E	NF1	NF1
NF1	A	NF2	NF1
	C	NF2	NF1
NF2	B	NF2	NF3
NF3	D	NF2	F
F	E	NF2	NF1



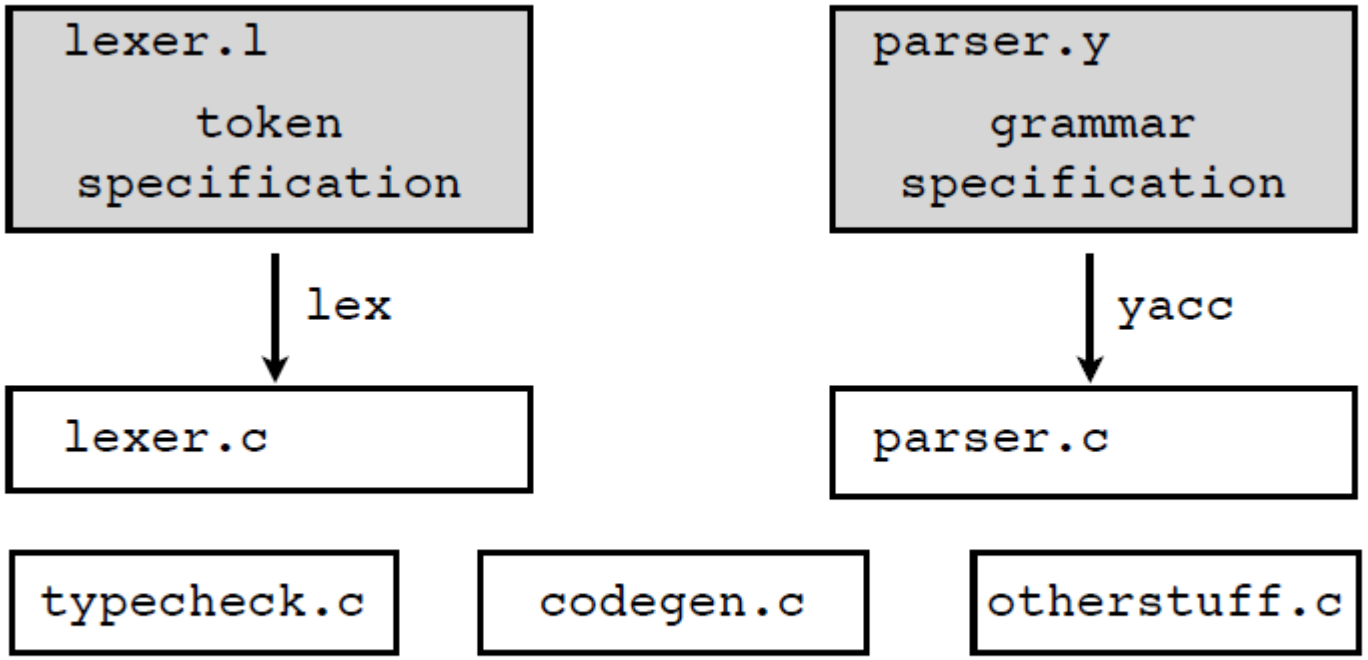


Lexx et Yacc

- Outils pour écrire des parsers
- Lex analyse lexicale
- Yacc (Yet Another Compiler compiler) → parser
- Outils anciens
  - Lex ~ 1975 (Mike Lesk, Eric Schmidt)
  - Yacc ~1970 (Stephen Johnson)
- Beaucoup, beaucoup de littérature
- Mais produit du C



Principe


lexer.l
token specification
parser.y
grammar specification
lex
yacc
lexer.c
parser.c
typecheck.c
codegen.c
otherstuff.c

Python PLY

- PLY = Python Lex Yacc
- version python de Lex et Yacc
- fonctionnalités équivalentes à celles de Lex et Yacc
- mais plus simple à manipuler que Lex et Yacc

Le package PLY

- Deux modules :
  - ply.lex
  - ply.yacc
- Il « suffit » d'installer le package (avec pip par exemple) et importer le module avec lequel on souhaite travailler
- PLY ne génère pas de code

ply.lex

- C'est le module pour écrire l'analyseur lexical
- Les tokens sont spécifiés en utilisant des expressions régulières
- Assez facile à utiliser

Exemple

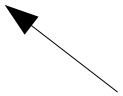
```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```



token

Exemple

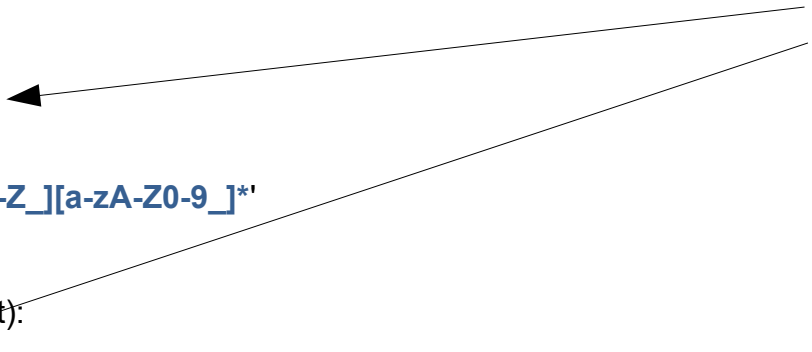
```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```



chaque token a une ER qui lui Correspond

Exemple

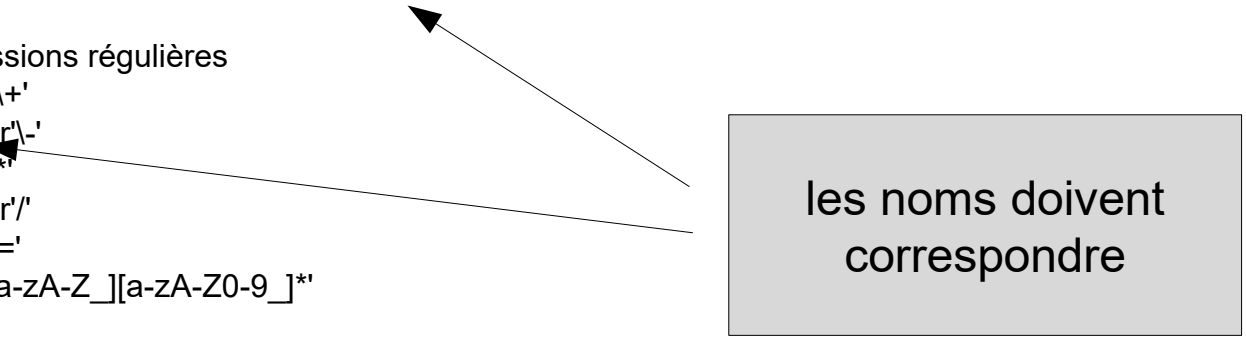
```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```



Exemple

```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```

pour des tokens simples, une chaîne

Exemple

```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```

une fonction quand une opération doit être effectuée

Exemple

```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```

Dans ce cas, c'est la docstring python qui stocke l'ER

Exemple

```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_ignore = ' \t'
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```

on rajoute souvent cette ER pour spécifier les caractères entre tokens

Exemple

```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_ignore = ' \t'
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```

Construit l'AL en construisant un super ER contenant l'ensemble des ER

Exemple

```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_ignore = ' \t'
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'


def t_NOMBRE(t):
    r'\d+'
    t.value = int(t.value)
    return t


lexer = lex.lex()
```

C'est l'introspection qui associe le token à son ER

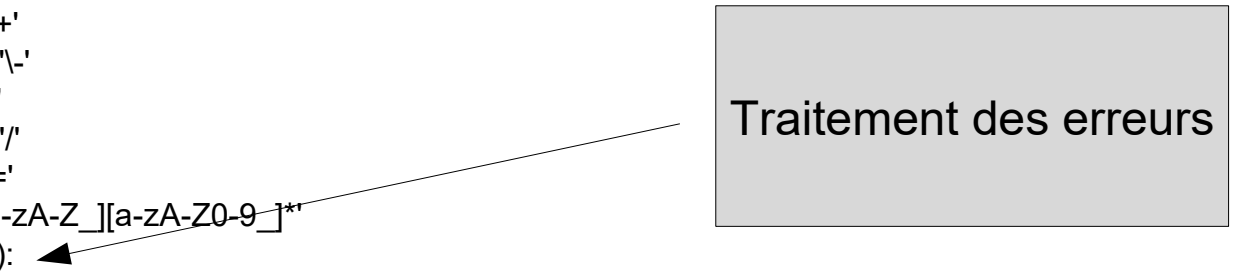
Exemple

```
import ply.lex as lex


# Liste des token
tokens = ('NOM','NOMBRE', 'PLUS','MOINS','FOIS','DIVISE','EGAL')

# les expressions régulières
t_ignore = ' \t'
t_PLUS = r'\+'
t_MOINS = r'\-'
t_FOIS = r'\*'
t_DIVISE = r'/'
t_EGAL = r'='
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)


def t_NOMBRE(t):
   r'\d+'
   t.value = int(t.value)
   return t
```



```
lexer = lex.lex()
```

Exemple

- 2 fonctions disponibles : input() et token()

```
lexer.input("x = 3 * 4 + 5 * 6")
    while True:
        tok = lexer.token()
        if not tok:
            break
        print(tok)
```

fournit le texte à analyser par l'AL

Exemple

- 2 fonctions disponibles : input() et token()

```
lexer.input("x = 3 * 4 + 5 * 6")
    while True:
        tok = lexer.token()
        if not tok:
            break
        print(tok)
```

fournit le prochain token ou None

Exemple

- 2 fonctions disponibles : input() et token()

```
lexer.input("x = 3 * 4 + 5 * 6")
while True:
    tok = lexer.token()
    if not tok:
        break
    print(tok)
```

4 attributs
tok.type
tok.value
tok.lineno
tok.lexpos

t_*NOM* = r'[a-zA-Z_][a-zA-Z0-9_]*'

Exemple

- 2 fonctions disponibles : input() et token()

```
lexer.input("x = 3 * 4 + 5 * 6")
    while True:
        tok = lexer.token()
        if not tok:
            break
        print(tok)
```

4 attributs
tok.type
tok.value
tok.lineno
tok.lexpos

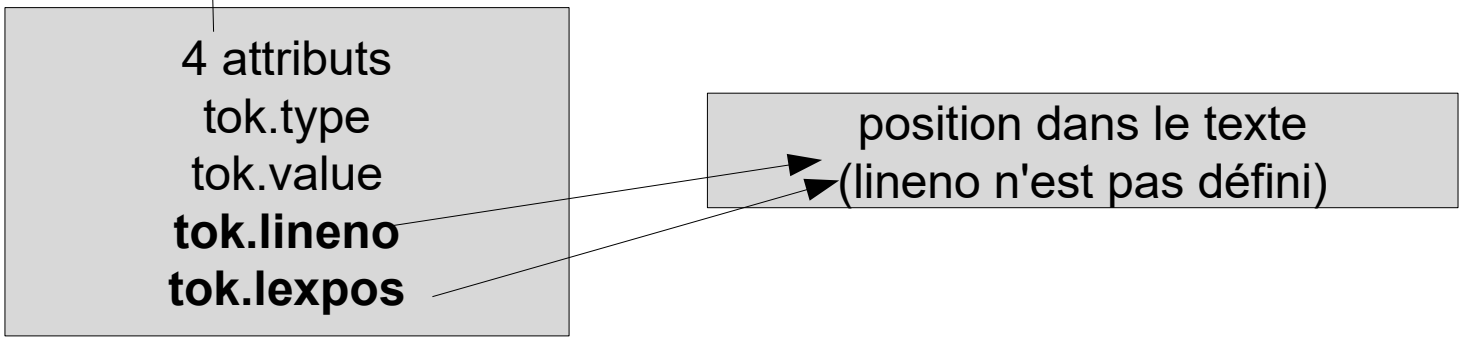
Le texte qui correspond à l'ER

```
t_NOM = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

Exemple

- 2 fonctions disponibles : input() et token()

```
lexer.input("x = 3 * 4 + 5 * 6")
while True:
    tok = lexer.token()
    if not tok:
        break
    print(tok)
```



Analyse syntaxique

Grammaires