Compilation pour l'Image Numérique

Fabrice Lamarche

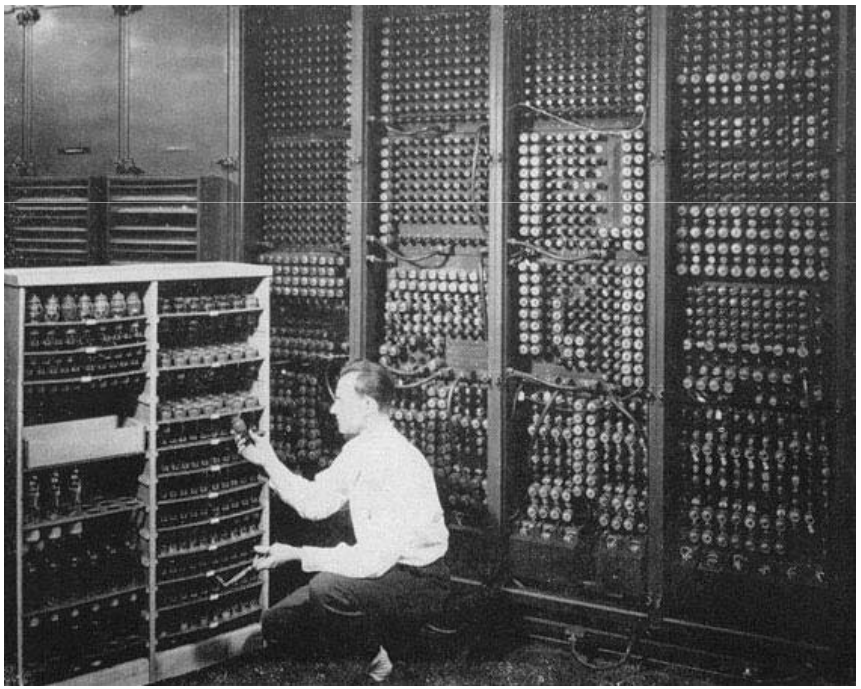
ESIR – IN

Historique

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

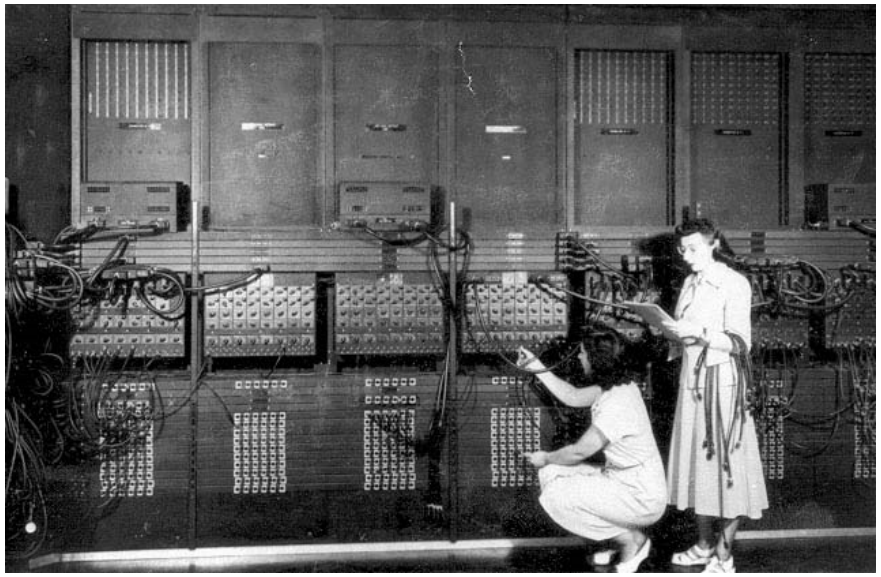
- Quelques chiffres
  - 17468 tubes à vide
  - 7200 diodes
  - 1500 relais
  - 70000 résistances
  - 10000 condensateurs
  - 30 tonnes
  - Occupe une surface de 67m^2
  - Consomme 150 Kilowatts

- 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 ☺

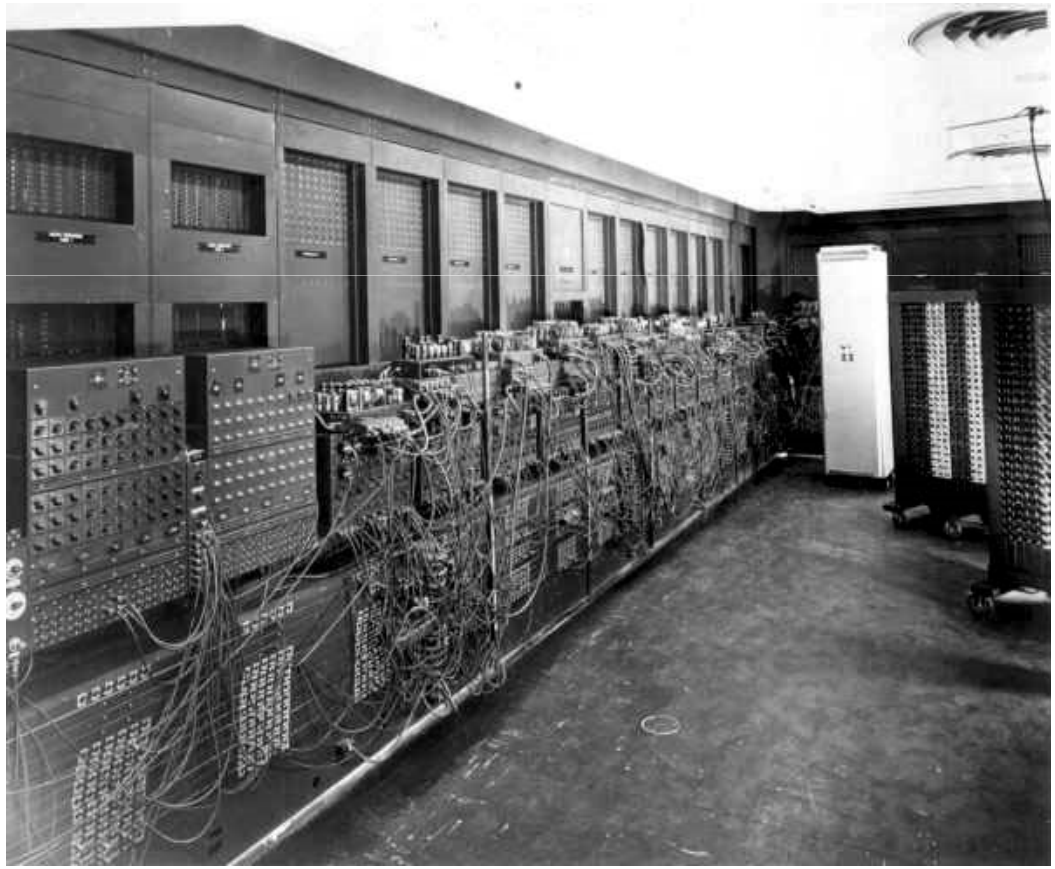


Historique

- 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 ☺

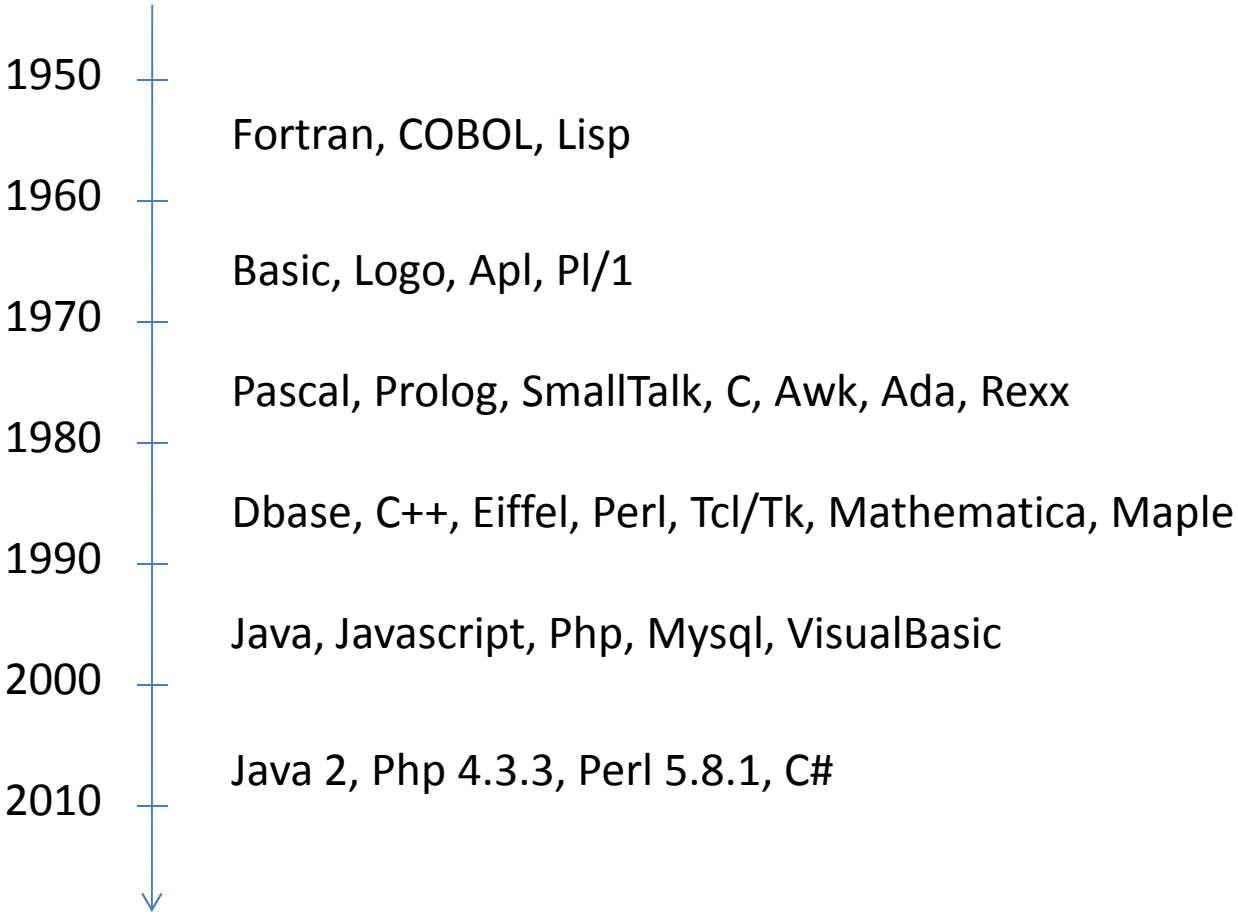


Historique

- 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 CIN ?

- 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 ?

- 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

- Certaines descriptions dans les langages standards peuvent être longues et fastidieuses
    - Création d'un compilateur : langage dédié, simple
    - 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

Logiciels et langages de script

- Développement de langages « dédiés »
  - Unreal development kit (UDK)
    - Proposition de l'unreal script
    - Utilisé, entre autre, pour la programmation des bots
  - 3DS Max
    - Maxscript
  - Pov ray
    - Moteur de rendu
    - Langage de script pour la description des scènes et des animation
    - Ouverture vers des interfaces de modélisation 3D
      - Communication via le langage de script

- Utilisation de langages de script existants
  - Google sketchup
    - Utillisation de Ruby avec interfaces vers fonctionnalités de Sketchup
  - Jeux vidéos
    - Utilisation de LUA (World of warcraft, Far cry, SimCity 4)

Qu’est-ce qu’un compilateur ?

Le compilateur

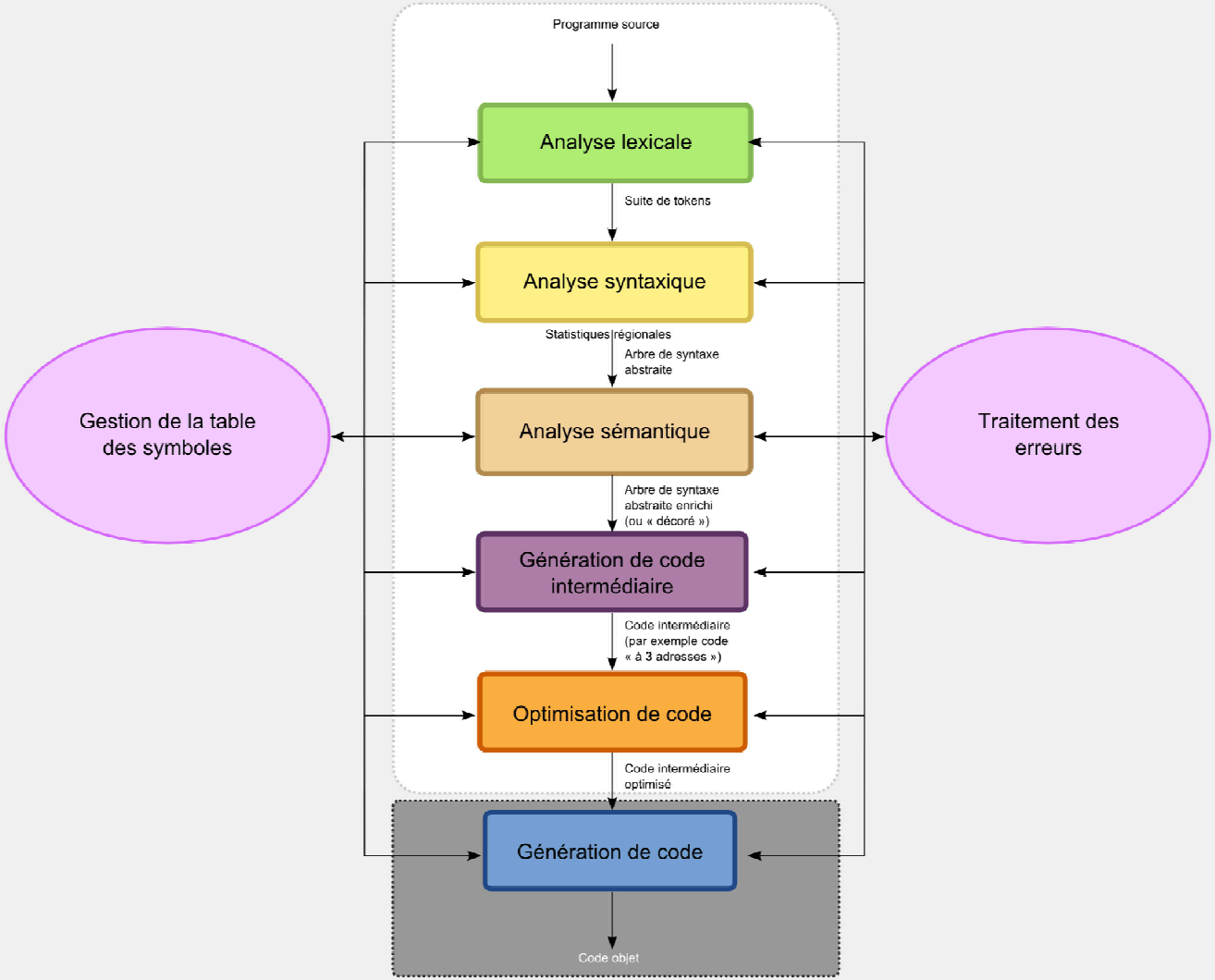
- Un **compilateur** est un programme informatique qui traduit un langage, le *langage source*, en un autre, appelé le *langage cible*, en préservant la signification du texte source

- Traduction d'un langage en instructions machines
  - Ex : C / C++

- Traduction d'un langage de haut niveau vers un autre
  - Ex : Traduction de Pascal en C

- Traduction d'un langage quelconque vers un langage quelconque
  - Ex : Word vers html, pdf vers ps, obj vers mesh

- Un langage décrit une information structurée
  - Utile dans **tous** les domaines



Chaîne de compilation

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

Chaine de compilation

- Analyse sémantique
  - Ajout d'information sémantique à l'arbre d'analyse
  - Construction de la table des symboles
    - Ex : noms de fonction, de variables etc…
  - Réalise des vérifications sémantiques
    - Vérification de type
    - Vérification de la déclaration des variables, des fonctions
    - Vérifie que les variables sont initialisées avant utilisation
    - …
  - Emission d'erreurs et / ou d'avertissements
  - Rejet des programmes « incorrects »

Chaine de compilation

- Génération de code intermédiaire
  - Code indépendant de la machine cible
  - Généralement : utilisation du code 3 adresses
    - Nombre « infini » de registres

```
int main(void)
{
  int i;
  int b[10];
  for (i = 0; i < 10; ++i)
  { b[i] = i*i; }
}
```

```
        i := 0 ; assignment
L1:     if i >= 10 goto L2 ; conditional jump
        t0 := i*i
        t1 := &b ; address-of operation
        t2 := t1 + i ; t2 holds the address of b[i]
        *t2 := t0 ; store through pointer
        i := i + 1
        goto L1
L2:
```

Chaine de compilation

- Optimisation de code
  - Accélération du programme
  - Suppression des calculs inutiles
  - Elimination des sous expressions communes
  - Propagation des copies
  - Propagation des constantes
  - Extraction des calculs invariants
  - Elimination de code mort
  - Inlining
  - …
- Transforme le code mais ne change pas sa sémantique
- La première optimisation est algorithmique !
  - Un compilateur ne changera pas votre algorithme…

Chaine de compilation

- Génération de code
  - Transformation du code intermédiaire en code machine
  - Connaissance des particularités du processeur
    - Nombre de registres
    - Utilisation du jeu d'instruction du processeur
      - Ex : instructions vectorielles de type SSE
    - Ordonnancement des instructions
- La seule phase réellement dépendante du processeur

- Possibilité d'avoir des compilateurs multi-cible
  - La différence réside dans la génération de code

Grammaires et expressions régulières

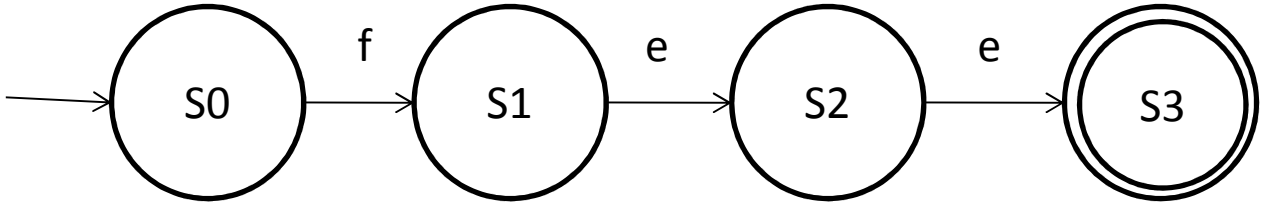
L’analyse lexicale

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

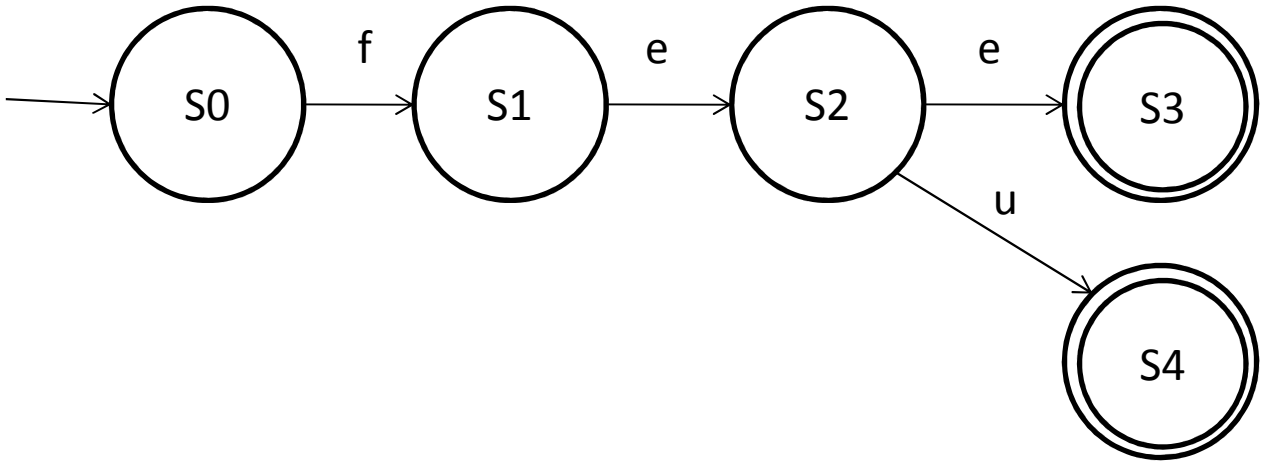
- Automate reconnaissant le mot ‘fee’



- 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 }

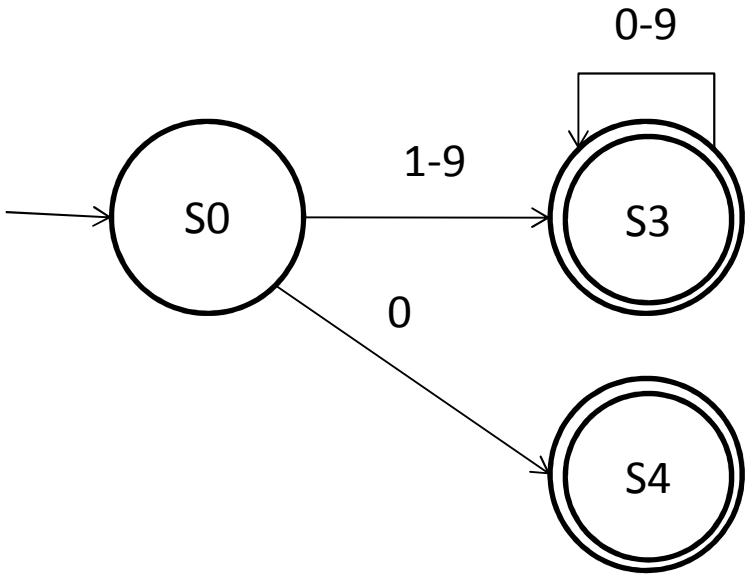
Automate fini déterministe

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



Automate fini déterministe

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



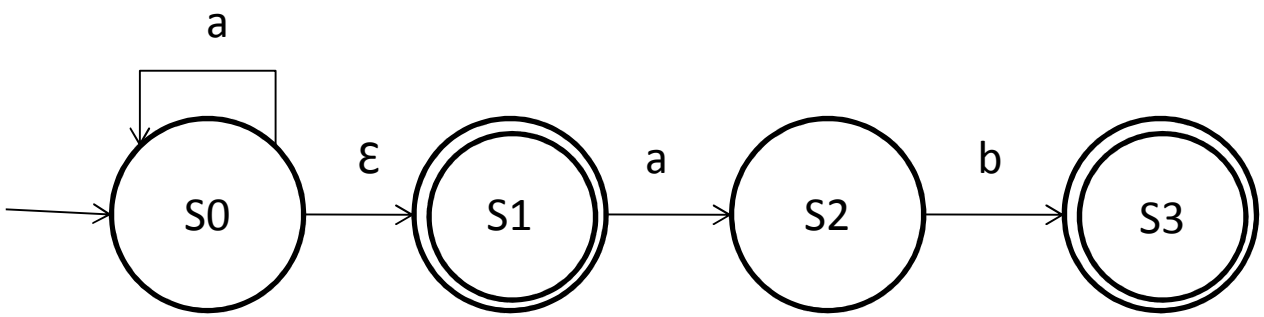
Que reconnait cet automate ?

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

Les expressions régulières

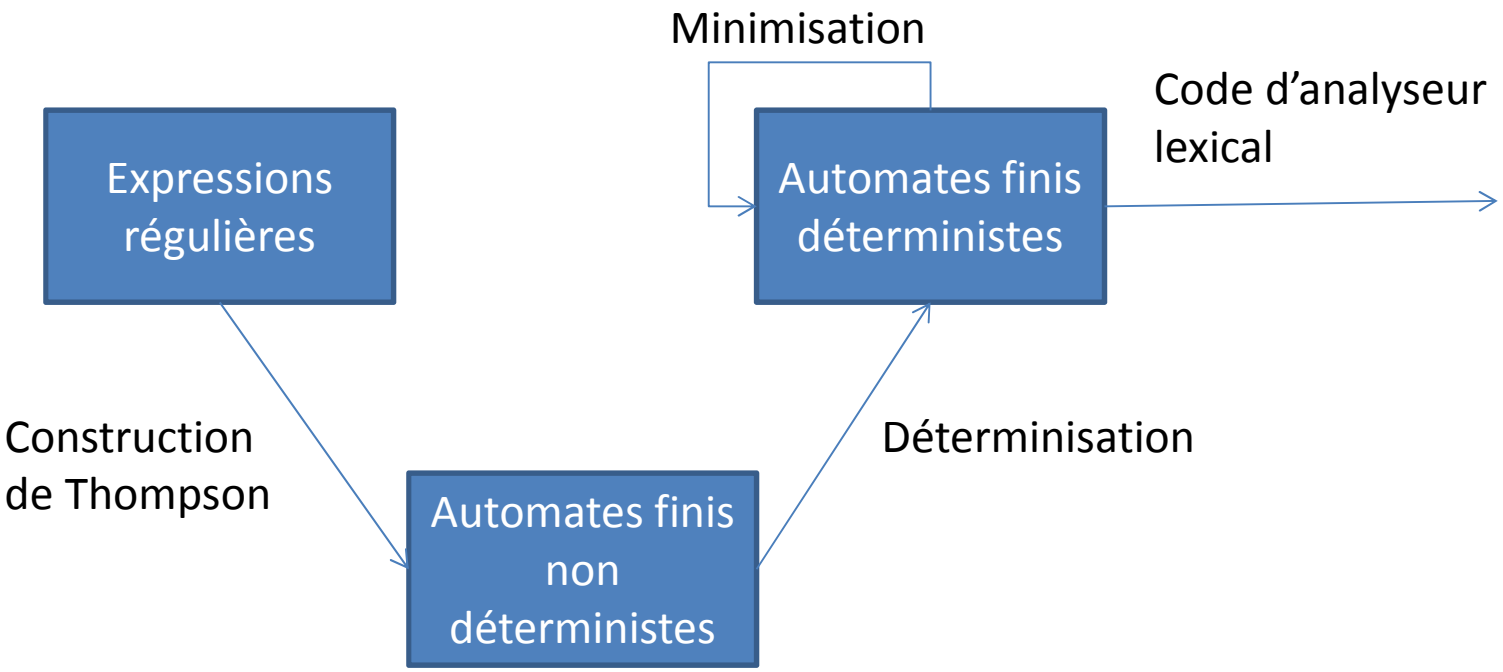
- Formule close désignant un ensemble de chaines de caractères construites sur un alphabet ∑

- Construction à partir des lettres de l'alphabet ∑ et de trois opérations sur les ensembles de chaines de caractères
  - Union de deux ensembles
    notée R | S = {s | s ϵ R ou s ϵ S }
  - La concaténation de deux ensembles R et S, notée RS.
    On note R^i pour la concaténation de R i fois.
  - La fermeture transitive de R, notée R* = $\bigcup_{i=0}^{\infty} R^i$

Les expressions régulières

- 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 réel ?
  - Rq : le mot vide est autorisé

Des expressions régulières aux automates non déterministes

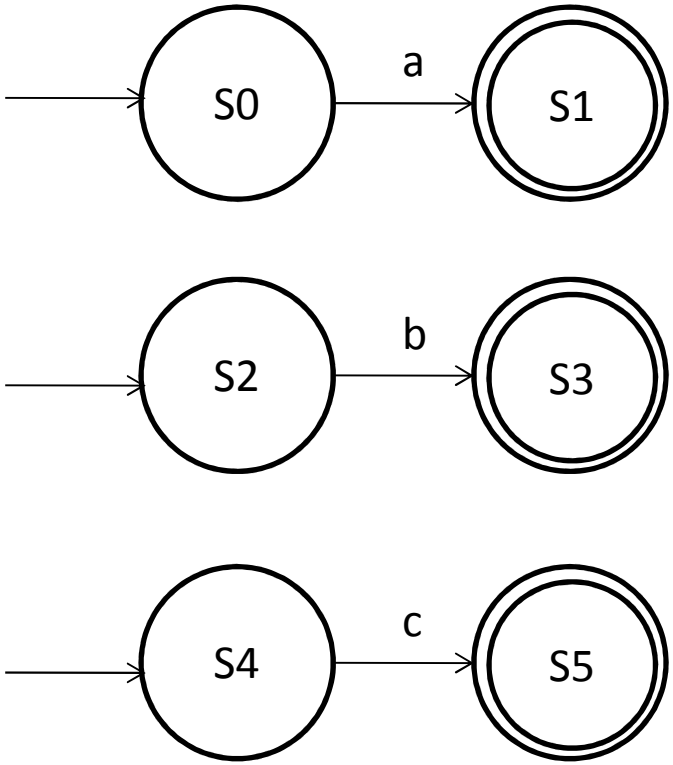


Des expressions régulières aux automates non déterministes

- 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

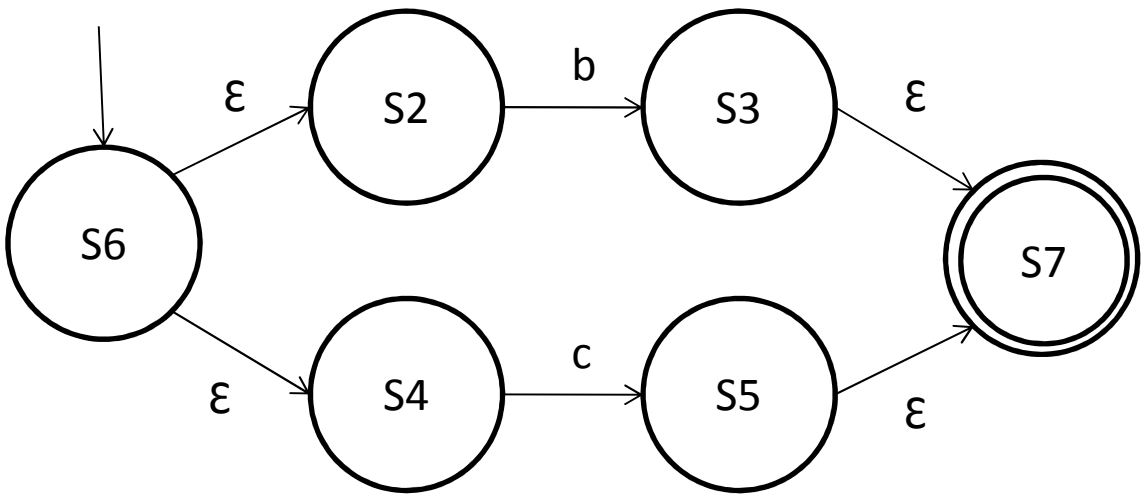


Automate reconnaissant a

Automate reconnaissant b

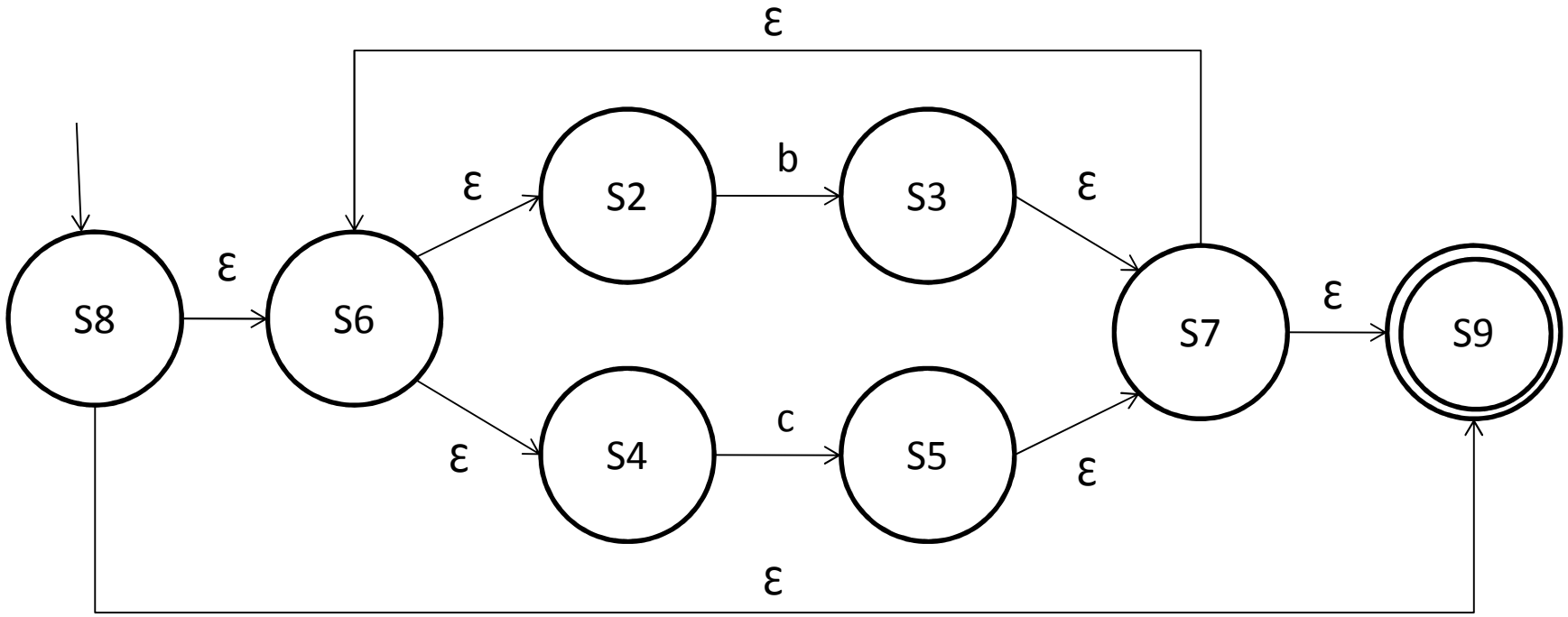
Automate reconnaissant c

Des expressions régulières aux automates non déterministes



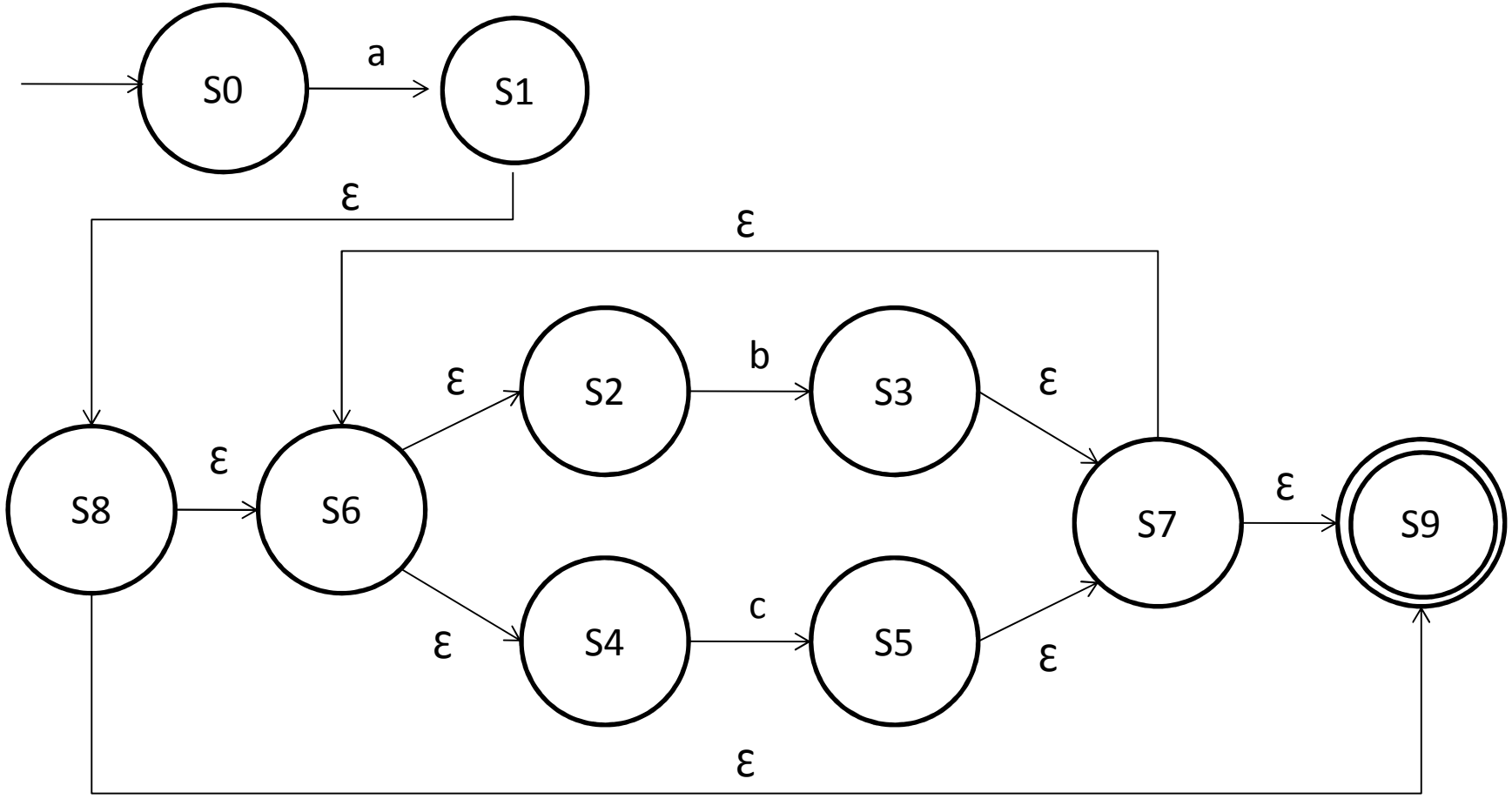
Automate reconnaissant b|c

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

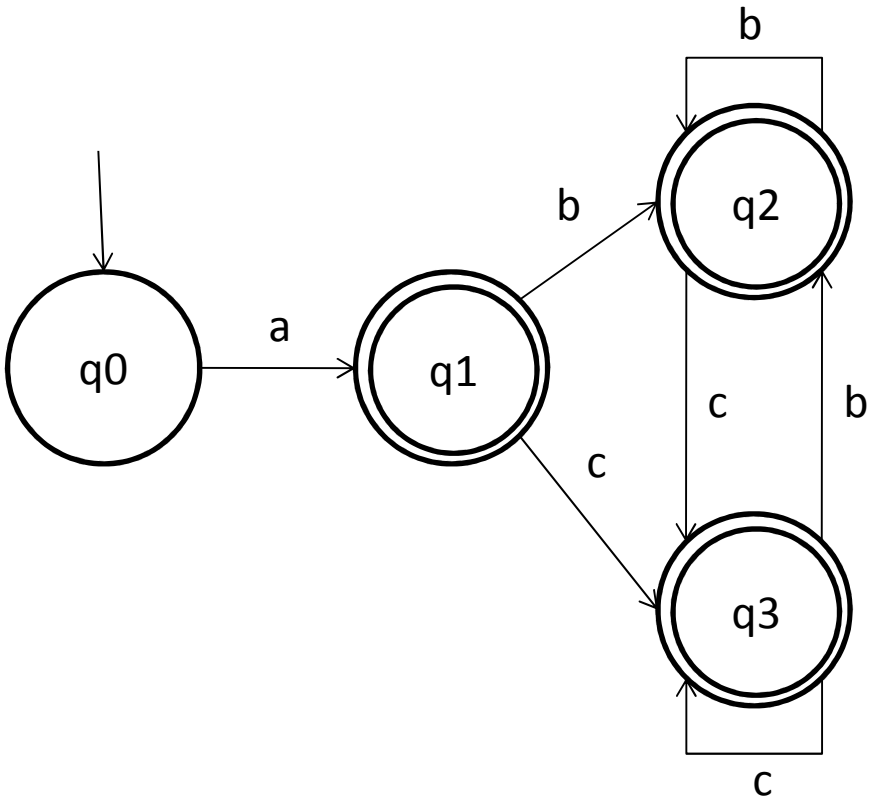
- s0 : état initial de l'automate
- Ɛ-fermeture(S) : collecte tous les états accessibles par Ɛ-transition depuis les états contenus dans S
- Δ(S,c) : collecte tous les états atteignables depuis les états de S en lisant le caractère c

```
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

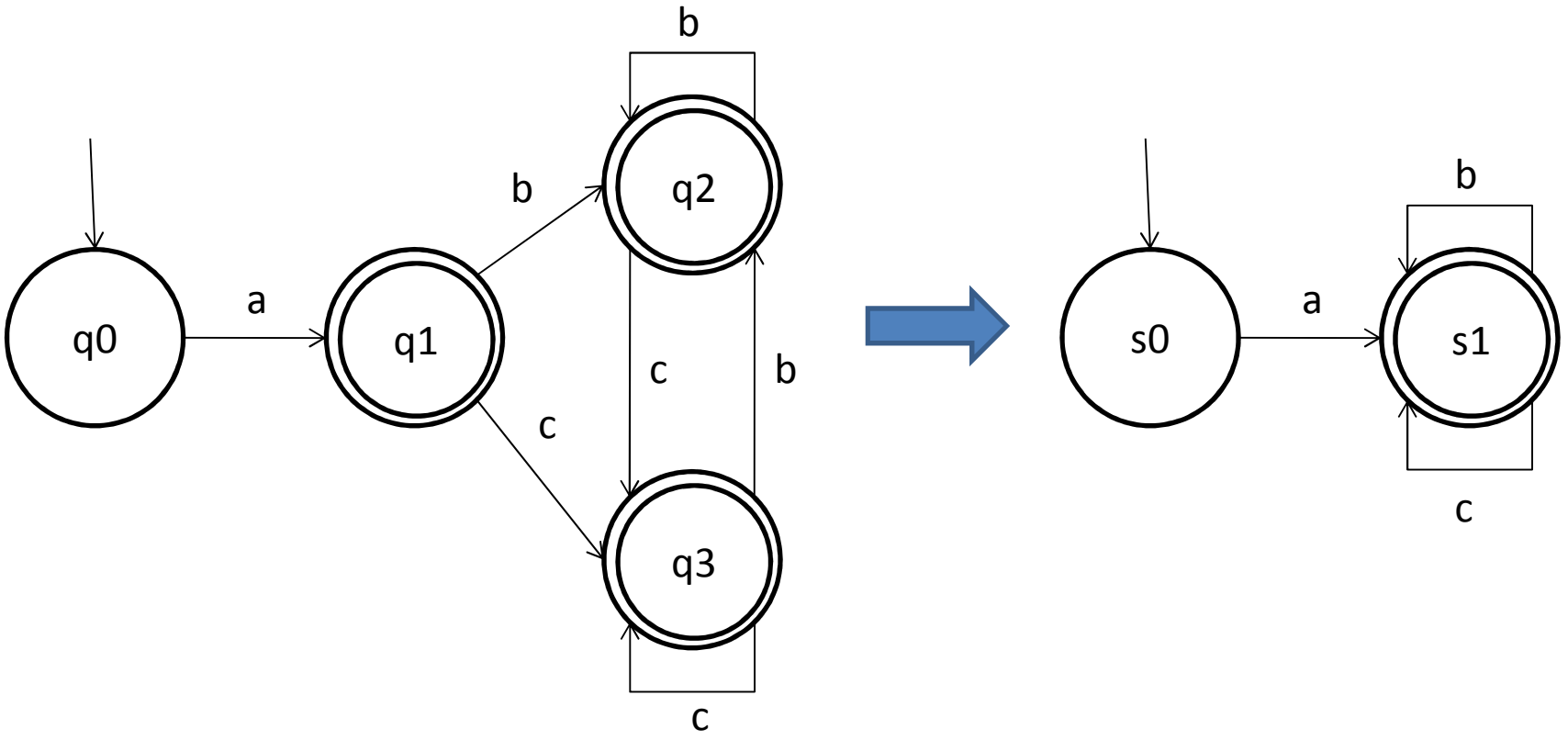
P <- { S_F, S- S_F }
Tant que P change
    T <- ensemble vide
    Pour chaque ensemble p de P
        T <- T union Partition(p)
    P <- T
Fin tant que

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)*



Analyseur lexical

- Production de l'analyseur lexical sur la base de l'automate
- Exemple de la table de transition

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

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

Les limites des langages réguliers

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

- Essayez de décrire un automate reconnaissant des expressions parenthèsées 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…

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

L’analyse syntaxique

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

- Il faut un moyen de décrire la structure d'une adresse avec ses variantes pour pouvoir l'analyser
  - i.e. décrire le langage associé à la rédaction des adresses

Les grammaires

- La grammaire d'un langage est constituée de quatre objets
  - **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)

- En fonction de la nature des règles, plusieurs classes de langages peuvent être identifiées
  - On notera **V**, l'union de N et T.

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 (1/2)

- 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…

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 $A \rightarrow w \in \mathbf{P}$
  - $m_0 \ *\rightarrow m_n$ s'il existe une suite m_k, k=0, .., n-1 et $m_k \rightarrow m_{k+1}$
- Langage engendré
  - **$L(G) = \{ m \in T^*, S \ *\rightarrow \ m \}$**
- 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

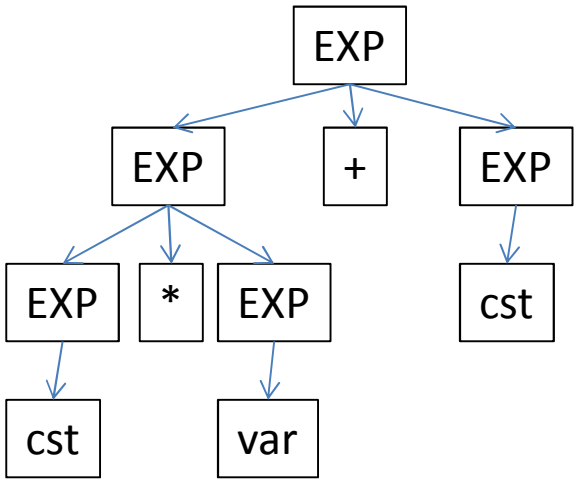
Grammaire algébrique
Arbre de dérivation syntaxique

- 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
  - }



Arbre de dérivation syntaxique pour cst * var + cst

- Exemple de dérivation de EXP
  - EXP -> EXP + EXP -> EXP * EXP + EXP
  - EXP *-> EXP * EXP + EXP
  - EXP *-> 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 parenthèsées
  - 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,
  - …

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* ";" } ;

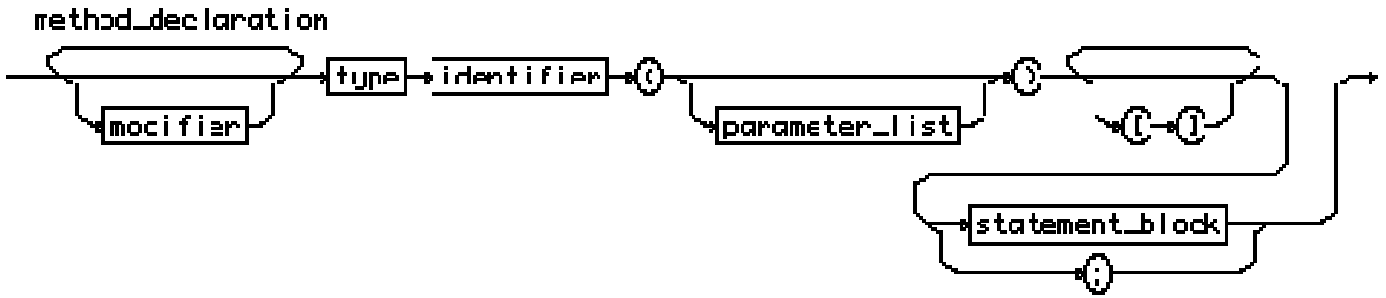
La grammaire pour les adresses
La grammaire en notation EBNF

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 numériques pouvant être parenthèsées et prenant en compte la priorité des opérateurs*
  - Opérateurs binaires: +, -, *, /
  - Opérateur unaire : -
  - Parenthèses : (,)
  - Valeurs 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)

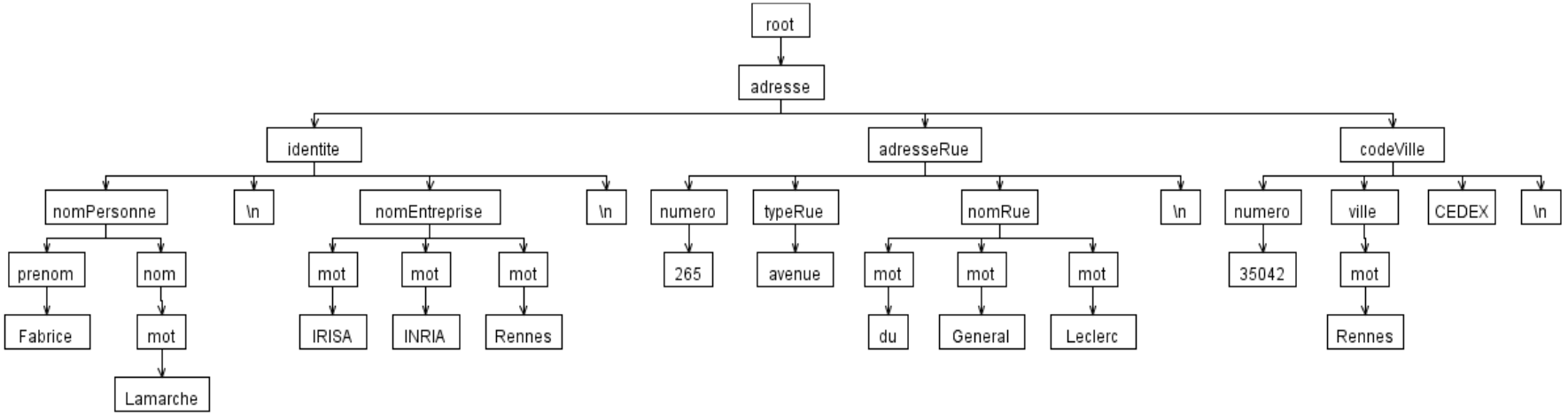
- Remarque 2 : de nos jours, grâce à certains outils (ANTLR par exemple), il n'est plus nécessaire d'effectuer cette transformation

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 syntaxiques abstraits
  - 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

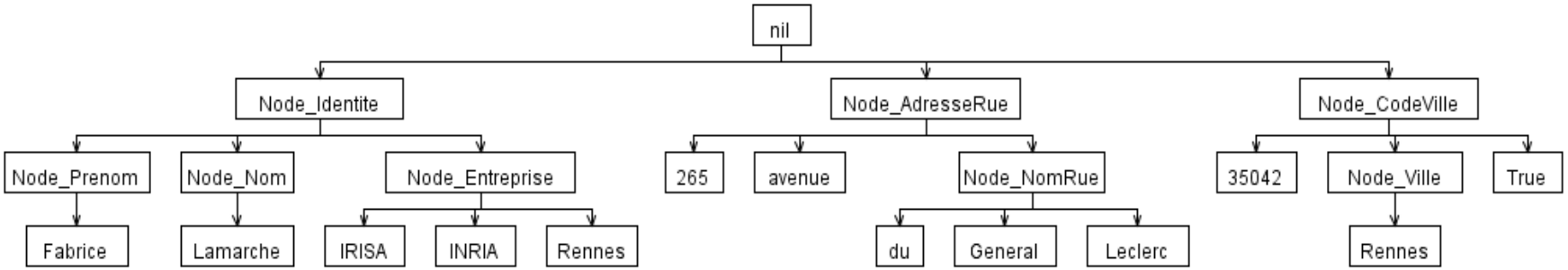
Retour sur les adresses
Arbre de dérivation syntaxique

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



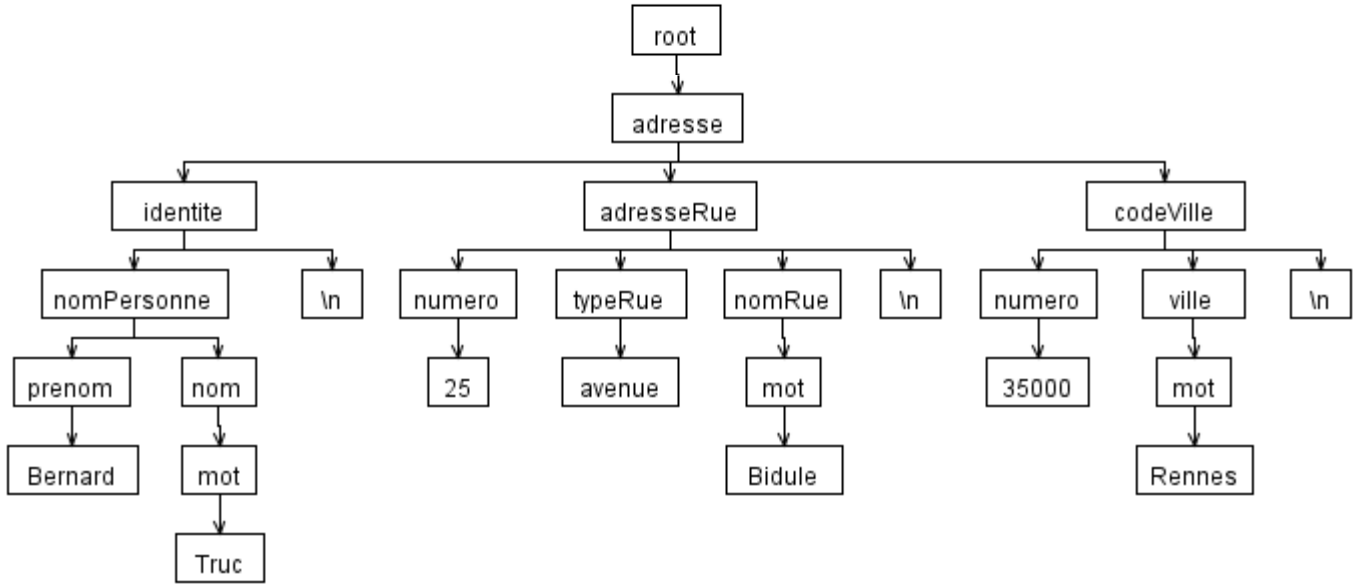
Retour sur les adresses
Arbre syntaxique abstrait

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



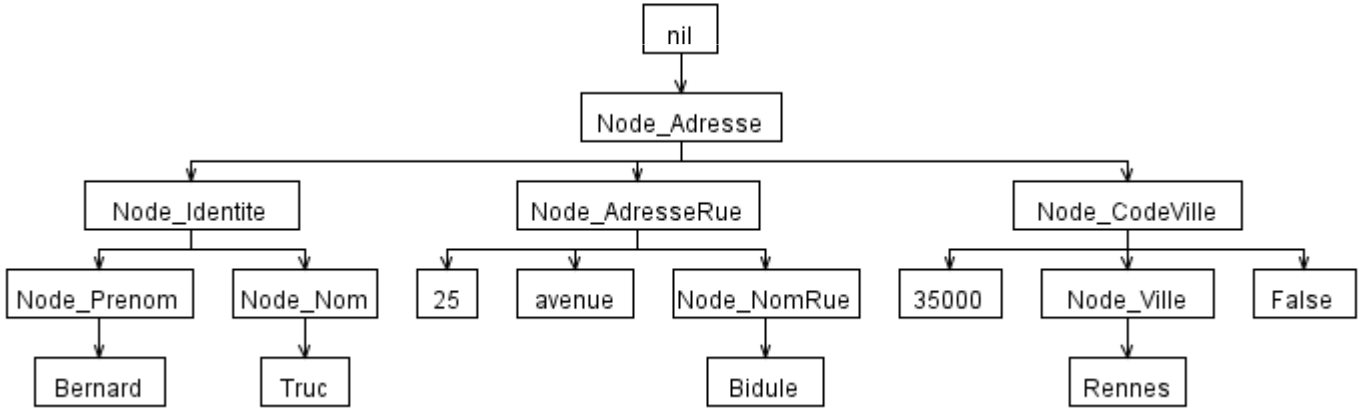
Retour sur les adresses
Arbre de dérivation syntaxique

- 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



ANTLR

Un compilateur de compilateurs

Introduction

- Un outils pour la reconnaissance de langages
  - Ecrit par Terence Parr en Java
- Plus facile d'utilisation que la plupart des outils similaires
- Propose un outils de mise au point
  - ANTLRWorks
    - Editeur graphique de grammaire et debuggeur
    - Ecrit par Jean Bovet
- Utilisé pour implémenter
  - De vrais langages de programmation
  - Des langages spécifiques à un domaine
    - Ex : fichiers de configuration
- Pour tout savoir : http://www.antlr.org
  - ANTLR + ANTRLWorks
  - Libre et open source

Introduction

- Compilateur de compilateur
  - Fichier source
    - Une description du langage
      - Grammaires EBNF
    - Éventuellement annotée
      - Association de différents types d'opérations aux règles de la grammaire
  - Produit
    - Le code pour l'analyseur lexical
    - Le code pour l'analyseur syntaxique
  - Le code produit peut être intégré dans les applications

- Supporte plusieurs langages cible
  - Java, Ruby, Pyton, C#, C et bien d'autres…

Introduction

- Supporte les grammaires LL(*)
  - Ne supporte que les grammaires récursives à droite
  - LL(k) : Regarde k lexèmes en avance pour lever des ambigüités
  - LL(*) : Regarde autant de lexèmes que nécessaire pour lever les ambigüités
- Avantage du LL(*)
  - Améliore la lisibilité des règles de la grammaire
  - Simplifie la description
  - Simplifie l'ajout de nouvelles règles

Introduction

- Trois grands types d'utilisation

1. Implémentation de « validateurs »
   - Valident si un texte en entrée respecte une grammaire
   - Pas d'action spécifique effectuée
2. Implémentation de « processeurs »
   - Valide un texte en entrée
   - Effectue les actions correspondantes
   - Les actions peuvent inclure
     - Des calculs (langages de script ou avoisinés)
     - De la mise à jour de base de données
     - L'initialisation d'une application à partir d'un fichier de configuration
     - …
3. Implémentation de « traducteurs »
   - Valide un texte
   - Le transforme dans un autre format
     - Ex : Java vers C++

Etapes de développement avec ANTLR

- Ecriture d'une grammaire

- Tester de debugger la grammaire sous ANTLRWorks
  - Offre la visualisation
    - d'arbres de dérivation syntaxique
    - d'arbres de syntaxe abstraits
  - L'exécution pas à pas de l'analyse d'un texte
  - Supporte la cible Java par défaut
    - i.e. vous pouvez inclure du code Java dans vos grammaires !

- Générer les classes depuis la grammaire
  - L'analyseur lexical et l'analyseur syntaxique

- Ecriture de l'application qui utilise les classes générées

- **Remarque** : dans les exemples et dans les TP, le langage Java sera utilisé

Structure d'un fichier de grammaire

- **grammar** *nom-grammaire* **;**
  - nom-gramaire : Nom donné à la grammaire
  - Doit correspondre au nom du fichier contenant la grammaire avec l'extension « .g »
- *options-de-grammaire* [optionnel]
  - Options permettant de décrire ce qui doit être généré par ANTLR
- *attributs-methodes-lexer-parser* [optionnel]
  - Ensemble des attributs et méthodes ajoutés aux classes d'analyseurs lexical et syntaxique
- *lexemes*
  - Ensemble des lexèmes utilisés dans la grammaire
- *regles*
  - Ensemble des règles définissant la grammaire

Les options des grammaires

- Syntaxe :

  option
  {
  name = *value* ;
  }

- Où *name* peut (principalement) prendre les valeurs suivantes
  - backtrack
    - Si backtrack = true : la grammaire est considérée comme LL(*)
  - language
    - Permet de signaler dans quel langage les classes doivent être générées
      - Ex : language = java, les classes sont générées en Java
  - output
    - La valeur associée à output définit le type de structure de donnée générée par l'analyseur syntaxique
      - Ex : output = AST, l'analyseur syntaxique génère un arbre syntaxique abstrait
  - k
    - Signale le nombre de lexèmes à utiliser pour choisir les règles
      - Ex : k = 1, signale que la grammaire est LL(1)

Ajout d'informations pour les classes et fichiers générés

- Ajout d'un en-tête au fichier généré
  - @lexer::header { … }
    - Signale des instructions à ajouter à l'en-tête du fichier contenant l'analyseur lexical
  - @parser::header { … }
    - Signale des instructions à ajouter à l'en-tête du fichier contenant l'analyseur syntaxique
  - Utile pour des inclusions de classes ou définition de package par exemple

- Ajout de méthodes / attributs
  - @lexer::members { … }
    - Fournit des méthodes / attributs définis par l'utilisateur pour l'analyseur lexical
  - @parser::members { … }
    - Fournit des méthodes / attributs définis par l'utilisateur pour l'analyseur syntaxique
  - Utile pour la factorisation de traitements pour les analyseurs ou encore pour ajouter des attributs permettant de partager des informations entre règles

Définition des lexèmes

- La syntaxe de définition des lexèmes est la suivante
  - *NomLexeme* **:** *expression-regulière* **;**
- Où
  - NomLexeme est l'identifiant d'un lexème
    - Cet identifiant DOIT commencer par une majuscule
  - expression-regulière est une expression régulière décrivant les mots associés au lexème
    - **'x'** : le caractère 'x'
    - **'a'..'t'** : l'ensemble des caractères dans l'intervalle ['a'; 't']
    - a**|**b : décrit l'alternative entre a et b
    - **.** : n'importe quel caractère
    - **~** : tous les caractères saufs ceux à droite de ~
    - **( )** : sert à regrouper des éléments
    - **+** : répétition 1 à n fois de l'élément à gauche de +
    - ***** : répétition 0 à n fois de l'élément à gauche de *
    - **?** : rend optionnel l'élément à gauche de ?

Définition des lexèmes

- Exemples
- Lexème reconnaissant un identifiant
  - ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
- Lexème reconnaissant un commentaire sur une ligne
  - COMMENT : '//' ~('\n'|'\r')* '\r'? '\n' ;
- Lexème reconnaissant un commentaire sur plusieurs lignes
  - COMMENT2 : '/*' .* '*/' ;
  - Attention : fonctionnement particulier du .*
    - Normalement, reconnait toute séquence de caractère
      - On ne s'arrête pas avant la fin du fichier
    - Mais avec ANTLR
      - On sort de la séquence dès que ce qui vient après est reconnu
      - Donc, dans l'exemple, dès que '*/' est trouvé
      - Pratique ☺

Définition des lexèmes

- Par défaut, l'analyseur lexical possède deux canaux de sortie pour les lexèmes
  - DEFAULT : le canal de sortie par défaut. Le lexème est consommé par l'analyseur syntaxique
  - HIDDEN : lorsqu'un lexème est sorti sur ce canal, il n'est pas traité par l'analyseur syntaxique
    - Utile pour les commentaires par exemple

- Lorsqu'un lexème est reconnu, une action peut-être effectuée avant que ce dernier soit transmis à l'analyseur syntaxique
  - Syntaxe :

    Lexeme **:** *expression-reguliere* **{** actions **} ;**

Définition des lexèmes

- Exemples d'actions
  - $channel = HIDDEN ;
    - Provoque la sortie sur le canal HIDDEN. Le lexème n'est donc pas traité par l'analyseur syntaxique
  - skip() ;
    - Le lexème qui vient d'être reconnu est simplement omis

- Retour sur l'exemple des commentaires :

```
COMMENT :  '//' ~('\n'|'\r')* '\r'? '\n'  {$channel=HIDDEN;}
   |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
   ;
```

  - Force les commentaires à être ignorés

Définition des règles

- Les règles sont décrites via la syntaxe EBNF inspirée par les expression régulières

- Syntaxe :

  regle : *corps-regle* ;

- Où
  - *regle* est le nom de la règle
    - Commence forcément par une minuscule
  - *corps-regle* est une expression décrivant ce qui doit être reconnu

Définition des règles

- Syntaxe du corps de la règle
  - *identifiant*
    - Si commence par une majuscule, réfère à un lexème préalablement déclaré qui doit être reconnu
    - Si commence par une minuscule, réfère à une règle qui doit être reconnue
  - ‘chaine’ : reconnait le mot ‘chaine’
    - Utilisé pour les mots clefs du langage par exemple
    - Ajoute « automatiquement » un lexème prioritaire
  - a **|** b : reconnait soit la sous-règle a, soit la sous règle b
  - a* : reconnait 0 à n fois la sous-règle a
  - a+ : reconnait 1 à n fois la sous-règle a
  - a? : rend optionnelle la reconnaissance de la sous règle a
  - () : sert à regrouper des sous-règles

Définition de règles

- Exemple de définition de règles

// Declaration de méthode à la C++

methodDecl : (*primitiveType* | *ID*) *ID* '(' (*varDecl* (',' *varDecl*)*)? ')' 'const'? ';'

// Types primitifs autorisés

primitiveType : 'char' | 'int' | 'float' | 'double' | 'bool' ;

// Déclaration de variable / paramètre

varDecl : (*primitiveType* | *ID*) *ID* ;

Définition de règles

- Vu d'ANTLR, une règle correspond à une méthode de l'analyseur syntaxique

- Elle peut posséder
  - Des paramètres
  - Une valeur de retour
  - Du code à exécuter
    - Au début de la règle
    - En fin de règle
    - À n'importe quel stade de l'analyse (en cours de règle)

Définition et utilisation des règles

- Syntaxe des règles

  regle **[** Type1 param1, Type2 param2**] returns [**TypeR1 id2, TypeR2 id2**]**
  @init { /* code en initialisation de règle */ }
  @after { /* code en fin de règle */ }
  : *corps-règle* { /*code en cours de règle */ } ;

- Manipulation des paramètres et valeurs de retour
  - $param1 : valeur du paramètre1
  - $id1 : valeur de retour

- Utilisation des paramètres et valeurs de retour
  - *regle2* : ret=*regle*[p1,p2] ;
  - p1 et p2 sont les paramètres
  - ret designe une structure contenant les valeurs de retour de *regle*
  - $ret.id1 et $ret.id2 désignent les deux valeurs de retour de *regle*

Exemple : la calculatrice en notation polonaise inverse en ~20 lignes

```
grammar calculatrice;

@parser::members
{ java.util.Stack<Integer> m_stack = new java.util.Stack() ; }

INT :  '0'..'9'+ ;

WS  :  (' ' | '\t' | '\r' | '\n') { $channel = HIDDEN ; } ;

expression
@after
{ System.out.println("Value on top: "+ m_stack.pop()) ; } :
        ( v = value { m_stack.push($v.val) ; }
        | v = operation[m_stack.pop(), m_stack.pop()] { m_stack.push($v.val) ; }
        )* ';' ;

value returns [Integer val] : a=INT { $val = Integer.valueOf($a.text) ; } ;

operation [Integer v1, Integer v2] returns [Integer val] :
        ('+'  {$val = $v1 + $v2 ; }
        | '-' {$val = $v1 - $v2 ; }
        | '*' {$val = $v1 * $v2 ; }
        | '/' {$val = $v1 / $v2 ; }
        ) ;
```

Exemple : la calculatrice en notation polonaise inverse

Evaluation interactive des expressions

```
import java.io.*;
import java.util.Scanner;
import org.antlr.runtime.*;

public class Processor {
	public static void main(String[] args) throws IOException, RecognitionException {
		new Processor().processInteractive();
	}

	private void processInteractive() throws IOException, RecognitionException  {
		Scanner scanner = new Scanner(System.in);
		while (true) {
			System.out.print("calculatrice> ");
			String line = scanner.nextLine().trim();
			if ("quit".equals(line) || "exit".equals(line)) break;
			Integer result = processLine(line);
			System.out.println("Resultat: "+result) ;
		}
	}

	private Integer processLine(String line) throws RecognitionException {
		calculatriceLexer lexer = new calculatriceLexer(new ANTLRStringStream(line));
		calculatricParser tokenParser = new calculatriceParser(new CommonTokenStream(lexer));
		calculatriceParser.expression_return parserResult = tokenParser.expression(); // start rule method
		return parserResult.resultat ;
	}
}
```

Exemple : la calculatrice en notation polonaise inverse

Evaluation des expressions à partir d'un fichier

```
import java.io.*;
import java.util.Scanner;
import org.antlr.runtime.*;

public class Processor {
    public static void main(String[] args) throws IOException, RecognitionException {
        if (args.length == 1) { // name of file to process was passed in
            Integer result = new Processor().processFile(args[0]);
            System.out.println(result) ;
        }
        else { // more than one command-line argument
            System.err.println("usage: java Processor file-name");
        }
    }

    private Integer processFile(String filePath) throws IOException, RecognitionException  {
        FileReader reader = new FileReader(filePath) ;
        calculatriceLexer lexer  =  new calculatriceLexer(new ANTLRReaderStream(reader));
        calculatriceParser parser  = new calculatriceParser(new CommonTokenStream(lexer)) ;
        calculatriceParser.expression_return parserResult = parser.expression() ;
        return parserResult.resultat ;
    }
}
```

Attributs des règles

- Les règles possèdent des attributs accessibles via
  - $nomRègle.attribut, accès direct à l'attribut
  - a=nomRègle et $a.attribut, récupération valeur de retour et accès à l'attribut

- Attributs prédéfinis
  - $r.text : String, le texte reconnu par la règle
  - $r.start : Token, le premier token à reconnaitre
  - $r.stop : Token, le dernier token à reconnaitre
  - $r.tree : Object, l'arbre AST généré par la règle

- Attributs correspondant aux valeurs de retour
  - Si valRet est une valeur de retour
  - $r.valRet permet d'accèder à la valeur de retour de la règle r

Attributs des tokens

- Les tokens possèdent des attributs accessibles via
  - $NomToken.attribut, accès direct à l'attribut
  - a=NomToken et $a.attribut, récupération valeur de retour et accès à l'attribut

- Attributs prédéfinis
  - $t.text : String, le texte associé au token t
  - $t.line : int, le numéro de ligne du premier caractère du token (commence à 1)
  - $t.pos : int, la position de ce caractère sur la ligne (commence à 0)

Le code généré

- Lors de la compilation, deux fichiers sont générés :
  - Un « lexer » (analyseur lexical)
  - Un « parser » (analyseur syntaxique)

- Ces fichiers contiennent une classe associée à chaque analyseur

- Soit une grammaire nommée Grammar
  - La classe GrammarLexer contiendra le « lexer »
  - La classe GrammarParser contiendra le « parser »

Le code généré pour l'analyseur syntaxique

- Les valeurs de retour des règles
  - Une classe générée par règle
  - Cette classe contient un attribut publique par valeur de retour de règle
  - Une référence vers l'arbre de syntaxe abstrait (si présent)
    - Accessible via getTree()
  - Nom normalisé : nomRegle_return

- Une règle = une méthode
  - Paramètres équivalents aux paramètres de la règle

- Les lexèmes (tokens) sont listés
  - Constantes de type int
  - Correspondent au « type » du lexème
  - Portent le nom du lexème
  - Permettra de faire la correspondance pour les arbres de syntaxe abstraits

Arbres de syntaxe abstraits

- ANTLR propose des fonctionnalités pour la construction d'arbres de syntaxe abstraits

- Objectifs d'un arbre de syntaxe abstrait
  - Stocker les lexèmes pertinents
  - Encoder la structure grammaticale sans information superflue
  - Etre facile à manipuler

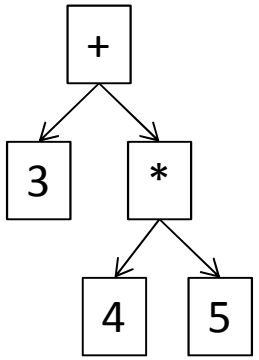
  ➢ *Il s'agit d'une structure encodant la sémantique du langage et simplifiant les traitements futurs*

Structure des arbres de syntaxe abstraits

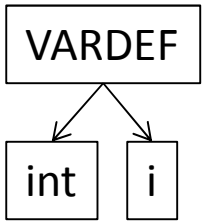
- En ANTLR un arbre de syntaxe abstrait :
  - Est composé de nœuds
  - Ces nœuds peuvent avoir de 0 à n fils
  - Les étiquettes associées aux nœuds sont des lexèmes
  - Rq : à partir des lexèmes, possibilité de récupérer le texte associé et donc de l'interpréter

- Construction d'un AST
  - ^(Token son1 son2)
    - Token est l'étiquette de l'AST
    - son1 et son2 sont des sous arbres
    - Rq : si son1 et / ou son2 sont des lexèmes, ils sont automatiquement convertis en AST
  - ^(ast1 son1 son2)
    - Construit un AST en ajoutant les sous arbres son1 et son2 à l'AST désigné par ast1

Exemples d'arbres de syntaxe abstraits



- Arbre correspondant à l'expression 3+4*5
- Encode la priorité des opérateurs
- Notation en ANTLR
  - ^('+' 3 ^('*' 4 5))



- Arbre correspondant à la déclaration d'une variable i de type int
- VARDEF est un lexème « imaginaire »
  - Non présent dans la grammaire
  - Ajouté pour donner une sémantique
- Notation en ANTLR
  - ^(VARDEF 'int' i)

Construction d'arbres de syntaxe abstraits à partir de règles de réécriture

- Manière recommandée de construire des arbres de syntaxe abstraits (AST)

- Notation :

  rule : « alt1 » **->** « créer-ceci-suivant-alt1 »
  | « alt2 » **->** « créer-ceci-suivant-alt2 »
  …
  | « alt2 » **->** « créer-ceci-suivant-alt2 »
  ;

- Remarques :
  - les règles peuvent générer 1 AST ou une liste d'AST
  - Par défaut : génération d'une liste d'AST correspondant aux éléments de la règle (si omission de l'opérateur ->)

Exemples simples

- stat: 'break' ';' -> 'break‘
  - Renvoie un AST composé d’un nœud avec l’étiquette ‘break’
  - Supprime le ‘;’

- decl : 'var' ID ':‘ type -> type ID ;
- type : ‘int’ | ‘float’
  - Renvoie deux AST contenant le type (int ou float) et l’identifiant
  - L’utilisation de type dans la production réfère à l’AST produit par la règle *type*
  - Rq : réordonne les éléments lus

- decl : ‘var’ ID ‘:’ type -> ^(VARDEF type ID)
  - Renvoie un AST avec comme racine le lexème VARDECL
  - Et comme fils l’AST renvoyé par type et un AST contenant le token ID

Les lexèmes ‘imaginaires’

- La règle précédente :
  - decl : ‘var’ ID ‘:’ type -> ^(VARDEF type ID)
  - Utilise un lexème nommé VARDEF
  - Ce lexème
    - n’a pas d’expression régulière associée
    - n’est pas présent dans la grammaire
    - Est ajouté pour donner de la **sémantique** à une opération

- Déclaration des lexèmes ‘imaginaires’
  - Sous la description des options, ajouter :

```
tokens
{
        VARDEF ;
        IM2 ; // Un autre lexeme imaginaire…
}
```

AST et règles de réécriture

- Collecte d'une suite d'éléments
  - list : ID (',' ID)* -> ID+
    - Retourne la liste des AST correspondant aux identifiants ID
  - formalArgs : formalArg (',' formalArg)* -> formalArg+
    - Retourne la liste des AST retournés par la règle formalArg
  - decl : 'int' ID (',' ID)* -> ^('int' ID+) ;
    - Retourne un AST avec 'int' comme étiquette et un ensemble de fils correspondant à la liste des identifiants
  - compilationUnit : packageDef? importDef* typeDef+
    -> ^(UNIT packageDef? importDef* typeDef+) ;
    - Retourne un AST ayant comme étiquette UNIT
    - Un fils optionnel contenant l'AST retourné par *packageDef*
    - Une suite de 0 à n fils contenant les AST retournés par *importDef*
    - Une suite de 1 à n fils contenant les AST retournés pas *typeDef*

AST et règles de réécriture

- Duplication de nœuds et d'arbres
  - dup : INT -> INT INT ;
    - Retourne deux AST désignant le même lexème
  - decl : 'int' ID (',' ID)* -> ^('int' ID)+ ;
    - Retourne 1 à n AST ayant 'int' pour étiquette et un AST contenant le token ID comme fils
  - decl : type ID (',' ID)* -> ^(type ID)+ ;
    - Retourne 1 à n AST en duplicant l'AST retourné par type et en ajoutant ID comme fils
  - decl : modifier? type ID (',' ID)* -> ^(type modifier? ID)+ ;
    - Retourne 1 à n AST comme dans la règle précédente mais ajoute l'AST retourné par *modifier* comme sous arbre si celui-ci est présent

AST et règles de réécriture

- Choix des arbres en fonction de conditions
  - Possibilité d'ajouter des conditionnelles en tête des règles de réécriture
  - Choix de la règle en fonction du résultat de la conditionnelle

```
a[int which] : ID INT -> {which==1}? ID
                      -> {which==2}? INT
                      -> // Rien : pas d'arbre
;
```

AST et règles de réécriture

- Possibilité d'utiliser des labels dans les règles de réécriture
  - decl : 'var' i=ID ':' t=type -> ^(VARDEF $t $i)
    - i désigne la valeur de retour de ID
    - t désigne la valeur de retour de t
    - $i et $t sont utilisés pour construire l'arbre
  - prog: main=method others+=method*
    -> ^(MAIN $main) $others* ;
    - main désigne la valeur de retour de method
    - others est une liste des valeurs (opérateur +=) de retour de method*
    - Produit une liste d'AST, le premier ayant pour racine MAIN les autres les AST correspondant à method*

AST et règles de réécriture

- Règles de réécriture dans les sous-règles

```
Ifstat :
'if' '(' equalityExpression ')' s1=statement
('else' s2=statement  -> ^('if' ^(EXPR equalityExpression) $s1 $s2)
 |                    -> ^('if' ^(EXPR equalityExpression) $s1)
)
;
```

  - Retourne deux arbres différents en fonction de la présence de ‘else’
  - La présence du ‘else’ indique la règle de réécriture à utiliser

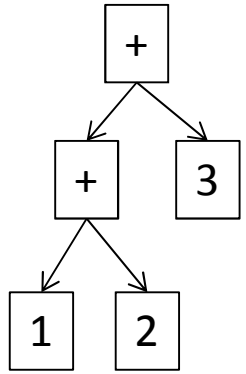
AST et règles de réécriture

- Parfois une simple réécriture n'est pas suffisante
  - Exemple

    expr : INT ('+' INT)*

    À l'analyse de 1+2+3 nous souhaiterons générer

    ^('+' ^('+' 1 2) 3)



- Il faut construire l'arbre itérativement
  - expr : (INT -> INT) ('+' i=INT -> ^('+' $expr $i))* ;
    - Où $expr réfère au dernier AST construit dans la règle expr

Comment utiliser un AST

- Par défaut, ANTLR utilise
  - **org.antlr.runtime.tree.CommonTree** pour représenter les AST
  - **org.antlr.runtime.Token** pour représenter les lexèmes

- Lorsque les AST sont générés, les types de retour des règles contiennent une méthode getTree() renvoyant l'AST
  - Attention l'arbre est renvoyé via une référence sur Object
  - (CommonTree)valRet.getTree() pour convertir en CommonTree
  - Pas très beau mais possibilité d'inclure ses propres classes d'AST
    - Cf. livre sur ANTLR

    The Definitive ANTLR Reference: Building Domain-Specific Languages
    Terence Parr, ISBN: 978-0-9787-3925-6

org.antlr.runtime.tree.CommonTree

- Manipulation du lexème associé à l'arbre
  - Token getToken()
    - Récupération du lexème
  - int getType()
    - Récupération du type du lexème

- Manipulation des fils
  - int getChildCount()
    - Nombre de fils
  - Tree getChild(int i)
    - Fils à partir de son index
  - List getChildren()
    - Liste des fils
  - Tree getFirstChildWithType(int type)
    - Le premier fils d'un type donné

Conclusion

- ANTLR est un outils puissant
  - Description des grammaires LL(*) en EBNF
  - Paramètres et valeurs de retour pour les règles
  - Association de code aux règles
  - Gestion des arbres de syntaxe abstraits
  - Règles de réécriture
  - Simple d'utilisation étant donné sa puissance… ☺
- Ne faites plus d'analyse à la main
  - ANTLR est votre ami…
  - Fichiers de configuration, lecture de fichiers structurés
  - Interaction avec des langages de script
- Utilisez les arbres de syntaxe abstraite
  - Ex : multiplicité des langages de description de géométrie pour la 3D
  - Plusieurs analyseurs mais 1 seul AST… i.e. une seule phase de génération par analyse de l'AST

Pour aller plus loin…

The Definitive ANTLR Reference: Building Domain-Specific Languages

Terence Parr, ISBN: 978-0-9787-3925-6

- Une description complète de ANTLR
- Le contenu détaillé de cette partie du cours
- Autres informations :
  - Gestion des erreurs
  - Utilisation des patrons pour la réécriture
  - …

Introduction à XML

Historique

- 1986 : SGML (Standard Generalized Markup Language)
  - Séparation entre la structure logique d'un document et sa mise en page (définie par des feuilles de style)
- 1991 : HTML (HyperText Markup Language)
  - Format de données conçu pour représenter des pages web
  - Dérivé de SGML
- 1998 : version 1.0 d'XML (eXtended Markup Language)
  - Langage extensible de balisage
  - Stockage / transfert de données structurées en champs arborescents
- 1999 : Redéfinition d'HTML 4.0 en XHTML à travers XML
  - Se fonde sur la syntaxe de XML plutôt de de SGML
  - Syntaxe plus rigoureuse
- 2004 : version 1.1 d'XML

Caractéristiques essentielles d'XML

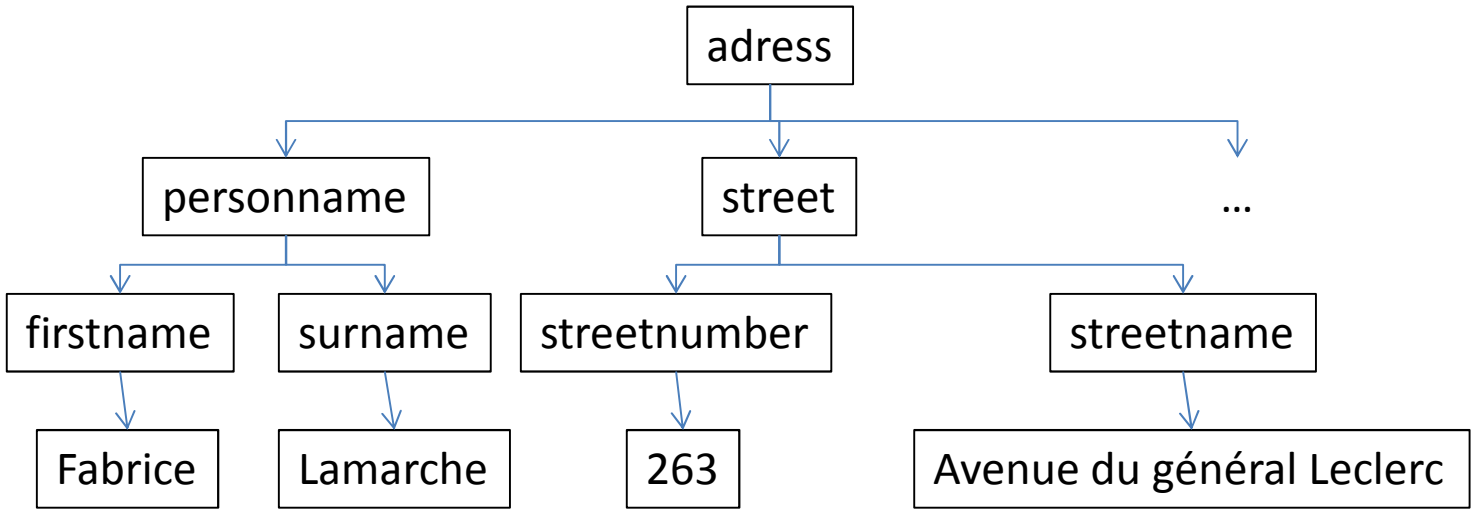
- Simplicité, universalité et extensibilité
- Format texte avec gestion de caractères spéciaux
- Structuration forte de l'information
- Séparation stricte entre contenu et présentation
- Modèles de documents (DTD et XML-Schémas)
  - Grammaires définissant la structure d'un document XML
- Modularité des modèles
- Validation du document par rapport au modèle
- Format libre
- Nombreuses technologies développées autours d'XML

Exemple : une adresse

```
<address>
    <personname>
        <firstname>Fabrice</firstname>
        <surname>Lamarche</surname>
    </personname>
    <street>
        <streetnumber>263</streetnumber>
        <streetname>Avenue du général Leclerc</streetname>
    </street>
    <zipcode>35042</zipcode>
    <city>Rennes</city>
    <email>fabrice.lamarche@irisa.fr</email>
</address>
```

Utilisation des balises pour
1) Structurer l'information
2) donner une sémantique à l'information contenue

Exemple : représentation aborescente



Composition d'un document XML

- Un document est composé de 3 parties
  - Prologue
    - Contient des déclaration facultatives
      - Type d'encodage des caractères
      - DTD ou Schema
  - Corps du document
    - Le contenu même du document
  - Commentaires
    - Peuvent être partout dans le document
    - Syntaxe : **<!–** commentaire **-->**
- Ex:

<?xml … ?>
…
<root-element>
…
<root-element/>

Prologue

Corps

Le prologue

- Constitué de
  - Entête XML (obligatoire)
    - Version de XML
    - Encodage des caractères
    - Savoir si le document référence des fichiers externes
  - La DTD (Document Type Definition) (optionnelle)
    - Définit la grammaire régissant la structure du document
    - Optionnelle
  - Des instructions de traitement (optionnelles)
    - Non explicité dans ce cours

Le prologue

- En-tête XML

  <?xml version="…" encoding= "..." standalone="…"/>

  – Version peut prendre les valeurs 1.0 ou 1.1
  – Encoding peut prendre les valeurs
    - US-ASCII, ISO-8859-1, UTF-8 et UTF-16
  – Standalone peut prendre les valeurs
    - « yes » ou « no » (en fonction du référencement de fichiers externes)
    - Optionel et valeur par défaut à « no »
- Ex:
  – <?xml version="1.0"?>
  – <?xml version='1.0' encoding='UTF-8'?>
  – <?xml version="1.1" encoding="iso-8859-1" standalone="no"?>

Corps du document

- Organisation hiérarchique
- L'unité de cette organisation s'appelle l'élément
- Chaque élément peut contenir
  - Du texte simple
  - D'autres éléments
  - Un mélange des deux

L’élément

- Un élément est formé
  - D’une balise ouvrante : **<**balise**>**
  - D’une balise fermante correspondante : **</**balise**>**
  - D’attributs écrits sous la forme
    - nomAttrib= "valeur "
    - Ex : **<**balise attrib1= "valeur1" attrib2= "valeur2"**>**
      L’élément balise a deux attributs : attrib1 et attrib2
  - D’un contenu
    - Texte, autres éléments, commentaires…
  - Un élément ne possédant pas de contenu possède une notation raccourcie:
    - <balise/> : élément sans contenu
    - <balise att= "v1"/> : élément sans contenu mais avec attribut(s)

Attribut vs contenu

- Utilisation du contenu du nœud pour donner les informations

```
<personname id="I666">
    <firstname>Gaston</firstname>
    <surname>Lagaffe</surname>
</personname>
```

- Utilisation des attributs

```
<personname id="I666" firstname="Gaston" surname="Lagaffe"/>
```

- Pas de règles pour savoir si une information « atomique » doit être mise en attribut ou dans le contenu du nœud

Structuration du document

- Choses non contraintes par XML
  - Noms des balises décrivant les éléments
  - Noms d'attributs associés aux éléments
  - Contenus des éléments
    - Texte, sous éléments..

- Cependant, besoin
  - d'une structure organisée
  - de définir des contraintes
    - Attributs des éléments
    - Sous éléments acceptés
    - Etc…

➢ Utilisation des schémas de données

Les schémas de données

Les DTD

Un schéma de données = un dialecte

- Un schéma spécifie un dialecte XML

- Pour les applications : une *grammaire*
  - vérifier la conformité d'un document bien formé vis à vis du dialecte considéré

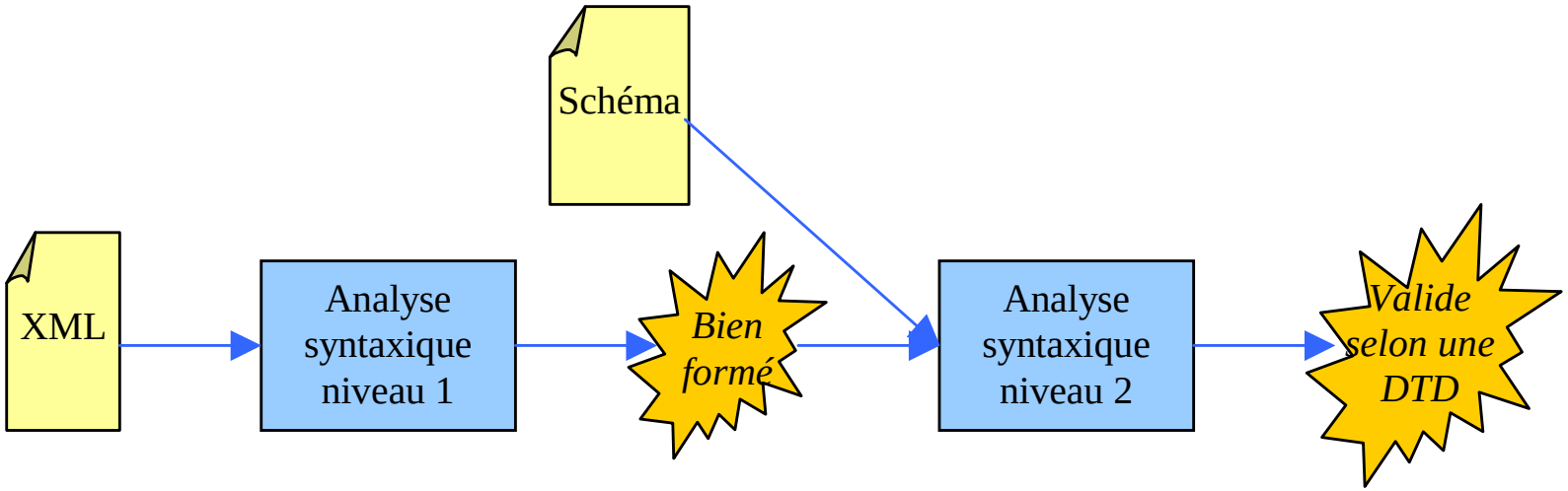
- Pour les utilisateurs : une *spécification*
  - spécifier, documenter, s'échanger un dialecte

Relation de conformité

« Est conforme à »

Programmation	Définition de classe	Instance d'objet
SGBD	Définition de table	Intance de table
XML	Schéma de données	Instance de document

Validation de document

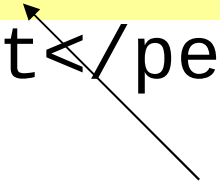


Les DTD

- DTD : Document Type Definition
- Une grammaire qui exprime les contraintes sur la structure d'un document
- Sa spécification et celle d'XML n'en font qu'un
- DTD où et comment ?
  - une description interne au document lui-même
  - un document séparé, référencé par le document lui-même

Exemple de document avec DTD

```
<?xml version="1.0"?>
<!DOCTYPE personne [
   <!ELEMENT personne (#PCDATA)>
]>
<personne>Jacques Dupont</personne>
```



Autres usages d’une DTD

- Définir des macros (entités)
- Modularité (éclatement physique d'une DTD)
- Donner des valeurs par défaut aux attributs
- Définir des contraintes d'unicité d'identification
  - attributs ID
- Définir des contraintes de référence
  - attributs IDREF

Localisation de la DTD

- Dans l'entête du document XML
- Trois méthodes
  - DTD interne dans le document

```
<!DOCTYPE nom [ … déclarations … ]>
```

  - DTD externe dans un fichier (pour la partager)

```
<!DOCTYPE nom SYSTEM "Fichier/URL">
<!DOCTYPE nom PUBLIC "identifiant" "Fichier">
```

  - DTD Mixte (interne/externe)

```
<!DOCTYPE élémentRacine SYSTEM "URL" [
 … déclarations …
]>
```

Déclaration d'élément

- Syntaxe

  `<!ELEMENT` *`nom modèleDeContenu`* `>`

  – *nom* : nom de l'élément

  – *modèleDeContenu* : expression définissant le contenu autorisé dans l'élément

    - Expression régulière sur les sous éléments

Sémantique	*Opérateur*
Enchaînement	... , ...
Choix	... | ...
Zéro ou 1	...?
Zéro ou plus	...*
Un ou plus	...+
Groupe	(...)
un et un seul	nom de l'élément

Exemples

```
<!-- choix parmi deux -->
<!ELEMENT mobile (train | avion)>

<!-- séquence d'éléments -->
<!ELEMENT adresse (numéro, voie, ville)>

<!-- Au moins un élément -->
<!ELEMENT carnetDAdresse (carteDeVisite+)>

<!-- séquence d'éléments dont le dernier peut
     apparaître 0 ou plusieurs fois -->
<!ELEMENT carteDeVisite (prénom, nom, organisme,
   adresse, tel*)>
```

5 types d'éléments

- élément vide

  `<!ELEMENT nom EMPTY>`
- élément avec contenu indifférent

  `<!ELEMENT nom ANY>`
- élément avec du texte seulement comme contenu

  `<!ELEMENT nom (#PCDATA)>`
- élément avec des éléments seuls comme contenu

  `<!ELEMENT nom (nom1 | nom2?)>`

  `<!ELEMENT nom (nom1 , (nom2 | nom3)*)>`
- élément mixte

  `<!ELEMENT nom (#PCDATA | nom1 | nom2)*>`

Déclaration d'attributs d'éléments

- Associer une liste d'attributs à un élément
- Schéma de la déclaration

```
<!ATTLIST nomElément
     nomAttribut  type  Contrainte
     ...
     nomAttribut  type  Contrainte
>
```

- Exemple

```
<!ATTLIST ville
     nom CDATA  #IMPLIED
     id  ID     #REQUIRED
>
```

Types d’attributs

Description	*Type*
Texte	CDATA
Type énuméré	(v1|v2|...)
Définition d'identifieur unique	ID
Référence à identifieur	IDREF
un mot sans espace	NMTOKEN
nom d'entités externes	ENTITY

Listes de valeurs

Type simple	Type liste
ID	IDS
IDREF	IDREFS
NMTOKEN	NMTOKENS
ENTITY	ENTITIES

Exemple : `<a ids="A123 A125 B12"/>`

Contraintes

Type de contrainte	Expression de la contrainte
Val par défaut d'un type énuméré	`'val'`
Val par défaut	`#DEFAULT val`
Obligatoire	`#REQUIRED`
Non obligatoire	`#IMPLIED`
Valeur constante	`#FIXED val`

Type énuméré

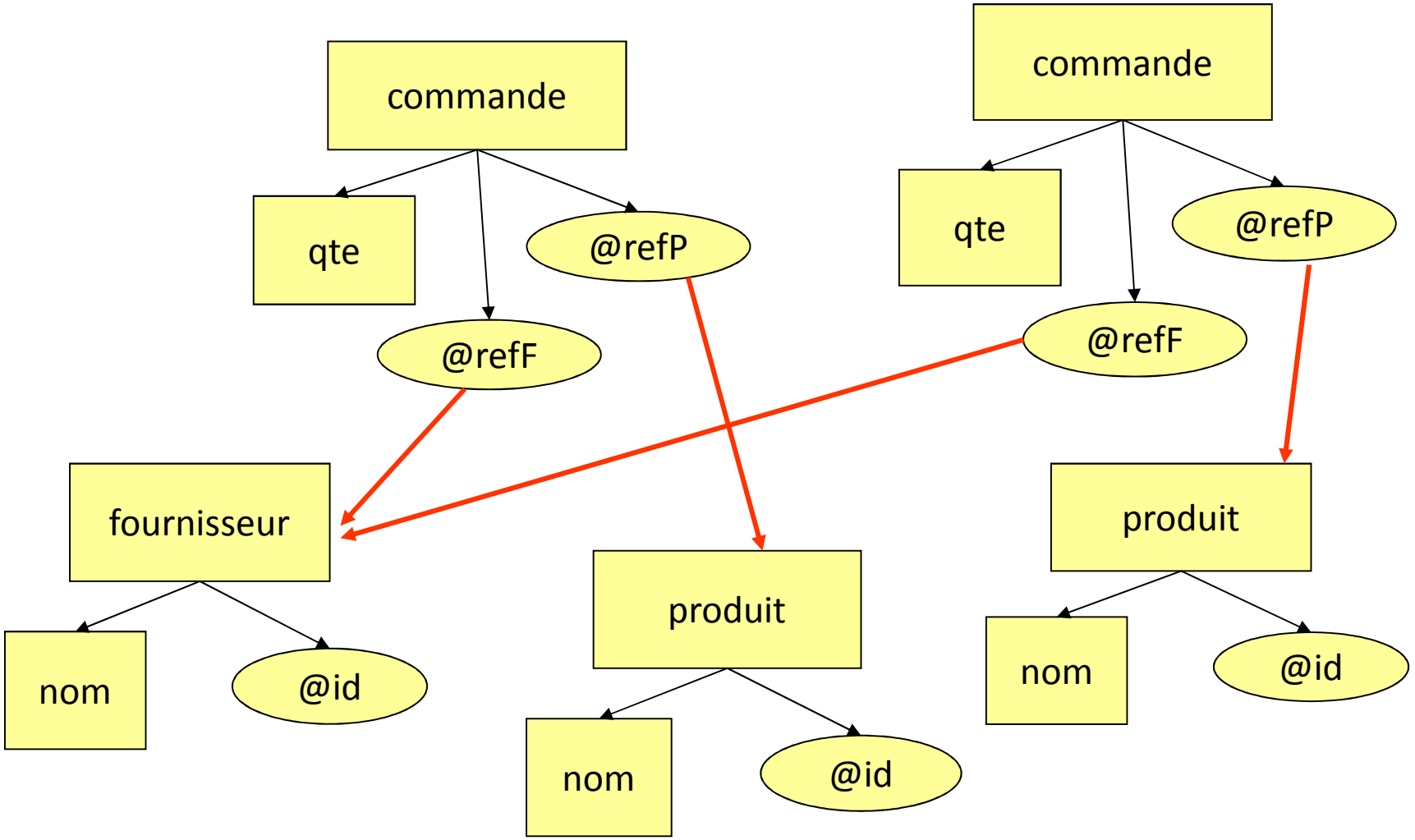
```
<!ELEMENT voie       (#PCDATA) >
<!ATTLIST voie
    type (rue | avenue | impasse | cours
    | square | boulevard | chemin | allée
    | quai | route | passage | place
    | rondPoint ) 'rue' >
```

ID et IDREF

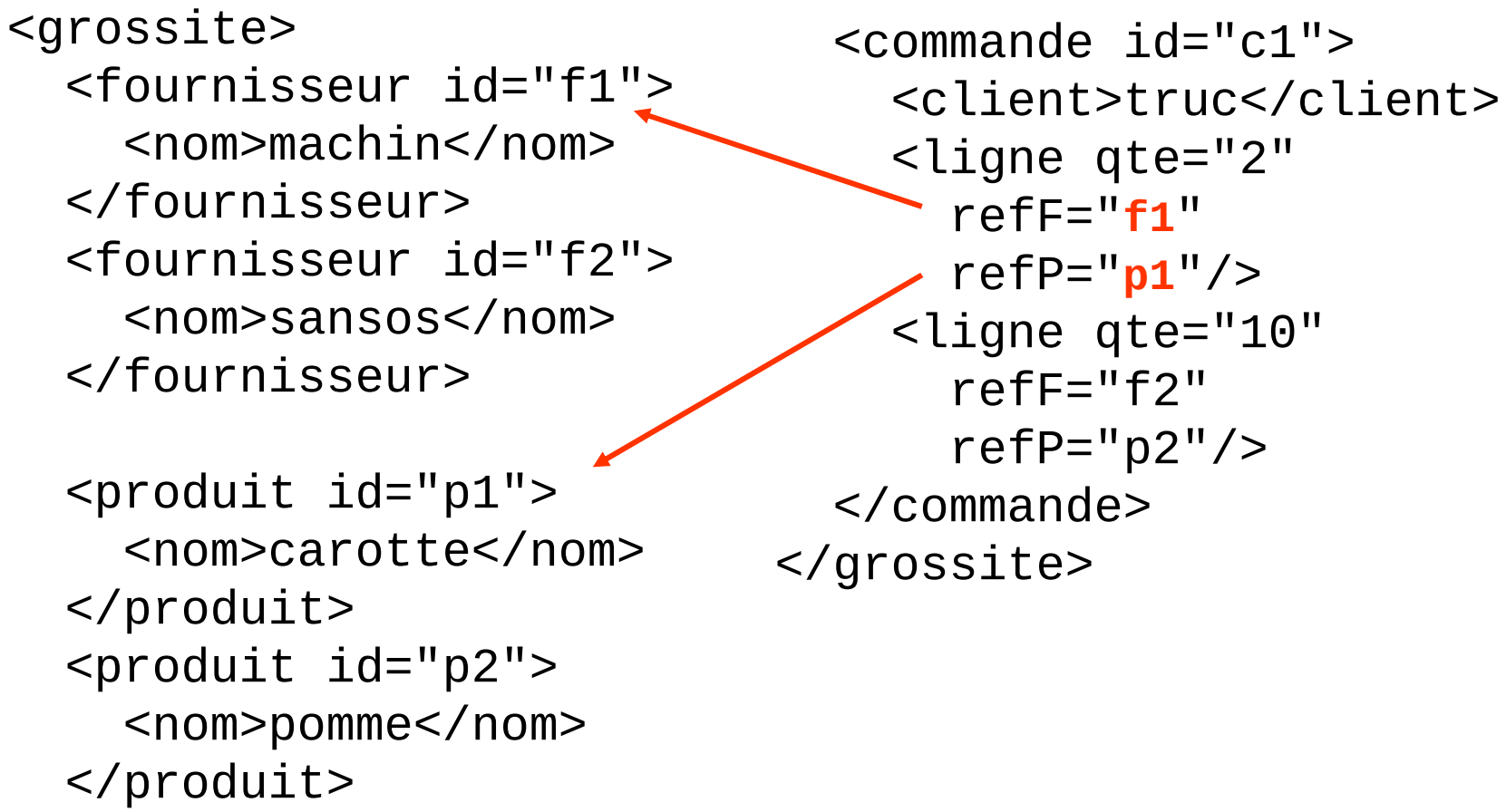
- Syntaxe
  - doivent respecter la syntaxe des noms d'éléments
- Contraintes
  - Un attribut ID doit identifier de manière unique un élément au sein d'un document considéré (*contrainte d'unicité*)
  - un attribut IDREF est contraint à prendre la valeur d'un attribut ID existant dans le document (*contrainte de référence*)

➢ Permet de créer des graphes

Exemple


commande
qte
@refP
@refF
commande
qte
@refP
@refF
fournisseur
nom
@id
produit
nom
@id
produit
nom
@id

Exemple de document



ENTITY

- Sorte de *d'abréviations* (ou de macro) qui associe
  - un nom d'entité
  - à un contenu d'entité qui est
    - Un simple texte ou un fragment de document XML
- Définition

```
<!ENTITY dtd "Document Type Definition">
<!ENTITY chap1 SYSTEM "chapitre1.xml">
```

- Utilisation on pose une référence
  - dans les contenus d'élément ou dans les valeurs d'attributs

`&dtd;` ou `&chap1;`

  - la référence est remplacée par le contenu de l'entité

ENTITY -Syntaxe

- Schéma de la définition

  `<!ENTITY nom [SYSTEM] "valeur">`

- Syntaxe des références

  `&nom;`

- La valeur associée peut contenir des balises :

```
<!ENTITY ifsic "Institut de Formation Supérieur …">
<!ENTITY piedDePage '<hr size="1"/>'>
```

ENTITY – Trois usages

- Créer une abréviation

```
<!ENTITY dtd "Document Type Definition">
```

- Créer un lien vers une source de données externe (construction modulaire)

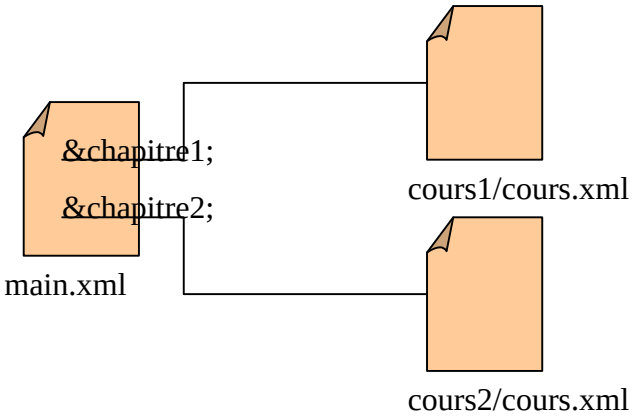
```
<!ENTITY chap1 SYSTEM "chapitre1.xml">
```

- Exprimer la transcriptions de signes spéciaux.

```
<!ENTITY euro "&#x20AC;">
```

Construction modulaire

```
<?xml version="1.0"?>
<!DOCTYPE livre [
    <!ELEMENT livre (html*)>
    <!ENTITY chapitre1 SYSTEM "cours1/cours.xml">
    <!ENTITY chapitre2 SYSTEM "cours2/cours.xml">
]>
<livre>
&chapitre1;
&chapitre2;
</livre>
```



Caractères spéciaux

- 5 entités prédéfinies

Référence	glyphe	Nom
&amp;	&	ampersand
&lt;	<	plus petit
&gt;	>	plus grand
&apos;	'	apostrophe
&quot;	"	double quote

- Utilisation : "*A >5*"

```
<if>A &gt; 5</if>
```

Deux sortes d'entités

- ***Entités générales*** pour insérer du texte
  - dans la DTD
  - `<!ENTITY dtd "Definition type document">`
  - dans le document XML, en dehors de la DTD
  - Référence : `&dtd;`
- ***Entités paramètres*** pour insérer du texte
  - Dans la DTD seulement
  - `<!ENTITY % contenuAdresse "ville,rue">`
  - Référence : `%contenuAdresse;`

Entités paramètres

Héritage de contenus d'élément

- Déclaration

```
<!ENTITY % pos "X,Y">
```

- Utilisation

```
<!ELEMENT carre (%pos;,lg)>
<!ELEMENT rectangle
  (%pos;,hauteur,largeur)>
<!ELEMENT cercle (%pos;,diametre)>
```

Entités paramètre

Héritage d'attributs

- Déclaration

```
<!ENTITY % pos
"X CDATA #REQUIRED
Y CDATA #REQUIRED">
```

- Utilisation

```
<!ELEMENT carre (EMPTY)>
<!ATTLIST carre %pos; lg CDATA #REQUIRED>
<!ELEMENT cercle (EMPTY)>
<!ATTLIST cercle %pos; diam CDATA
  #REQUIRED>
```

Outils de validation

- Les compilateurs en général
- Les éditeurs XML (qui permettent une saisie contextuelle assistée grâce aux DTDs)
  - XMLSpy
  - Eclipse avec des plug-ins *éditeurs XML*
    - WTP
    - XMLBuddy
    - Nombreux autres éditeurs

La force des DTDs

- Un moyen compact de spécifier un dialecte XML
- 20 ans d'expérience d'utilisation des DTDs dans les milieux SGML
  - Des centaines de langages normalisés par leur DTD (HTML en premier…)

Limites dans la spécification des DTD

- Les DTDs ne permettent pas de *typer* les chaînes contenues dans les éléments et dans les attributs
  - On aimerait exprimer qu'un contenu d'élément ou d'attribut est un *entier positif*, ou un *entier dans l'intervalle [0..99]*
  - Tout ce que l'on sait exprimer c'est qu'un contenu est une *chaîne* !

- La spécification de la cardinalité d'un élément est pauvre : ?,+,* (*0 ou 1, 1 ou plus, 0 ou plus*)
  - On aimerait contraindre la cardinalité d'un élément sur un intervalle quelconque (entre 3 et 7 par exemple)

Autres problèmes des DTDs

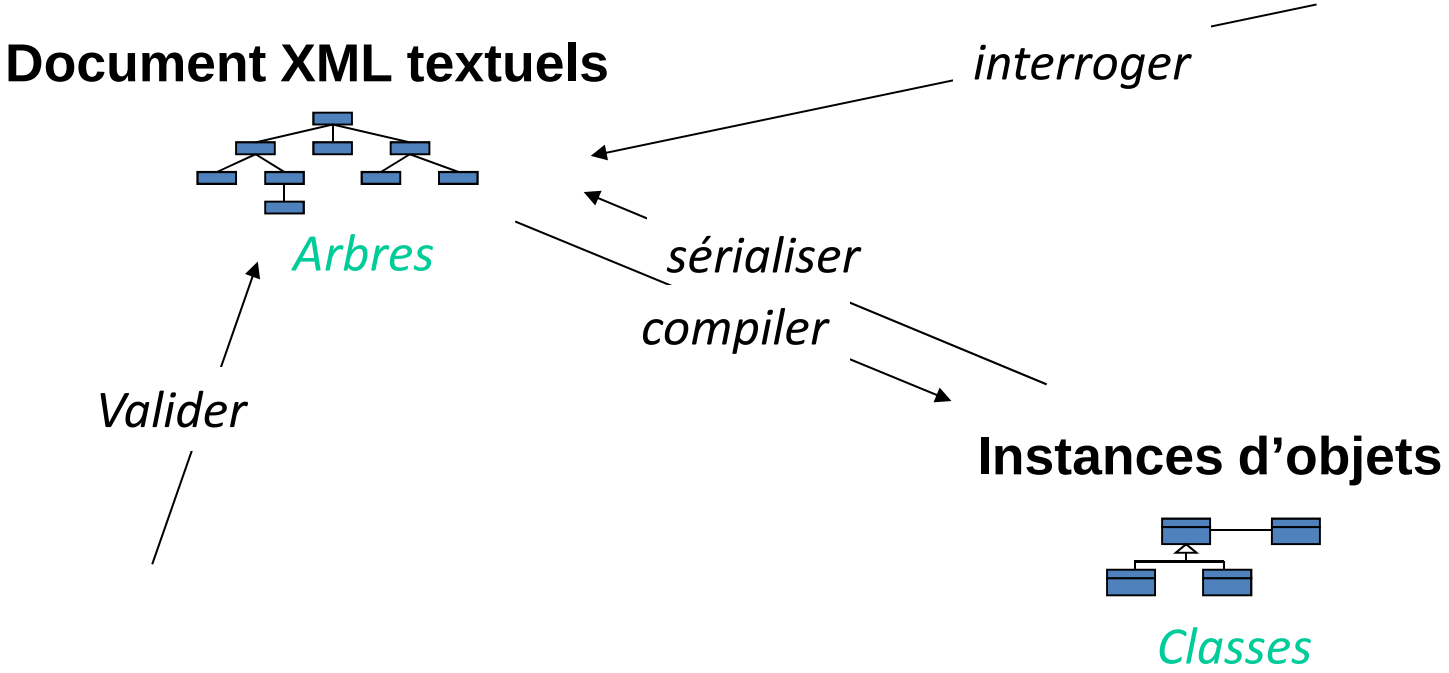
- N'utilise pas la syntaxe XML et n'offre pas de version en XML
- Sa spécification n'est pas séparée de celle d'XML (difficile de la faire évoluer …)
- Très peu de support pour la modularité et la réutilisation des descriptions
- Pas de notion d'héritage, d'extension
- Portée globale pour les éléments et les attributs
- Des valeurs par défaut pour les attributs mais pas pour les éléments

Quelle suggestion ?

- Il y a de la marge pour la définition d'une nouvelle technologie de modélisation plus précise …
- C'est la brèche ouverte par les *Schémas XML* et les nombreuses autres propositions équivalentes
- Schemas XML : Une autre description de la grammaire d'un fichier XML
  - Plus rigoureuse
  - Possibilité d'exprimer plus de contraintes
  - Au format XML

Principe de compilation de fichiers XML

Manipuler des documents XML



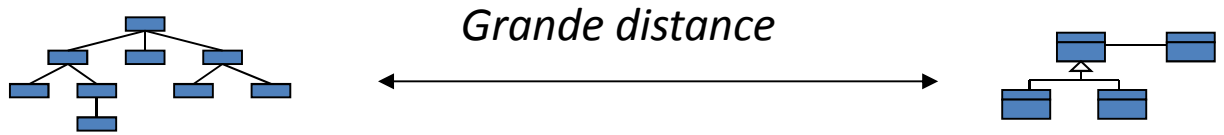
Importer un document XML

- Utiliser un outil clé en main
  - DOM Document Object Model, Norme conçue par le W3C Solution
    - Générique, indépendante des langages : Javascript, C, Java, Perl, Latex …
    - Construire rapidement un arbre de nœuds à partir d'un document textuel et vis-versa

- Fabriquer compilateur et sérialiseur
  - SAX Simple API for XML
    - Norme conçue par un consortium de constructeur
    - Outils pour construire des compilateurs poussés par des évènements
  - Compilation classique
    - tirée par un analyseur lexical

DOM

- Intérêt :
  - Compilation facile à mettre en œuvre (sans investissement)
  - Permettent le style *programmation extrême* …
- Mais **pas forcement** la méthode la plus
  - *Rapide* pour arriver au résultat
  - *Extensible* (permettant de revenir sur la spécification)
  - *Efficace* (en temps et mémoire)
- Son défaut **trop souvent oublié**
  - *on construit un arbre générique*
    - *Parcours laborieux*
    - *Modifications laborieuses*



arbre générique

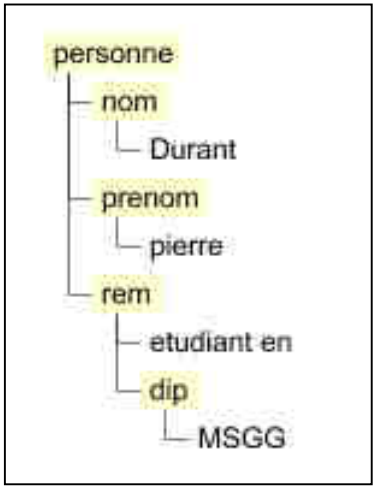
Modèle dédié

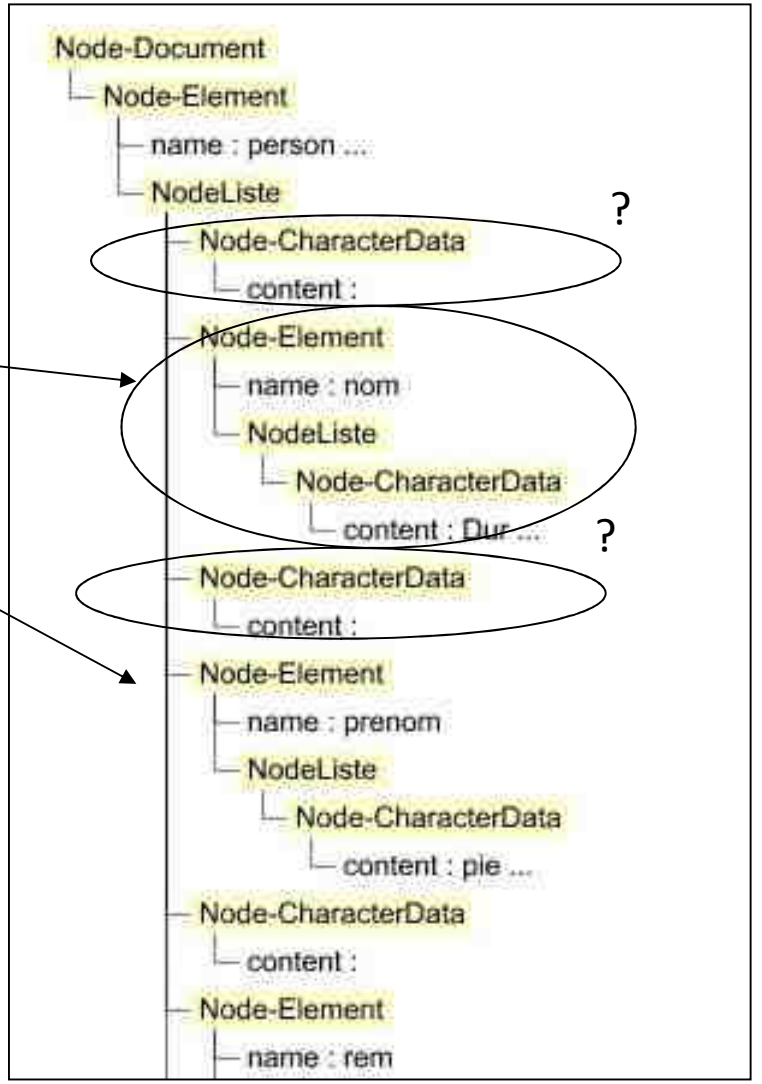
DOM

- Document Object Model
  - Modèle d'arbre générique permettant de représenter dans un programme un document html ou xml par des instances objets
  - Modèle indépendant des langages de programmation, s'applique à tout langage
    - C++, java, JavaScript, perl, Latex …
- Pourquoi
  - parcourir l'arbre du document
  - ajouter, effacer, modifier un sous-arbre
  - sélectionner par un sous-arbre par son contenu
- Par qui :
  - Conçu par le W3C
  - intégrée par Sun dans Java 1.4 sous le nom de Jaxp 1.0

Arbre DOM

```
<personne>
  <nom>Durant</nom>
  <prenom>pierre</prenom>
  <rem>etudiant en <dip>MSGG</dip></rem>
</personne>
```

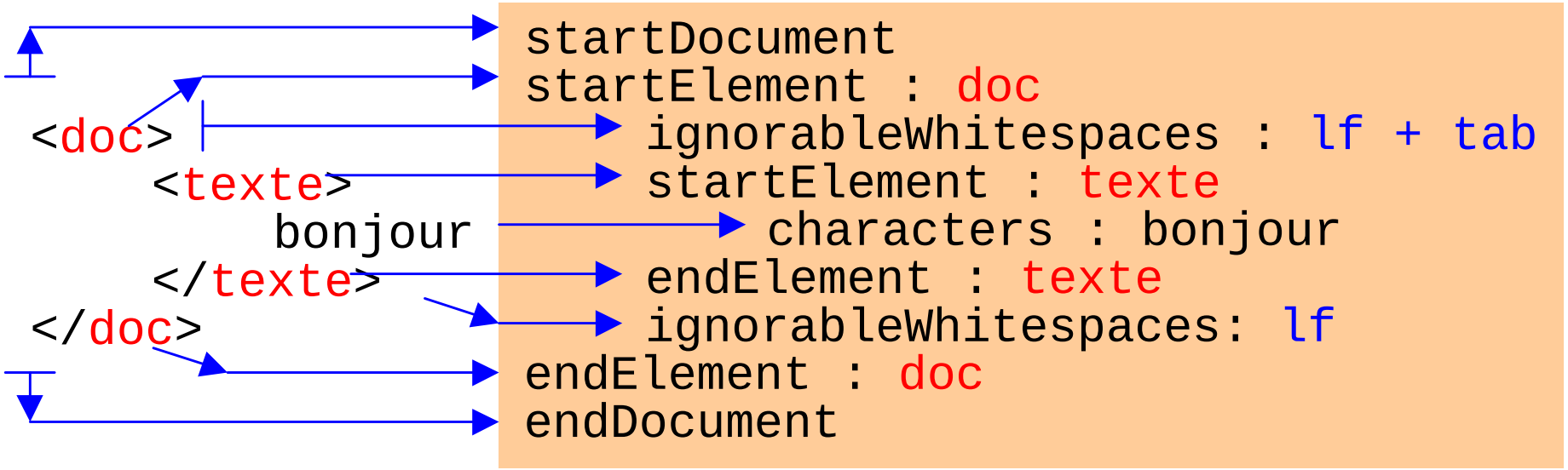




SAX

- SAX : Simple API for XML
  - Modèle évènementiel
- Pourquoi
  - pour concevoir des compilateurs XML spécifiques
- Par qui
  - Un consortium de fabriquants
  - Adopté par Sun dans Java 1.4 sous le nom Jaxp 1.0
- Comment
  - pas d'image mémoire du document lu
  - séquence d'événements à interpréter
  - ➢ Très gros gain en mémoire par rapport à DOM

Séquence d'événements



- Six types d'évènements
  - début et fin de document
  - début et fin d'éléments
  - caractères
  - caractères ignorables

Conclusion

- SAX ou DOM ?
  - DOM est souvent plus lent que SAX
  - DOM est plus consommateur de mémoire
    - Problème pour le traitement de gros documents
  - SAX permet de construire une représentation à la volée
- XML
  - Format standardisé, pratique pour échanger des données
  - Beaucoup d'outils dans tous les langages de programmation
  - Inconvénient :
    - Verbeux, difficile de demander à un utilisateur de saisir du XML
    - consommateur d'espace

Conclusion CIN

- Utilité
  - Savoir lire la grammaire d'un langage
  - Connaitre
  - Lecture de fichiers structurés
  - Savoir créer un langage
    - Description de structures de données
    - Langages de script

- Vos remarques pour l'amélioration du cours sont les bienvenues