Théorie des Langages et Compilation (TLC)

Fabrice Lamarche

fabrice.lamarche@irisa.fr

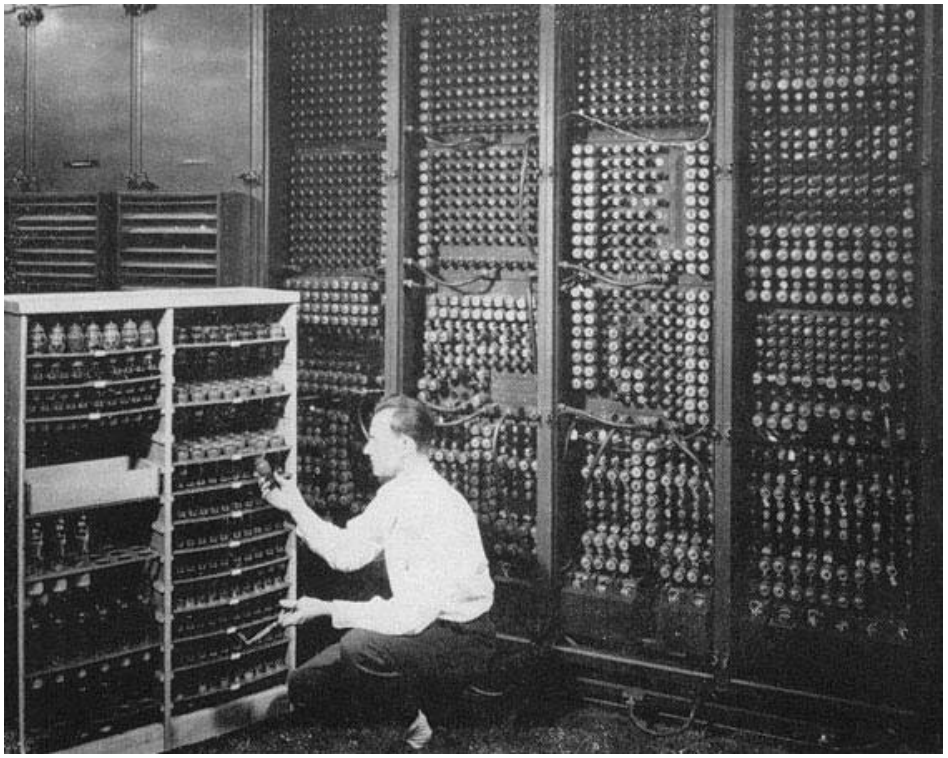
ESIR

Historique

- 1946 : Création de l'ENIAC (Electronic Numerical Integrator and Computer)
  - P. Eckert et J. Marchly

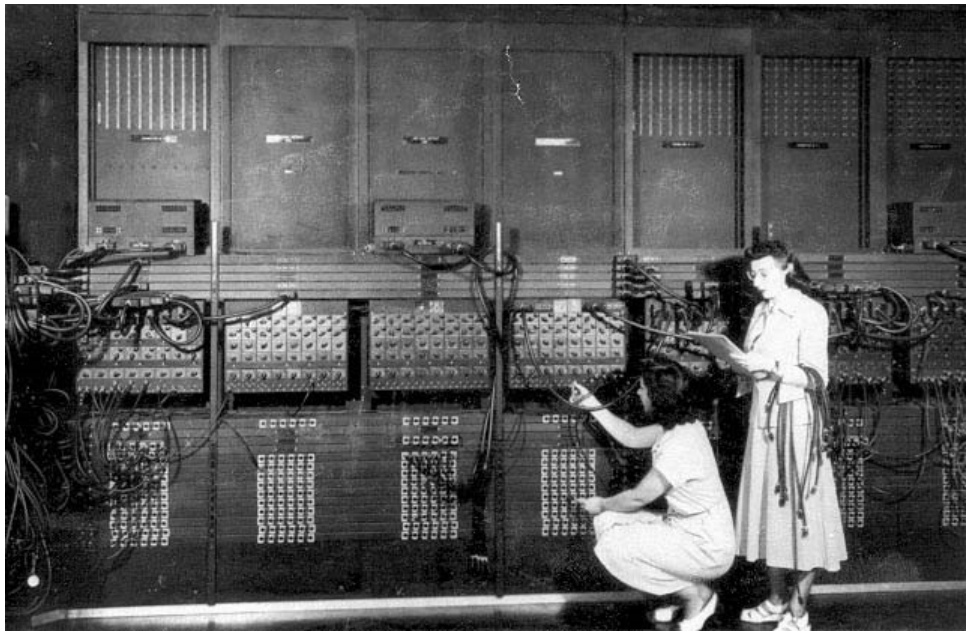
- Quelques chiffres
  - 17468 tubes à vide
  - 7200 diodes
  - 1500 relais
  - 70000 résistances
  - 10000 condensateurs
  - 30 tonnes
  - Occupe une surface de 67m^2
  - Consomme 150 Kilowatts

- Performances
  - Horloge à 100KHz
  - 5000 additions / seconde
  - 330 multiplications / seconde
  - 38 divisions / seconde



Historique

- ENIAC
  - 30 Unités autonomes
    - 20 accumulateurs 10 digits
    - 1 multiplicateur
    - 1 Master program capable de gérer des boucles
  - Mode de programmation
    - Switchs
    - Câblage des unités entre elles ☺

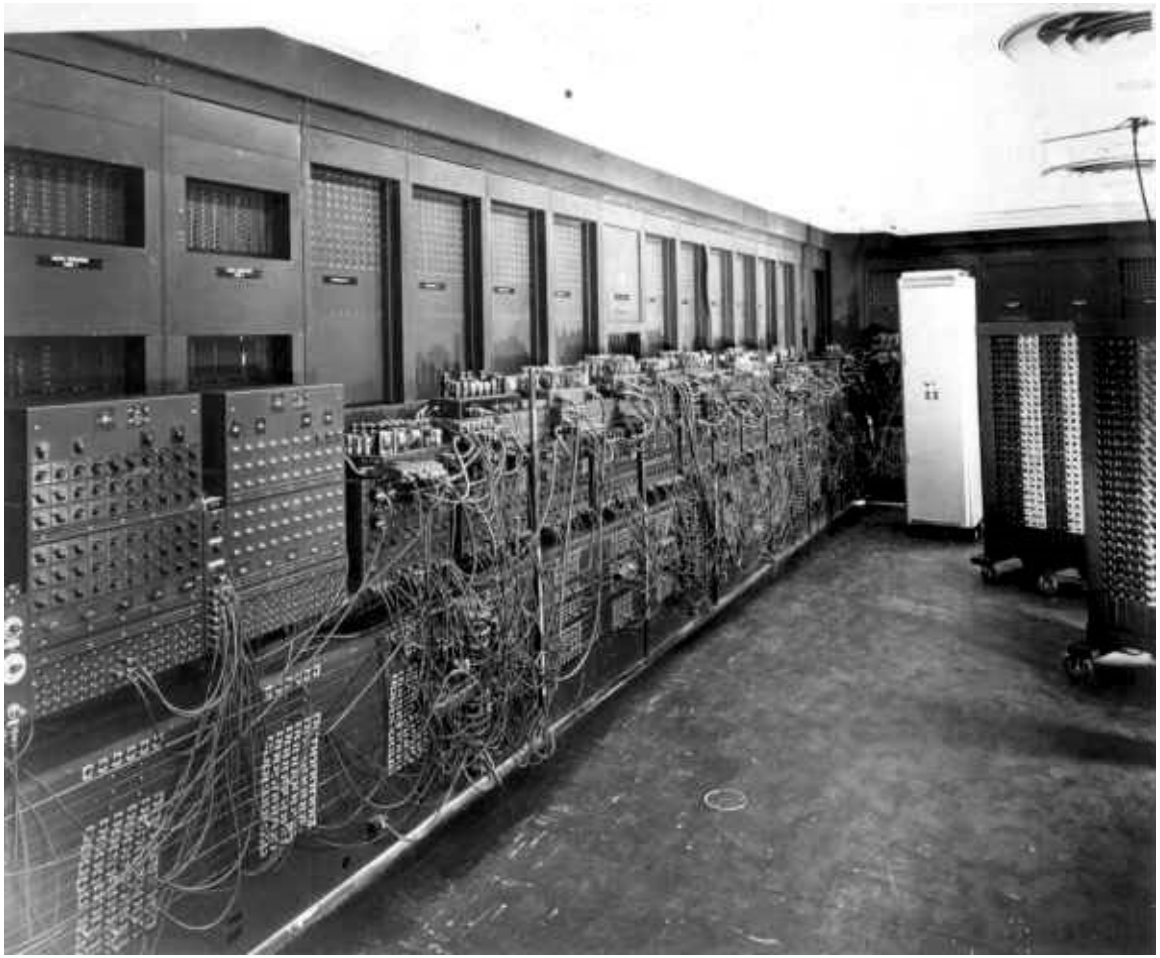


Historique

- Ceci est un programme sur l'ENIAC…

Une cause fréquente de panne était la combustion d'un insecte sur un tube chaud.

Naissance du mot BUG ☺

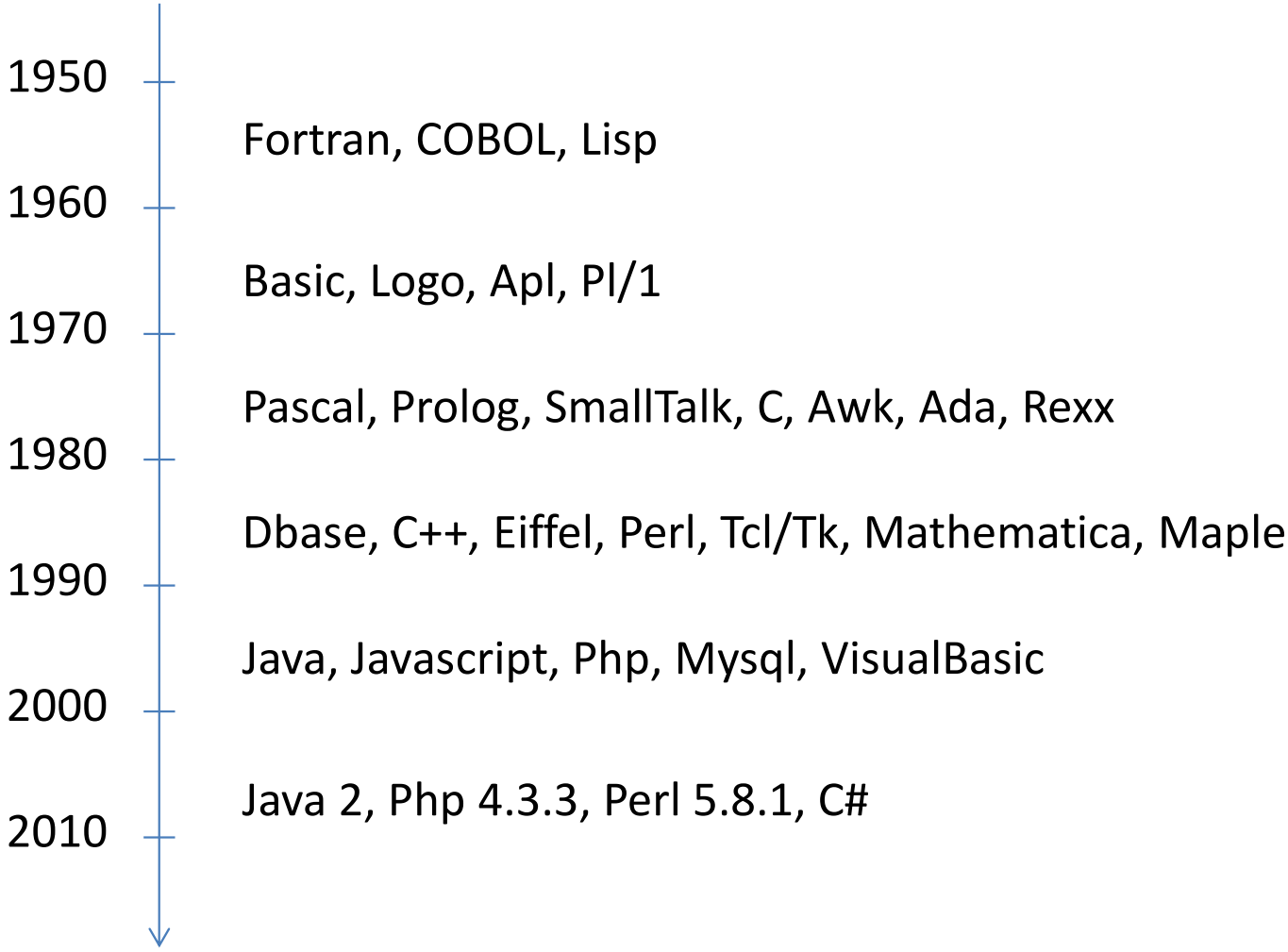


Historique

- 1950 : Invention de l'assembleur par Maurice V. Wilkes de l'université de Cambridge
  - Avant : programmation en binaire
- 1951 : Grace Hopper crée le premier compilateur pour UNIVAC 1 : le A-O system.
  - Permet de générer un programme binaire à partir d'un « code source »
- 1957 : Grace Hopper travaille chez IBM
  - défend l'idée qu'un programme devrait pouvoir être écrit dans un langage proche de l'Anglais
- 1959 : Grace Hopper crée le langage COBOL

"It's better to ask forgiveness than it is to get permission." -- the late Rear Admiral Grace Hopper, who led the effort to create COBOL.

Historique


1950
Fortran, COBOL, Lisp
1960
Basic, Logo, Apl, Pl/1
1970
Pascal, Prolog, SmallTalk, C, Awk, Ada, Rexx
1980
Dbase, C++, Eiffel, Perl, Tcl/Tk, Mathematica, Maple
1990
Java, Javascript, Php, Mysql, VisualBasic
2000
Java 2, Php 4.3.3, Perl 5.8.1, C#
2010

Les langages de programmation

- Langages de programmation compilés
  - Génération de code exécutable i.e. en langage machine
  - Ex : C / C++
- Les langages de programmation interprétés
  - Un langage interprété est converti en instructions exécutables par la machine au moment de son exécution
  - Ex : PHP, Javascript, Ruby
- Les langages P-code
  - Langages à mi-chemin entre l'interprétation et la compilation
  - Java
    - La phase de compilation génère du « byte code »
    - Le « byte code » est interprété par une machine virtuelle lors de l 'exécution
  - Exécution plus rapide que les langages interprétés
  - Exécution plus lente que les langages compilés
  - Machine virtuelle pour chaque processeur => portabilité du code compilé

Catégories de langages de programmation

- Programmation impérative
  - la **programmation impérative** est un paradigme de programmation qui décrit les opérations en termes de séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.
  - Programmation procédurale
    - Paradigme de programmation basé sur le concept d'appel procédural. Une procédure, aussi appelée *routine*, *sous-routine* ou *fonction* contient simplement une série d'étapes à réaliser. N'importe quelle procédure peut être appelée à n'importe quelle étape de l'exécution du programme, incluant d'autres procédures voire la procédure elle-même
  - Programmation objet
    - Paradigme de programmation qui consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique. Il possède une structure interne, un comportement et sait communiquer avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; la communication entre les objets via leur relation permet de réaliser les fonctionnalités attendues, de résoudre le ou les problèmes.

Catégories de langages de programmation

- Programmation fonctionnelle
  - La **programmation fonctionnelle** est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques et rejette le changement d'état et la mutation des données. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état.
  - Ex: CAML

- Programmation logique
  - La **programmation logique** est une forme de programmation qui définit les applications à l'aide d'un ensemble de faits élémentaires les concernant et de règles de logique leur associant des conséquences plus ou moins directes. Ces faits et ces règles sont exploités par un démonstrateur de théorème ou moteur d'inférence, en réaction à une question ou requête.
  - Ex : PROLOG

Pourquoi TLC ?

- Culture de l'ingénieur
  - Vous utilisez des compilateurs, des interpréteurs etc…
    - Savez vous comment ils fonctionnent ?
  - Qu'est-ce qu'un langage ?
  - Comment le définir ?
  - Comment le reconnaitre ?
  - Comment l'interpréter, le compiler ?

- Les concepts sous tendant la compilation sont utilisés partout
  - Les formats de fichiers définissent une structuration de l'information
    - Ils ont donc un langage associé…
  - Les gros logiciels offrent des langages de script parfois dédiés
  - Et bien d'autres…

- Certaines descriptions dans les langages standards peuvent être longues et fastidieuses
    - Création d'un compilateur : langage dédié, simple (Domain Specific Language: DSL)
    - Le compilateur s'occupe de la partie fastidieuse et systématique

Logiciels et langages de script

- Les gros logiciels offrent souvent des langages de script
  - Ouverture du logiciel vers les utilisateurs
    - Ajout de nouvelles fonctionnalités
    - Automatisation de tâches
  - Proposition de langages pertinents par rapport au domaine applicatif
    - Souvent plus simples que Java / C++
    - Pas forcément besoin d'être informaticien pour l'utiliser
  - Fermeture de l'API du programme
    - Parfait contrôle des fonctionnalités mises à disposition
    - Contrôle de l'utilisation des fonctionnalités

THÉORIE DES LANGAGES

Motivation historique

- Reconnaissance *automatique* de structure dans des *séquences finies*
  - Structure : loi, régularité
  - Séquence : agencement spatial ou temporel
    - Avant / après, gauche / droite, dessus / dessous…

➢ *Il faut s'assurer que le reconnaisseur est correct !*

Exemples

- Télécommunications : séquences de signaux (Shanon 1916-2001)
- Langue naturelle : séquences de mots (Chomsky 1928-…)
- Formats de fichiers : séquences de codes

Exemples

- Protocoles :
  - Séquences de symboles formant des messages
  - Séquences de messages
  - ➢ *2 niveaux d'analyse*

- Langages de programmation
  - Séquences de caractères pour former des mots
  - Séquences de mots
  - ➢ *2 niveaux d'analyse*

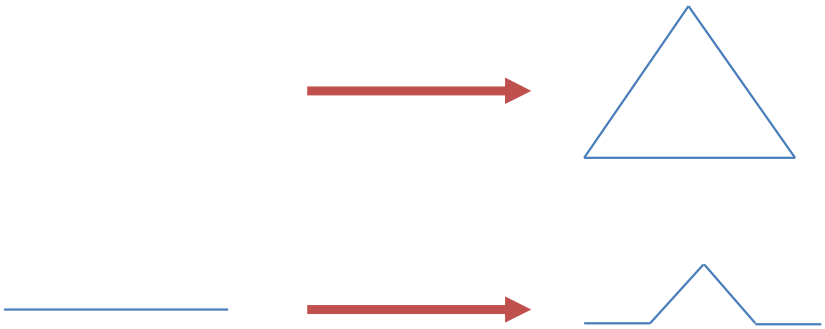
Exemple : le morse

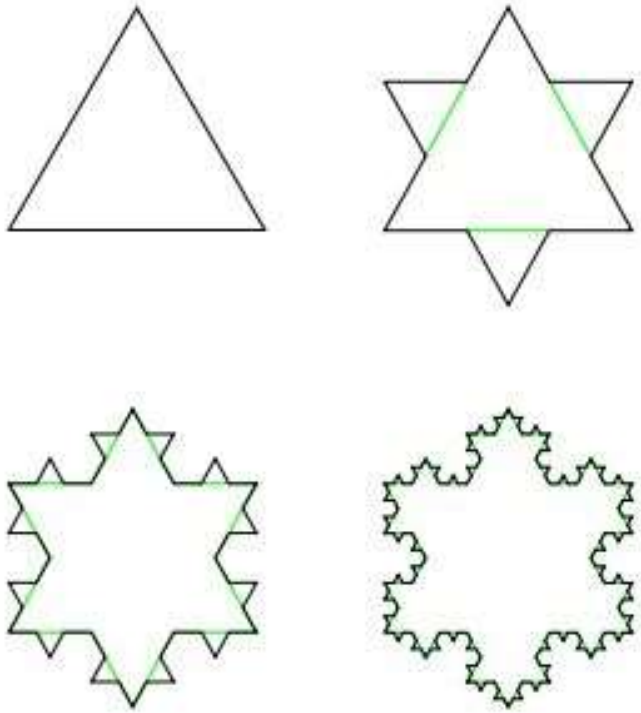
- Suite de traits et de points forme des lettres
- Les silences séparent les lettres et les mots
  - Jamais deux silences à suivre

Exemple : grammaire de formes

- Description de la structure d'une forme géométrique

- Le « flocon de Koch »





Théorie des langages

- Etudie les moyens de caractériser des langages
  - Ces moyens sont eux-mêmes des langages
  - Notion de ***métalangage***
- Théorie très outillée car très formalisée
- Champ d'application immense
  - Texte, séquences d'événements, chimie, musique…

Applications (1)

- Spécification syntaxique
  - Le langage est le plus souvent infini
    - Point de vue extensionnel
  - Trouver une description formelle finie
    - Point de vue intentionnel
    - *description* => $\mathcal{L}$(*description*)
    - $\mathcal{L}$(description) = langage engendré

- Difficultés
  - Sur-générer : $\mathcal{L}(description) \supsetneq langage$
  - Sous-générer :$\mathcal{L}(description) \subsetneq langage$

Applications (2)

- Vérification syntaxique pour la conformité
  - Langages de programmation, protocoles réseau, format de fichier…

- La vérification doit être **automatique**
  - ➢ Il faut un algorithme
  - Calculabilité de $document \in \mathcal{L}(description)$?
  - Complexité de $document \in \mathcal{L}(description)$?
  - Question qui se pose par exemple pour la langue naturelle…

Applications (3)

- Analyse syntaxique
  - Recherche du sens (sémantique)

- Difficultés
  - Ambiguïté syntaxique
    - Jean a vu Pierre avec ses lunettes
    - « ses » réfère à Jean ou à Pierre ?
  - Ambiguïté sémantique
    - Cet avocat véreux est dégoutant
    - Quel « avocat », le fruit ou l'homme de loi ?

Notions de base

Lettres, mots, langages

Lettre

- Utilisation d'un alphabet Σ (ou V)
  - Ensemble fini de **lettres** ou de **symboles**
  - On parle aussi de **vocabulaire**

- Symboles génériques
  - Chiffre, ponctuation, majuscule….
  - Correspond à des **classes** de symboles

Mot

- Un mot *m* (ou *w*)
  - Suite finie de symboles appartenant à Σ
  - On note ε le mot vide

- Σ^* est l'ensemble de tous les mots

Langage

- Un langage $\mathcal{L}$ est un ensemble de mots (possiblement vide ou infini)
- $\emptyset$ est le langage vide
- Σ^* est le plus gros langage sur Σ
  - Ensemble infini de tous les mots finis que l'on peut former sur Σ
- $\mathcal{P}(\Sigma*)$ est l'ensemble infini de tous les langages sur Σ
  - Rq : $\mathcal{L} \in \mathcal{P}(\Sigma^*) \Leftrightarrow \mathcal{L} \subseteq \Sigma^*$

Attention

- $\emptyset$: le langage vide
  - Aucun mot
- ε : le mot vide
  - Un mot, aucune lettre
- $\{\varepsilon\}$: le langage du mot vide
  - Un seul mot, le mot vide

Opérations sur les mots

- Concaténation (noté $\cdot$)

$$\cdot \quad : \quad \Sigma* \times \Sigma* \rightarrow \Sigma*$$

$$m_1, m_2 \mapsto m$$

  - Tel que m contient les lettres de m1 puis de m2
  - Ex : $aaa \cdot b = aaab$
  - On écrit aussi bien $m_1 \cdot m_2$ que $m_1 m_2$

Opérations sur les mots

- Propriétés
  - $m \cdot \varepsilon = \varepsilon \cdot m = m$
    - ε est un élément neutre
  - $m_1 \cdot (m_2 \cdot m_3) = (m_1 \cdot m_2) \cdot m_3$
    - La concaténation est associative
  - En général : $m_1 \cdot m_2 \neq m_2 \cdot m_1$
    - La concaténation n'est pas commutative

- $(\Sigma, \cdot)$ forme un monoïde libre
  - i.e. un groupe sans élément symétrique

Opérations sur les langages

- Somme de langages : $\mathcal{L}_1 + \mathcal{L}_2$ (ou $\mathcal{L}_1 \mid \mathcal{L}_2$)

$$+ \quad : \quad \mathcal{P}(\Sigma*) \times \mathcal{P}(\Sigma*) \to \mathcal{P}(\Sigma*)$$

$$\mathcal{L}_1, \mathcal{L}_2 \mapsto \{m \mid m \in \mathcal{L}_1 \vee m \in \mathcal{L}_2\}$$

Ex : $\{a, b, c\} + \{d, e\} = \{a, b, c, d, e\}$

- Propriétés
  - $\mathcal{L} + \emptyset = \emptyset + \mathcal{L} = \mathcal{L}$
  - $\mathcal{L} + \{\varepsilon\} = \{\varepsilon\} + \mathcal{L} \supseteq \mathcal{L}$

Opérations sur les langages

- Produit de langages : $\mathcal{L}_1 \cdot \mathcal{L}_2$ (ou $\mathcal{L}_1\ \mathcal{L}_2$)

  $\cdot$: $\mathcal{P}(\Sigma*) \times \mathcal{P}(\Sigma*) \to \mathcal{P}(\Sigma*)$

  $\mathcal{L}_1, \mathcal{L}_2 \mapsto \{m_1 \cdot m_2 | m_1 \in \mathcal{L}_1 \wedge m_2 \in \mathcal{L}_2\}$

  Ex : $\{\mathrm{a}, \mathrm{b}, \mathrm{c}\} \cdot \{1,2\} = \{a1, b1, c1, a2, b2, c2\}$

- Propriétés
  - $\mathcal{L} \cdot \emptyset = \emptyset \cdot \mathcal{L} = \emptyset$
  - $\mathcal{L} \cdot \{\varepsilon\} = \{\varepsilon\} \cdot \mathcal{L} = \mathcal{L}$

Opérations sur les langages

- Fermeture de Kleene : $\mathcal{L}^*$

$$^*: \quad \mathcal{P}(\Sigma*) \rightarrow \mathcal{P}(\Sigma*)$$

$$\mathcal{L} \mapsto \{\varepsilon\} \cup \mathcal{L} \cup \mathcal{L} \cdot \mathcal{L} \cup \mathcal{L} \cdot \mathcal{L} \cdot \mathcal{L} \cup \ldots$$

Ex : $\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aab, \ldots\}$

Conclusion

- Les mots sont des séquences de lettres
  - Concaténation
- Les langages sont des ensembles de mots
  - Opérations ensemblistes
  - Produit
  - Fermeture

Expressions régulières

Motivation

- Expression de langages en termes de langages plus simples
- Décrire un nouveau langage en combinant des langages existants
- Décrire des langages en utilisant des opérations sur les langages

Expressions régulières (RE)

- Formule désignant un ensemble de mots construits sur un alphabet ∑

- Construction à partir des lettres de l'alphabet ∑ et de trois opérations sur les mots
  - L'union notée R | S = {s | s ϵ R ou s ϵ S }
  - Le produit de deux ensembles R et S, notée R · S.
  - La fermeture de Kleene de R, notée $R^* \bigcup_{i=0}^{\infty} R^i$

Expression régulières (RE)

- Définition plus formelle
  - Soit ∑ un alphabet
  - Si $a \in \sum$, a est une RE
  - ε est une RE
  - $\emptyset$ est une RE
  - r^* est une RE si r est une RE
  - $r_1|r_2$ est une RE si r_1 et r_2 sont des RE
  - $r_1 \cdot r_2$ est une RE si r_1 et r_2 sont des RE

Expressions régulières

- Langage engendré et propriétés

- Soit $\mathcal{L}$(r) le langage engendré par la RE r
  - $\mathcal{L}$ est une fonction de $\mathrm{RE} \rightarrow \mathcal{P}(\Sigma*)$
  - $\mathcal{L}(a) = \{a\}$ si $a \in \sum$
  - $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
  - $\mathcal{L}(\emptyset) = \emptyset$
  - $\mathcal{L}(r^*) = \mathcal{L}(\mathrm{r})^*$ si r est une RE
  - $\mathcal{L}(r_1|r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
  - $\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$

Variations de notations

- Objectif : simplifier la description
  - [abc]=(a|b|c)
  - [a-z] = (a | b | c | … | z)
  - $r^{0|1} = r? = (r \mid \varepsilon)$
  - $r^n = \underbrace{r \cdot r \cdot \ \ldots \cdot r}_{\text{n fois}}$
  - $r_1 - r_2 \ \ tq \ \ \mathcal{L}(r_1 - r_2) = \mathcal{L}(r_1) - \mathcal{L}(r_2)$

Exemples

- Expression régulière correspondant à un entier
  - 0 | [1..9][0..9]*
- Expression régulière correspondant à un identifiant
  - ([a..z] | [A..Z])([a..z] | [A..Z] | [0..9])*
- Quelle est l'expression régulière décrivant un flottant ?
  - Rq : le mot vide est autorisé

Les limites des langages réguliers

- Les expressions régulières construisent des langages dits réguliers sur la base de trois opérations
  - La concaténation, l'union, la fermeture

- Essayez de décrire une RE reconnaissant des expressions parenthésées composées des identificateurs 'a' et 'b', de '+', '-', '*', '/' et '(', ')'
  - Quel problème se pose ?

- La langages réguliers ne sont donc pas suffisants pour décrire et analyser la syntaxe des langages que nous utilisons…

Conclusion

- Description qui emploie les opérations sur les langages
  - Construire un langage complexe à partir de langages simples
- Les expressions régulières sont des grammaires de langages réguliers
- Une notation qui fait partie du langage de l'informaticien
  - Utilisée dans les utilitaires unix, presentes dans les bibliothèques standard de nombreux langages (java, c++, c#, python…)…
- Mais comment les reconnaitre ?

Les automates à états finis

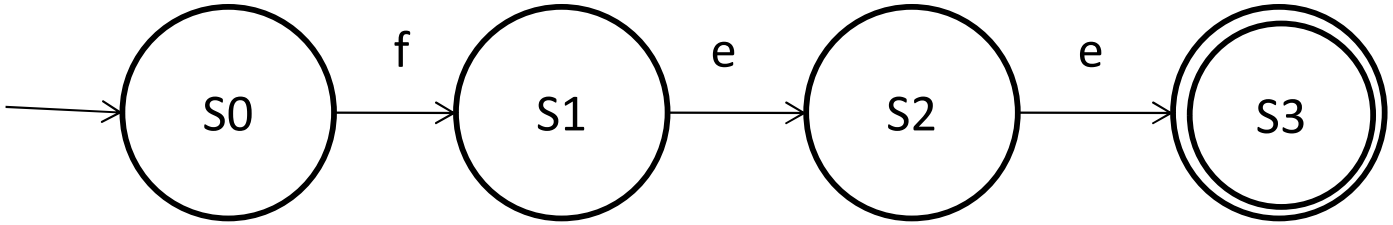
Reconnaissance d’expressions régulières

Automate fini déterministe

- Un automate fini déterministe est donné par un quintuplet $(S, \sum, \delta, s_0, S_F)$
  - S est un ensemble fini d'états
  - $\sum$ est un alphabet fini
  - $\delta : S \times \sum \rightarrow S$ est la fonction de transition
  - s_0 est l'état initial
  - S_F est l'ensemble des états finaux
- Automate déterministe
  - l'état courant + un caractère défini un unique état suivant.
- La structure de l'automate ainsi que ses transitions définissent des mots reconnus sur l'alphabet $\sum$

Automate fini déterministe

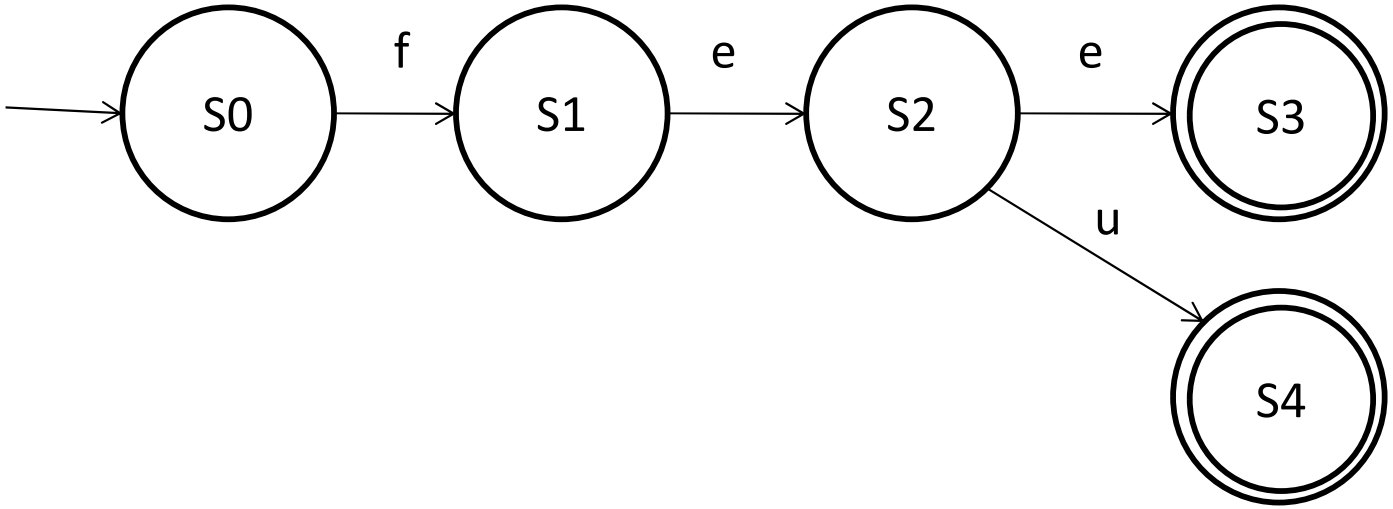
- Automate reconnaissant le mot ‘fee’



- S = {S0, S1, S2, S3}
- ∑ = {f,e}
- δ = {δ(S0,f)->S1, δ(S1,e)->S2, δ(S2,e)->S3}
- S0 : état initial
- S_F = { S3 }

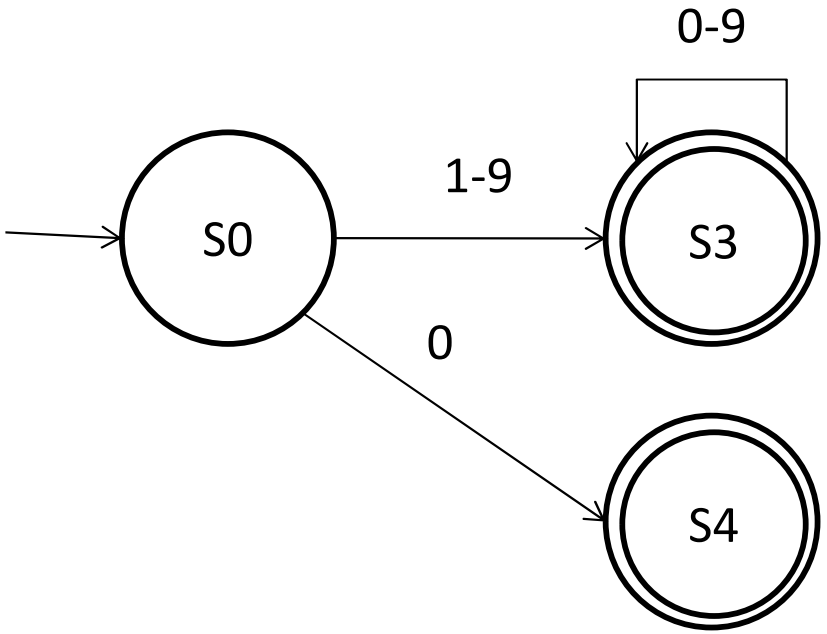
Automate fini déterministe

- Automate reconnaissant les mots ‘fee’ et ‘feu’



Automate fini déterministe

- Possibilité de reconnaitre des mots de longueur infinie
  - Rebouclage sur un état



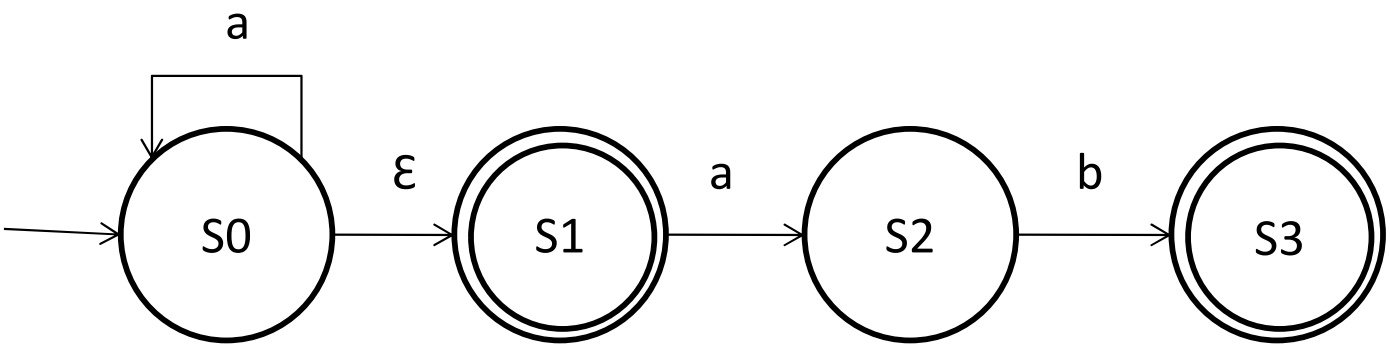
Que reconnait cet automate ?

Automates finis non déterministes

- Automates pouvant atteindre plusieurs états en lisant un seul caractère
- Deux représentations possibles
  - On autorise plusieurs transitions sur un même caractère
    - δ devient une fonction de $S \times \sum \rightarrow 2^S$
  - On autorise les Ɛ-transitions
    - Transitions sur le mot vide

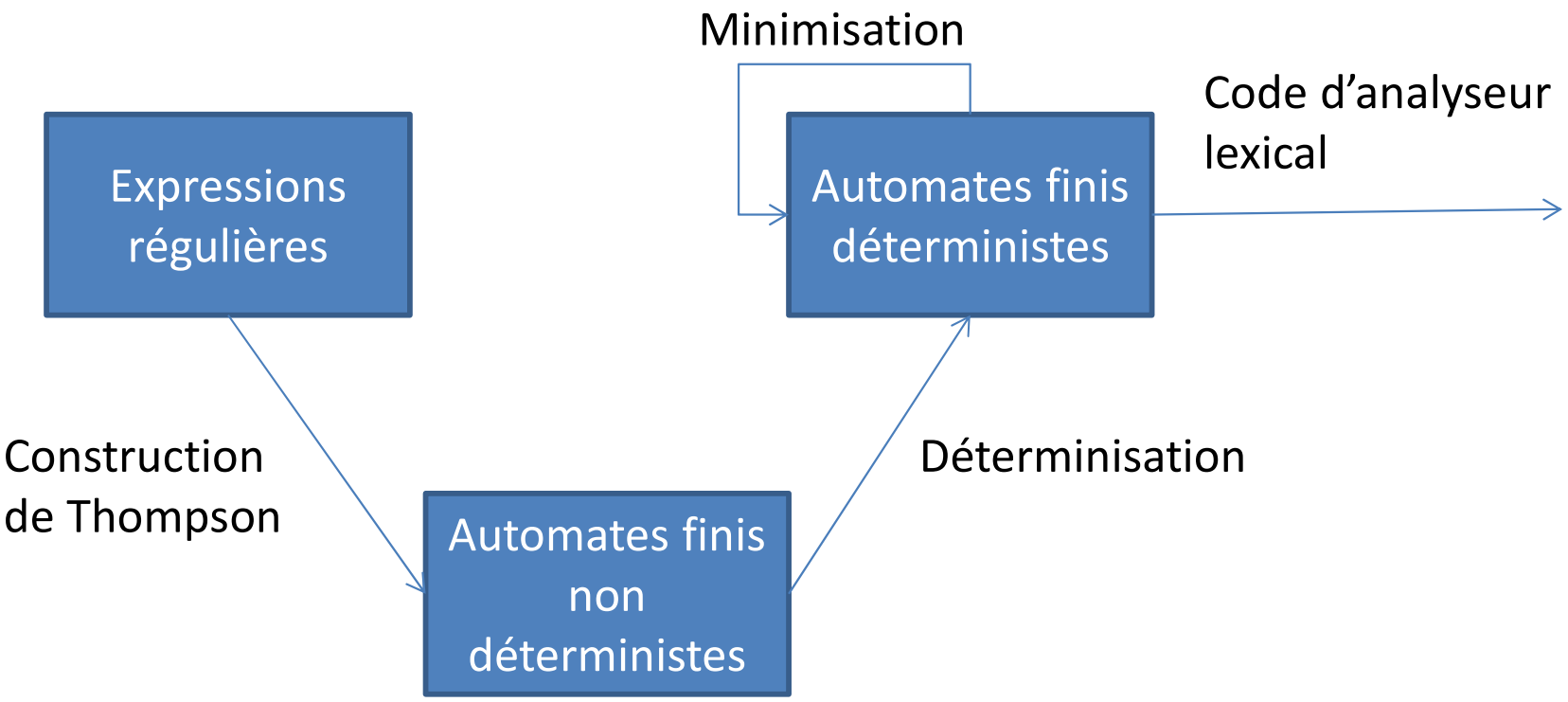
- Remarque : on peut démontrer que les automates finis déterministes et non déterministes sont équivalents en terme de langage reconnu

Automates finis non déterministes



- Reconnaissance d'un mot vide, composé uniquement de a ou se terminant par ab

Des expressions régulières aux automates non déterministes

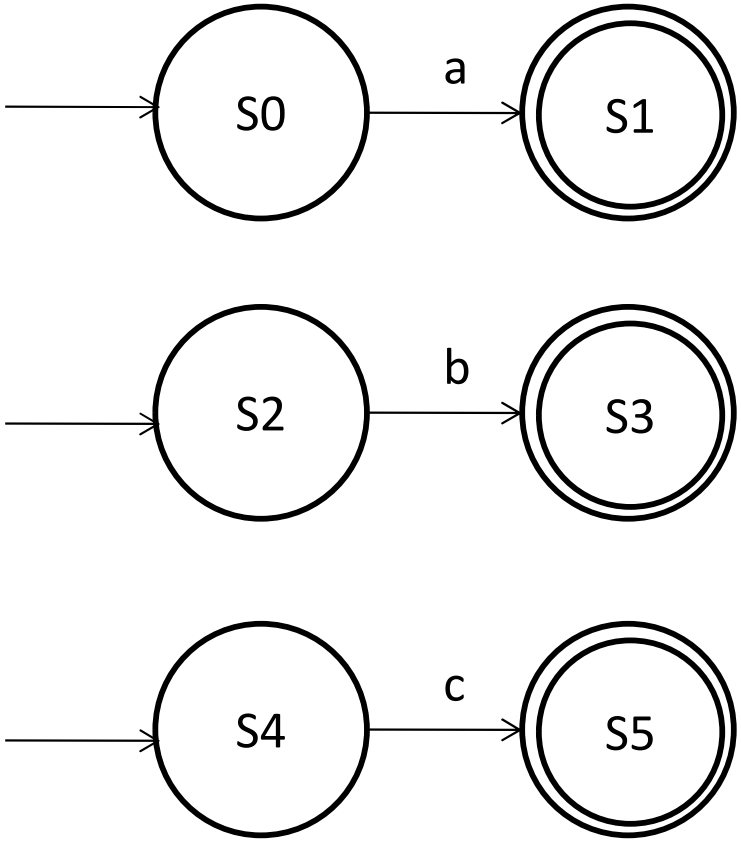


Des expressions régulières aux automates non déterministes

- Pour construire un automate reconnaissant un langage régulier, il suffit d'avoir un mécanisme qui reconnait
  - La concaténation
  - L'union
  - La fermeture

- Exemple sur a(b | c)*

Des expressions régulières aux automates non déterministes

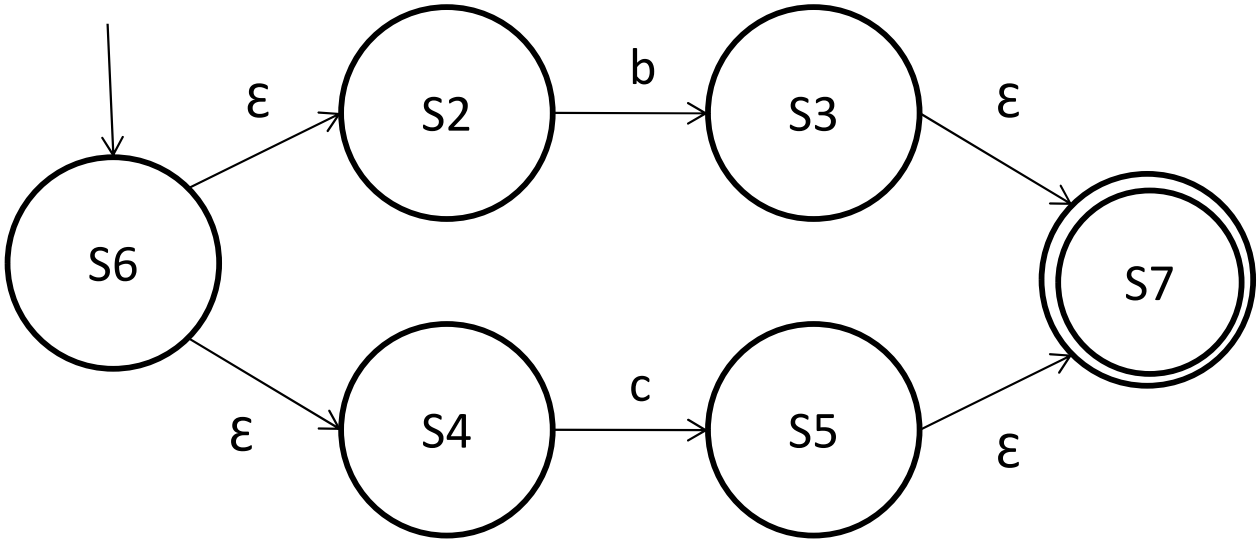


Automate reconnaissant a

Automate reconnaissant b

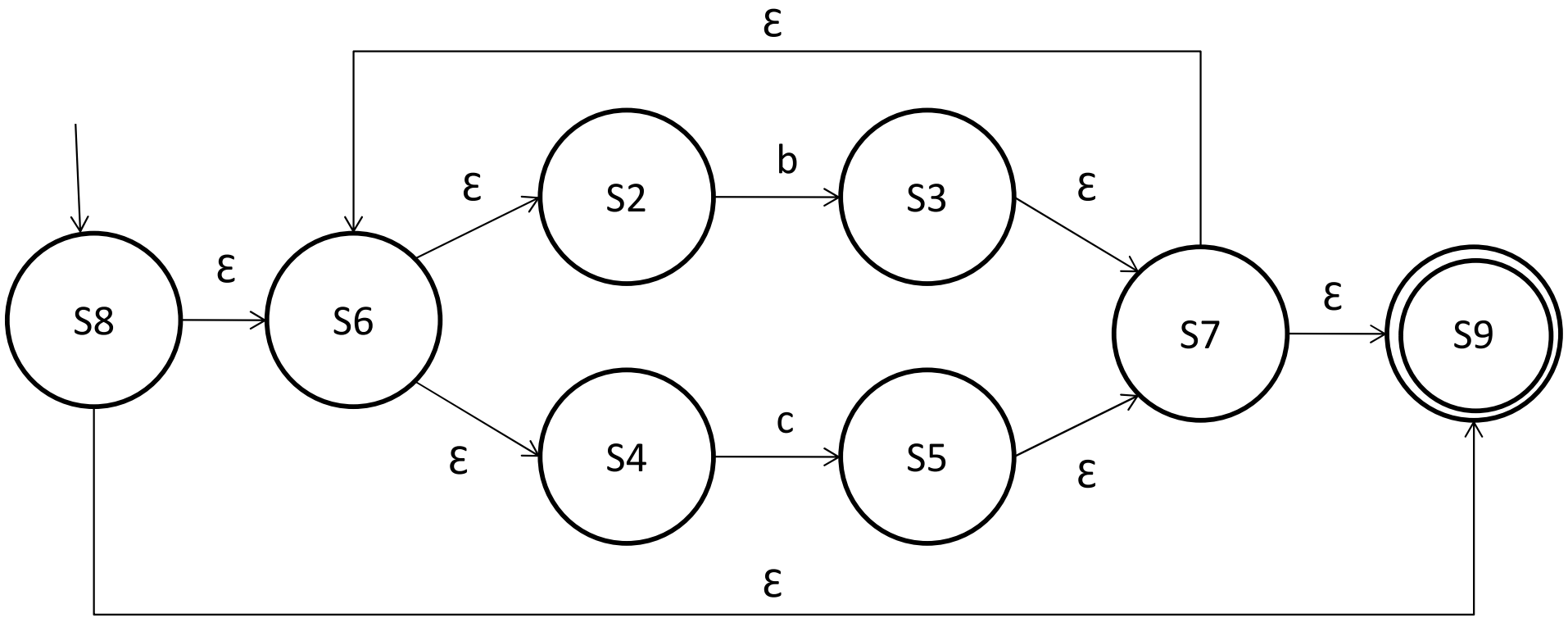
Automate reconnaissant c

Des expressions régulières aux automates non déterministes



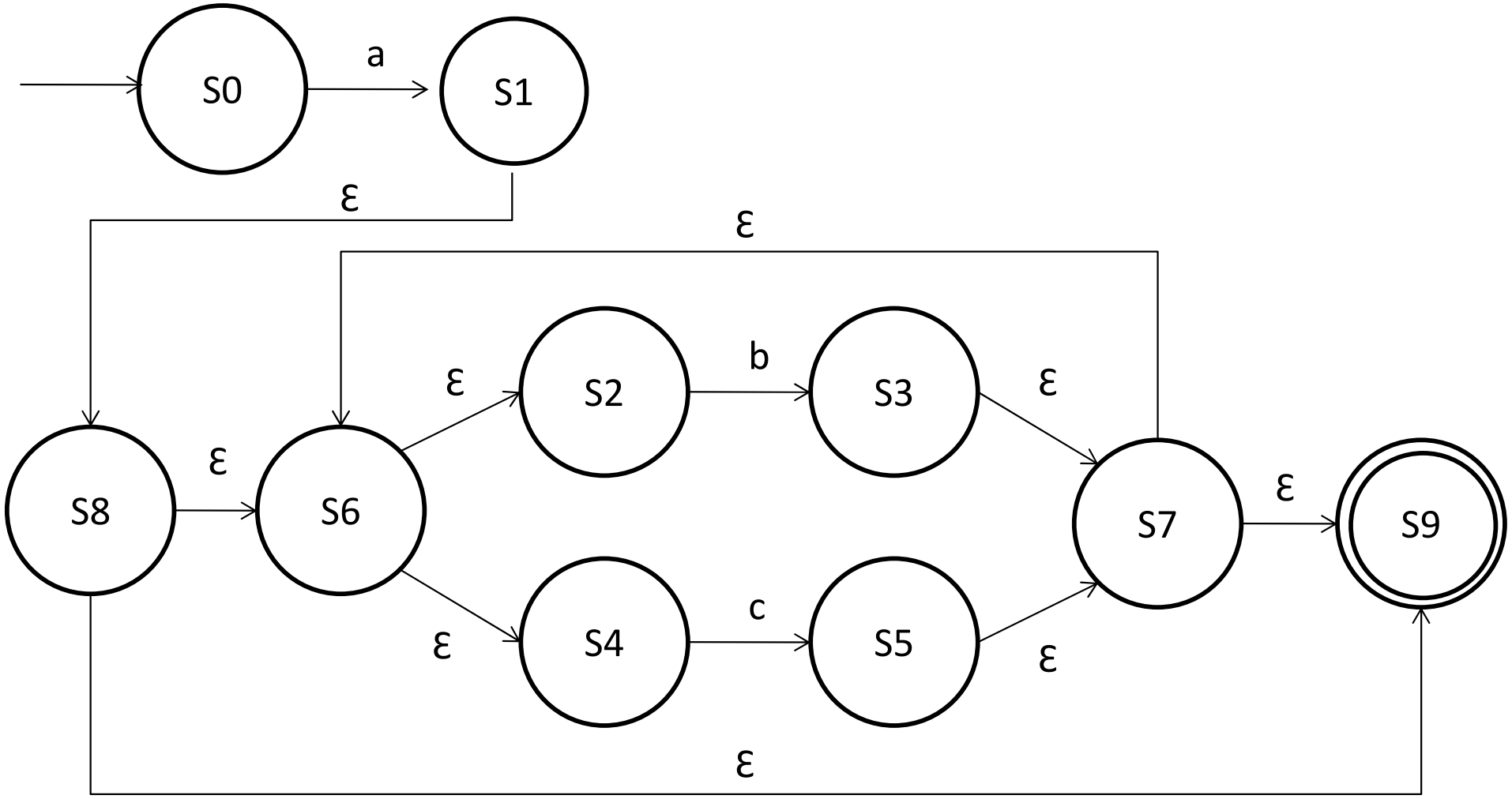
Automate reconnaissant b|c

Des expressions régulières aux automates non déterministes



Automate reconnaissant (b|c)*

Des expressions régulières aux automates non déterministes



Automate reconnaissant a(b|c)*

Des expressions régulières aux automates non déterministes

- Oui mais…

- Les automates non déterministes sont peu efficaces
  - nécessitent l'exploration de plusieurs états suite à la lecture d'une lettre de l'alphabet

- Transformation d'un automate non déterministe en automate déterministe
  - Algorithme de déterminisation et de minimisation

Algorithme de déterminisation

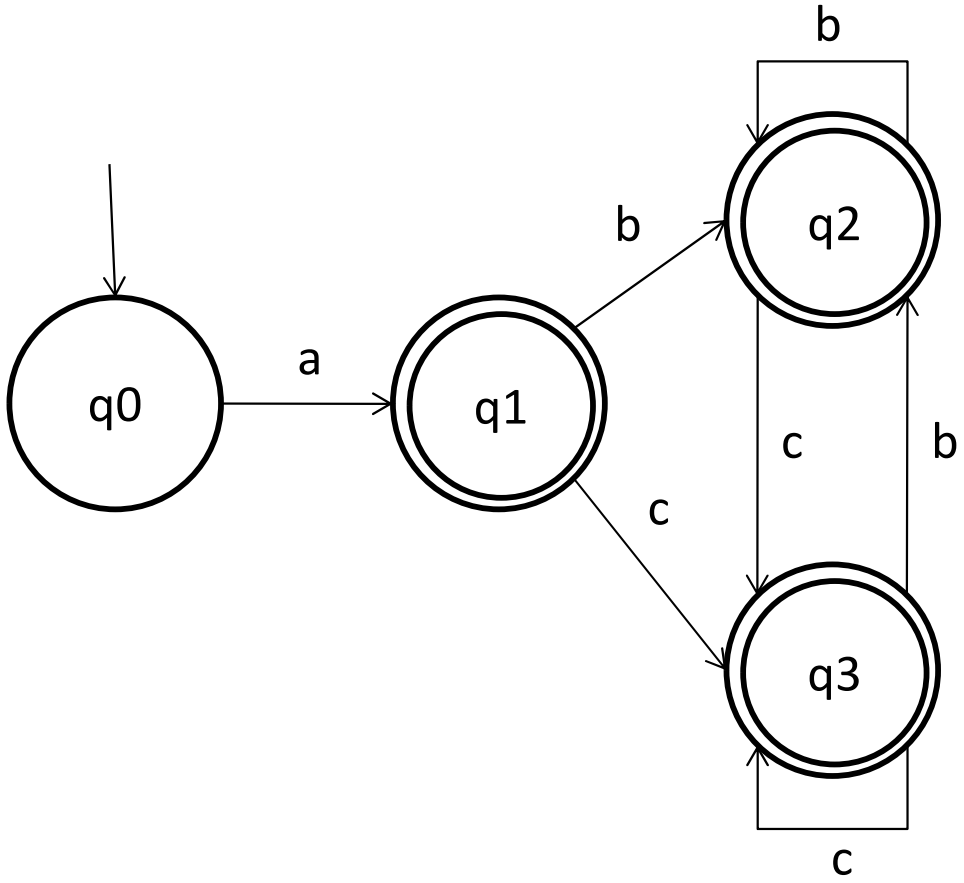
- s0 : état initial de l'automate
- Ɛ-fermeture(S) : collecte tous les états accessibles par Ɛ-transition depuis les états contenus dans S
- Δ(S,c) : collecte tous les états atteignables depuis les états de S en lisant le caractère c

```
q0 <- Ɛ-fermeture(s0)
 initialiser Q avec q0
WorkList <- q0
Tant que WorkList non vide
        Prendre qi dans WorkList
        Pour chaque caractère c de ∑
                q <- Ɛ-fermeture(Δ(qi, c))
                Δ[qi,c] <- q
                Si q n'est pas dans Q
                        Ajouter q à WorkList
                        Ajouter q à Q
                Fin si
        Fin pour
Fin tant que
```

Exercice
Déterminisez l'automate reconnaissant a(b|c)*

Algorithme de déterminisation

- Déterminisation de l'automate pour a(b|c)*



Algorithme de minimisation

- S : ensemble des états de l'automate
- S_F : ensemble des états finaux de l'automate

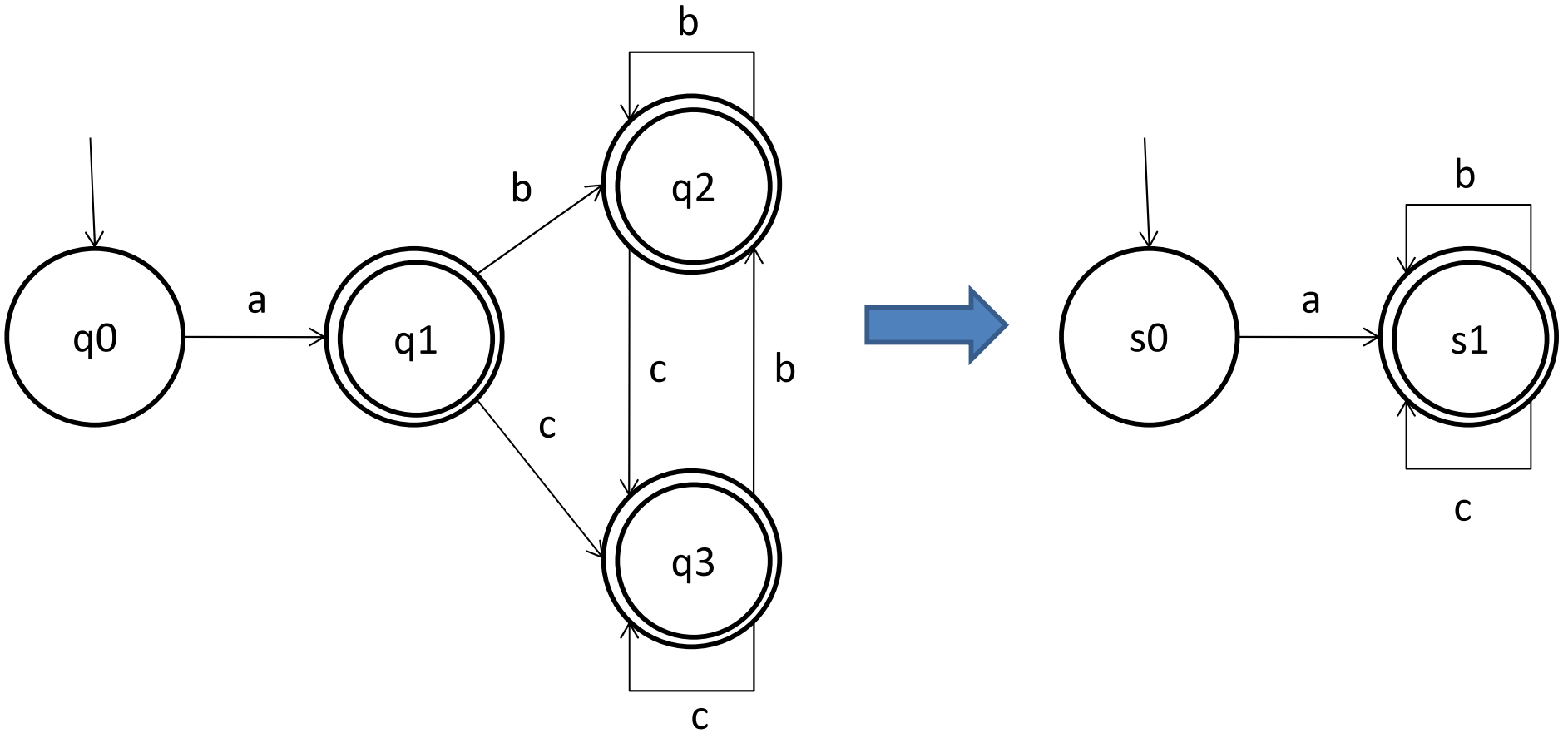
P <- { S_F, S- S_F }
Tant que P change
    T <- ensemble vide
    Pour chaque ensemble p de P
        T <- T union Partition(p)
    P <- T
Fin tant que

Partition(p)
    Pour chaque c de ∑
        Si c sépare p en { p_1,…,p_k } return { p_1,…,p_k }
    Return p

Exercice
Minimisez l'automate reconnaissant a(b|c)*

Algorithme de minimisation

- Minimisation de l'automate pour a(b|c)*



Algorithme de reconnaissance

- Codage de l'automate par une table de transition

char <- prochain caractère
state <- s0
Tant que char != EOF
    State <- δ(state, char), erreur si pas de transition
    Char <- prochain caractère
si state est dans S_F
    alors accepter
    sinon rejeter

δ	a	b	c	autre
s0	s1	-	-	-
s1	-	s1	s1	-
-	-	-	-	-

Conclusion

- Les expressions régulières décrivent ce qui doit être reconnu
- Les automates reconnaissent
- Les RE et les automates finis parlent le même langage
- Méthodologie
  - Spécifier avec une expression régulière
  - Transformer l'expression régulière en automate
  - Reconnaitre avec l'automate

Analyse lexicale

- Les langages réguliers sont utilisés pour décrire les jetons (ou tokens) reconnus lors de l'analyse lexicale
  - Constantes numériques
  - Chaines de caractères
  - Mots clefs du langage
  - Identifiants
  - …
- Pour reconnaitre ces différents jetons, il faut :
  - Construire un automate non déterministe par jeton
    - Chaque état final identifie le jeton reconnu
  - Construire un automate non déterministe correspondant au « ou » entre tous les jetons
  - Déterminiser et minimiser l'automate obtenu
- A partir de l'automate obtenu, il est possible d'écrire / générer un analyseur lexical
  - Entrée : texte à analyser
  - Sortie : suite de jetons utiles à l'analyse syntaxique

Les classes de grammaires

La hiérarchie de Chomsky

(formalisée en 1959)

Grammaire : définition

Une grammaire est un formalisme permettant de décrire une syntaxe et donc un langage formel i.e. un ensemble de mots admissibles sur un alphabet donné.

Les grammaires : formalisation

- La grammaire d'un langage est constituée de quatre objets
  - **T** est l'alphabet **terminal**
    - Ensemble des symboles qui constituent les phrases à reconnaitre (Cf. sections précédentes)
  - **N** est l'alphabet **non terminal**
    - Un non-terminal est une variable syntaxique désignant des ensembles de chaines de symboles terminaux
  - **S** est un symbole particulier de **N**, l'**axiome**
    - Symbole non-terminal qui désigne l'intégralité du langage
  - **P** est un ensemble **de règles de production** (ou de dérivation)
    - Leur forme générale est la suivante : $A \rightarrow B$
      avec $A \in (T \cup N)^* N (T \cup N)^*$ et $B \in (T \cup N)^*$
    - Cela signifie que si l'on a A, il est correct de le remplacer par B (production)
    - Ou que si l'on trouve B, nous avons en réalité reconnu A (analyse)

Les grammaires : formalisation

- En fonction de la nature des règles de production, plusieurs classes de langages peuvent être identifiées

- La hiérarchie de Chomsky comporte 5 classes de grammaires :
  - Générales (type 0)
  - Contextuelles (type 1)
  - Algébriques (type 2)
  - Régulières (type 3)
  - A choix fini (type 4)

- Pour simplifier les notations, posons $V = T \cup N$

Grammaires générales (type 0)

- Règles de la forme $A \rightarrow B$ avec $A \in V^*NV^*$ et $B \in V^*$
- Génère un très grand nombre de mots
- Temps pour savoir si un mot appartient ou non à la grammaire n'est pas forcément fini
  - Appartenance : temps fini
  - Non appartenance : possibilité de bouclage sans réponse
  - Indécidable…

Grammaires contextuelles (type 1)

- Règles de la forme $\alpha A \beta \rightarrow \alpha \gamma \beta$ avec
  - $A \in N$
  - $\alpha, \beta, \gamma \in V^*$
  - $\gamma \neq \varepsilon$

- Contextuelles car le remplacement d'un non terminal peut dépendre des éléments autour de lui

- Appartenance : décidable et PSPACE complet

Grammaires algébriques (type 2)

- Règles de la forme $A \rightarrow \gamma$ avec
  - $A \in N$
  - $\gamma \in V^*$

- Les non-terminaux sont traités individuellement, sans prise en compte du contexte

- Reconnaissance des langages algébriques
  - Algorithmes efficaces

Grammaires régulières (type 3)

- Règles de la forme
  1. $A \rightarrow Ba$ et $A \rightarrow a$ avec $A, B \in N$ et $a \in T$
  2. $A \rightarrow aB$ et $A \rightarrow a$ avec $A, B \in N$ et $a \in T$
  - ➢ On ne peut pas autoriser les deux types de règles simultanément au risque de sortir des langages réguliers

- Reconnaissance des langages réguliers
  - Equivalent aux expressions régulières
  - Reconnus par automates à états finis
    - Algorithmes de reconnaissance très efficaces
    - Cf. partie précédente

Grammaire à choix fini (type 4)

- Règles de la forme $A \rightarrow a$ avec $A \in N$ et $a \in T$
- Classe très restreinte…
- Utilisé pour décrire des macros

Les langages de programmation usuels

- La description d'un programme est un texte
  - Une simple suite de caractères

- L'analyse du texte se fait en deux étapes
  1. On découpe la suite de caractères en mots, en vérifiant que les mots appartiennent au langage
  2. On analyse la suite de mots pour vérifier que les phrases sont correctes

Les langages de programmation usuels

1. Analyse lexicale
   - Découpe du texte en petits morceaux appelés jetons (tokens) correspondant à des suites de caractères
   - Chaque jeton est une unité atomique du langage
     - Mots clés, identifiants, constantes numériques…
   - Les jetons sont décrits par un langage régulier
     - Description via des expression régulières
     - Reconnaissance via des automates à état finis

- Le logiciel effectuant l'analyse lexicale est appelé analyseur lexical ou scanner

Les langages de programmation usuels

2. Analyse syntaxique
   - Analyse de la séquence de jetons pour identifier la structure syntaxique du langage
   - S'appuie sur une grammaire algébrique définissant la syntaxe du langage
   - Produit généralement un arbre syntaxique qui pourra être analysé et transformé par la suite
   - Détection des erreurs de syntaxe
     - Constructions ne respectant pas la grammaire

- Le logiciel effectuant l'analyse syntaxique est appelé analyseur syntaxique ou parser

Chaine de compilation

- Analyse lexicale
  - Découpe du texte en petits morceaux appelés jetons (tokens)
  - Chaque jeton est une unité atomique du langage
    - Mots clés, identifiants, constantes numériques…
  - Les jetons sont décrits par un langage régulier
    - Détection via des automates à état finis
    - Description via des expression régulières

- Le logiciels effectuant l'analyse lexicale est appelé analyseur lexical ou scanner

Chaine de compilation

- Analyse syntaxique
  - Analyse de la séquence de jetons pour identifier la structure syntaxique du langage
  - S'appuie sur une grammaire formelle définissant la syntaxe du langage
  - Produit généralement un arbre syntaxique qui pourra être analysé et transformé par la suite
  - Détection des erreurs de syntaxe
    - Constructions ne respectant pas la grammaire

La hiérarchie de chomsky (1/2)

- Extraction de 5 classes de grammaires
  - Générales (type 0)
    - Règles de la forme : $\alpha \rightarrow \beta$ avec $\alpha \in V^*NV^*, \beta \in V^*$
    - Génère un très grand nombre de mots
    - Temps pour savoir si un mot appartient ou non à la grammaire n'est pas forcément fini
      - Appartenance : temps fini
      - Non appartenance : possibilité de bouclage sans réponse
      - Indécidable…
  - Contextuelles (type 1)
    - Règles de la forme $\alpha A \beta \rightarrow \alpha \gamma \beta$ avec $A \in N, \alpha, \beta, \gamma \in V^*, \gamma \neq \epsilon$
    - Contextuelles car le remplacement d'un non terminal peut dépendre des éléments autour de lui

La hiérarchie de chomsky (2/2)

- Hors contexte (type 2)
  - Règles de la forme $A \rightarrow \gamma$ avec $A \in N, \gamma \in V^*$
  - Les éléments terminaux sont traités individuellement, sans prise en compte du contexte
  - Reconnaissance des langages algébriques

- Régulières (type 3)
  - Règles de la forme $A \rightarrow Ba$ et $A \rightarrow aB$ et $A \rightarrow a$ avec $A, B \in N, a \in T$
  - Reconnaissance des langages rationnels (reconnus par automates à états finis)

- A choix fini(type 4)
  - Règles de la forme $A \rightarrow a$ avec $A \in N, a \in T$
  - Classe très restreinte…

Les grammaires algébriques

L’analyse syntaxique

Grammaire algébrique (hors contexte)
Définition

- **Définition**: une grammaire algébrique est un quadruplet

- **G** = (**N, T, S, P**) où:
  - **N** est l'alphabet **non terminal**
  - **T** est l'alphabet **terminal**
  - **S** est un symbole particulier de **N**, l'**axiome**
  - **P** est un ensemble **de règles de production** (ou de dérivation), de la forme:

    $A \rightarrow w$, avec $A \in \mathbf{N}$ et $w \in (\mathbf{N} \cup \mathbf{T})^*$

Grammaire algébrique
Langage engendré

- Dérivations
  - $m \rightarrow m'$ si $m = uAv$ et $m'=uwv$ et $A \to w \in \mathbf{P}$
  - $m_0 \ast\rightarrow m_n$ s'il existe une suite m_k, k=0, .., n-1 et $m_k \rightarrow m_{k+1}$
- Langage engendré
  - **$L(G) = \{ m \in T^*, S \ast\rightarrow m \}$**
- Langage algébrique:
  - **Définition**: un langage L est dit algébrique (**ALG**) ou "context free" (**CFL**) si il existe une grammaire algébrique **G**, tel que L = L(**G**)
  - Propriété des langages algébriques
    - la partie gauche d'une règle est un non-terminal

Grammaire algébrique
Arbre de dérivation syntaxique

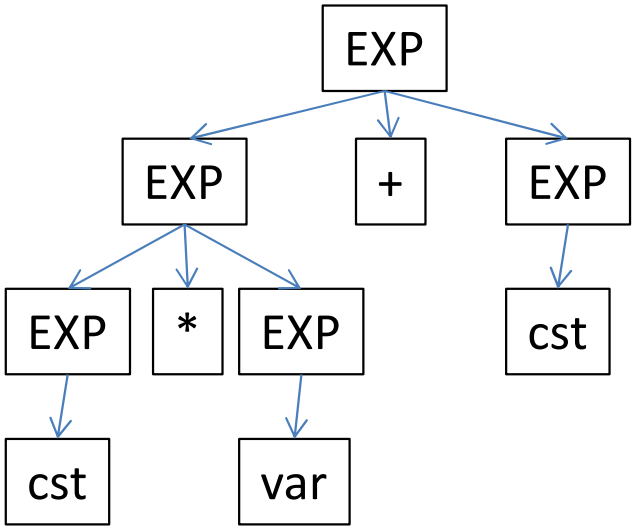
- Un mot m est reconnu par une grammaire algébrique s'il existe un arbre de dérivation syntaxique
  - Les nœuds internes contiennent des symboles non terminaux
  - Les feuilles contiennent des symboles terminaux
  - La racine est le symbole initial
  - Dans un parcours infixe, les feuilles donnent le mot m
  - Si un nœud interne étiqueté X a des sous-arbres $t_1 \ldots t_n$ alors $X \rightarrow t_1 \ldots t_n$ est une règle de la grammaire

- Attention: ne pas confondre
  - Arbre de dérivation syntaxique : associé à la grammaire
  - Arbre de syntaxe abstraite : associé au langage

Grammaire algébrique
Exemple

- Exemple de grammaire décrivant des expressions simples (sans parenthèses)
  - N = {EXP}, S = {EXP}, T = {var, cst, +, *}
  - P = {
    EXP -> var,
    EXP -> cst,
    EXP -> EXP + EXP,
    EXP -> EXP * EXP
  - }

- Exemple de dérivation de EXP
  - EXP -> EXP + EXP -> EXP * EXP + EXP
  - EXP *-> EXP * EXP + EXP
  - EXP *-> cst * var + cst



Arbre de dérivation syntaxique pour cst * var + cst

- Le langage engendré est un langage algébrique et un ***langage régulier***
  - (var | cst) ((+ | *) (var | cst))*

Grammaire algébrique
Exemple

- Exemple de grammaire décrivant des expressions parenthésées
  - N = {EXP}, S = {EXP}, T = {var, cst, +, *, (,) }
  - P = {
    EXP -> var,
    EXP -> cst,
    EXP -> EXP + EXP,
    EXP -> EXP * EXP,
    EXP -> (EXP)
  - }

- Exemple de dérivation de EXP
  - EXP -> EXP * EXP -> EXP * (EXP) -> EXP * (EXP + EXP)
  - EXP *-> EXP * (EXP + EXP)
  - EXP *-> cst * (var + cst)

- Le langage engendré est algébrique et n'est pas régulier
  - L'utilisation de la règle EXP -> (EXP) permet de s'assurer qu'une parenthèse fermante est toujours associée à une parenthèse ouvrante

Grammaire algébrique
Alphabet et langage de programmation

- L'alphabet **non terminal N** correspond
  - soit à des **constructions sémantiques** du langage de programmation: programme, déclaration, instruction, boucle, etc..
  - soit à des **constructions syntaxiques**: liste, etc..

- L'**axiome S** correspond à la construction compilable de plus haut niveau:
  - classe ou interface pour Java.
  - définition de type, fonction, implémentation de méthode pour C++

- L'alphabet **terminal T** correspond aux **jetons** (unités lexicales) renvoyées par l'analyseur lexical:
  - mots clés,
  - identificateurs,
  - séparateurs,
  - opérateurs,
  - …

Un exemple de texte structuré

- Exemple de deux adresses postales

Fabrice Lamarche
IRISA
263 avenue du général Leclerc
35062 Rennes CEDEX

Bernard Truc
25 avenue Bidule
35000 Rennes

- Que peut-on remarquer ?
  - Une structure similaire
    - Un nom, un numéro et un nom de rue, un code postal et une ville
  - Mais quelques variantes
    - Le nom peut être complété avec un nom d'entreprise ou être simplement un nom d'entreprise
    - La ville peut être suivie de « CEDEX » ou non

Notations usuelles des grammaires

- BNF : Backus Naur Form ou Backus Normal Form
  - Description de grammaires algébriques ou context free

- Notations
  - Définition d'une règle : opérateur **::=**
    - Non terminal **::=** expression
  - Non terminaux : **<**non-terminal**>**
    - Identifiant entre < >
  - Terminaux : **"**terminal**"**
    - Chaine de caractères entre " "
  - Alternative : opérateur **|**
    - "toto" **|** <regle>
    - signifie une alternative entre *toto* et ce qui est reconnu par le non terminal <regle>

- Cette notation est suffisante pour décrire les grammaires algébriques

Grammaire en notation BNF
La grammaire pour les adresses

<adresse> **::=** *<identite> <adresse-rue> <code-ville>*
<identite> **::=** *<nom-personne> <EOL>* |
<nom-personne> <EOL> <nom-entreprise> <EOL>
<nom-personne> **::=** *<prenom> <nom>*
<prenom> **::=** *<initiale>* "." | *<mot>*
<nom> **::=** *<mot>*
<nom-entreprise> **::=** *<mot>* | *<mot> <nom-entreprise>*
<adresse-rue> **::=** *<numero> <type-rue> <nom-rue> <EOL>*
<type-rue> **::=** "rue" | "boulevard" | "avenue"
<nom-rue> **::=** *<mot>* | *<mot> <nom-rue>*
<code-ville> **::=** *<numero> <ville> <EOL>* | <numero> <ville> "CEDEX" <EOL>
<ville> **::=** *<mot>* | *<mot> <ville>*

Grammaires en notation BNF

- Certaines choses sont « longues » à décrire et pas forcément lisibles
  - Rendre quelque chose d'optionnel
    - **<**rule-optional**> ::= <**element**> | ""**
    - Rq : "" est le mot vide
  - Répétition d'un élément 0 à n fois
    - **<**rule-repeat**> ::= <**element**> <**rule-repeat**> | ""**
    - Utilisation de la récursivité et du mot vide
  - Répétition d'un élément 1 à n fois
    - **<**rule-repeat-one**> ::= <**element**> <**rule-repeat**>**
    - Utilisation de deux règles car pas de groupement

- Des extensions à la notation BNF ont été proposées
  - EBNF, ABNF etc…
  - Rôle : augmenter la lisibilité et simplifier l'écriture des grammaires

Notations usuelles des grammaires
Grammaires en notation EBNF

- EBNF : Extended BNF
  - Même expressivité que les grammaire BNF
  - Plus facile à écrire et à comprendre

- Notations :
  - Définition d'une règle : operateur **=**
  - Fin de règle : symbole **;**
  - Alternative : symbole **|**
  - Elément optionnel : utilisation de **[** … **]**
  - Répétition de 0 à n fois : **{** … **}**
  - Groupement : utilisation de **(** … **)**
  - Terminal : "terminal" ou 'terminal'
  - Commentaire : **(*** commentaire ***)**

- Apparition d'une notation pour les élément optionnels, les groupements et la répétition
  - Simplification de notation et amélioration de la lisibilité
  - Evite la récursivité (pour les cas simples) et l'utilisation du mot vide

Exemple BNF vs EBNF

- Exemple: langage de déclaration de plusieurs variables de type int ou float.

  – Grammaire en notation BNF

  <declaration-vars> ::= <declaration-var> <declaration-vars> | ""

  <declaration-var> ::= <type> <identifiant> ";"

  <type> ::= "int" | "float"

  – Grammaire en notation EBNF

  declarationVars = { ("int" | "float") *identifiant* ";" } ;

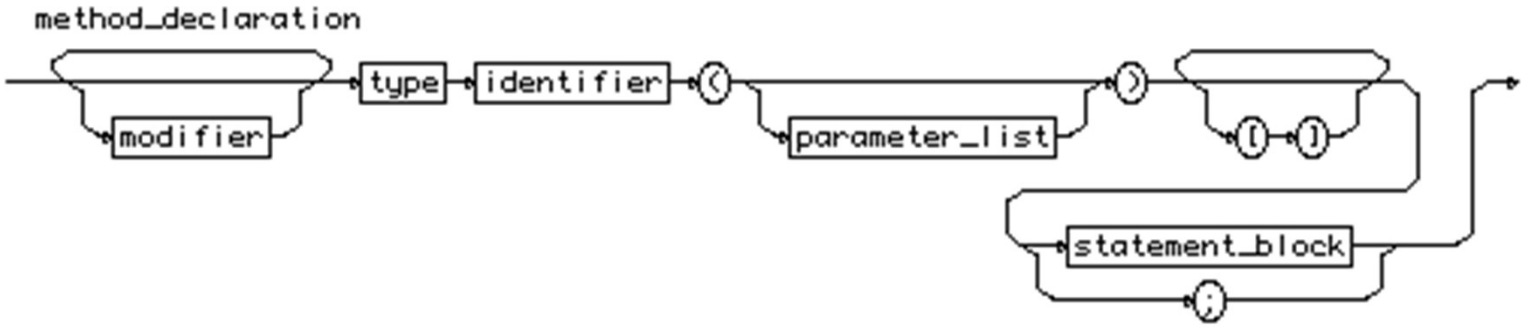
La grammaire pour les adresses
La grammaire en notation EBNF

adresse = *identite adresseRue codeVille ;*
identite = *nomPersonne EOL* [*nomEntreprise EOL*] *;*
nomPersonne = *prenom nom ;*
prenom = *initiale* "." | *mot ;*
nom = *mot ;*
nomEntreprise = *mot* {*mot*} ;
adresseRue = *numero typeRue nomRue EOL ;*
typeRue = "rue" | "boulevard" | "avenue" ;
nomRue = *mot* {*mot*} ;
codeVille = *numero ville* ["CEDEX"] *EOL ;*
ville = *mot* {*mot*} ;

La notation graphique
Diagrammes de syntaxe

- Exemple sur la règle de déclaration de méthode en Java

method_declaration ::= { modifier } type identifier
"**(**" [parameterList] "**)**" { "**[**" "**]**" }
(statement_block | "**;**")



Notations usuelles des grammaires
Variante basée sur expressions régulières

- Simple d'utilisation
- Notation similaire à celle utilisée pour l'analyseur lexical
- Notation utilisée dans certains logiciels
  - Ex : ANTLR qui sera utilisé en TP.

- Notations
  - Alternative : symbole **|**
  - Groupement : utilisation de **(** … **)**
  - Répétition 0 à n fois : utilisation de *****
  - Répétition 1 à n fois : utilisation de +
  - Elément optionnel : utilisation de ?

- Retour sur l'exemple précédent dans cette notation :
  - *declarationVars* = **(** **(**"int" **|** "float"**)** *identifiant* ";" **)*** ;

La grammaire pour les adresses
Variante EBNF basée exp. régulières

adresse = *identite adresseRue codeVille ;*
identite = *nomPersonne EOL* (*nomEntreprise EOL*)? *;*
nomPersonne = *prenom nom ;*
prenom = *initiale* "." | *mot ;*
nom = *mot ;*
nomEntreprise = *mot+* ;
adresseRue = *numero typeRue nomRue EOL ;*
typeRue = "rue" | "boulevard" | "avenue" ;
nomRue = *mot+* ;
codeVille = *numero ville* "CEDEX"? *EOL ;*
ville = *mot+* ;

Exercice

- *Ecrivez la grammaire (notation EBNF) d'un langage reconnaissant des expressions booléennes pouvant être parenthèsées et prenant en compte la priorité des opérateurs*
  - Opérateurs binaires: and, or
  - Opérateur unaire : not
  - Parenthèses : (,)
  - Valeurs (true, false) ou identifiants pour les termes

Reconnaissance du langage

- Une fois la grammaire écrite, il faut la reconnaitre
- Deux grands types d'approche
  - L'analyse ascendante
    - Construction de l'arbre de dérivation syntaxique à partir des feuilles
    - Utilisée dans des logiciels type YACC
  - L'analyse descendante
    - Construction de l'arbre de dérivation syntaxique à partir de la racine
    - Utilisée dans des logiciels type ANTLR(utilisé en TP)

L’analyse ascendante

- Aussi appelée analyse LR
  - Il analyse l'entrée de gauche à droite (**L**eft to right) et produit une dérivation à droite (**R**ightmost derivation)
  - On parle d’analyseur LR(k) où k est le nombre de symboles anticipés et non consommés utilisés pour prendre une décision d’analyse syntaxique

- Construit l’arbre de dérivation syntaxique en partant des feuilles
- On garde en mémoire une pile.
  - Cette pile contient une liste de non-terminaux et de terminaux.
  - Correspond à la portion d’arbre reconstruit
- Deux opérations
  - Lecture (shift) : on fait passer le terminal du mot à lire vers la pile
  - Réduction (reduce) : on reconnait sur la pile la partie droite x1…xn d’une règle X ::= x1…xn et on la remplace par X
- Le mot est reconnu si on termine avec le mot vide à lire et le symbole initial sur la pile.

L’analyse descendante

- Aussi appelée analyse LL
  - Il analyse l'entrée de gauche à droite (**L**eft to right) et en construit une dérivation à gauche (**L**eftmost derivation)
  - On parle d’analyseur LL(k) où k est le nombre de symboles anticipés et non consommés utilisés pour prendre une décision d’analyse syntaxique

- Reconstruction de l’arbre syntaxique à partir de la racine
  - Entrée : un mot *m* (suite de jetons produits par l’analyseur lexical)
  - Une fonction associée à chaque terminal et non terminal
    - Reconnaissance de la structure du non terminal / terminal
  - Suppose qu’étant donné le non terminal et le mot, on peut décider de la règle à appliquer
  - Pour reconnaitre le mot, on part de la fonction associée au symbole initial et on vérifie que l’on atteint la fin de la chaine

- Construit (implicitement) un arbre de racine le symbole initial dont les feuilles forment le préfixe de *m*

- Chaque fonction renvoie
  - le reste du mot *m* à analyser
  - Eventuellement une information associée (ex : arbre de dérivation syntaxique)

Limites de l'analyse descendante

- On ne peut pas toujours décider facilement de la règle à appliquer
  - Combien de terminaux faut-il explorer pour déterminer la règle à appliquer ?
  - Si k terminaux à explorer : analyseur LL(k)
  - La plupart du temps, on s'intéresse aux grammaires LL(1)
- Les grammaires récursives gauches (X::=Xm) ne sont pas LL(1)
- Par contre, les grammaires récursives droite (X::=mX) le sont

- Exemple de grammaire LL(2)
  - bonjour = bonjourMonsieur | bonjourMadame ;
  - bonjourMonsieur = "bonjour" "monsieur" **;**
  - bonjourMadame = "bonjour" "madame" **;**

- Même grammaire en LL(1)
  - bonjour = "bonjour" (monsieur | madame)
  - monsieur = "monsieur" **;**
  - madame = "madame" **;**

- Remarque : le plus souvent, il est possible de passer d'une grammaire LL(k) à une grammaire LL(1)

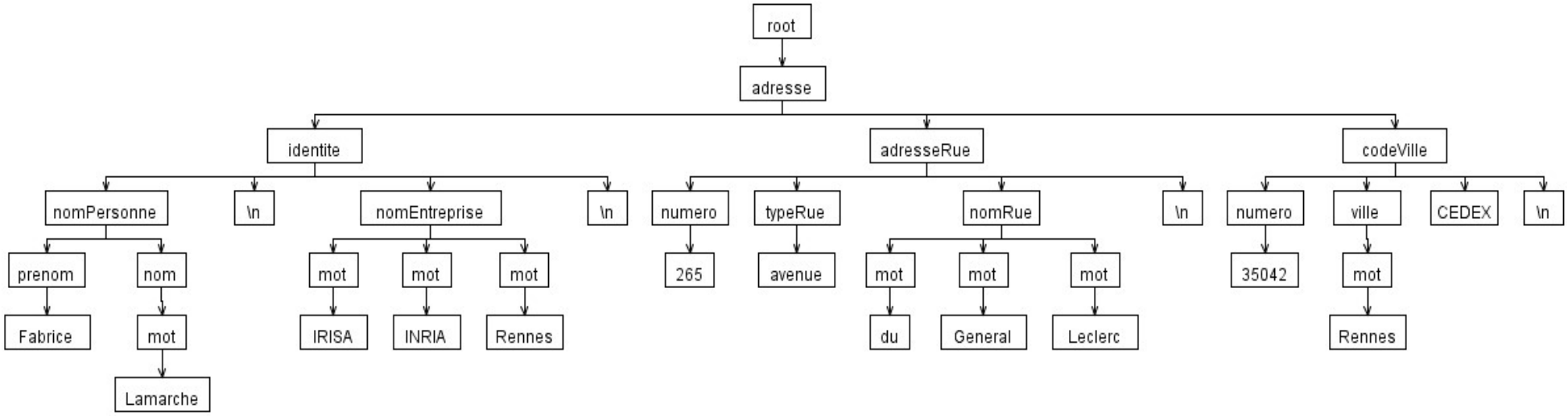
- Remarque 2 : de nos jours, grâce à certains outils (ANTLR par exemple), il n'est plus nécessaire d'effectuer cette transformation

Vers les arbres syntaxiques abstraits

- Arbre de dérivation syntaxique
  - Résulte du parcours de la grammaire lors de l'analyse syntaxique
  - Possède beaucoup d'informations « inutiles »
    - Lexèmes servant à désambigüiser l'analyse
    - Lexèmes servant à identifier les priorités sur les opérations
      - Exemple : parenthèses dans une expression numérique
    - Délimitation de début / fin de bloc
    - …
- Arbre syntaxique abstrait
  - Plus compacte que l'arbre de dérivation syntaxique
  - Représente la sémantique du langage
  - Supprime les informations « inutiles »
    - Ex : Une syntaxe sous forme d'arbre représente implicitement les priorités des opérations, pas besoin de parenthèses
  - S'obtient par réécriture de l'arbre de dérivation syntaxique

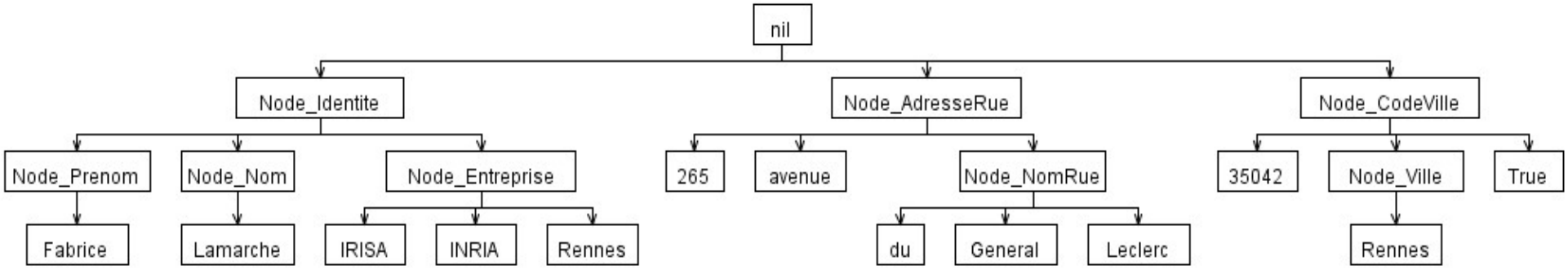
Retour sur les adresses
Arbre de dérivation syntaxique

- Adresse exemple :
  Fabrice Lamarche
  IRISA INRIA Rennes
  265 avenue du General Leclerc
  35042 Rennes CEDEX



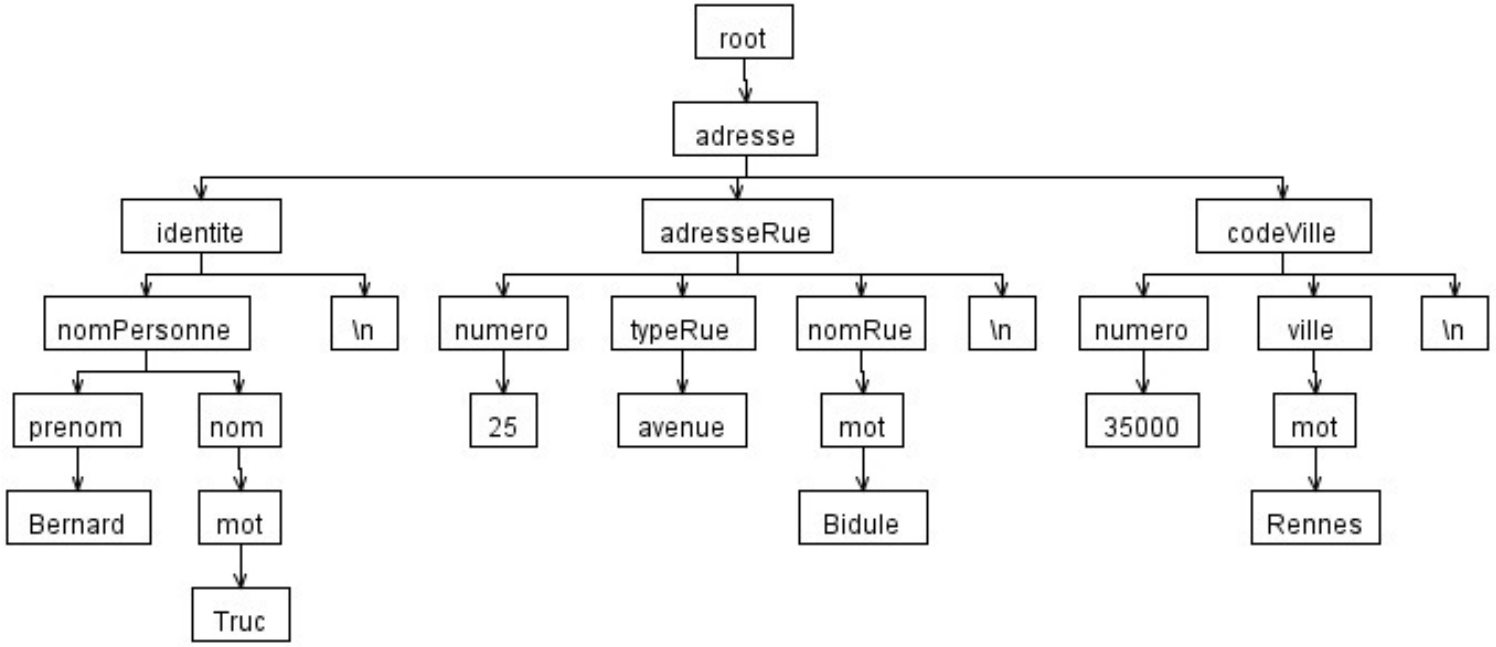
Retour sur les adresses
Arbre syntaxique abstrait

- Adresse exemple :
  Fabrice Lamarche
  IRISA INRIA Rennes
  265 avenue du General Leclerc
  35042 Rennes CEDEX



Retour sur les adresses
Arbre de dérivation syntaxique

- Adresse exemple :
  Bernard Truc
  25 avenue Bidule
  35000 Rennes



Retour sur les adresses
Arbre syntaxique abstrait

- Adresse exemple :
  Bernard Truc
  25 avenue Bidule
  35000 Rennes