INFO416 : Compilation

Dr. Etienne Kouokam, Gérard Ndenoka

UMMISCO-LIRIMA, Département d'Informatique
Université de Yaoundé I, Cameroun

Année académique 2017-2018

Plan

Plan

Plan

Objectifs

1. Les microprocesseurs disposent généralement d'un jeu d'instructions très sommaire, peu convivial et qui varie d'un processeur à l'autre. Afin de fournir à l'utilisateur de systèmes informatiques des formalismes universels, expressifs et concis, de nombreux langages de programmation ont été créés. **L'objet de la théorie des langages est de les définir.**
2. La compilation, pour sa part, à pour objet de transformer les programmes écrits dans ces langages en code machine. L'objectif de ce cours est de **fournir à l'étudiant les connaissances conceptuelles qui lui permettront de mieux comprendre les langages informatiques ainsi que leur compilation**.

Que va-t-on apprendre dans ce cours ? (entre autres)

1. Comment définir formellement un modèle pour décrire
   - un langage (de programmation ou autre)
   - un système (informatique)
2. Comment déduire des propriétés sur ce modèle
3. Ce qu'est
   - un compilateur
   - un outil de traitement de données
4. La notion d'outil permettant de construire d'autres outils
   Exemple : générateur de (partie de) compilateur
5. Comment construire un compilateur ou un outil de traitement de données
   - en programmant
   - en utilisant ces outils
6. Quelques notions de décidabilité

Démarche

Montrer la démarche scientifique et de l'ingénieur qui consiste à

1. Comprendre les outils (mathématiques / informatiques) disponibles pour résoudre le problème
2. Apprendre à manipuler ces outils
3. Concevoir à partir de ces outils un système (informatique)
4. Implémenter ce système

Les outils ici sont :

- les formalismes pour définir un langage ou modéliser un système
- les générateurs de parties de compilateurs

Prérequis + Bibliographie

Prérequis : Eléments d'algèbre.

Ouvrages de référence :

1. J. E. Hopcroft, R. Motwani, and J. D. Ullman ; Introduction to Automata Theory, Languages, and Computation, Second Edition, Addison-Wesley, New York, 2001.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers : Principles, Techniques and Tools", Addison-Wesley, 1986
3. Jean-Michel Autebert. Théorie des langages et des automates. Masson, 1994.

Historique

Machine de Turing (1936)

Conçue pour modéliser une machine qui résout le plus de problèmes possibles

Automates (années 40 à 50)

Utilisé dans la modélisation de certaines fonctionnalités du cerveau

Les grammaires formelles (fin des années 50, Chomsky)

Largement utilisées en Compilation, feront leur apparition pour modéliser les langues naturelles

Ainsi, la solution à un problème ne peut être déterminée que si les éléments du langages sont bien connus

Théorie des langages & Concepts

Objectif

Comprendre le fonctionnement des langages (formel), vus comme moyen de communication, d'un point de vue mathématique.
Algorithmes, Programmes, etc.

Définition (Langage - mot)

- Un langage est un ensemble de mots.
- Un mot (ou lexème ou chaîne de caractères ou string en anglais) est une séquence de symboles (signes) élémentaires dans un alphabet donné.

Alphabet, Mot, Langage

Exemples

Alphabet	Mots	Langages
$\Sigma = \{0, 1\}$,	$\epsilon, 0, 1, 00, 01$	$\{\epsilon, 0, 1, 00, 01\}, \{\epsilon\}, \emptyset$
$\{a, b, c, \ldots, z\}$,	*salut*, *patron*	$\{$*salut*, *patron*, $\epsilon\}$
$\{\alpha, \beta, \gamma, \delta, \mu, \nu, \pi, \sigma, \eta\}$,	$\eta\gamma\alpha\mu$	$\{\epsilon, \eta\gamma\alpha\mu\}$

Notation

Nous utiliserons habituellement des notations conventionnelles :

Notation	Exemple
Alphabet : Σ	$\Sigma = \{0, 1\}$
Mot : $a, b, c, \ldots$	$x = 00011$
Langage : $L, L_1, \ldots$	$L = \{\epsilon, 00, 11, \mu\}$

Théorie des langages & Concepts (suite)

Quelques concepts

- La notion de grammaire (formelle) qui définit la syntaxe d'un langage,
- La notion d'automate permettant de déterminer si un mot fait partie d'un langage (et donc permettant également de définir un langage (l'ensemble des mots acceptés)), constituant ainsi son lexique,
- La notion d'expression régulière qui permet de dénoter un langage.

Motivations et applications

Applications pratiques de la théorie des langages

- la définition formelle de la syntaxe et sémantique de langages de programmation,
- la construction de compilateurs,
- la modélisation abstraite d'un système informatique, électronique, biologique, ...

Motivations théoriques

Liées aux théories de

- la calculabilité (qui détermine en particulier quels problèmes sont solubles par un ordinateur)
- la complexité qui étudie les ressources (principalement temps et espace mémoire) requises pour résoudre un problème donné

Définition (Compilation)

Traduction $C = f(L_C, L_S \rightarrow L_O)$ d'un langage dans un autre avec

- L_C le langage avec lequel est écrit le compilateur
- L_S le langage source à compiler
- L_O le langage cible ou langage objet : celui vers lequel il faut traduire

Pour $C = f(L_C, L_S \rightarrow L_O)$

L_C	L_S	L_O
C	Assembleur RISC	Assembleur RISC
C	C	Assembleur P7
C	Java	C
Java	LateX	HTML
C	XML	PDF

Remarque

- Si $L_C = L_S$: peut nécessiter un bootstrapping pour compiler $C = f(L_C, L_S \rightarrow L_O)$.
- Il en ressort que la compilation ne concerne
  - pas seulement des langages de programmation : langages de script, d'interrogation, de description (LaTeX, XML, ...)
  - pas seulement pour obtenir un programme exécutable : enjoliveurs, éditeurs syntaxiques, ...
  - concepts et outils utilisés dans de nombreux autres domaines : web, bases de donnés, traitement des langues naturelles ...

Terminologie

- Traducteur : passage d'un langage à un autre, par exemple
  - préprocesseur : d'un sur-langage de L vers L (C par exemple)
  - assembleur : du langage d'assemblage vers le code machine binaire
  - éditeur de liens : d'un ensemble de sous-programmes en binaire relogeable (les .o de linux par exemple) vers un programme exécutable
- compilateur : traduction d'un langage source (généralement) de haut niveau vers un langage de plus bas niveau
- décompilateur : programme qui essaye de reconstruire le programme source à partir du code objet

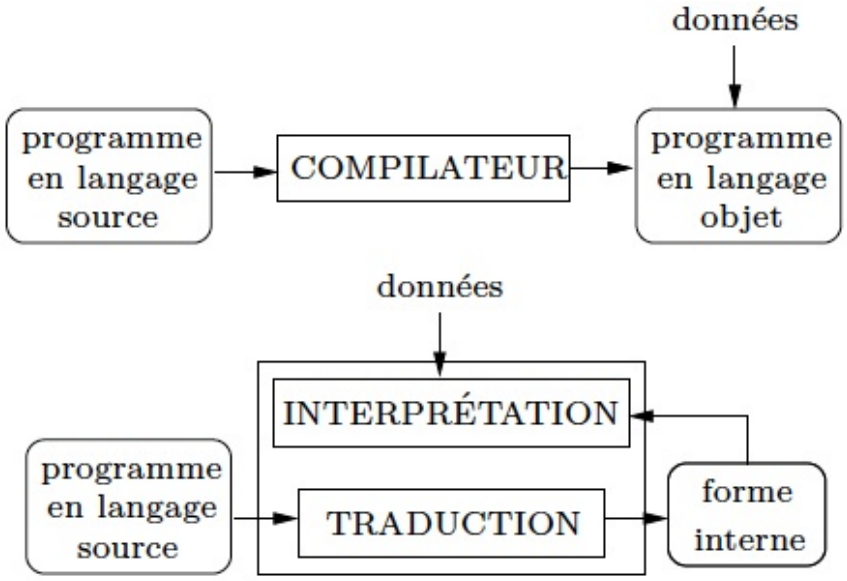


FIGURE: Compilateur Vs Interprète

Compilateur Vs Interprète

- Interpréteur = outil qui analyse, traduit, mais aussi exécute un programme écrit dans un langage informatique.
- L'interpréteur se charge donc également de l'exécution au fur et à mesure de son interprétation.

Avantages et inconvénients de chaque modèle

- compilateur : produit du code machine $\rightarrow$ programme efficace mais pas portable
- interprète : programme portable mais moins efficace, mais qui peut manipuler le code source (méta-programmation)

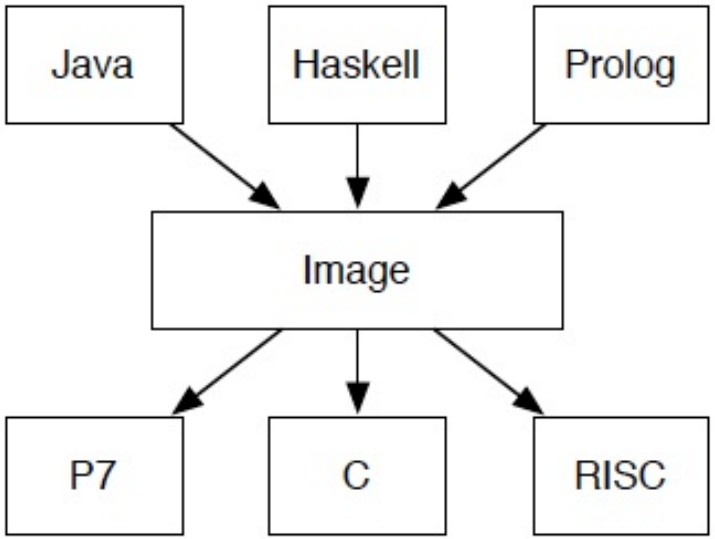
Structure d'un compilateur

Structure

Un langage intermédiaire L_I est généralement utilisé. Le compilateur est donc formé de :

- une partie avant ou front-end : $L_S \rightarrow L_I$
- une partie arrière ou back-end : $L_I \rightarrow L_O$

Ce qui facilite la construction de nouveaux compilateurs



Type de compilateur

Compilateur monolithique

- les premiers compilateurs étaient monolitiques : ils pouvaient travailler en 1 seule passe (les langages étaient encore simples)
- autant de compilateurs que de couples (langage, machine cible), soit $m \times n$

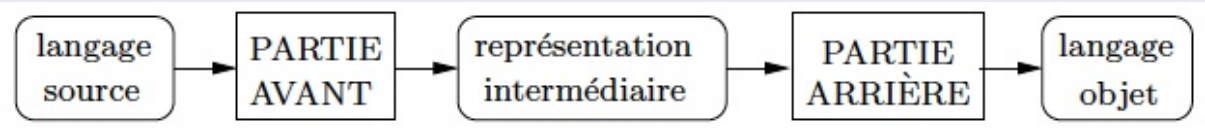


FIGURE: Compilateur monolithique

Compilateur modulaire

- partie avant (analyse) : analyses lexicale, syntaxique, sémantique
- partie arrière (synthèse) : optimisation, production de code
- avantages de cette décomposition : m parties avant + n parties arrières permettent d'avoir $m \times n$ compilateurs
- le langage objet peut être celui d'une machine virtuelle (JVM . . .) : le programme résultant sera portable
- on peut interpréter la représentation intermédiaire

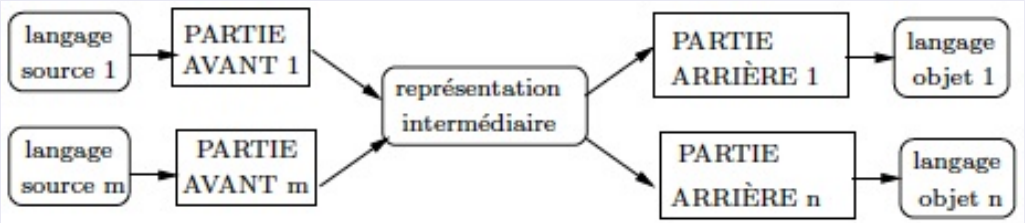


FIGURE: Compilateur modulaire

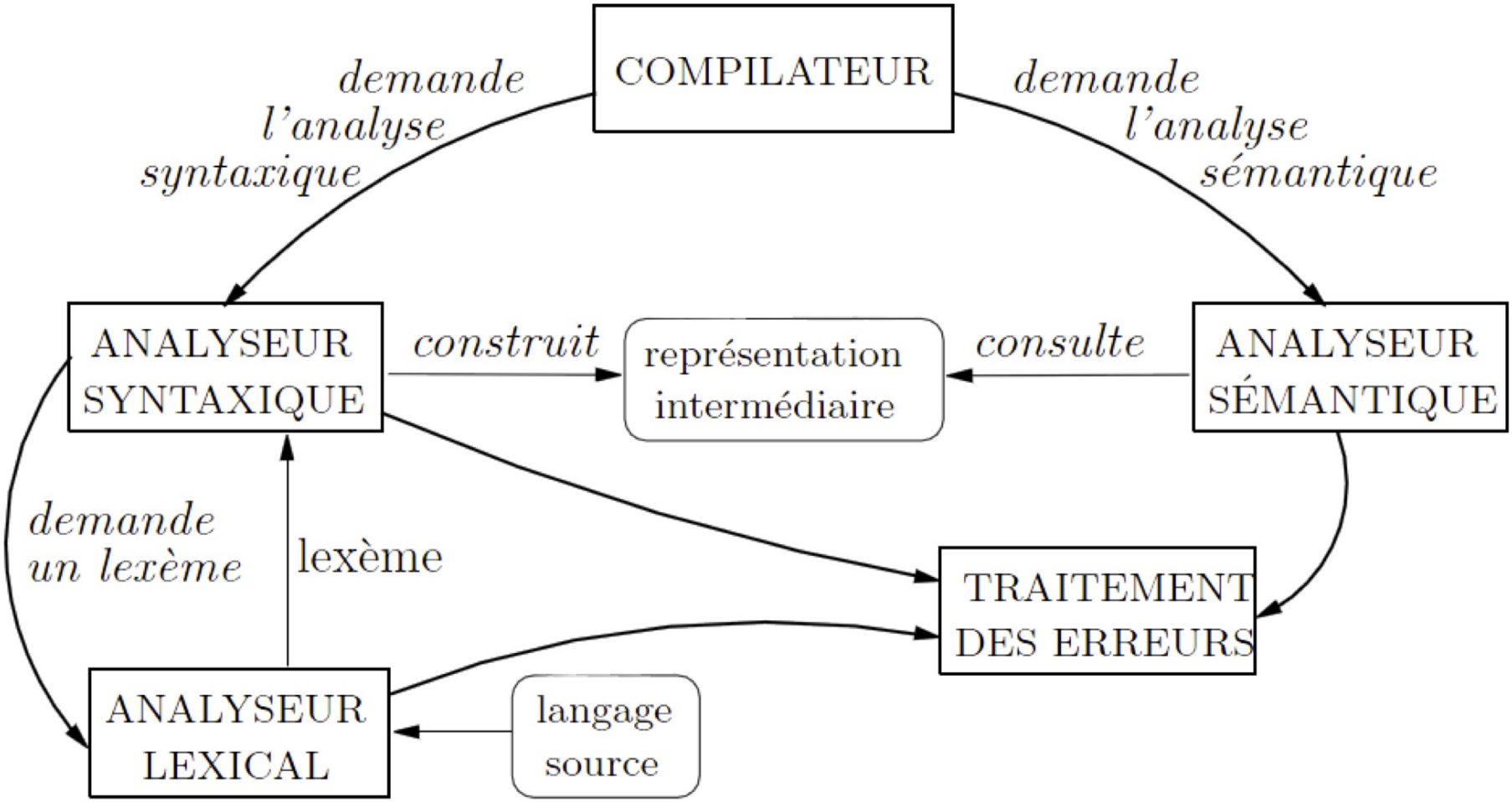


FIGURE: Schéma synthétique de la partie avant

Pourquoi la partie avant : Point de vue lexico-syntaxique

- le programme source est plein de "bruits" (espaces inutiles, commentaires . . .)
- pour la grammaire, de nombreux symboles sont équivalents (identificateurs, nombres . . .)
- ce qui justifie le pré-traitement du texte source

Pourquoi la partie avant : Point de vue sémantique

- existence de références en avant (utilisation d'un identificateur avant sa déclaration par exemple)
- unification du traitement de constructions équivalentes ($attr = 0$ et $this.attr = 0$ par exemple) ou proches (boucles notamment)
- ce qui justifie la mémorisation (sous forme intermédiaire) du texte à compiler

Concepts et structures de données utilisés

- analyse lexicale : langages réguliers, expressions régulières, automates finis pour l'essentiel, mais aussi tables d'adressage dispersé, arithmétique
- analyse syntaxique : grammaires hors-contexte, automates à pile (analyseurs descendants ou ascendants), attributs sémantiques
- analyse sémantique : diverses sémantiques formelles (mais l'analyse sémantique est souvent codée à la main), équations de type, table de symboles
- représentation intermédiaire : arbre ou graphe le plus généralement

Propriétés d'un bon compilateur

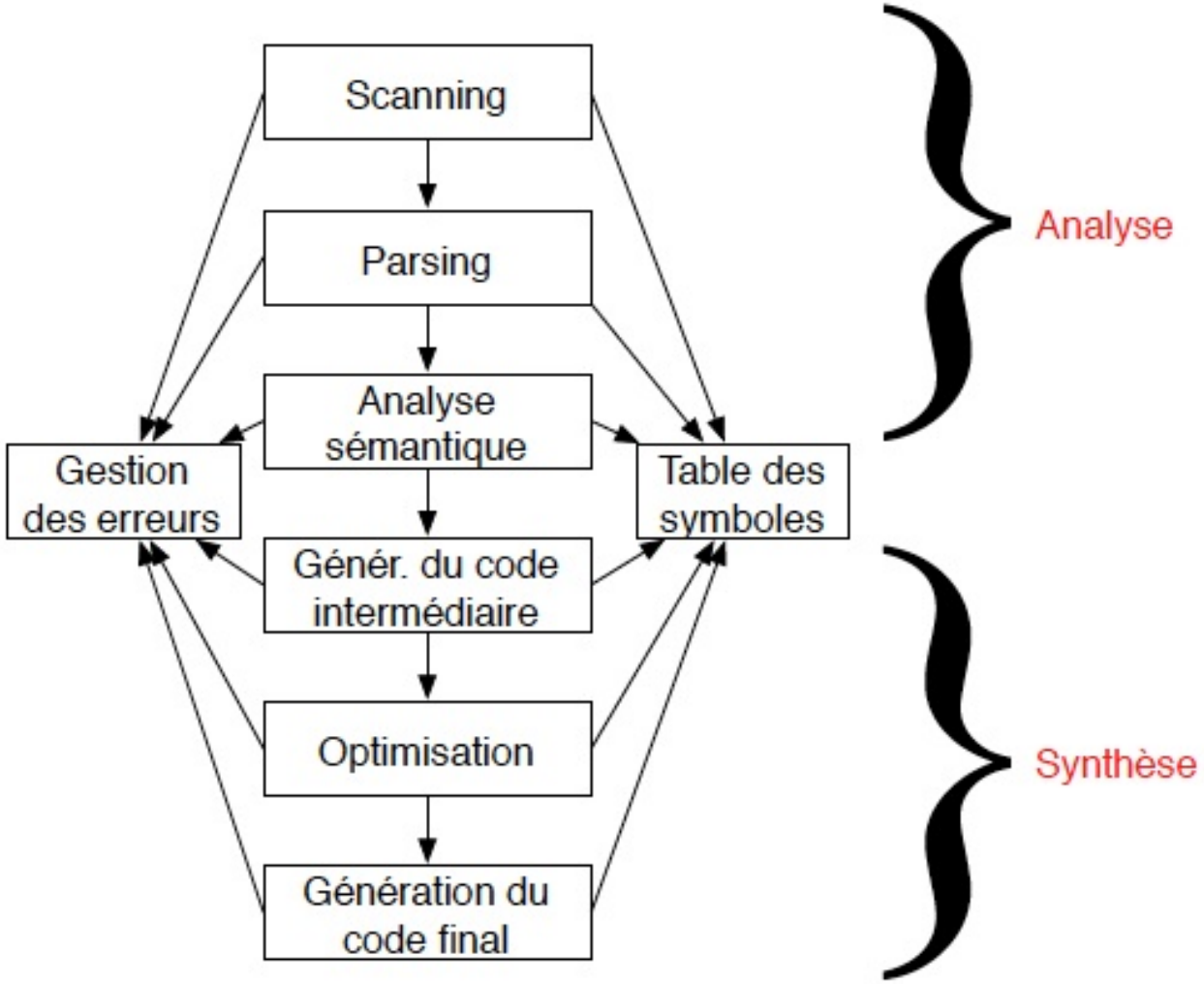
- reconnaître tout le langage, et rien que le langage (ou alors avoir un mode permettant d'informer des constructions non standards)
  - sinon, accepte des programmes non portables
  - ne pas se restreindre à des constructions humainement lisibles : beaucoup de programmes sont créés par des programmes
  - indiquer de fa¸con claire les erreurs et éviter les messages d'erreur en cascade (bien que la source de l'erreur ne soit pas toujours évidente)
- traduire correctement vers le langage cible, et en donnant le meilleur code possible
- être raisonnablement rapide (utiliser des algorithmes quasi linéaires)

Caractéristiques d'un compilateur

- Efficacité
- Robustesse
- Portabilité
- Fiabilité
- Code debuggable
- A une passe
- A n passes (70 pour un compilateur PL/I !)
- Optimisant
- Natif
- Cross-compilation

Les Phases de la Compilation

Les phases de la compilation



Compilation découpée en 2 étapes

1. L'analyse décompose et identifie les éléments et relations du programme source et construit son image (représentation hierachique du programme avec ses relations),
2. La synthèse qui construit à partir de l'image un programme en langage cible

Contenu de la table des symboles

Un enregistrement par identificateur du programme à compiler contenant les valeurs des attributs pour le décrire.

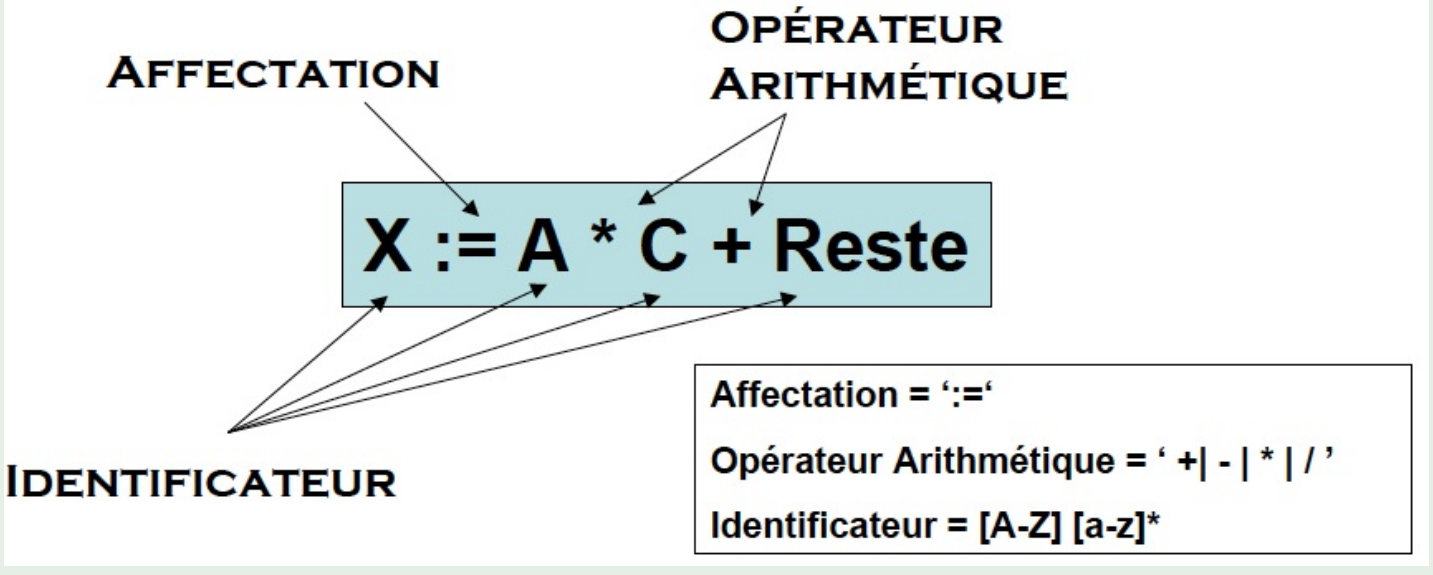
Contenu de la table des symboles

En cas d'erreur, le compilateur peut essayer de se resynchroniser pour éventuellement essayer de reporter d'autres erreurs

Analyse lexicale (Scanning)

- Un programme peut être vu comme une "phrase" ; le rôle principal de l'analyse lexicale est d'identifier les "mots" de la phrase.
- Le scanner décompose le programme en lexèmes en identifiant les unités lexicales de chaque lexème.

Exemple



Analyse lexicale (Scanning)

Définition (Unité lexicale (ou token))

Type générique d'éléments lexicaux (correspond à un ensemble de strings ayant une sémantique proche).
Exemple : identificateur, opérateur relationnel, mot clé "begin"...

Définition (Lexème (ou string))

Occurrence d'une unité lexicale.
Exemple : N est un lexème dont l'unité lexicale est identificateur

Définition (Modèle (pattern))

Règle décrivant une unité lexicale
En général un modèle est donné sous forme d'expression régulière (voir ci-dessous)

Relation entre lexème, unité lexicale et modèle

unité lexicale = lexème|modèle(lexème)

Résultat et autres buts du scanning

Exemple (d'unités lexicales et de lexèmes)

unité lexicale	lexème
identificateur	int
identificateur	main
par-ouvrante	(
par-fermante	)
...	

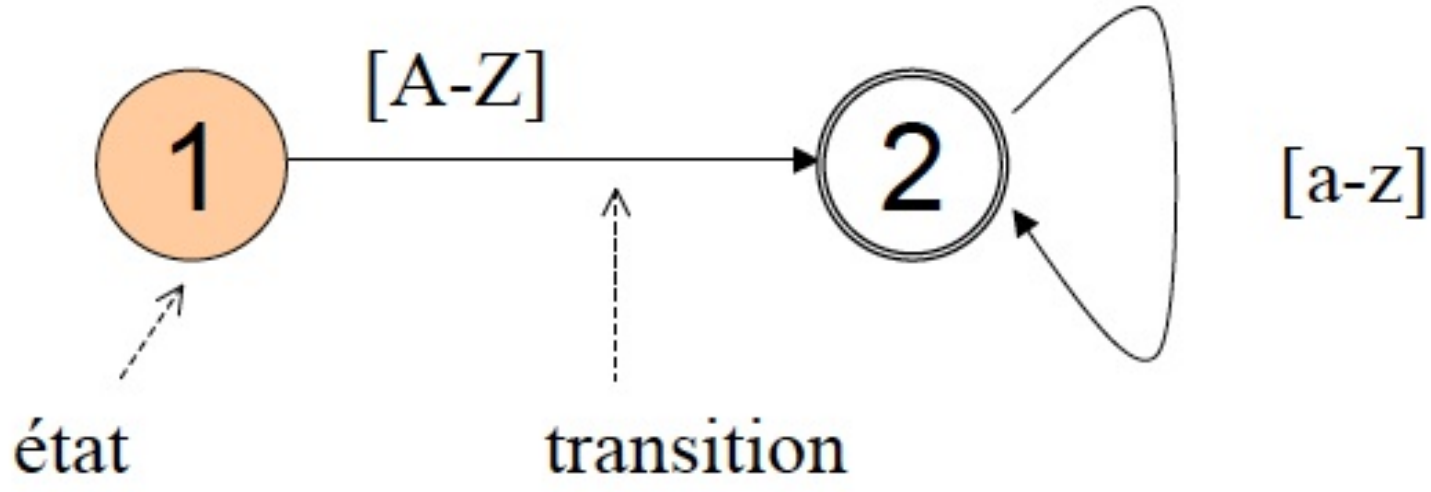
Autres buts du scanning

- Met (éventuellement) les identificateurs (non prédéfinis) et littéraux dans la table des symboles
- Produit le listing / lien avec un éditeur intelligent
- Nettoie le programme source (supprime les commentaires, espaces, tabulations, ...)

Analyse lexicale (Scanning)

Les unités lexicales sont reconnues à l'aide d'automates

Identificateur = [A-Z] [a-z]*



Analyse syntaxique (Parsing)

- Le rôle principal de l'analyse syntaxique est de trouver la structure de la "phrase" ? (le programme) : i-e de construire une représentation interne au compilateur et facilement manipulable de la structure syntaxique du programme source.
- Le parser construit l'arbre syntaxique correspondant au code.

L'ensemble des arbres syntaxiques possibles pour un programme est défini grâce à une grammaire (context-free).

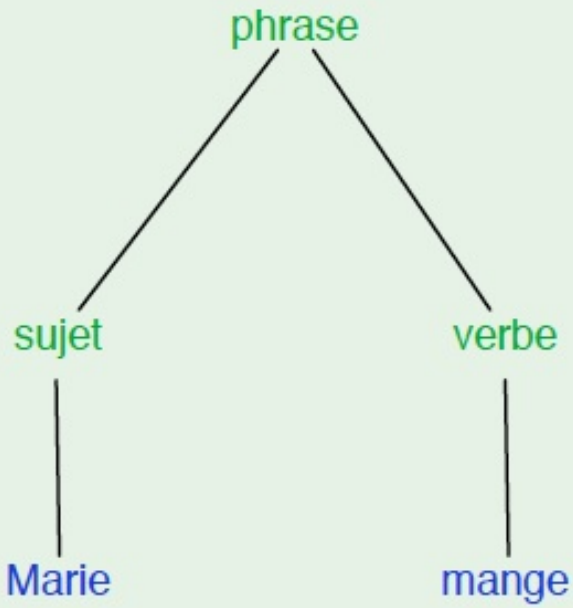
Exemple de grammaire

Exemple (grammaire d'une phrase)

- phrase = sujet verbe
- sujet = "**Jean**" | "**Marie**"
- verbe = "**mange**" | "**parle**"

peut donner

- **Jean mange**
- **Jean parle**
- **Marie mange**
- **Marie parle**



Arbre syntaxique de la phrase
Marie mange

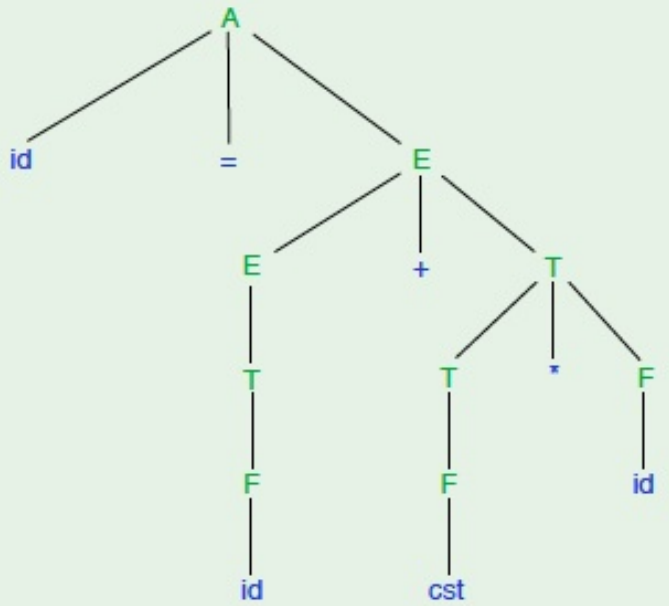
Exemple de grammaire

Exemple (grammaire d'une expression)

- A = "**id**" "**=**" E
- E = T | E "**+**" T
- T = F | T "*****" F
- F = "**id**" | "**cst**" | "**(**" E "**)**"

peut donner :

- **id = id**
- **id = id + cst * id**
- ...



Arbre syntaxique de la phrase
id = id + cst * id

Exemple de grammaire (suite)

Exemple (grammaire d'une expression)

- A = "**id**" "**=**" E
- E = T | E "**+**" T
- T = F | T "*****" F
- F = "**id**" | "**cst**" | "**(**" E "**)**"

peut donner :

- **id = id**
- **id = id + cst * id**
- ...

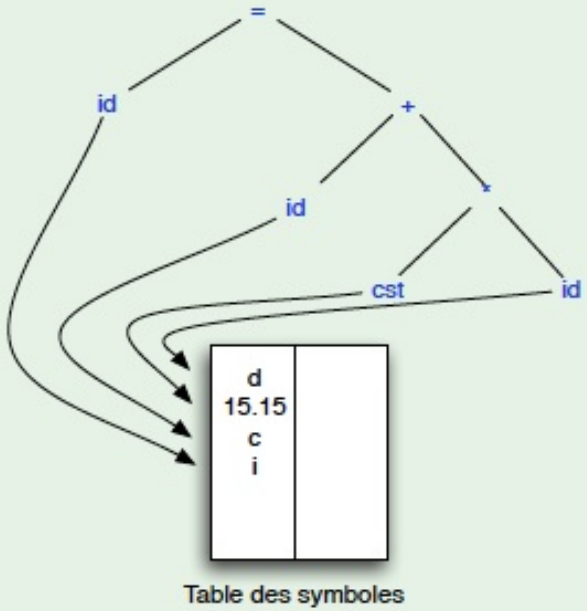


Table des symboles

Arbre syntaxique abstrait avec références à la table des symboles de la phrase **i = c + 15.15 * d**

Analyse sémantique

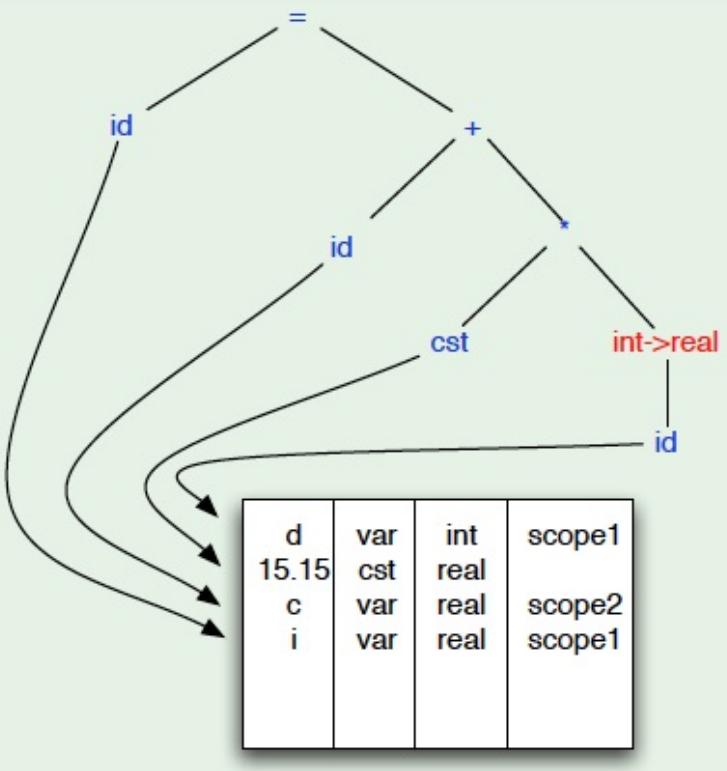
Rôle de l'analyse sémantique

Pour un langage impératif, l'analyse sémantique (appelée aussi gestion de contexte) s'occupe des relations non locales ; elle s'occupe ainsi :

1. du contrôle de visibilité et du lien entre les définitions et utilisations des identificateurs (en utilisant/construisant la table des symboles)
2. du contrôle de type des objets, nombre et type des paramètres de fonctions
3. du contrôle de flot (vérifie par exemple qu"un goto est licite - voir exemple plus bas)
4. de construire un arbre syntaxique abstrait complété avec des informations de type et un graphe de contrôle de flot pour préparer les phases de synthèse.

Exemple de grammaire (suite)

Exemple (pour l'expression $i = c + 15.15 * d$)



Arbre syntaxique abstrait modifié avec références à la table des symboles de la phrase **i = c + 15.15 * d**

Synthèse

Phase de synthèse

Pour un langage impératif, la synthèse comporte les 3 phases

1. Génération de code intermédiaire sous forme de langage universel qui
   - utilise un adressage symbolique
   - utilise des opérations standard
   - effectue l'allocation mémoire (résultat dans des variables temporaires...)
2. Optimisation du code
   - supprime le code "mort"
   - met certaines instructions hors des boucles
   - supprime certaines instructions et optimise les accès à la mémoire
3. Production du code final
   - Allocation de mémoire physique
   - gestion des registres

Exemple de grammaire (suite) (pour le code $i = c + 15.15 * d$)

1. Génération de code intermédiaire

$$
\begin{array}{lcr}
\text{temp1} & \leftarrow & 15.5 \\
\text{temp2} & \leftarrow & \mathit{Int2Real}(\mathit{id}3) \\
\text{temp2} & \leftarrow & \mathit{temp}1 * \mathit{temp}2 \\
\text{temp3} & \leftarrow & \mathit{id}2 \\
\text{temp3} & \leftarrow & \mathit{temp}3 + \mathit{temp}2 \\
\text{id1} & \leftarrow & \mathit{temp}3
\end{array}
$$

2. Optimisation du code

$$
\begin{array}{lcr}
\text{temp1} & \leftarrow & \mathit{Int2Real}(\mathit{id}3) \\
\text{temp1} & \leftarrow & 15.15 * \mathit{temp}1 \\
\text{id1} & \leftarrow & \mathit{id}2 + \mathit{temp}1
\end{array}
$$

3. Production du code final

Exemple de grammaire (suite) (pour le code $i = c + 15.15 * d$)

1. Génération de code intermédiaire
2. Optimisation du code

$$\begin{array}{lcl} \text{temp1} & \leftarrow & \mathit{Int2Real}(\mathit{id3}) \\ \text{temp1} & \leftarrow & 15.15 * \mathit{temp1} \\ \text{id1} & \leftarrow & \mathit{id2} + \mathit{temp1} \end{array}$$

3. Production du code final

MOVF	id3, R1
ITOR	R1
MULF	15.15, R1, R1
ADDF	id2, R1, R1
STO	R1, id1

Plan

Définition

Une grammaire c'est quoi ?

4 composantes : $G = (V_n, V_t, S, P)$. On pose $\Sigma = V_n \cup V_t$ où :

- V_t = Alphabet des symboles terminaux : Les éléments du langage (variable, identificateur, ...). Ils sont notés en minuscules. a, b, c ...
- V_n = Symboles non-terminaux : symboles auxiliaires dénotant les types de construction (boucle, expression booléenne, ...). Ils sont notés en majuscules. A, B, C ...
- S = Le but (symbole de départ) appelé axiome : dénote n'importe quelle phrase.
- P = Productions : Les règles de réécriture utilisées pour reconnaître et générer des phrases. Elles sont de la forme
  - $\alpha \to \beta$ avec $\alpha \in \Sigma^* V_n \Sigma^*$ et $\beta \in \Sigma^*$
  - (α, β), où $\alpha \in \Sigma^+$ et $\beta \in \Sigma^*$

Définition

4 composantes : $G = (V_n, V_t, S, P)$. On pose $\Sigma = V_n \cup V_t$ où :

Dérivation

- Les ensembles de symboles V_t et V_n doivent être disjoints
- Seuls les symboles terminaux formes les mots du langage
- Les symboles non-terminaux sont là juste pour aider à générer le langage
- a génération du langage commence toujours par le symbole de départ S

Remarque

Souvent, les lettres grecques sont utilisées pour désigner les chaînes construites d'éléments terminaux ou non terminaux comme dans la production précédente.

Dérivation & réécriture

Dérivation

La grammaire G permet de dériver v de u en une étape (notation $u \Rightarrow v$) si et seulement si :

- $u = xu'y$ (u peut être décomposé en trois parties x, u' et y ; les parties x et y peuvent éventuellement être vides),
- $v = xv'y$ (v peut être décomposé en trois parties x, v' et y),
- $u' \rightarrow v'$ (la règle (u', v') est dans P).

$\longrightarrow^+$ désigne la fermeture transitive de $\longrightarrow$

Une grammaire G défini un langage L(G) sur l'alphabet Σ dont les éléments sont les mots engendrés par dérivation à partir du symbole de départ S.

$$L(G) = \{w \in \Sigma^* | S \longrightarrow^+ w\}$$

Hiérarchie de Chomsky

Hiérarchie de Chomsky

On détermine la classe et la complexité des langages (et des grammaires) en fonction d'un certain nombre de contraintes sur la forme des règles de production. 4 ou 5 types de grammaire.

Types de grammaire

- Type 0
- Type 1
- Type 2
- Type 3
- (Type 4)

Hiérarchie de Chomsky

Grammaire de type 0 ou grammaire générale

Les règles ne sont sujettes à aucune restriction. Il suffit que chaque règle fasse intervenir (au moins) un non terminal à gauche.

Exemple / Contre exemple

- $aAbb \rightarrow ba$ ou $aAbB \rightarrow \epsilon$ OK
- $ab \rightarrow ba$ ou $\epsilon \rightarrow aa$ $\overline{OK}$

- Elle correspond aux langages récursivement énumérables
- Le problème de l'analyse pour de tels langages est indécidable. Un langage est décidable si pour toute phrase on peut savoir en temps fini si elle est du langage ou pas
- Les langages correspondants sont reconnus par des machines de Turing.

Hiérarchie de Chomsky

Exemple 1 : Grammaire de type 0 ou grammaire générale

Une grammaire qui engendre tous les mots qui contiennent un nombre égal de a, b et c.

$$G_1 \begin{cases} S \to SABC & AC \to CA & A \to a \\ S \to \epsilon & CA \to AC & B \to b \\ AB \to BA & BC \to CB & C \to c \\ BA \to AB & CB \to BC & \end{cases}$$

Cette grammaire fonctionne en produisant des chaînes de la forme $(ABC)^n$ puis en permutant les non terminaux, et enfin en produisant les chaînes terminales.

Hiérarchie de Chomsky

Exemple 2 : Les mots jumeaux

Langage vu plus pour les machines de Turing.

$$G_2 \begin{cases} S \to \$S'\$ & Aa \to aA & \$a \to a\$ \\ S' \to aAS' & Ab \to bA & \$b \to b\$ \\ S' \to bBS' & Ba \to aB & A\$ \to \$a \\ S' \to \epsilon & Bb \to bB & B\$ \to \$b \\ \$\$ \to \# & & \end{cases}$$

Hiérarchie de Chomsky

Grammaire de type 1 ou grammaire sensible au contexte (context-sensitive) ou monotone

Les règles sont de la forme $\alpha \to \beta$ avec $|\alpha| \leq |\beta|$. On dit alors que le langage engendré est propre. Exceptionnellement, afin d'engendrer le mot vide, on introduit $S \to \epsilon$ pour autant que S n'apparaisse pas dans le membre de droite d'une production.

Définition alternative des grammaires de type 1

Les règles sont de la forme $\alpha A \gamma \to \alpha \beta \gamma$ avec $\beta \in (V_t \cup V_n)^*$
Ou bien de la forme $S \to \epsilon$ et S n'apparaît dans aucun membre de droite. On préfère souvent $S' \to S + \epsilon$ (langage propre).

- Autrement dit, toute règle comprend un non-terminal entouré de deux mots qui décrivent le contexte dans lequel la variable peut être remplacée.

Définition alternative des grammaires de type 1

- Grammaires dites contextuelles car le remplacement d'un élément non-terminal peut dépendre des éléments autour de lui : son contexte.
- Les langages contextuels qui en résultent sont exactement ceux reconnus par une machine de Turing non déterministe à mémoire linéairement bornée, appelés couramment automates linéairement bornés.

Exemple de grammaire de type 1

Grammaire contextuelle pour le langage $L(G_3) = \{a^{2^n} | n > 0\}$

$$G_3 \begin{cases} S \to DT & T \to XT & Xaa \to aaXa \\ S \to AA & T \to aF & XaF \to aaF \\ Daaa \to aaDaa & DaaF \to aaaa & \end{cases}$$

Hiérarchie de Chomsky

Grammaire de type 2 : Grammaire hors-contexte (context-free) ou algébrique

Règles de la forme $A \to \beta$ où $A \in V_n$ et pas de restriction sur β, i-e $\beta \in \Sigma^*$

- Elles sont particulièrement étudiées
- Les langages algébriques correspondants sont reconnus par des automates à pile non-déterministes.
- L'analyse de ces langages est polynomiale.

Exemple

$$P = \{S \longrightarrow aSb, S \longrightarrow ab\}$$

$$\Longrightarrow L(G) = \{a^n b^n \mid n > 0\}$$

Hiérarchie de Chomsky

Grammaire de type 3 : Grammaire régulière

Les règles peuvent prendre deux formes :

- linéaire à gauche : $\alpha \to \beta$ où $\alpha \in V_n$ et $\beta \in V_n V_t^*$.
  i-e $A \to Bw$ ou $A \to w$ avec $A, B \in V_n$ et $w \in V_t^*$
- linéaire à droite : $\alpha \to \beta$ où $\alpha \in N$ et $\beta \in V_t^* V_n$.
  i-e $A \to wB$ ou $A \to w$ avec $A, B \in V_n$ et $w \in V_t^*$

- Les langages réguliers correspondants sont construits et reconnus par des automates finis
- L'analyse de ces langages est polynomiale.

Exemple

$$P = \{A \to aB, A \to a, B \to b\}$$

Hiérarchie de Chomsky

Grammaire de type 4

Les parties droites de toutes les règles sont des terminaux.
Les règles sont de la forme $X \to \alpha$ où $X \in V_n$ et $\alpha \in V_t^*$

- Une telle grammaire ne fait qu'énumérer les phrases de son langage sur V_t .

Hiérarchie de Chomsky

Théorème : Chomsky

1. Les langages réguliers (type 3) sont strictement contenus dans les langages algébriques (type 2).
2. Les langages algébriques propres (type 2) sont strictement contenus dans les langages contextuels (type 1).
3. les langages contextuels (type 1) sont strictement contenus dans les langages récursifs.
4. les langages récursifs sont strictement contenus dans les langages récursivement énumérables (type 0).

Inclusion des grammaires

Au final, on a

$$\textit{Type}\ 4 \subseteq \textit{Type}\ 3 \subseteq \textit{Type}\ 2 \subseteq \textit{Type}\ 1 \subseteq \textit{Type}\ 0$$

Analyse syntaxique

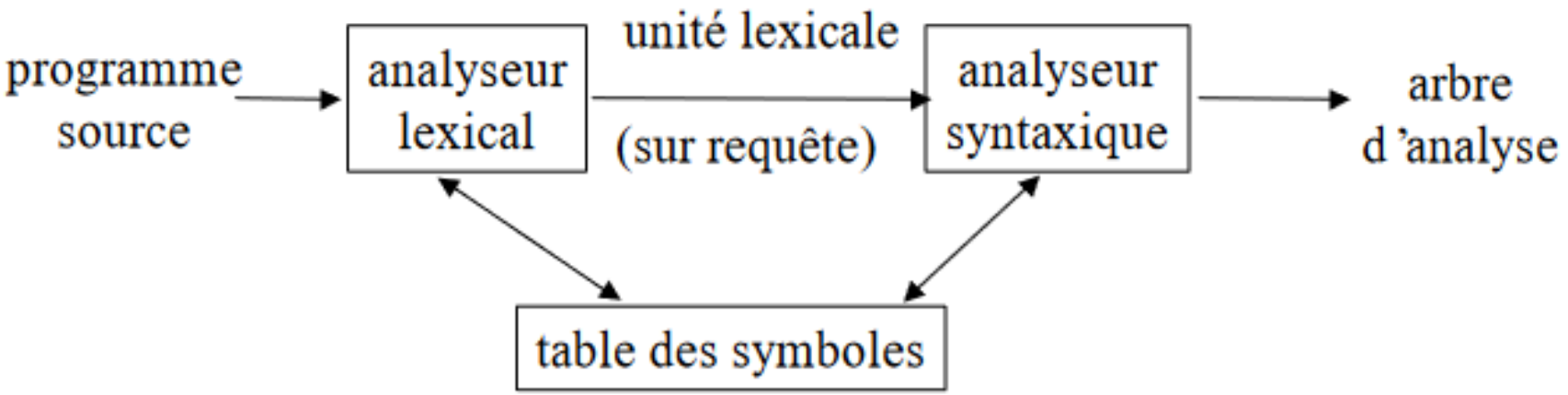
Basée sur les grammaires, l'analyse syntaxique peut vouloir dire deux choses

- Décision (le programme est correct ou non)
- Décomposition syntaxique ou "Parsing" (une représentation de la structure syntaxique est construite pour les programmes corrects).

Composante essentielle de l'implémentation d'un langage de programmation, l'analyse syntaxique a pour rôle de :

- Structurer la suite de lexèmes en fonction d'une grammaire
- Construire un arbre syntaxique
- Détecter et corriger les éventuelles erreurs
- Collecter les informations dans la table des symboles

Rôle de l'analyseur syntaxique



Il joue le rôle principal de la partie frontale. A ce titre, il :

- Active l'analyseur lexical
- Vérifie la conformité syntaxique
- Construit l'arbre d'analyse
- Prépare ou anticipe la traduction
- Gère les erreurs communes de syntaxe.

En bref

- Il s'agit de reconnaître la structure du programme.
- On construit pour cela une représentation structurée (sous forme d'arbre de syntaxe abstraite) du programme.
- Pour l'analyse, on s'appuie sur une grammaire du langage à analyser, en particulier, une grammaire hors-contexte.

Exemple

Considérons l'instruction **"If A=1 then . . . fi"**

Anal. Lexicale : MotReservé(**If**) + Variable(**A**) + OpRel(**=**)+ entier(**1**) + MotRéservé(**then**) + . . . + MotRéservé(**fi**)
Anal. Syntaxique : Si(**MotReservé(If)**) + Expression(**Variable(A)) + OpRel(=)+ entier(1))** + Alors(**MotRéservé(then)**) + BlocInstruction(**. . . + MotRéservé(fi)**)

Types d'analyseurs syntaxiques

- Méthodes universelles (Cocke - Younger - Kasami), permettent une analyse de n'importe quelle grammaire algébrique, mais performance en $O(n^3)$
- Méthodes linéaires en O(n), sur certaines grammaires :
  - analyse ascendante, la plus intuitive, se prête bien à certains types de traductions ;
  - analyse descendante, plus sophistiquée, la plus utilisée par les générateurs automatiques d'analyseurs syntaxiques, car ne nécessite que peu d'adaptations de la grammaire.

Traitement des erreurs

- Diagnostic (messages)
- Redémarrage
  - Mode panique, jusqu'à resynchronisation
  - Correction, difficile si l'erreur est antérieure à sa détection ;
  - Règles d'erreurs, intégrées à la grammaire.

Syntaxe spécifiée par des règles de grammaire (algébrique)

Les grammaires

- Symboles terminaux (= unités lexicales) alphabet A
- Symboles intermédiaires ou variables (= catégories grammaticales) alphabet X
- Règles de grammaire $x \to w$ où $x \in X$ et $w \in (A \cup X)^*$. w est donc un mot quelconque, même vide

Exemples

- *instr* $\to$ **si** *expr* **alors** *instr* **sinon** *instr*
- *phrase* $\to$ sujet verbe complément

- Axiome (= programme)

Langage engendrés

Langage engendré = mots terminaux dérivant de l'axiome.

Rôle de l'analyse syntaxique

Analyse syntaxique

Étant donné un mot terminal, déterminer s'il est ou non engendré par la grammaire ; si oui, en donner un arbre d'analyse.
En général, il existe 3 méthodes bien connues pour ce faire

Méthode universelle : essayer toutes les dérivations à partir de l'axiome, jusqu'à trouver le mot. Des règles de longueur (après modifications) permettent d'éliminer les impasses

Méthode descendante (descente récursive) : une procédure par variable, les terminaux servant à choisir la dérivation (prévision) et à la validation.

Méthode ascendante : on lit le mot (décalage) jusqu'à identifier des dérivation, qu'on réduit et empile (réduction).

Arbre de dérivation

Toute dérivation peut être représentée graphiquement par un arbre appelé arbre de dérivation, défini de la manière suivante :

- la racine de l'arbre est le symbole de départ
- les noeuds intérieurs sont étiquetés par des symboles non terminaux
- si un noeud intérieur e est étiqueté par le symbole S et si la production $S \rightarrow S_1, S_2, \ldots S_k$ a été utilisée pour dériver S alors les fils de e sont des noeuds étiquetés, de la gauche vers la droite, par $S_1, S_2, \ldots S_k$
- les feuilles sont étiquetées par des symboles terminaux et, si on allonge verticalement les branches de l'arbre (sans les croiser) de telle manière que les feuilles soient toutes à la même hauteur, alors, lues de la gauche vers la droite, elles constituent la chaîne w

Arbre de dérivation

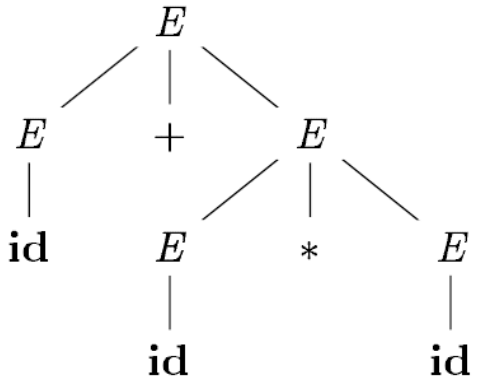
- Grammaire $G_A = (V_n, V_t, S, P)$ pour les expressions arithmétiques où : $V_n = \{E\},\ V_t = \{int, (,), +, -, *, /\},\ S = E,\ P = \{E \rightarrow E + E, E \rightarrow E - E, E \rightarrow E * E, E \rightarrow E/E, E \rightarrow (E), E \rightarrow id\}$.
- Dérivation : $int + int * int \in L(G_A)$ comme le montrent les dérivations suivantes :
  - $E \rightarrow E + E \rightarrow E + E * E \rightarrow id + E * E \rightarrow id + id * E \rightarrow id + id * id$
  - $E \rightarrow E + E \rightarrow id + E \rightarrow id + E * E \rightarrow id + id * E \rightarrow id + id * id$
  - $E \rightarrow E * E \rightarrow E + E * E \rightarrow id + E * E \rightarrow id + int * E \rightarrow id + id * id$

Remarque

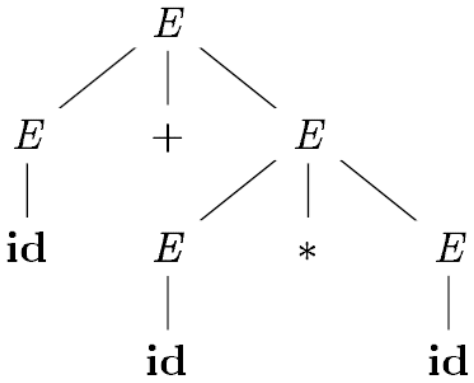
Les deux premières dérivations correspondent à la même façon de comprendre le mot, en voyant $+$ reconnu plus haut que $*$, ce qui n'est pas le cas pour la dernière.

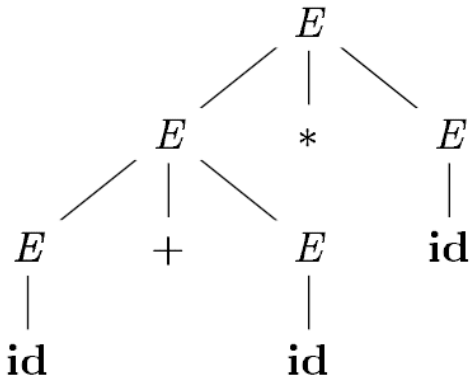
Ambigüité

- Exemple : Arbre de dérivation pour l'expression $id + id * id \in L(G_A)$ pour la dérivation 1 (et 2).



- S'il existe différentes dérivations gauches pour une même chaîne de terminaux, la grammaire est dite ambigüe.





Exemple 1 (1/3) :

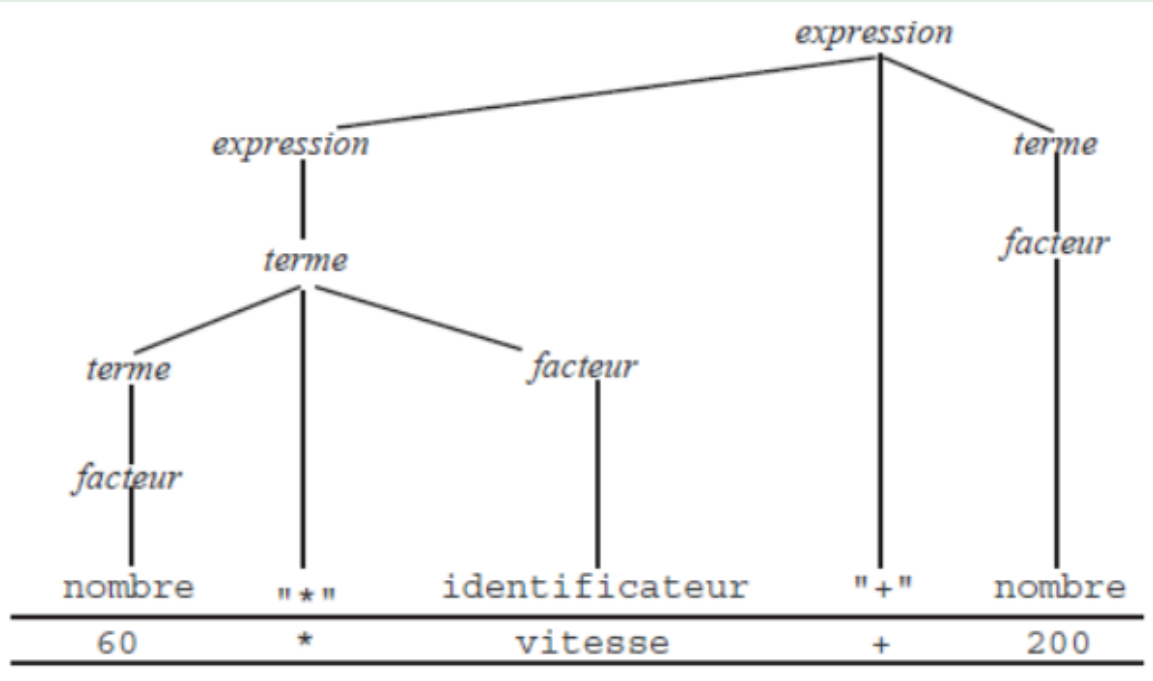
- Grammaire $G_1 = (V_n, V_t, S, P)$ définie par :

$$P = \left\{ \begin{array}{lcl} E & \rightarrow & E + T|T \\ T & \rightarrow & T * F|F \\ F & \rightarrow & (E)|int \end{array} \right\} \quad (1)$$

- L'analyse lexicale de "$200 + 60 * vitesse$" produit :
  $w = (nombre\text{"}+\text{"}\ nombre\text{"}*\text{"}\ identificateur) \in L(G_1)$:

$$\left\{ \begin{array}{lcl} expression & \rightarrow & expression\text{"}+\text{"}\ terme \\ & \rightarrow & terme\text{"}+\text{"}\ terme \\ & \rightarrow & terme\text{"}*\text{"}\ facteur\text{"}+\text{"}\ terme \\ & \rightarrow & facteur\text{"}*\text{"}\ facteur\text{"}+\text{"}\ terme \\ & \rightarrow & nombre\text{"}*\text{"}\ facteur\text{"}+\text{"}\ terme \\ & \rightarrow & nombre\text{"}*\text{"}\ identificateur\text{"}+\text{"}\ terme \\ & \rightarrow & nombre\text{"}*\text{"}\ identificateur\text{"}+\text{"}\ facteur \\ & \rightarrow & nombre\text{"}*\text{"}\ identificateur\text{"}+\text{"}\ nombre \end{array} \right. \quad (2)$$

Exemple 1 (2/3) :



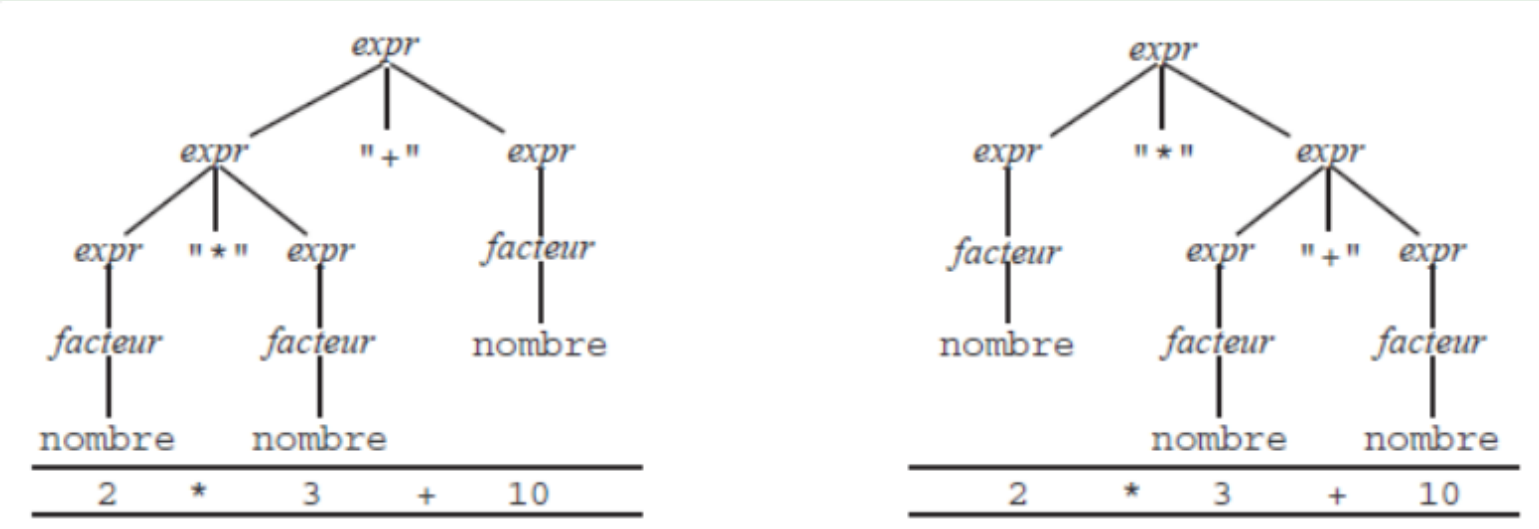
La dérivation précédente est appelée **dérivation gauche** car entièrement composée de dérivations en une étape où à chaque fois c'est le non-terminal le plus à gauche qui est réécrit. On définit de même une **dérivation droite** : à chaque étape c'est le non-terminal le plus à droite qui est réécrit.

Exemple 2 (3/3) :

- Grammaire $G_2 = (V_n, V_t, S, P)$ définie par :

$$P = \begin{cases} expr & \rightarrow & expr\ "+"\ expr | expr\ "*"\ expr | facteur \\ facteur & \rightarrow & nombre | identificateur | "("\ expression\ ")" \end{cases} \quad (3)$$

- 2 arbres de dérivation distincts pour la chaîne "2 * 3 + 10" :



Ambigüité & Suppression de l'ambigüité

Quelques remarques

1. L'ambiguïté est une propriété des grammaires et non des langages.
2. Idéalement, pour permettre le parsing, une grammaire ne doit pas être ambiguë. En effet, l'arbre de dérivation détermine le code généré par le compilateur.
3. On essayera donc de modifier la grammaire pour supprimer ses ambiguïtés.
4. Il n'existe pas d'algorithme pour supprimer les ambiguïtés d'une grammaire context-free.
5. Pour un même langage, certaines grammaires peuvent être ambiguës, d'autres non.

En bref

- D'ailleurs, certains langages context-free sont ambigus de façon inhérente (toutes les grammaires définissant ce langage sont ambiguës). Un tel langage est dit intrinsèquement ambigü.
- L'ambiguïté est, pour nous, nuisible. Il est donc souhaitable de l'éviter. De plus, un analyseur syntaxique déterministe sera plus efficace. Malheureusement, déterminer si une grammaire algébrique est ou non ambiguë est un problème indécidable.
- L'analyse syntaxique consiste, étant donné un mot, à dire s'il est engendré par une grammaire donnée. Si oui, à produire un arbre de dérivation. Les techniques classiques d'analyse descendante et ascendante ne s'appliquent qu'à des grammaires non ambiguës.

Ambigüité & Suppression de l'ambigüité

Exemple de langage intrinsèquement ambigü

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

$$P = \left\{ \begin{array}{lcl} S & \rightarrow & AB \mid C \\ A & \rightarrow & aAb \mid ab \\ B & \rightarrow & cBd \mid cd \\ C & \rightarrow & aCd \mid aDd \\ D & \rightarrow & bDc \mid bc \end{array} \right\} \quad (4)$$

$a^i b^i c^i d^i, \forall i \geq 0$, G a 2 arbres de dérivation. On peut ensuite démontrer que toute autre Grammaire Hors-contexte pour L est ambiguë.

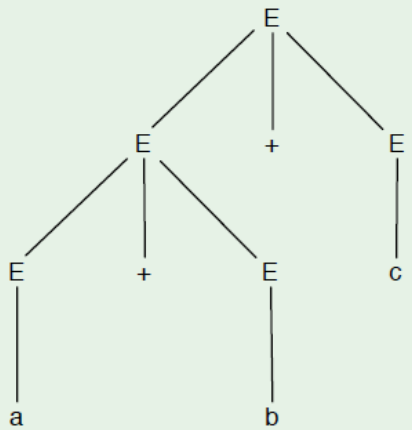
Priorité et associativité : causes d'ambigüité

Lorsque le langage définit des chaînes composés d'instructions et d'opérations, l'arbre syntaxique (qui va déterminer le code produit par le compilateur) doit refléter

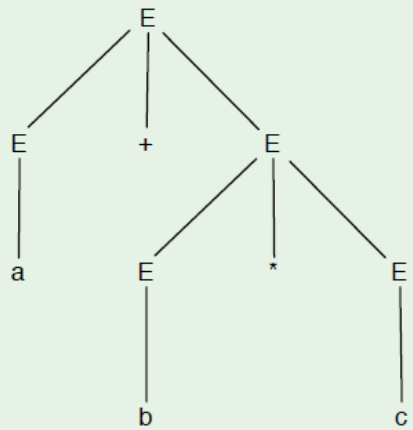
- les priorités et
- les associativités

Arbres associés à des expressions

$a + b + c$



$a + b * c$



Priorité et associativité

- Pour respecter l'associativité à gauche, on n'écrit pas

$$E \rightarrow E + E|T \text{ mais } E \rightarrow E + T|T$$

- Pour respecter les priorités on définit plusieurs niveaux de variables / règles (le symbole de départ ayant le niveau 0) : les opérateurs les moins prioritaires sont définis à un niveau plus bas (plus proche du symbole de départ) que les plus prioritaires. On écrit en 2 niveaux

$$\text{Non pas } P = \left\{ \begin{array}{lll} E & \rightarrow & T + E \mid T * E \mid T \\ T & \rightarrow & id \mid (E) \end{array} \right\} \quad (5)$$

Mais plutôt ??? (6)

Priorité et associativité

- Pour respecter l'associativité à gauche, on n'écrit pas

$$E \to E + E | T \text{ mais } E \to E + T | T$$

- Pour respecter les priorités on définit plusieurs niveaux de variables / règles (le symbole de départ ayant le niveau 0) : les opérateurs les moins prioritaires sont définis à un niveau plus bas (plus proche du symbole de départ) que les plus prioritaires. On écrit en 2 niveaux

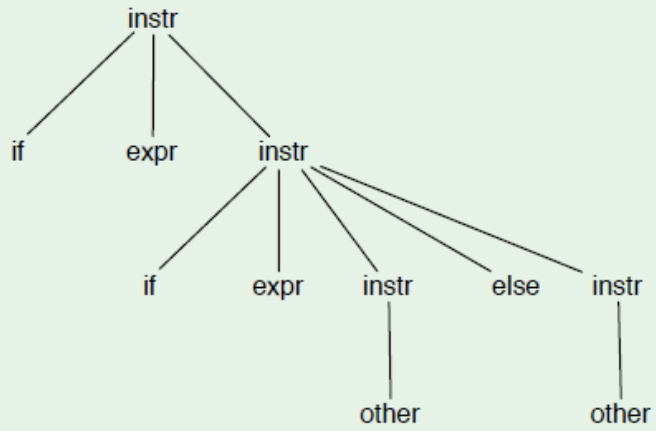
$$\text{Non pas } P = \left\{ \begin{array}{lll} E & \to & T + E \mid T * E \mid T \\ T & \to & id \mid (E) \end{array} \right\} \tag{7}$$

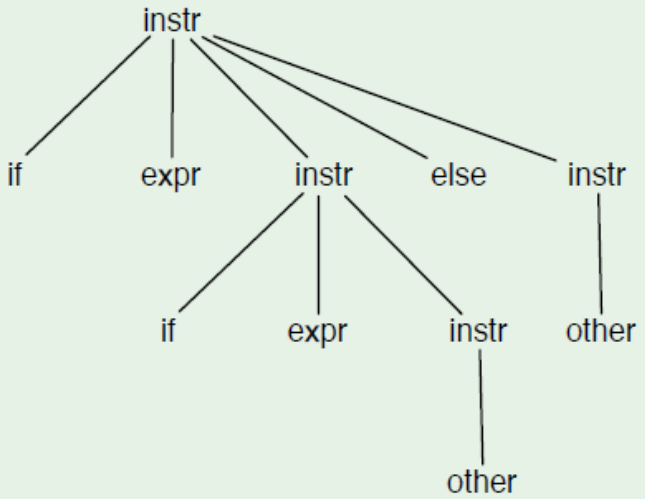
$$\text{Mais plutôt } P = \left\{ \begin{array}{lll} E & \to & T + E \mid T \\ T & \to & F * T \mid F \\ F & \to & id \mid (E) \end{array} \right\} \tag{8}$$

Dérivation

Associativité de l'instruction If

instr $\rightarrow$ *if expr instr*|*if expr instr else instr*|*other* est ambigüe





Dans les langages impératifs habituels c'est l'arbre de gauche qu'il faut générer. Grammaire pouvant être transformée . . .

Associativité de l'instruction If

$instr \rightarrow if\ expr\ instr | if\ expr\ instr\ else\ instr | other$ est ambigüe

- En réalité, par convention, on fait correspondre chaque sinon avec le si qui le précède, le plus proche et sans correspondant.
- L'idée est qu'une instruction apparaissant entre un alors et un sinon doit être "close", c-à-d qu'elle ne doit pas se terminer par un alors sans correspondant.
- Une instruction close est soit une instruction si-alors-sinon ne contenant pas d'instruction non close, soit une instruction non conditionnelle quelconque.

$$\text{Et on a } P = \left\{ \begin{array}{lcl} instr & \rightarrow & open | close \\ close & \rightarrow & if\ expr\ close\ else\ close | other \\ open & \rightarrow & if\ expr\ instr \\ open & \rightarrow & if\ expr\ close\ else\ open \end{array} \right\} \quad (9)$$

Simplification des grammaires

Processus de simplification des grammaires

Il existe des algorithmes pour transformer une grammaire hors-contexte en une grammaire équivalente plus simple. Cela se passe en trois étapes :

- Etape 1 Élimination des symboles inutiles.
- Etape 2 Suppression des ϵ-productions.
- Etape 3 Productions unitaires.

Etape 1 : Élimination des symboles inutiles

1. Eliminer les variables d'où ne dérive aucun mot en symboles terminaux. Pour cela
   - 1.1 Les variables dont une production au moins ne contient que des terminaux sont utiles
   - 1.2 Les variables dont une production au moins ne contient que des terminaux et des symboles utiles sont utiles.
2. Eliminer tous les symboles (terminaux ou non) n'appartenant à aucun métamot dérivé de S.
   - 2.1 L'axiome est utile
   - 2.2 Les symboles apparaissant dans les productions de symboles utiles sont utiles.

A chaque étape (1 et 2), les symboles non retenus sont inutiles, on les enlève et on a encore une grammaire équivalente. L'ordre a de l'importance.

Application

Exemple :

- Grammaire $G_2 = (V_n, V_t, S, P)$ définie par :

$$P = \left\{ \begin{array}{lllllllll} S & \rightarrow & AB \,|\, CA, & A & \rightarrow & a, & B & \rightarrow & AB \,|EA, \\ C & \rightarrow & aB \,|b, & D & \rightarrow & aC, & E & \rightarrow & BA \end{array} \right\} \quad (10)$$

- Qu'obtient-on à l'étape 1 ? ? ?
  Il reste donc :

$$S \rightarrow CA,\ A \rightarrow a,\ C \rightarrow b,\ D \rightarrow aC$$

- Qu'obtient-on à l'étape 2 ? ? ?
- Que reste-t-il finalement ? ? ?

Application

Exemple :

- Grammaire $G_2 = (V_n, V_t, S, P)$ définie par :

$$P = \left\{ \begin{array}{lllllllll} S & \rightarrow & AB \,|\, CA, & A & \rightarrow & a, & B & \rightarrow & AB \,|EA, \\ C & \rightarrow & aB \,|b, & D & \rightarrow & aC, & E & \rightarrow & BA \end{array} \right\} \quad (11)$$

- L'étape 1 de l'algo conserve A et C puis S et D.
  Il reste donc :

$$S \rightarrow CA,\ A \rightarrow a,\ C \rightarrow b,\ D \rightarrow aC$$

- L'étape 2 de l'algo élimine D.
  Il reste finalement :

$$S \rightarrow CA,\ A \rightarrow a,\ C \rightarrow b$$

Etape 2 : Suppression des ϵ-productions.

1. Les ϵ-productions ont comme inconvénient que, dans une dérivation, la longueur des métamots peut décroître. Pour des besoins d'analyse, il est préférable d'éviter cette situation. Si ϵ appartient à L(G), il n'est bien sûr pas possible d'éviter toutes les ϵ-productions.
2. On traite donc uniquement les langages non contextuels dont on a éventuellement enlevé le mot vide.
3. On cherche récursivement les variables annulables, celles d'où dérive le mot vide ; on part d'une grammaire sans symbole inutile :
   - Les variables qui se réécrivent ϵ sont annulables ;
   - Les variables dont une production au moins ne contient que des variables annulables sont annulables.
4. L'ensemble ANNUL(G) des variables annulables de G étant déterminé, on modifie les productions contenant des variables annulables.
5. On remplace dans une production $A \to a$ les symboles annulables

Application

Exemple :

- La grammaire suivante engendre les mots bien parenthésés :

$$S \to S(S) \mid \epsilon$$

Celle qui suit engendre les mêmes mots, sauf le mot vide.

$$S \to S(S) \mid (S) \mid S() \mid ()$$

- Pour obtenir le mot vide, on tolère une seule production vide, sur l'axiome, et sans autoriser celui-ci à apparaître en partie droite de production. On rajoute donc un nouvel axiome, ici T :

$$T \to S \mid \epsilon,\ S \to S(S) \mid (S) \mid S() \mid ()$$

Etape 3 : Productions unitaires.

- Il s'agit des productions $A \to B$, où B est une variable.
- On suppose que G n'a aucune variable inutile, ni production vide.
- On cherche toutes les dérivations de la forme $A \Rightarrow^* B$. Cela se fait récursivement à partir des productions unitaires. Chaque fois qu'une telle dérivation est obtenue, on ajoute aux productions de A toutes les productions non unitaires de B. Enfin, on efface les productions unitaires.
- La grammaire ainsi obtenue a peut-être des symboles inutiles, qu'on supprime. La grammaire finale est équivalente à la grammaire de départ, n'a pas de symboles inutiles, de productions vides ni de productions unitaires.
- **Exemple :** Pour la grammaire précédente, on obtiendrait :

$$T \to \epsilon \mid S(S) \mid (S) \mid S() \mid () \quad S \to \mid S(S) \mid (S) \mid S() \mid ()$$

Récursivité à gauche

- Une grammaire est récursive à gauche s'il existe un non-terminal A et une dérivation de la forme $A \Rightarrow^* A\alpha$ où α est une chaîne quelconque. Une récursivité à gauche est simple si la grammaire possède une production $A \rightarrow A\alpha$.
- Exemple : G_1 est récursive à gauche, et même simplement

$$P = \left\{ \begin{array}{lcl} \textit{expression} & \rightarrow & \textit{expression}\,''+''\,\textit{terme} | \textit{terme} \\ \textit{terme} & \rightarrow & \textit{terme}\,''*''\,\textit{facteur} | \textit{facteur} \\ \textit{facteur} & \rightarrow & \textit{nombre} | \textit{identificateur} | ''('' \textit{expression} '')'' \end{array} \right\} ($$

- Une grammaire G est non ambiguë, si pour tout mot terminal il existe au plus une dérivation gauche.
- Idéalement, pour permettre le parsing, une grammaire ne doit pas être ambiguë. En effet, l'arbre de dérivation détermine le code généré par le compilateur.
- Souvent, on essaie donc de modifier la grammaire pour supprimer ses ambiguïtés.

Elimination de la récursion gauche

- Il existe une méthode pour obtenir une grammaire non récursive à gauche équivalente à une grammaire donnée.
- Dans le cas de la récursivité à gauche simple, cela consiste à remplacer une production telle que $A \rightarrow A\alpha \mid \beta$ par les deux productions $A \rightarrow \beta\ A'$ et $A' \rightarrow \alpha\ A' \mid \epsilon$
- En appliquant à G_1 caractérisée par P, on a P'.

$$P = \left\{ \begin{array}{lcl} \textit{expression} & \rightarrow & \textit{expression}\text{ '' + '' }\textit{terme} | \textit{terme} \\ \textit{terme} & \rightarrow & \textit{terme}\text{ '' * '' }\textit{facteur} | \textit{facteur} \\ \textit{facteur} & \rightarrow & \textit{nombre} | \textit{identificateur} | \text{'' ('' }\textit{expression}\text{'')''} \end{array} \right\} (13)$$

$$P' = ??? \quad (14)$$

Elimination de la récursion gauche

- Il existe une méthode pour obtenir une grammaire non récursive à gauche équivalente à une grammaire donnée.
- Dans le cas de la récursivité à gauche simple, cela consiste à remplacer une production telle que $A \rightarrow A\alpha \mid \beta$ par les deux productions $A \rightarrow \beta\ A'$ et $A' \rightarrow \alpha\ A' \mid \epsilon$
- En appliquant à G_1 caractérisée par P, on a P'.

$$P = \left\{ \begin{array}{lcl} \textit{expression} & \rightarrow & \textit{expression}\ \text{"}+\text{"}\ \textit{terme} | \textit{terme} \\ \textit{terme} & \rightarrow & \textit{terme}\ \text{"}*\text{"}\ \textit{facteur} | \textit{facteur} \\ \textit{facteur} & \rightarrow & \textit{nombre} | \textit{identificateur} | \text{"(" } \textit{expression} \text{")"} \end{array} \right\} ($$

$$P' = \left\{ \begin{array}{lcl} \textit{expression} & \rightarrow & \textit{terme fin\_expression} \\ \textit{fin\_expression} & \rightarrow & \text{"}+\text{"}\ \textit{terme fin\_expression} \mid \epsilon \\ \textit{terme} & \rightarrow & \textit{facteur fin\_terme} \\ \textit{fin\_terme} & \rightarrow & \text{"}*\text{"}\ \textit{facteur fin\_terme} | \epsilon \\ \textit{facteur} & \rightarrow & \textit{nombre} | \textit{identificateur} | \text{"(" } \textit{expression} \text{")"} \end{array} \right\} ($$

Elimination de la récursion gauche

- Dans le cas de la récursivité à gauche indirecte, comme dans l'exemple $S \rightarrow A\ a \mid b,\ A \rightarrow Ac \mid Sd \mid \epsilon$, on peut appliquer l'algorithme suivant :

Algorithme de suppression de la récursivité à gauche

Ranger les non-terminaux dans un ordre $A_1, \ldots, A_n$.
for (i in 1..n) {
    for (j in 1..i-1) {
        remplacer chaque production de la forme $A_i \rightarrow A_j\gamma$
        par les productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ où
        $Aj \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
            }
Eliminer les récursions simples parmi les production A_i
        }

Factorisation gauche

- Un analyseur syntaxique étant prédictif, à tout moment le choix entre productions qui ont le même membre gauche doit pouvoir se faire, sans risque d'erreur, en comparant le symbole courant de la chaîne à analyser avec les symboles susceptibles de commencer les dérivations des membres droits des productions en compétition.
- Une grammaire contenant des productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ viole ce principe car lorsqu'il faut choisir entre les productions $A \rightarrow \alpha\beta_1$ et $A \rightarrow \alpha\beta_2$ le symbole courant est un de ceux qui peuvent commencer une dérivation de α et on ne peut choisir à coup sûr entre $\alpha\beta_1$ et $\alpha\beta_2$
- La factorisation à gauche est donc cette transformation simple qui corrige ce défaut (si les symboles susceptibles de commencer une réécriture de β_1 sont distincts de ceux pouvant commencer une réécriture de β_2) :

Exemples

- **Classique :** Les grammaires de la plupart des langages de programmation définissent ainsi l'instruction conditionnelle

$$instr\_si \rightarrow si\ expr\ alors\ instr | si\ expr\ alors\ instr\ sinon\ instr$$

Pour avoir un analyseur prédictif dans ce cas, il faudra opérer une factorisation à gauche pour avoir

$$P = \left\{ \begin{array}{lcl} instr\_si & \rightarrow & si\ expr\ alors\ instr\ fin\_instr\_si \\ fin\_instr\_si & \rightarrow & sinon\ instr \mid \epsilon \end{array} \right\} \quad (17)$$

- La factorisation de la grammaire $S \rightarrow ABCD | ABaD | Aba$ aboutit à

$$\left\{ \begin{array}{lcl} S & \rightarrow & AE \\ E & \rightarrow & BF \mid ba \\ F & \rightarrow & CD | aD \end{array} \right\} \quad (18)$$

Définition

Une grammaire hors-contexte est sous Forme Normale de Chomsky si ses règles ont l'une des deux formes :

$$\left.\begin{array}{lcl} X & \rightarrow & YZ \\ X & \rightarrow & a \end{array}\right\} avec \left\{\begin{array}{lcl} X, Y, Z & \in & V_n \\ a & \in & V_t \end{array}\right. \tag{19}$$

Une grammaire hors-contexte est sous CNF étendue si ses règles peuvent également prendre les formes :

$$\left.\begin{array}{lcl} X & \rightarrow & YZ \\ X & \rightarrow & Y \\ X & \rightarrow & a \end{array}\right\} avec \left\{\begin{array}{lcl} X, Y, Z & \in & V_n \\ a & \in & V_t \end{array}\right. \tag{20}$$

Mise sous CNF

Théorème

Pour toute grammaire hors-contexte, il existe une grammaire hors-contexte équivalente sous forme normale.

Théorème

- Définition
  Deux grammaires sont dites équivalentes si elles peuvent produire les mêmes chaînes de symboles terminaux.
- Grammaires de type 2 : passage d'un arbre syntaxique à un arbre équivalent sous CNF = facile

Méthode (En 3 temps :)

1. Suppression des règles de type : $X \rightarrow \alpha t_i \beta$ (où t_i est un terminal et α et/ou β sont non vides)
   - 1.1 Créer un non-terminal T_i
   - 1.2 Ajouter la règle $T_i \rightarrow t_i$
   - 1.3 Remplacer la règle $X \rightarrow \alpha t_i \beta$ par $X \rightarrow \alpha T_i \beta$
2. Suppression des règles de type : $X \rightarrow Y$
   - 2.1 Pour chaque règle $Z \rightarrow \alpha X \beta$, ajouter une règle $Z \rightarrow \alpha Y \beta$
   - 2.2 Supprimer $X \rightarrow Y$.
3. Suppression des règles de type : $X \rightarrow YZ\alpha$
   - 3.1 Créer un nouveau non-terminal X_i
   - 3.2 Ajouter la règle $X_i \rightarrow Z\alpha$
   - 3.3 Remplacer la règle $X \rightarrow YZ\alpha$ par $X \rightarrow YX_i$

Cette méthode de mise sous CNF augmente considérablement le nombre de non-terminaux et de règles.

Application

Exemple

$$R = \left\{ \begin{array}{llll} P & \rightarrow & SN\ SV, & SN \rightarrow Det\ N, \\ SN & \rightarrow & Det\ N\ SP, & SP \rightarrow Prep\ SN, \\ SV & \rightarrow & V, & SV \rightarrow V\ SN, \\ SV & \rightarrow & V\ SN\ SP, & SV \rightarrow mange \end{array} \right\} \quad (21)$$

Règle	Forme initiale	Règle	Forme Normale de Chomsky
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$

Application

Exemple

Règle	Forme initiale	Règle	Forme Normale de Chomsky
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$

Application

Exemple

Règle	Forme initiale	Règle	Forme Normale de Chomsky
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$:	$X_1 \rightarrow N\ SP$
		$R_{3.2}$:	$SN \rightarrow Det\ X_1$

Application

Exemple

Règle	Forme initiale	Règle	Forme Normale de Chomsky
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \rightarrow N\ SP$ $SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$

Application

Exemple

Règle	Forme initiale	Règle	Forme Normale de Chomsky
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$:	$X_1 \rightarrow N\ SP$
		$R_{3.2}$:	$SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow V$	$R_{1.2}$	$P \rightarrow SN\ V$

Application

Exemple

Règle	Forme initiale	Règle	Forme Normale de Chomsky
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$:	$X_1 \rightarrow N\ SP$
		$R_{3.2}$:	$SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow SV$	$R_{1.2}$	$P \rightarrow SN\ V$
R_6 :	$SV \rightarrow V\ SN$	R_6 :	$SV \rightarrow V\ SN$

Application

Exemple

Règle	Forme initiale	Règle	Forme Normale de Chomsky
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \rightarrow N\ SP$ $SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow SV$	$R_{1.2}$	$P \rightarrow SN\ V$
R_6 :	$SV \rightarrow V\ SN$	R_6 :	$SV \rightarrow V\ SN$
R_7 :	$SV \rightarrow V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \rightarrow SN\ SP$ $SV \rightarrow V\ X_2$

Application

Exemple

Règle	Forme initiale	Règle	Forme Normale de Chomsky
R_1 :	$P \rightarrow SN\ SV$	R_1 :	$P \rightarrow SN\ SV$
R_2 :	$SN \rightarrow Det\ N$	R_2 :	$SN \rightarrow Det\ N$
R_3 :	$SN \rightarrow Det\ N\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_1 \rightarrow N\ SP$ $SN \rightarrow Det\ X_1$
R_4 :	$SP \rightarrow Prep\ SN$	R_4 :	$SP \rightarrow Prep\ SN$
R_5 :	$SV \rightarrow SV$	$R_{1.2}$	$P \rightarrow SN\ V$
R_6 :	$SV \rightarrow V\ SN$	R_6 :	$SV \rightarrow V\ SN$
R_7 :	$SV \rightarrow V\ SN\ SP$	$R_{3.1}$: $R_{3.2}$:	$X_2 \rightarrow SN\ SP$ $SV \rightarrow V\ X_2$
R_8 :	$SV \rightarrow mange$	R_8 :	$SV \rightarrow mange$

GNF & Théorème

- Une grammaire hors-contexte est sous Forme Normale de Greibach si ses règles sont de la forme :

$$X \to a\alpha \text{ avec } a \in V_t \text{ et } \alpha \in V_n^*$$

- **Théorème :** Tout langage non contextuel sans le mot vide peut être engendré par une grammaire sans symbole inutile dont toutes les productions sont de la forme $X \to aB_1B_2 \ldots B_m$ où $m \geq 0$.
- **Démonstration :** On part d'une grammaire G sous forme normale de Chomsky.
  On ordonne (arbitrairement) les variables : $A_1, A_2, \ldots, A_m$; c'est une bonne idée de supposer que la première est l'axiome.

Les productions considérées seront de trois types possibles :

type 1 $A_i \to a\alpha$, où $\alpha \in \{A_i + B_j\}^*$;

type 2 $A_i \to A_j\alpha$, où $\alpha \in \{A_i\}\{A_i + B_j\}^*$ *et* $j > i$;

type 3 $A_i \to A_j\alpha$, où $\alpha \in \{A_i\}\{A_i + B_j\}^*$ *et* $j \leq i$;

Au départ, les trois types sont les seuls présents (Chomsky).
Dans un premier temps, on va éliminer les productions de type 3, quitte à introduire de nouvelles variables B_i et des productions de la forme $B_i \to A_j\beta$, où $\beta \in \{A_i + B_j\}^*$.
Supposons qu'on a pu éliminer les productions de type 3 pour les variables jusqu'à A_{i-1} en introduisant les B_j correspondantes.
On considère les productions de type 3 dans l'ordre croissant des indices j : $A_i \to A_1\alpha$ et on remplace A_1 par ses productions. On fait ainsi apparaître des productions des trois types, mais plus aucune de type 3 commençant par A_1. Puis on fait de même pour $A_2, \ldots, A_{i-1}$.
Les seules productions de type 3 restantes sont directement récursives à gauche.

Les productions de A_i sont des types :

type 1 $A_i \to a\alpha$, où $\alpha \in \{A_i + B_j\}^*$;

type 2 $A_i \to A_j\alpha$, où $\alpha \in \{A_i\}\{A_i + B_j\}^*$ *et* $j > i$;

type 3 $A_i \to A_i\gamma$, où $\gamma \in \{A_i\}\{A_i + B_j\}^*$;

On élimine le type 3 par élimination de la récursivité directe à gauche : on introduit donc la variable B_i et on obtient des productions : $A_i \to a\alpha \mid a\alpha B_i,\ A_i \to A_j\alpha \mid A_j\alpha B_i,\ B_i \to \gamma \mid \gamma B_i$. Les productions restantes sont bien de types 1 ou 2, et les productions de B_i commencent bien par une variable A_j.

On considère maintenant les productions de type 2 par ordre décroissant des indices.

La dernière variable A_m ne peut en avoir. Dans les productions de A_{m-1} de type 2, seule apparaît A_m, qu'on remplace par ses productions et ainsi de suite jusqu'à A_1.

- Maintenant, toutes les productions des A_i sont de type 1. Dans les productions des B_i, on remplace le premier symbole, qui est un A_i, par les productions de celui-ci.
  La grammaire obtenue est sous forme normale de Greibach.

- **Remarques :**
  Cela accroît le nombre de productions de façon exponentielle.
  Dans la pratique, il n'est pas indispensable que la grammaire soit sous forme normale de Chomsky, mais le déroulement de l'algorithme peut en être perturbé.
- **Exemple :** On considère la grammaire initiale :
  $A_1 \to A_2A_3, A_2 \to A_3A_1 \mid b,\ A_3 \to A_1A_2 \mid a$

- Première étape : A_1 et A_2 sans changements ; pour A_3, garder $A_3 \to a$; puis $A_3 \to A_2A_3A_2$, puis $A_3 \to A_3A_1A_3A_2 | bA_3A_2$; les productions de A_3 sont donc : $A_3 \to A_3A_1A_3A_2$ et $A_3 \to bA_3A_2 | a$.
- Élimination de la récursivité directe à gauche :
  $A_3 \to bA_3A_2 | a | bA_3A_2B_3 | aB_3$ et $B_3 \to A_1A_3A_2 | A_1A_3A_2B_3$.
- Comme prévu, les productions de A_3 sont de type 2. Utilisons-les pour récrire les productions de A_2 :
  $A_2 \to bA_3A_2A_1 | aA_1 | bA_3A_2B_3A_1 | aB_3A_1 | b$.
- Réécrivons maintenant les productions de A_1 :
  $A_1 \to bA_3A_2A_1A_3 | aA_1A_3 | bA_3A_2B_3A_1A_3 | aB_3A_1A_3 | bA_3$,
- puis les productions de B_3 :
  $B_3 \to bA_3A_2A_1A_3A_3A_2 | aA_1A_3A_3A_2 | bA_3A_2B_3A_1A_3A_3A_2$
  $B_3 \to aB_3A_1A_3A_3A_2 | bA_3A_3A_2 | bA_3A_2A_1A_3A_3A_2B_3$
  $B_3 \to aA_1A_3A_3A_2B_3 | bA_3A_2B_3A_1A_3A_3A_2B_3$
  $B_3 \to aB_3A_1A_3A_3A_2B_3 | bA_3A_3A_2B_3$.

Grammaires linéaires droites

- C'est une grammaire dont les seules productions sont $A \to aB$ ou $A \to a$.
- Elles sont sous forme normale de Greibach.

Théorème

Un langage est engendré par une grammaire linéaire droite si et seulement s'il est régulier sans le mot vide.

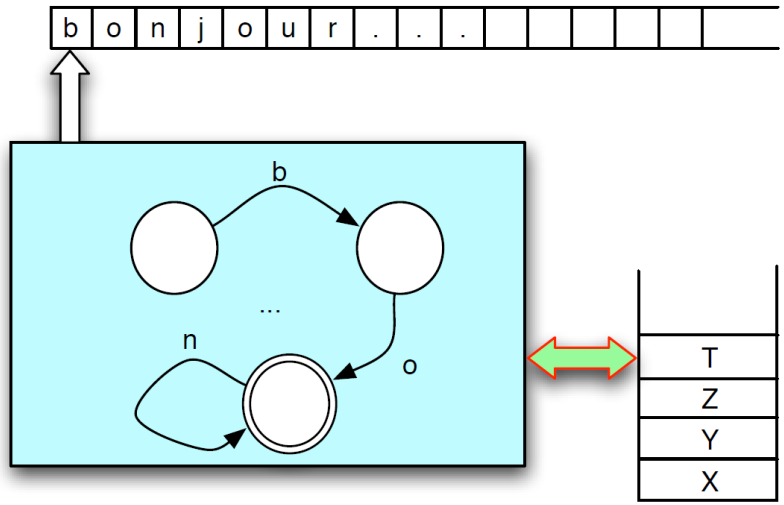
- ($\Rightarrow$) Suppose $G = (V_n, V_t, S, P)$ linéaire droite. $M = (\{a\}, V_n \cup \{f\}, q_0, \{f\}, \delta)$ un AFN où f correspond aux états finaux. On montre par récurrence sur la longueur de w=ax que $(f \in \delta(A, w)) \Leftrightarrow A \Rightarrow^* w$. D'où $L(G) = L(M)$
- ($\Leftarrow$) Réciproquement, soit $M = (V, T, d, q_0, F)$ un AFD. Soit $G = (V, T, P, q_0)$ la grammaire d'axiome q_0 dont les productions P sont : $q_i \to aq_j$ si $q_j = \delta(q_i, a) \notin F$ et $q_i \to aq_j \mid a$ si $q_j = \delta(q_i, a) \in F$. On a bien $L(M) = L(G)$.

Plan

Automate à Pile ou PDA (PDA pour Push Down Automata)

Un PDA est essentiellement un $\epsilon - AFN$ avec une pile (stack).
Lors d'une transition, le PDA :

1. Consomme un symbole d'entrée (ou non si ϵ-transition)
2. Change d'état de contrôle
3. Remplace le symbole T en sommet de la pile par un string (ϵ (pop), T (pas de changement), AT (push un A))



Présentation informelle

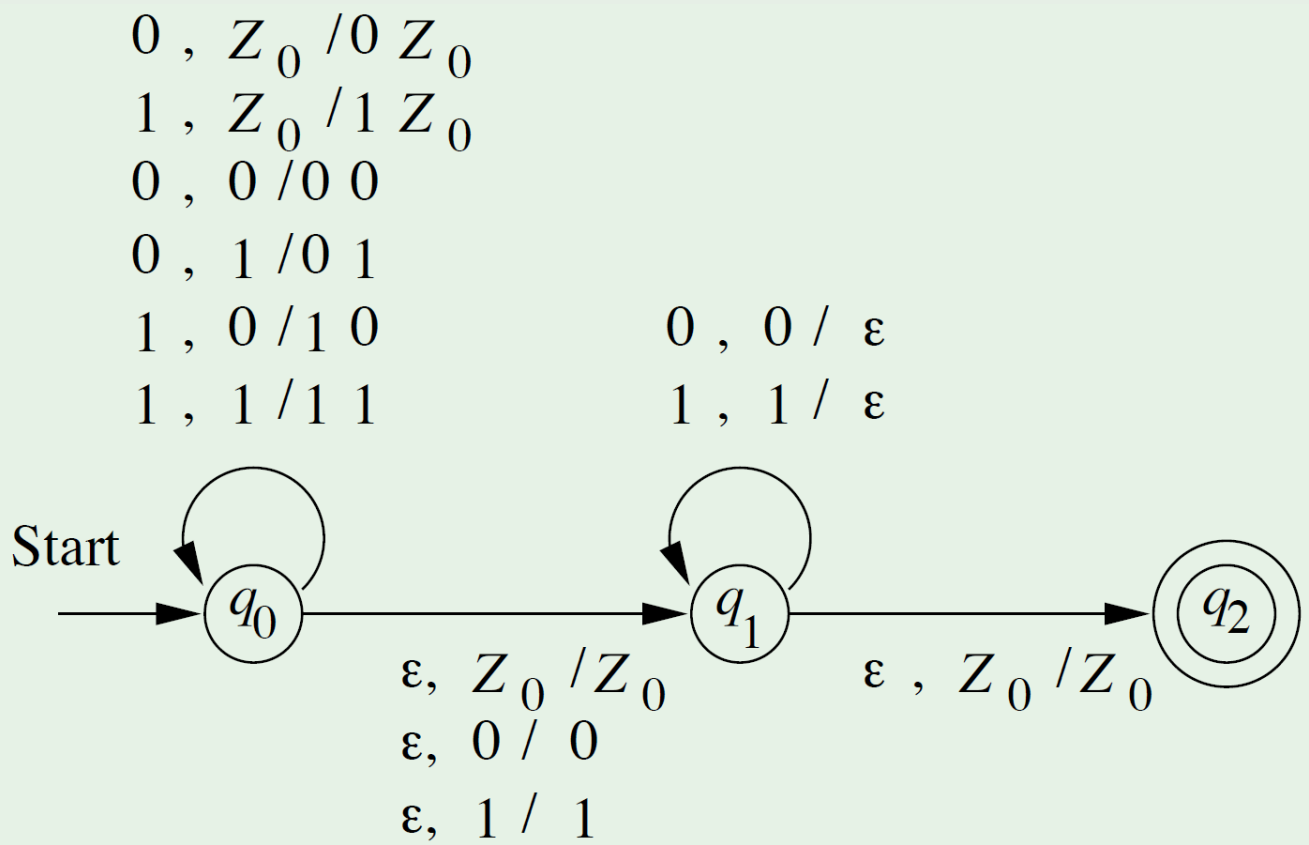
Exemple (PDA pour $L_{ww^R} = \{WW^R \mid W \in \{0,1\}\}$))

La grammaire correpondante est $P \rightarrow 0P0 \mid 1P1 \mid \epsilon$.
On peut construire un PDA équivalent à 3 états qui fonctionne comme suit :

- E0 Il peut supposer (guess) lire w : push le symbole sur la pile
- E0 Il peut supposer être au milieu de l'input : va dans l'état 1
- E1 Il compare ce qui est lu et ce qui est sur la pile : s'ils sont identiques, la comparaison est correcte, il pop le sommet de la pile et continue (sinon bloque)
- E2 S'il retombe sur le symbole initial de la pile, va dans l'état 2 (accepteur).

Présentation informelle

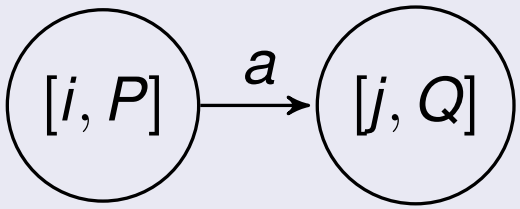
Exemple (PDA pour $L_{WW^R} = \{WW^R \mid W \in \{0,1\}\}$))



Définition formelle d'un PDA

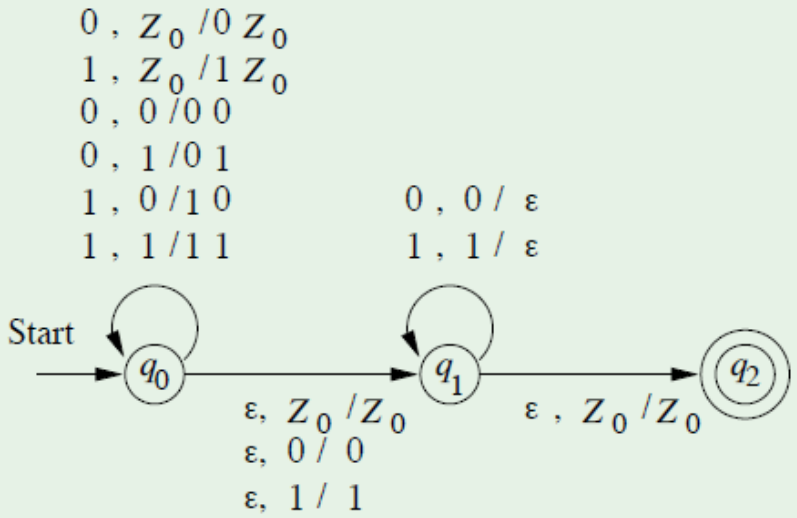
Un PDA est un heptuplet $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ tel que :

- Q est l'ensemble fini des états dont q_0 est l'initial (unique)
- Σ est un ensemble fini de symboles (alphabet d'entrée)
- Γ est un ensemble fini de symboles (pile) dont Z est l'initial.
- $\delta : (Q \times \Sigma^* \times V^*) \rightarrow (Q \times V^*)$ est l'ensemble des relations de transitions
- La transition $([i, P], a, [j, Q])$ fait passer de l'état i à l'état j **en dépilant P** puis **en empilant Q**.



- $F \subseteq Q$ est un ensemble d'états accepteurs

Exemple Illustration (1/3)

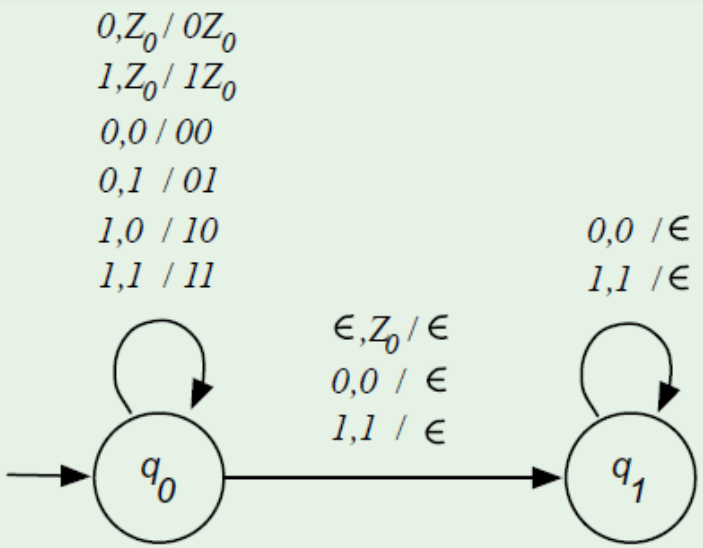


$$P = \langle \{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\} \rangle$$

δ	$(0, Z_0)$	$(1, Z_0)$	$(0, 0)$	$(0, 1)$	$(1, 0)$	$(1, 1)$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00)\}$	$\{(q_0, 01)\}$	$\{(q_0, 10)\}$	$\{(q_0, 11)\}$
q_1	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$
$*q_2$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

δ	(ϵ, Z_0)	$(\epsilon, 0)$	$(\epsilon, 1)$
$\rightarrow q_0$	$\{(q_1, Z_0)\}$	$\{(q_1, 0)\}$	$\{(q_1, 1)\}$
q_1	$\{(q_2, Z_0)\}$	$\emptyset$	$\emptyset$
$*q_2$	$\emptyset$	$\emptyset$	$\emptyset$

Exemple Illustration (2/3)



$$P = \langle \{q_0, q_1\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \emptyset \rangle$$

δ	$(0, Z_0)$	$(1, Z_0)$	$(0, 0)$	$(0, 1)$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00), (q_1, \epsilon)\}$	$\{(q_0, 01)\}$
q_1	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$	$\emptyset$

δ	$(1, 0)$	$(1, 1)$	(ϵ, Z_0)
$\rightarrow q_0$	$\{(q_0, 10)\}$	$\{(q_0, 11), (q_1, \epsilon)\}$	$\{(q_1, \epsilon)\}$
q_1	$\emptyset$	$\{(q_1, \epsilon)\}$	$\{(q_1, \epsilon)\}$

Autres variantes

Table de transitions

Un automate à pile peut se décrire aussi à l'aide d'une table de transitions dans laquelle les transitions sont spécifiées.

- Les lignes correspondent aux états,
- Les colonnes correspondent aux étiquettes possibles.
- Dans les cases, figurent les triplets (état d'arrivée ; symbole dépilé ; symbole empilé) ; il peut y en avoir plusieurs par case, ou aucun. Les états particuliers sont signalés dans les dernières colonnes.

Applications

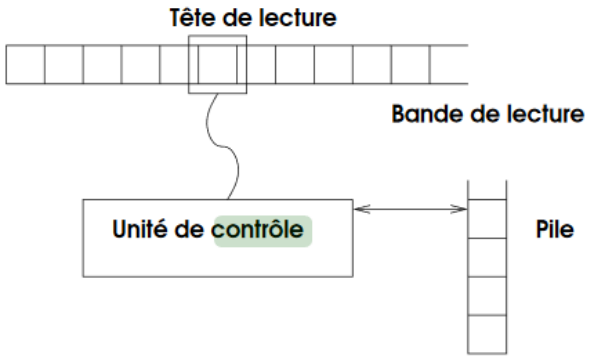
Exemple 1 : Pour $a^n b^n$

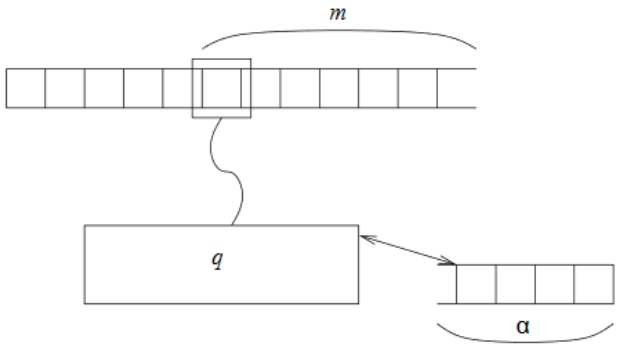
	a	**b**	ϵ	**accept**
S_0	$(1, \epsilon, u)$			x
1	$(1, \epsilon, u)$	$(2, u, \epsilon)$		
2		$(2, u, \epsilon)$		x

Exemple 2 : Le palindrome

	a	**b**	ϵ	**accept**
S_0	(S_0, ϵ, P) $(1, \epsilon, \epsilon)$	(S_0, ϵ, Q) $(1, \epsilon, \epsilon)$	$(1, \epsilon, \epsilon)$ $(1, \epsilon, \epsilon)$	
1	$(1, \epsilon, u)$	$(2, u, \epsilon)$		
2		$(2, u, \epsilon)$		x

Configuration & Mode de reconnaissance





Définition

Configuration = triplet $(q, m, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, où :

- q est l'état courant de l'unité de contrôle
- m représente la partie du mot à reconnaître non encore lue. Le symbole le plus à gauche de m est le caractère sous la tête de lecture.
- α représente le contenu de la pile. Le symbole le plus à gauche de α est le sommet de la pile.

Changement de configuration

- Comportement : Lors du mouvement de transition on note

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha) \text{ si } (q', \gamma) \in \delta(q, a, Z)$$

  - l'unité de contrôle passe de q à q'
  - le symbole a a été lu
  - la tête de lecture s'est déplacée d'une case vers la droite
  - le symbole Z a été dépilé et le mot γ empilé.

- Pour un automate à pile A, une chaîne $w \in \Sigma^*$ est reconnue par état acceptant et pile vide s'il y a une suite de transitions partant de l'état initial avec la pile vide, dont la suite des étiquettes est w , et aboutissant à un état acceptant où la pile est vide.

Configuration & Mode de reconnaissance

Mode de reconnaissance d'un mot dans un PDA

Un mot reconnu dans un PDA peut l'être de 2 façons :

- **Par état final :** Une chaîne testée est acceptée si, à partir de l'état initial, elle peut être entièrement lue en arrivant à un état de F, ceci quel que soit le contenu de la pile à ce moment-là. La pile est supposée vide au départ, mais on peut aussi convenir d'un contenu initial qui sera imposé.
- **Par pile vide :** Mode de reconnaissance autorisant un automate à pile à accepter une chaîne si, à partir de l'état initial, elle peut être entièrement lue en vidant la pile. Il n'y a donc pas lieu de préciser F, il est sous-entendu que $F = Q$ (tous les états sont acceptants).

Configuration & Mode de reconnaissance

Il existe des variantes qui ne portent pas sur la forme des transitions, mais sur l'état dans lequel doit être la pile pour qu'une chaîne soit acceptée.

Remarque

- Pour un PDA P, 2 langages (à priori différents) sont définis :
  - **N(P) (acceptation par pile vide) et**
  - **L(P) (acceptation par état final)** .
- N(P) n'utilise pas F et n'est donc pas modifié si l'on définit $F = \emptyset$;

Remarque

- $L(P) = \{w \mid w \in \Sigma^* \wedge \exists q \in F, \gamma \in \Gamma^*(q_0, w, Z_0) \vdash_P^* (q, \epsilon, \gamma)\}$
- $N(P) = \{w \mid w \in \Sigma^* \wedge \exists q \in Q, (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \epsilon)\}$

Configuration

Configuration et langage accepté

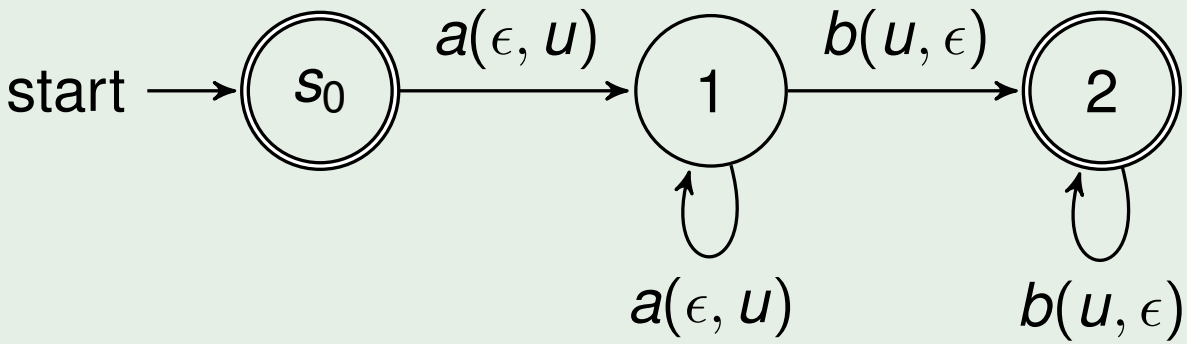
Configuration = triplet $(q, m, \alpha) \in Q \times \Sigma^* \times \Gamma^*$

- Configuration initiale : (q_0, w, Z_0) où w est la chaîne à accepter
- Configuration finale avec acceptation par pile vide : (q, ϵ, ϵ) (q quelconque)
- Configuration finale avec acceptation par état final : (q, ϵ, γ) avec $q \in F$ ($\gamma \in \Gamma^*$ quelconque)

Remarque

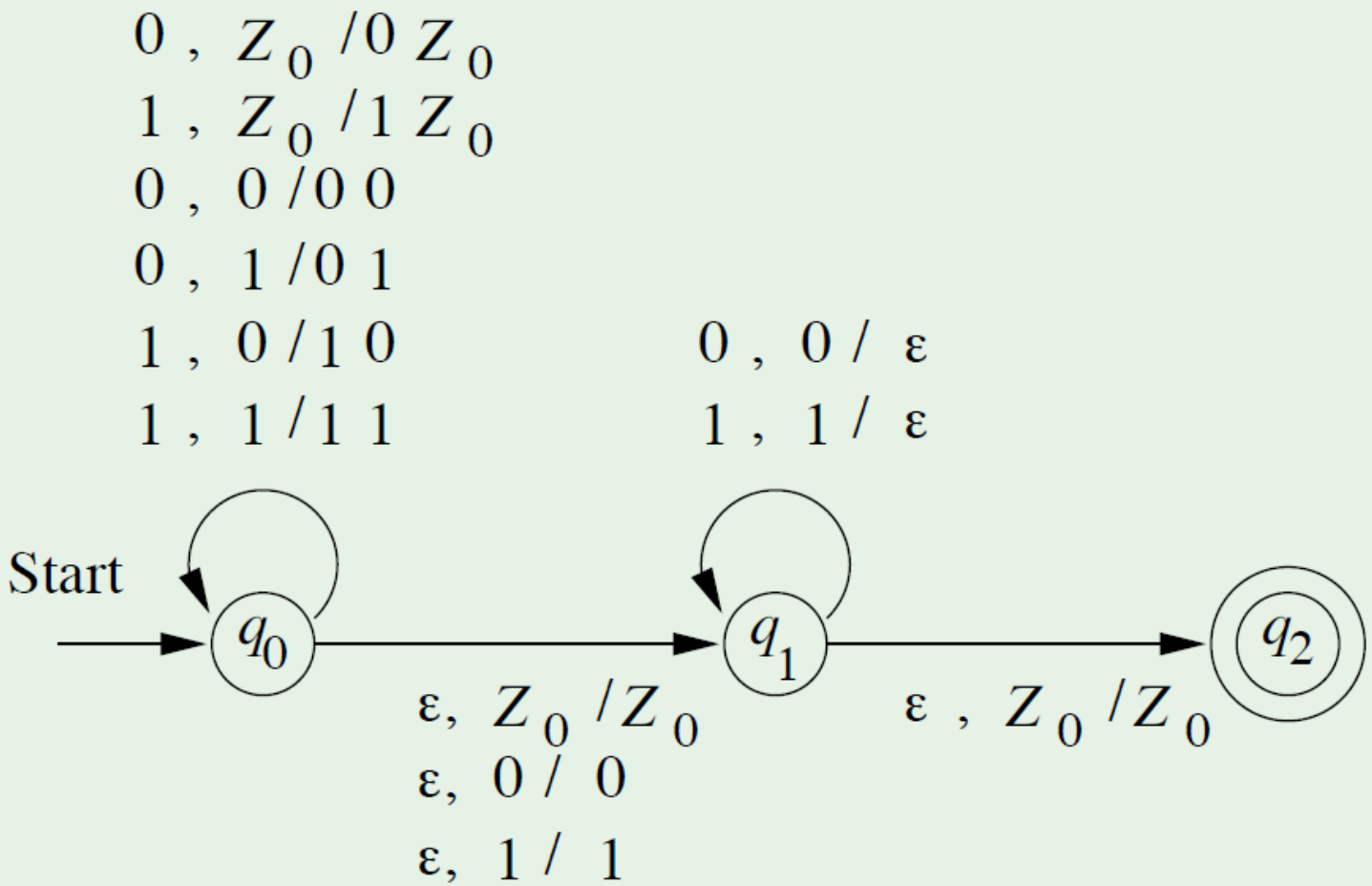
- Si $\gamma = \epsilon$, la pile a été dépilée (sauf si $Z = \epsilon$)
- $a = \epsilon$, le changement d'état et la modification de la pile se font sans mouvement de la tête.
- $Z = \epsilon$, il s'agit d'une transition permise quel que soit le symbole sur la pile.
- $\gamma = \epsilon$ et $Z = \epsilon$, alors la pile est inchangée.
- Donc la transition $(q, \epsilon, \epsilon) \vdash (q, \epsilon)$ est un changement d'état sans autre modification ($\Leftrightarrow$ ϵ-transition dans AFD)

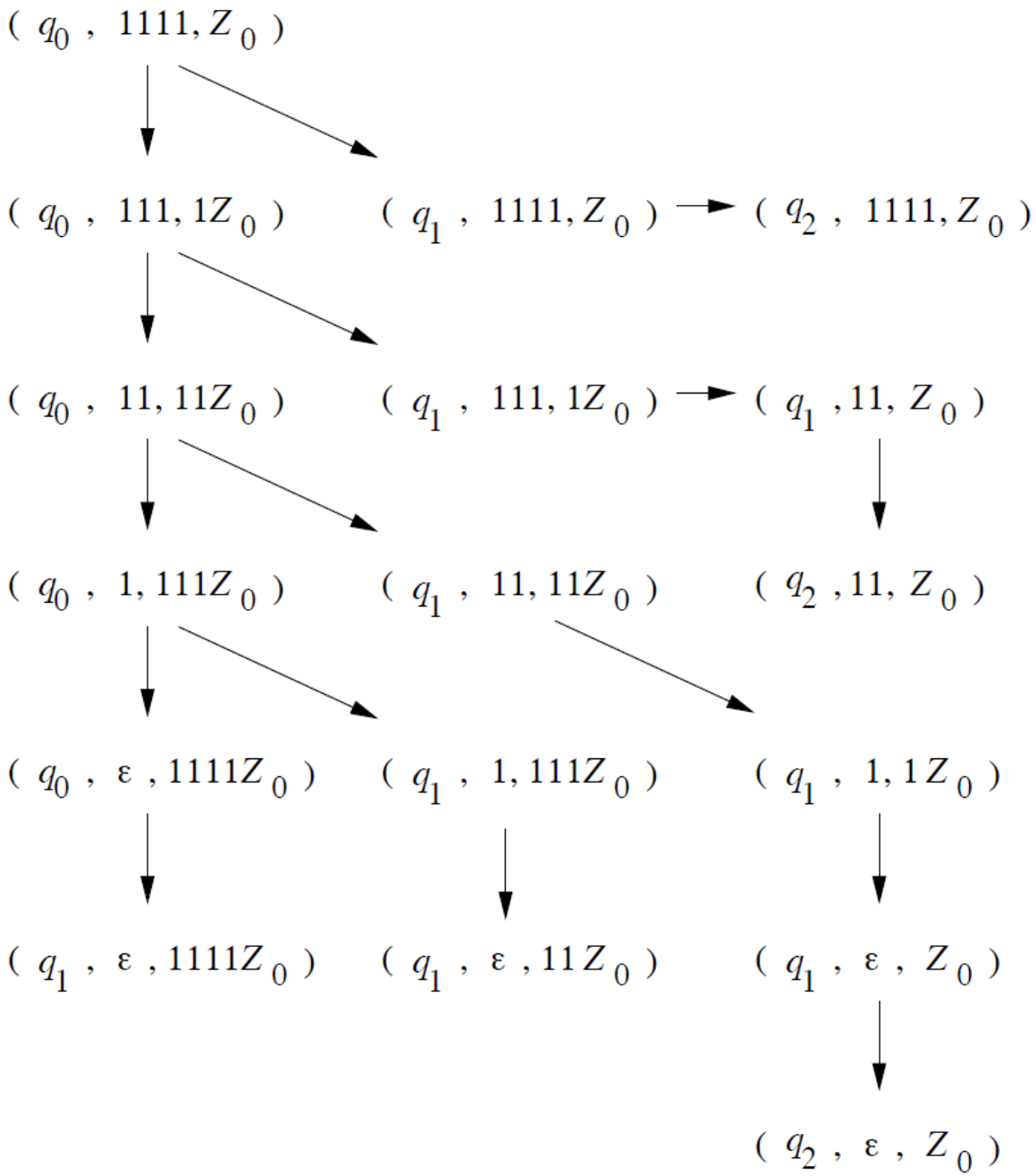
Exemple (3/3) $L = \{a^n b^n \mid n \geq 0\}$ pour le PDA ($\Gamma = \{u\}$)



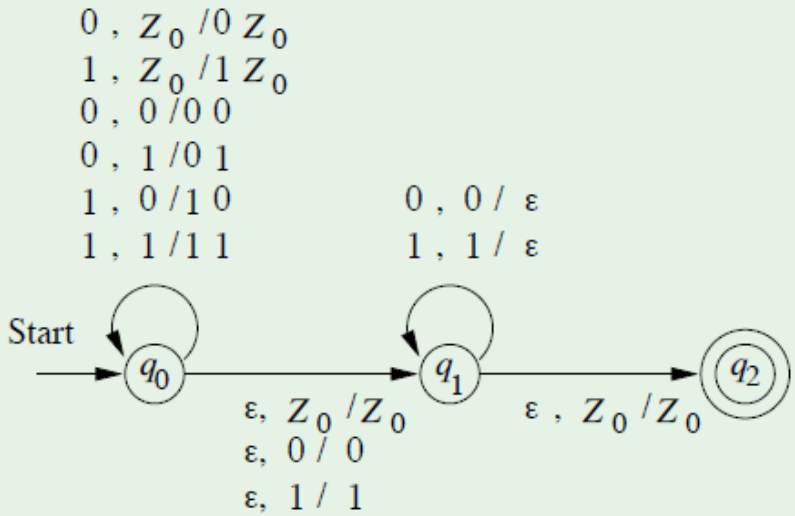
Exemple de Séquence d'exécution

PDA pour $L_{WW^R} = \{WW^R \mid W \in \{0,1\}\}$ et exécution sur 1111




$(q_0, 1111, Z_0)$
$(q_0, 111, 1Z_0)$
$(q_1, 1111, Z_0)$
$(q_2, 1111, Z_0)$
$(q_0, 11, 11Z_0)$
$(q_1, 111, 1Z_0)$
$(q_1, 11, Z_0)$
$(q_0, 1, 111Z_0)$
$(q_1, 11, 11Z_0)$
$(q_2, 11, Z_0)$
$(q_0, \varepsilon, 1111Z_0)$
$(q_1, 1, 111Z_0)$
$(q_1, 1, 1Z_0)$
$(q_1, \varepsilon, 1111Z_0)$
$(q_1, \varepsilon, 11Z_0)$
(q_1, ε, Z_0)
(q_2, ε, Z_0)

Miroir : $L(P) = \{ww^R \mid w \in \{0,1\}^*\}$ et $N(P) = \emptyset$

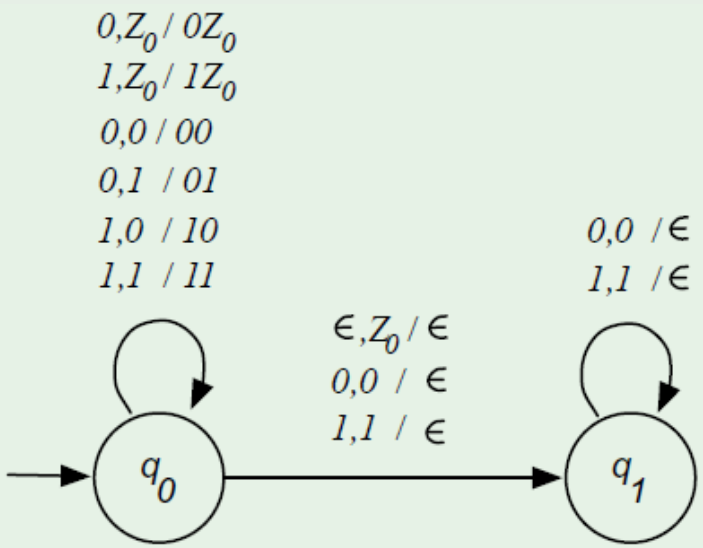


$$P = \langle \{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\} \rangle$$

δ	$(0, Z_0)$	$(1, Z_0)$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00)\}$	$\{(q_0, 01)\}$	$\{(q_0, 10)\}$	$\{(q_0, 11)\}$
q_1	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$
$*q_2$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

δ	(ϵ, Z_0)	$(\epsilon, 0)$	$(\epsilon, 1)$
$\rightarrow q_0$	$\{(q_1, Z_0)\}$	$\{(q_1, 0)\}$	$\{(q_1, 1)\}$
q_1	$\{(q_2, Z_0)\}$	$\emptyset$	$\emptyset$
$*q_2$	$\emptyset$	$\emptyset$	$\emptyset$

Palindrome : $\emptyset$ et $L(P) = N(P) = \{ww^R \mid w \in \{0,1\}^*\}$



$$P = \langle \{q_0, q_1\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \emptyset \rangle$$

δ	$(0, Z_0)$	$(1, Z_0)$	$(0,0)$	$(0,1)$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00), (q_1, \epsilon)\}$	$\{(q_0, 01)\}$
q_1	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$	$\emptyset$

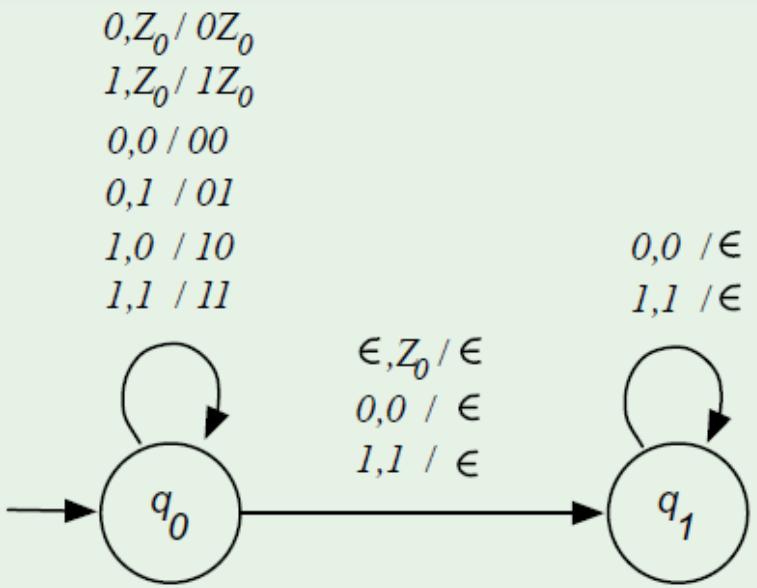
δ	$(1,0)$	$(1,1)$	(ϵ, Z_0)
$\rightarrow q_0$	$\{(q_0, 10)\}$	$\{(q_0, 11), (q_1, \epsilon)\}$	$\{(q_1, \epsilon)\}$
q_1	$\emptyset$	$\{(q_1, \epsilon)\}$	$\{(q_1, \epsilon)\}$

Définition : PDA déterministe (DPDA)

Un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ est déterministe si :

- $\delta(q, a, X)$ est toujours soit vide soit 1 singleton ($a \in \Sigma \cup \{\epsilon\}$)
- Si $\delta(q, a, X)$ est non vide alors $\delta(q, \epsilon, X)$ est vide

Palindrome : $L(P) = \emptyset$ et $N(P) = \{ww^R \mid w \in \{0,1\}^*\}$



$$P = \langle \{q_0, q_1\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \emptyset \rangle$$

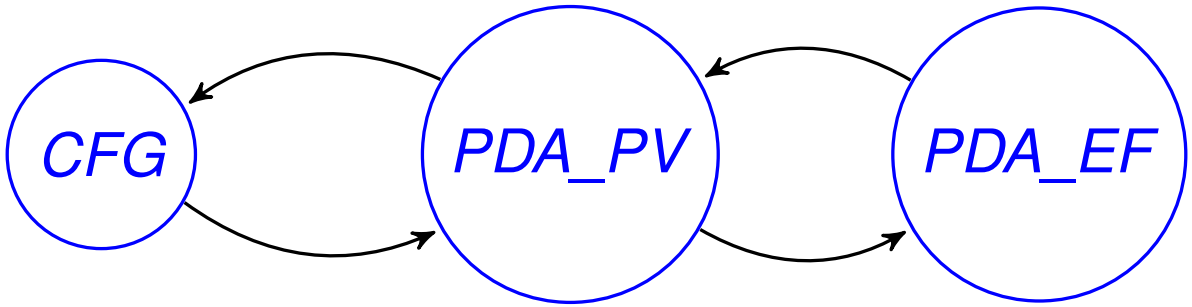
Théorème

La classe des langages définis par un PDA déterministe est strictement incluse dans la classe des langages définis par un PDA (général)

Preuve

On peut montrer que le langage L_{ww^r} défini au slide 124 ne peut être défini par un PDA déterministe

On va montrer les inclusions (flèches) suivantes
PDA_PV = PDA par Pile Vide, *PDA_EF* = PDA par Etat Final



Ce qui prouvera que

Théorème

Les trois classes de langages suivants sont équivalentes :

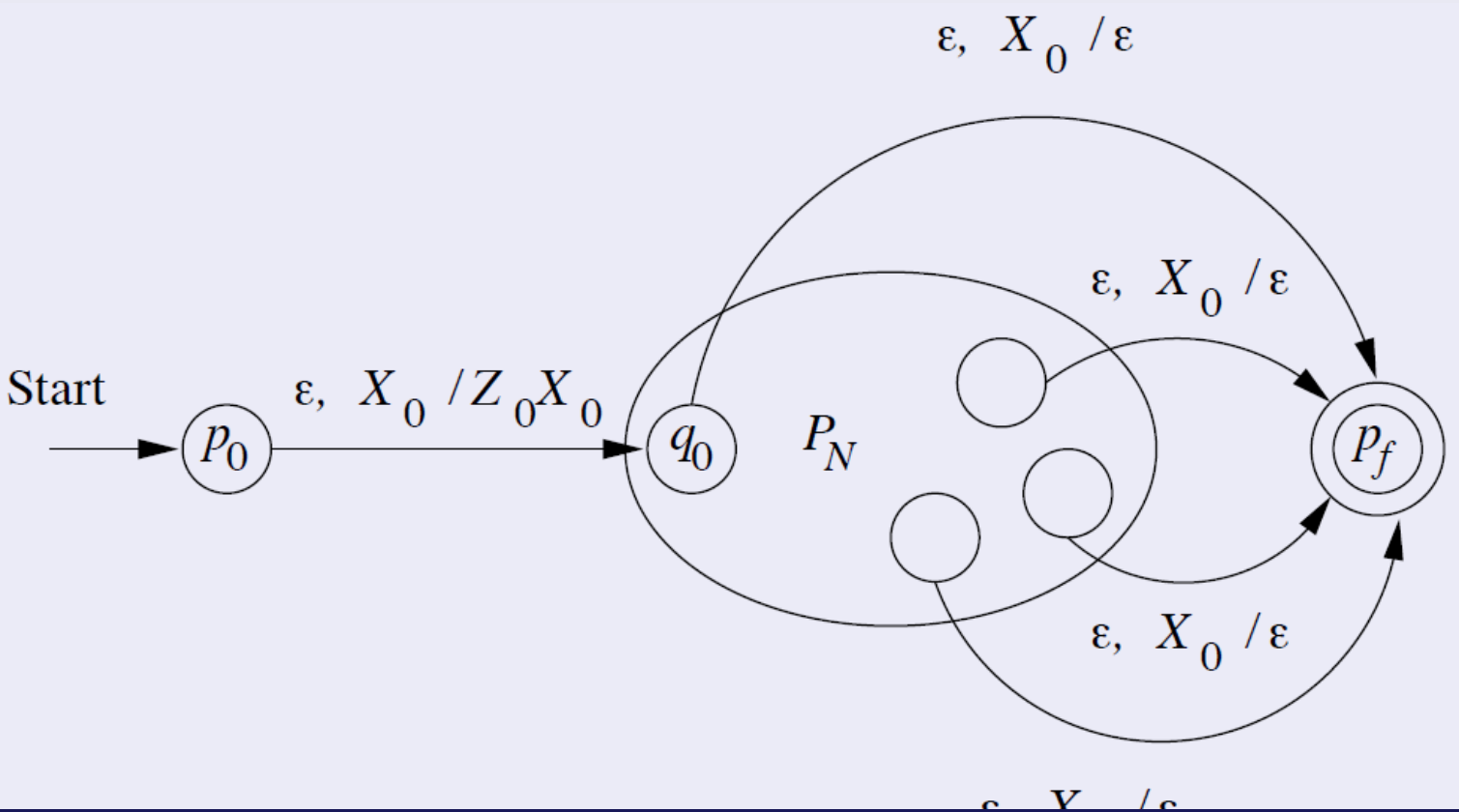
- Les langages définis par une grammaire hors contexte (i.e les langages hors contexte)
- Les langages définis par un PDA avec acceptation par pile vide
- Les langages définis par un PDA avec acceptation par état final

Équivalence état final - pile vide : Cas $P_N \Rightarrow P_F$

Théorème

Pour tout PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0, \emptyset)$ on peut définir un PDA P_F avec $L(P_F) = N(P_N)$

Construction

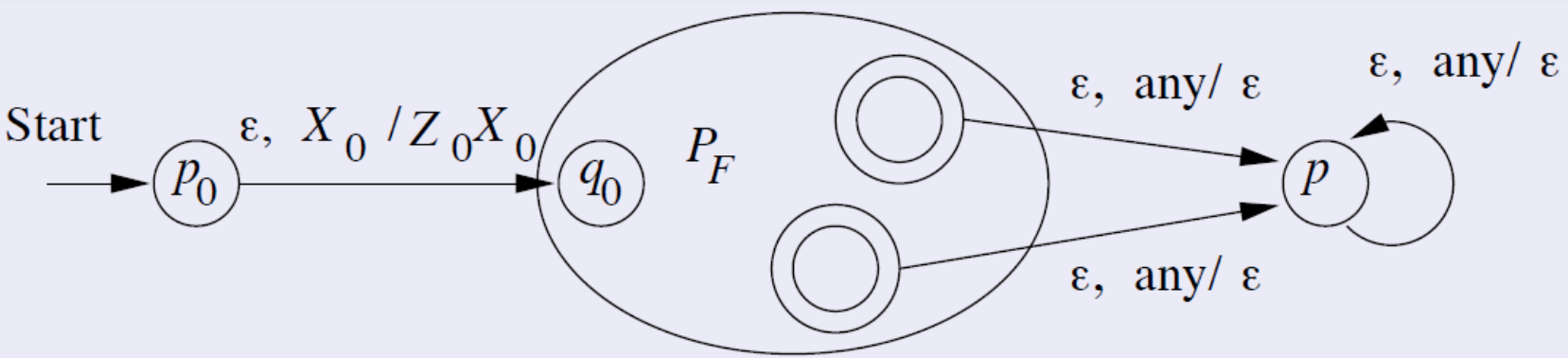


Équivalence état final - pile vide $P_F \Rightarrow P_N$

Théorème

Pour tout PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$ on peut définir un PDA P_N avec $N(P_N) = L(P_F)$

Construction



Théorème

Pour toute CFG G on peut définir un PDA M avec $L(G) = N(M)$

Principe de la preuve

Ayant G = (V, T, P, S) une CFG, on construit un PDA M à un seul état qui simule les dérivations gauches de G :

$$P = (\{q\}, T, V \cup T, \delta, Q, S, \emptyset) \text{ avec}$$
$$\forall A \to X_1 X_2 \dots X_k \in P, (q, X_1 X_2 \dots X_k) \in \delta(q, \epsilon, A)$$
$$\forall a \in T : \delta(q, a, a) = \{(q, \epsilon)\}$$

- Au départ le symbole de départ S est sur la pile
- Toute variable A au sommet de la pile avec $A \to X_1 X_2$ *ldots*$X_k \in P$ peut être remplacée par sa partie droite $A \to X_1 X_2$ *ldots*X_k avec $X1$ au sommet de la pile
- Tout terminal au sommet de la pile qui est égal au prochain symbole en entrée est matché avec l'entrée (on lit l'entrée et pop le symbole)
- À la fin, si la pile est vide, le string est accepté

Équivalence entre PDA et CFG

Théorème

Pour tout PDA P on peut définir une CFG G avec $L(G) = N(P)$

Quelques questions qu'on peut se poser sur des langages L, $L1$, $L2$

- L est-il context-free ?
- Pour quels opérateurs les langages context-free sont-ils fermés ?
- $w \in L$?
- L est-il vide, fini, infini ?
- $L1 \subseteq L2$, $L1 = L2$?

L est-il context-free ?

Astuces

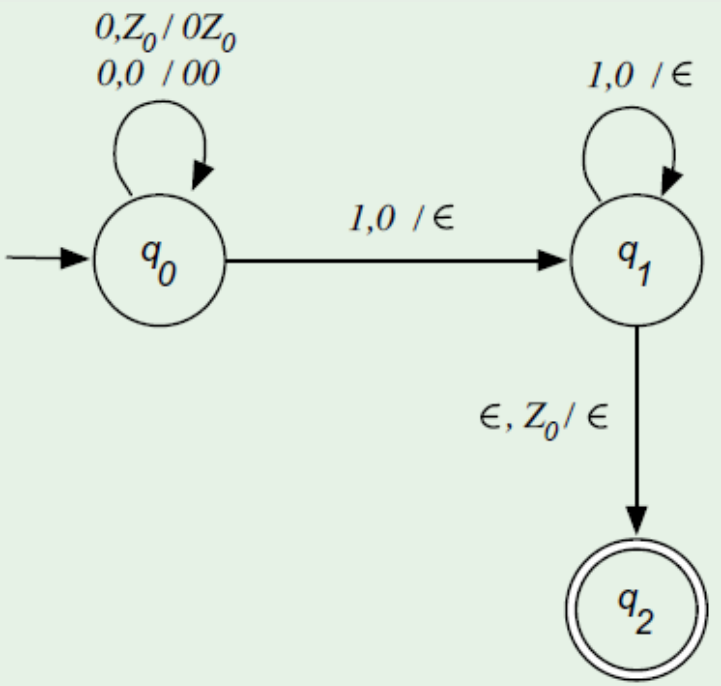
Pour montrer que L est context-free, il faut :

- Trouver une CFG (ou un PDA) G et
- montrer que $L = L(G)$ (ou $L = N(G)$ si G est un PDA avec acceptation par pile vide i.e
  - $L \subseteq L(M)$, et donc tout mot de L est accepté par M
  - $L(M) \subseteq L$, et donc tout mot accepté par M est dans L

L est-il context-free ?

Exemple : $L = \{0^i 1^j \mid i \in \mathbb{N}\}$

Pour montrer que L est context-free, on définit par exemple le PDA P et on démontre (par induction) que $L = L(P)$.



L est-il context-free ?

Pour montrer que L n'est pas context-free, on exploite parfois le pumping lemma associé aux langages hors contexte

Pumping Lemma pour les langages algébriques

Soit L un langage algébrique. Il existe alors un entier p (appelé longueur du pompage) tel que tout mot $t \in L$ de taille $\mid t \mid \geq p$, s se factorise en $t = uvwxy$, avec $u, v, w, x, y \in \Sigma^*$ où :

(i) $\mid vx \mid \geq 1$ (i.e l'un au moins de v et x est $\neq \epsilon$),

(ii) $\mid vwx \mid \leq p$

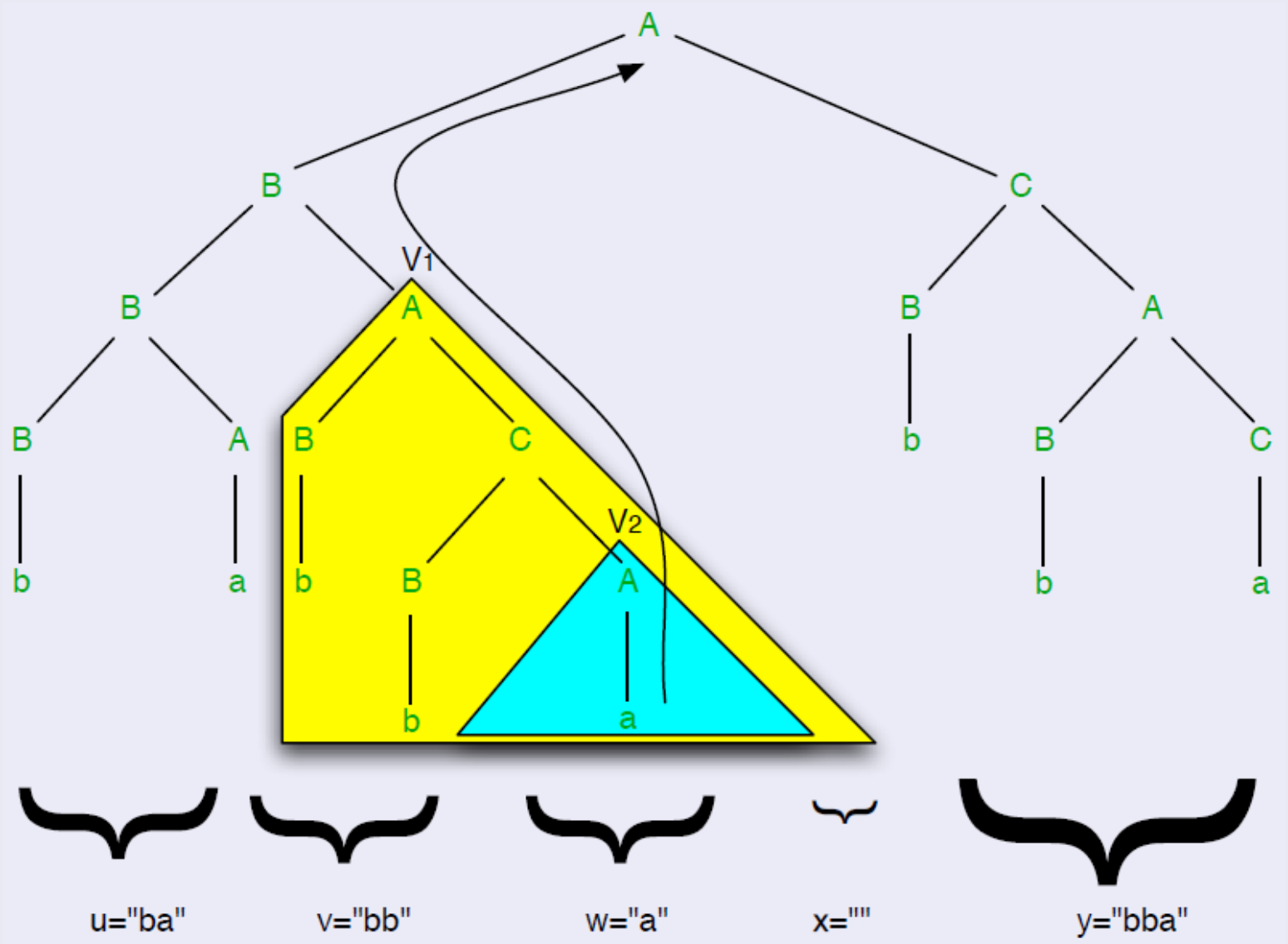
(iii) $\forall i \geq 0$, $uv^i wx^i y \in L$.

Pumping Lemma pour les CFG

Preuve

- Soit L un langage algébrique.
- L (ou $L \setminus \{\epsilon\}$) est reconnu par une CFG en forme normale de Chomsky G
- Soit k le nombre de variables de G
- Prenons $p = 2^k$ k et $\mid z \mid \geq p$
- Prenons un des plus longs chemins de l'arbre de dérivation de z : plusieurs noeuds du chemin ont le même label (une variable)
- Dénotons v_1 et v_2 2 sommets sur ce chemin labellés par la même variable avec v_1 ancêtre de v_2 et on choisi v_1 "le plus bas possible"
- Découpons z comme indiqué sur la figure

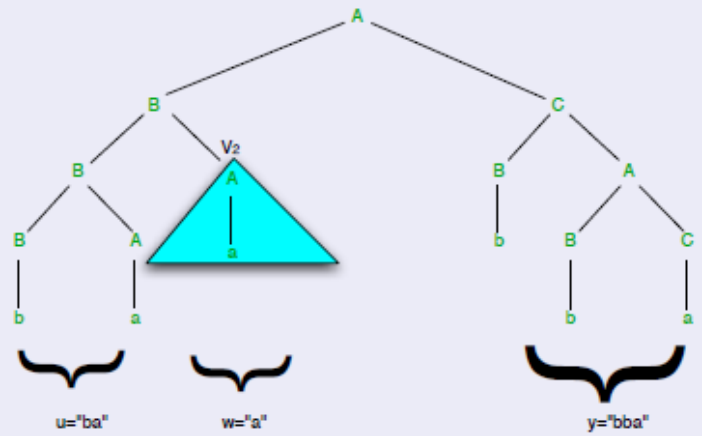
Pumping Lemma pour les CFG

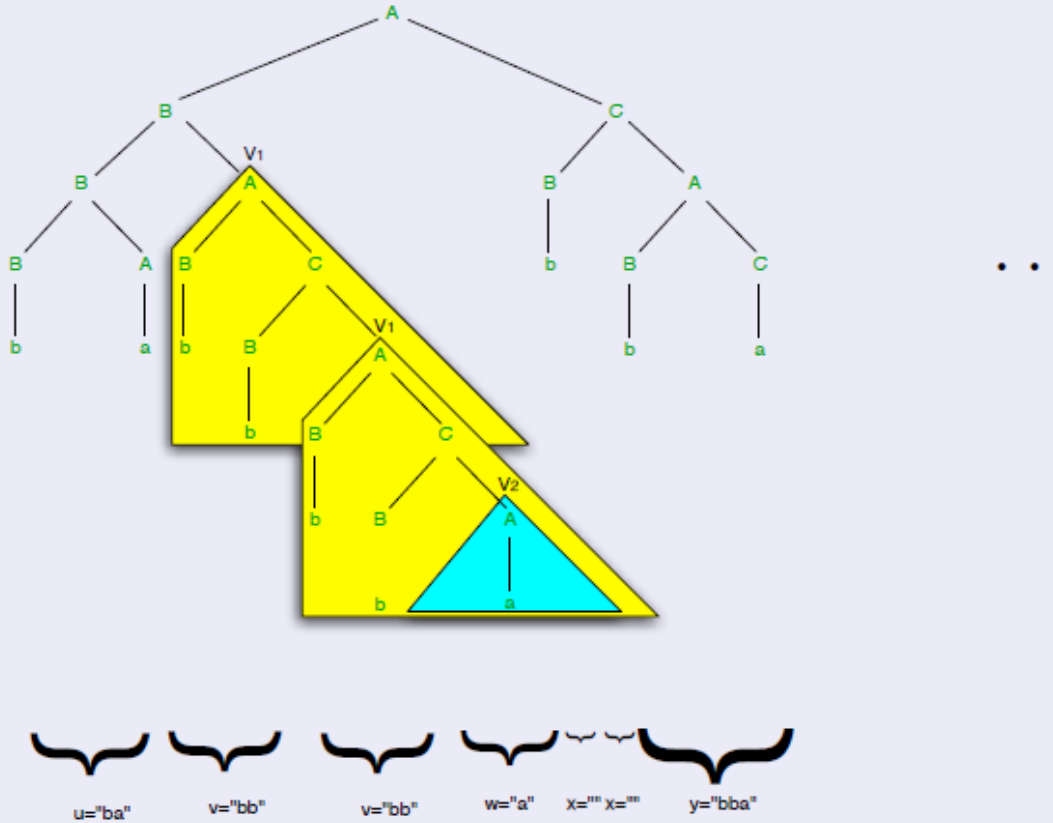


Preuve (suite)

- Et donc z=uvwxy
- La hauteur maximale de l'arbre de racine v_1 est k +1 et génère un mot vwx avec $\mid vwx \mid \leq 2^k = p$
- De plus comme G est en forme normale de Chomsky, v_2 est soit dans le fils droit, soit dans le fils gauche de l'arbre de racine v_1
- De toute façon, soit $v \neq \epsilon$ soit $x \neq \epsilon$
- On peut enfin voir que $uv^i wx^i y \in L(G)$
  - Si on remplace le sous arbre de racine v_1 par le sous arbre de racine v_2, on obtient $uwy \in L$
  - On peut plutôt remplacer le sous arbre de racine v_2 par un nouvelle instance de sous arbre de racine v_1, on obtient $uv^2wx^2y \in L$
  - On peut répéter l'opération précédente avec l'arbre obtenu et on obtient que $\forall i : uv^i wx^i y \in L$.

Pumping Lemma pour les CFG





Et donc $\forall i : uv^i wx^i y \in L$

Pumping Lemma

Exemple

$\Sigma = \{a, b, c\}$. Montrer que $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ n'est pas algébrique. Soit k l'entier dont le lemme de pompage garantit l'existence. Considérons le mot $t = a^k b^k c^k$. il doit alors exister une factorisation $t = uvwxy$ pour laquelle on a (i) $\mid vx \mid \geq 1$, (ii) $\mid vxx \mid \leq k$ et (iii) $\forall i \geq 0$, $uv^i wx^i y \in L$. La propriété (ii) assure que le facteur vwx ne peut pas contenir simultanément les lettres a et c : en effet, tout facteur de t comportant un a et un c doit avoir aussi le facteur b^k, et donc être de longueur $\geq k + 2$. Supposons que vwx ne contienne pas la lettre c (l'autre cas étant complètement analogue) : en particulier, ni v ni x ne la contient, donc le mot $uv^i wx^i y$, qui est dans L d'après (iii), a le même nombre de c que le mot t initial ; mais comme son nombre de a ou bien de b est différent (d'après (i)), on a une contradiction.

Opérations de clôture

Théorème

L_1 et L_2 algébriques $\Rightarrow L_1 \cup L_2$, $L_1.L_2$ et L^* algébriques.

Preuve

Soit $G_1 = (V_1, T_1, P_1, S_1)$ et $G_2 = (V_2, T_2, P_2, S_2)$ 2 grammaires n'ayant aucune variable en commun et $S \notin V_1 \cup V_2$:

- Union : Pour $G_\cup = (V_1 \cup V_2 \cup S, T_1 \cup T_2, P, S)$ avec $P = P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}$, on a $L(G_\cup)) = L(G1) \cup L(G2)$
- Concaténation : Pour $G_{concat} = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S)$ avec $P = P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}$, on a : $L(G_{concat}) = L(G1).L(G2)$
- Fermeture de Kleene : Pour $G_* = (V_1 \cup \{S\}, T_1, P, S)$ avec $P = P_1 \cup \{S \rightarrow S_1 S_2 \mid \epsilon\}$, on a : $L(G_*) = L(G_1)*$

Opérations de clôture pour Miroir

Théorème

Si L est un langage algébrique alors L^R l'est aussi.

Preuve

L est reconnu par une CFG $G = (V, T, P, S)$.
À partir de G on peut construire une CFG $G^R = (V, T, P^R, S)$ avec $P^R = \{A \to \alpha^R \mid \to \alpha \in P\}$
On peut prouver que

$$L(G^R) = L^R$$

Opérations de clôture ? ? ?

Théorème

Si L et M sont algébriques alors $L \cap M$ peut ne pas être algébrique.

Preuve

Il suffit de donner un contre exemple !

- $L_1\{a^l b^l c^m \mid l, m \in \mathbb{N}\}$ est un langage algébrique (démontrer)
- $L_2\{a^l b^m c^m \mid l, m \in \mathbb{N}\}$ est un langage algébrique (démontrer)
- Mais $L_1 \cap L_2$ n'est pas un langage algébrique

Opérations de clôture ? ? ?

Théorème

Si L est algébrique et R est régulier alors $L \cap R$ est algébrique.

Preuve

- Ayant le PDA A_L avec $L(A_L) = L : A_L = (Q_L, \Sigma, \Gamma, \gamma_L, q_L, Z_0, F_L)$
- Et l'AFD A_R avec $L(A_R) = R$, $A_R = (\Sigma, Q_R, q_R, \delta_R, F_R)$
- On définit le PDA $A_{L\cap R}$ par $A_{Q_L \cap Q_R}, \Sigma, \Gamma, \delta_{L\cap R}, (q_L, q_R), Z_0, F_L \times F_R$ avec $\delta_{L\cap R}(p, q, a) = \{(p', q') \mid p' \in \delta_L(p, a) \wedge q' \in \delta_R(q, a)\}$ $a \in \Sigma \cup \{\epsilon\}$. On peut démontrer que $L(A_{L\cap R}) = L \cap R$

Opérations de clôture ? ? ?

Théorème

Si L est un langage algébrique alors $\overline{L}$ ne peut être algébrique

Preuve

Une des lois de De Morgan est :

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Donc comme l'intersection n'est pas fermée pour les langages algébriques mais que l'union l'est, le complément n'est pas non plus fermé pour les CFL.

Opérations de clôture ? ? ?

Théorème

Si L et M sont des langages algébriques alors $L \setminus M$ peut ne pas être algébrique

Preuve

$$\overline{(L)} = \Sigma^* \setminus M$$

Et Σ^* est régulier.

$w \in L$, L vide, fini, infini ? ? ?

Astuce

Pour tester que :

- $w \in L$ voir si une CFG en forme normale de Chomsky génère w Il existe un algorithme en O(n^3) où n est la longueur de w
- L vide (L=$\emptyset$), voir si dans G avec $L(G) = L$, le symbole de départ produit quelque chose
- L infini ? voir si une CFG en forme normale de Chomsky qui définit L est récursive

Problèmes indécidables

Les problèmes suivants sur les langages Hors-contexte sont indécidables (il n'existe pas d'algorithme pour les résoudre de façon générale)

- la grammaire algébrique G est-elle ambiguë ?
- le langage algébrique L est-il ambigu de façon inhérente ?
- l'intersection de 2 CFL est-elle vide ?
- le langage algébrique L $=\Sigma^*$?
- $L_1 \subseteq L_2$?
- $L_1 = L_2$?
- $\overline{(L)}$ est-il un langage hors-contexte
- $L(G)$ est-il déterministe ?
- $L(G)$ est-il régulier ?

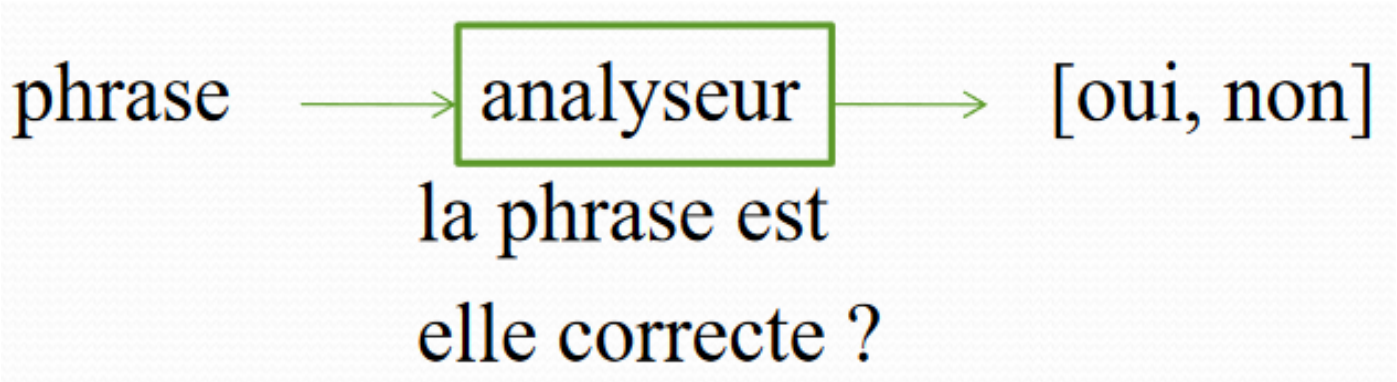
Plan

Introduction aux méthodes d'analyse

Questions posées

En supposant qu'on ait une phrase ω, Les différentes questions que l'on peut se poser par rapport aux phrases engendrées par une grammaire sont :

- comment analyser ω ?
- ω appartient-elle au langage $L(G)$?
- si $\omega \in L(G)$, donner une dérivation



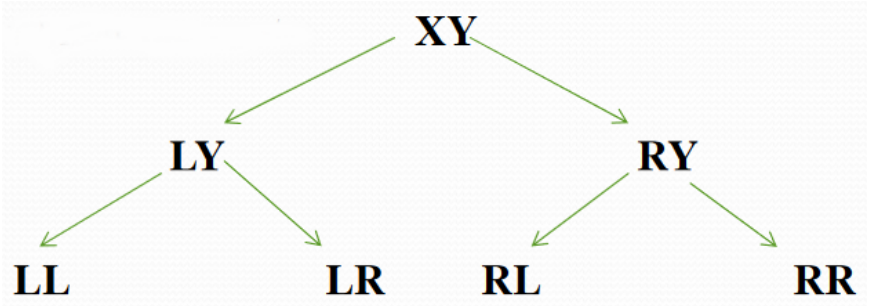
Introduction aux méthodes d'analyse

Critères de classification

Il existe 2 critères de classification pour distinguer les différentes méthodes d'analyse syntaxique :

- le sens X de parcours de la chaîne analysée : gauche $\rightarrow$ droite (L) ou droite $\rightarrow$ gauche (R). $X \notin \{L, R\}$
- le sens Y d'application des productions : dérivation ou réduction (Leftmost ou Rightmost). $Y \notin \{L, R\}$

On distingue donc 4 méthodes d'analyse désignées par un couple XY



L'analyse LL(k)

LL(k) : Avantages et Inconvénients

LL(k)="scanning the input from **L**eft to right producing a **L**eftmost derivation, with **k** symbols of look-ahead".

- Avantages : cette technique est simple, très efficace et facile à implémenter.
- Inconvénients : elle ne fonctionne sans conflits que sur une sous-classe des langages algébriques déterministes.

- On dit qu'un langage L est LL(k) s'il existe une grammaire LL(k) qui engendre L.
- Un langage est LL(k) s'il est LL(1).
- On parlera de la classe des langages LL.

L'analyse LL(1)

Définition

- C'est une analyse avec une inspection d'un symbole terminal en avance sur le symbole courant traité.
- Dans la pratique ce sont les grammaires les plus utilisées.
- Le but de l'analyse LL(1) est de pouvoir (toujours) connaître la règle de production à appliquer et ceci en se basant sur le terminal suivant non encore traité de la phrase en cours d'examen.
- Deux concepts nouveaux se révèlent nécessaires pour faciliter l'analyse LL(1) :
  - PREMIER (FIRST)
  - SUIVANT (FOLLOW)

Définition

Une grammaire est LL(1) si elle remplit la condition suivante : Quel que soit A un non terminal tel que : $A \rightarrow \alpha$ et $A \rightarrow \beta$ où α et β sont deux symboles grammaticaux, alors :

$$S \Longrightarrow^* \omega A \gamma \Longrightarrow \begin{cases} \omega\alpha\gamma & \Longrightarrow^* & \omega x \\ \omega\beta\gamma & \Longrightarrow^* & \omega y \end{cases}$$

- Si $PREMIER(x) = PREMIER(y)$ alors $\alpha = \beta$
- Si $PREMIER(x) \neq PREMIER(y)$ alors $\alpha \neq \beta$

Plus clairement, PREMIER représente l'ensemble des premiers symboles terminaux qui commencent x et $x \neq \epsilon$.

i-e qu'à un instant donné de la dérivation, s'il y a plusieurs chemins pour arriver à la phrase que l'on analyse, ces chemins sont identiques et la connaissance du prochain symbole de la phrase est suffisante pour déterminer la règle de production à utiliser.

Analyse LL(1) : Ensembles **PREMIER** et **SUIVANT**

Les notions PREMIER et SUIVANT sont utiles d'une part pour montrer qu'une grammaire est LL(1) et d'autre part pour la construction de l'analyseur syntaxique.

L'ensemble **PREMIER**

- si a est un terminal, alors : PREMIER(a) = a de même : $PREMIER(\epsilon) = \{\epsilon\}$
- si $A \rightarrow b\alpha$, alors $b \in PREMIER(A)$
- si $A \rightarrow B\alpha$, alors $PREMIER(B) \subset PREMIER(A)$
  De plus, si $B \Rightarrow^* \epsilon$, alors
  $((PREMIER(B) \backslash \epsilon) \cup PREMIER(\alpha)) \subset PREMIER(A)$

D'une autre manière, PREMIER(α) inclut toujours l'ensemble PREMIER du premier symbole de α. Si celui-ci peut produire ϵ, il inclut aussi l'ensemble PREMIER du 2nd symbole de α, ...
Si tout symbole de α peut produire ϵ alors $\epsilon \in$ PREMIER(α).

Analyse LL(1) : Ensembles **PREMIER** et **SUIVANT**

L'ensemble **SUIVANT**

Nous appliquons les règles suivantes jusqu'à ce qu'aucun terminal ne puisse être ajouté aux ensembles SUIVANT.

- mettre \$ dans **SUIVANT** de l'axiome.
- s'il y a une production : $A \rightarrow \alpha B \beta$, le contenu de **PREMIER(β)**, excepté la chaîne vide est ajouté à **SUIVANT(B)**
- s'il y a une production : $A \rightarrow \alpha B$ ou $A \rightarrow \alpha B \beta$ tq $\beta \Rightarrow^* \epsilon$, les éléments de **SUIVANT(A)** sont ajoutés à **SUIVANT(B)**

Exemple

Soit la grammaire :

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow id \mid (E) \end{aligned}$$

$PREMIER(F) = \{id, (\}$
$PREMIER(T) = \{id, (\}$
$PREMIER(E) = \{id, (\}$
$SUIVANT(E) = \{\$, +, -,)\}$
$SUIVANT(T) = \{\$, +, -, *, /,)\}$
$SUIVANT(F) = \{\$, +, -, *, /,)\}$

La grammaire équivalente :

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid -TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid /FT' \mid \epsilon \\ F &\rightarrow id \mid (E) \end{aligned}$$

$PREMIER(F) = \{id, (\}$
$PREMIER(T') = \{*, /, \epsilon\}$
$PREMIER(T) = \{id, (\}$
$PREMIER(E') = \{+, -, \epsilon\}$
$PREMIER(E) = \{id, (\}$
$SUIVANT(E) = \{\$,)\}$
$SUIVANT(E') = \{\$,)\}$
$SUIVANT(T) = \{\$, +, -,)\}$
$SUIVANT(T') = \{\$,), +, -\}$
$SUIVANT(F) = \{\$, +, -, *, /,)\}$

Conditions d'analyse d'une grammaire LL(1)

Pour qu'une grammaire soit de type LL(1), c'est-à-dire qu'elle soit analysable par une méthode descendante déterministe, il faut que ses règles de production respectent 2 conditions :
soit une paire de règles : $A \rightarrow \alpha$ et $A \rightarrow \beta$ on a :

Contition 1 : $PREMIER(\alpha) \cap PREMIER(\beta) = \emptyset$
Exemple : $A \rightarrow a\ B | a\ C$???

Contition 2 : si $\epsilon \in PREMIER(\beta)$ alors
$PREMIER(\alpha) \cap SUIVANT(A) = \emptyset$

Bref,

G est LL(1)

$$\Updownarrow$$

$$PREMIER(\alpha SUIVANT(A)) \cap PREMIER(\beta SUIVANT(A)) = \emptyset$$

Exemple

Exemple 1 : Considérons la grammaire des expressions arithmétiques

$$\begin{cases} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid nb \end{cases}$$

Exemple 2 : Soit la grammaire suivante

$$\begin{cases} S & \rightarrow & Xa \\ S & \rightarrow & \epsilon \\ X & \rightarrow & Xb \\ X & \rightarrow & Y \\ Y & \rightarrow & aX \\ Y & \rightarrow & \epsilon \end{cases}$$

Exemple 3 : Montrer qu'une grammaire LL(1) ne peut pas avoir des productions récursives à gauche, i-e :

$$A \rightarrow A\alpha \mid \beta$$

Construction d'un analyseur LL(1)

Outils pour l'analyse

L'intérêt de cet analyseur est de reconnaître et de fournir l'unique dérivation gauche des phrases d'un langage.
Un analyseur LL(1) est implanté sous forme d'un automate à pile.

Il est constitué de :

- un tampon qui contient la chaîne du programme à analyser, suivie par le symbole $.
- une pile qui contient une séquence de symboles grammaticaux, avec $ au fond de la pile.
- une table d'analyse à deux dimensions qui permet de choisir la règle de production à utiliser.

Schéma d'un analyseur LL(1)

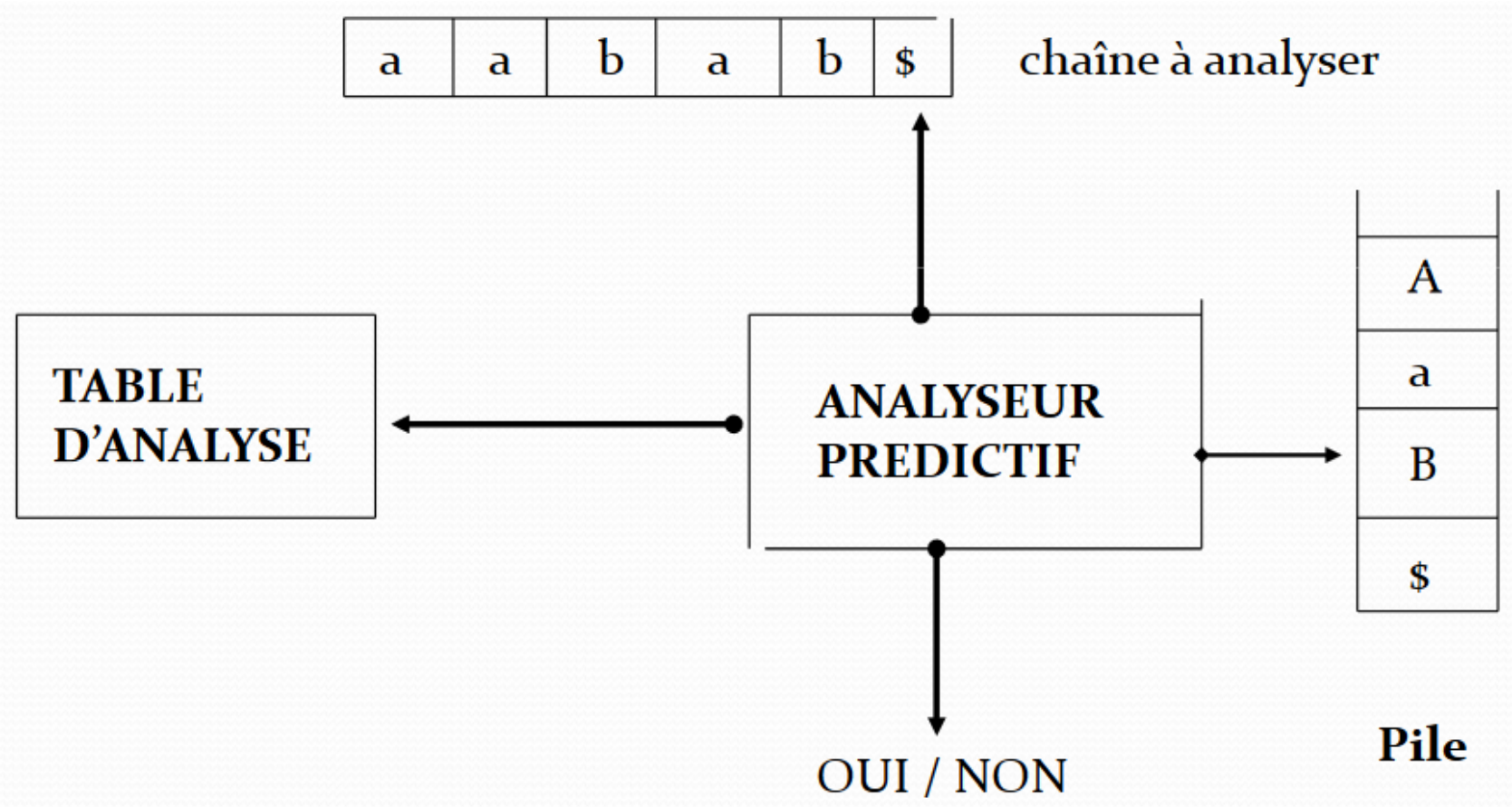


Table d'analyse

Description de la table d'analyse :

Elle est constituée de l'ensemble des règles à utiliser : DEPILER, ERREUR, ACCEPTE.

Donnée : une grammaire.

Résultat : une table d'analyse **M** pour **G**.

Méthode :
$$M[A,a] = \begin{cases} A \rightarrow \alpha \text{ si } a \in PREMIER(\alpha) \\ A \rightarrow \epsilon \text{ si } \alpha \rightarrow^* \epsilon \\ Erreur \text{ sinon} \end{cases}$$

$$M[a,b] = \begin{cases} DEPILER & si & a = b \\ ACCEPTE & si & a = b = \$ \\ Erreur & \text{sinon} & \end{cases}$$

Table d'analyse

Algorithme de construction de la table d'analyse syntaxique

Début $M \leftarrow \emptyset$

  Pour chaque règle $A \rightarrow \alpha$ **faire**

    Pour chaque $a \in PREMIER(\alpha)$ **faire**

      $M[A, a] \leftarrow M[A, a] \cup \{A \rightarrow \alpha\}$

    fpour

    Si $\epsilon \in PREMIER(\alpha)$ **Alors**

      Pour chaque $b \in SUIVANT(A)$ **faire**

        $M[A, b] \leftarrow M[A, b] \cup \{A \rightarrow \alpha\}$

      fpour

    fsi

  fpour

fin

Applications 1/2

Grammaire symbolisant if-then-else après factorisation (elle est ambiguë) :

$$M[A, a] = \begin{cases} S & \rightarrow & iEtSS' | a \\ S' & \rightarrow & eS | \epsilon \\ E & \rightarrow & b \end{cases}$$

	S	**S'**	**E**
PREMIER	?	?	?
SUIVANT	?	?	?

	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Applications 1/2

Grammaire symbolisant if-then-else après factorisation (elle est ambiguë) :

$$M[A,a] = \begin{cases} S & \rightarrow & iEtSS' | a \\ S' & \rightarrow & eS | \epsilon \\ E & \rightarrow & b \end{cases}$$

	S	**S'**	**E**
PREMIER	i, a	e, ϵ	b
SUIVANT	$\$, e$	$\$, e$	t

	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow a$	Erreur	Erreur	$S \rightarrow iEtSS'$	Erreur	Erreur
S'	Erreur	Erreur	$S' \rightarrow eS$ $S' \rightarrow \epsilon$	Erreur	Erreur	$S' \rightarrow \epsilon$
E	Erreur	$E \rightarrow b$	Erreur	Erreur	Erreur	Erreur

Applications 2/2

$E \rightarrow TE'$
$E' \rightarrow +TE'|\epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'|\epsilon$
$F \rightarrow (E)|nb$

	E	**E'**	**T**	**T'**	**F**
PREMIER	$(, nb$	$+, \epsilon$	$(, nb$	$*, \epsilon$	$(, nb$
SUIVANT	$\$,)$	$\$,)$	$+, \epsilon, \$,)$	$+, \epsilon, \$,)$	$*, \epsilon, +, \$$

	nb	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow nb$			$F \rightarrow (E)$		

Trace d'analyse :

- Pour analyser une chaîne, on initialise l'automate à pile, en empilant $ puis l'axiome.
- Ensuite, on combine les symboles du texte d'entrée, les uns après les autres, avec le symbole courant au sommet de la pile.
- Lorsque cette combinaison nous donne, à partir de la table d'analyse, une règle de production à appliquer, on remplace le symbole au sommet de la pile par la partie droite de la production.

	Pile	**Tampon**
Etat initial	S\$	w\$
Etat final	\$	\$

- La trace d'analyse, nous donne l'arbre d'analyse, et le processus de l'unique dérivation gauche de la phrase analysée.

Exemple

Soit la phrase à analyser "nb + nb * nb". On a :

Pile	Tampon	Action
E$	nb + *nb* ∗ *nb*$	$E \rightarrow TE'$
TE'$	*nb* + *nb* ∗ *nb*$	$T \rightarrow FT'$
FT'E'$	*nb* + *nb* ∗ *nb*$	$F \rightarrow nb$
nbT'E'$	*nb* + *nb* ∗ *nb*$	DEPILER
T'E'$	+*nb* ∗ *nb*$	$T' \rightarrow \epsilon$
E'$	+*nb* ∗ *nb*$	$E' \rightarrow +TE'$
+TE'$	+*nb* ∗ *nb*$	DEPILER
TE'$	*nb* ∗ *nb*$	$T \rightarrow FT'$

Exemple

Analyse de "nb + nb * nb" (Suite et fin).

Pile	Tampon	Action
FT'E'$	$nb * nb$$	$F \rightarrow nb$
nbT'E'$	$nb * nb$$	DEPILER
T'E'$	$*nb$$	$T' \rightarrow *FT'$
*FT'E'$	$*nb$$	DEPILER
FT'E'$	$nb$$	$F \rightarrow nb$
nbT'E'$	$nb$$	DEPILER
T'E'$	$$	$T' \rightarrow \epsilon$
E'$	$$	$E' \rightarrow \epsilon$
$	$$	ACCEPTE

Les analyses LR(k) et LALR

LL(k) : Avantages et Inconvénients

LR(k)="scanning the input from **L**eft to right producing a **R**ightmost derivation, with **k** symbols of look-ahead".

On appelle grammaire **LR(k)**, une grammaire algébrique **G** qui permet, en connaissant les **k** caractères suivants du mot à analyser, de décider (de façon déterministe) quelle opération appliquer pour arriver à la dérivation d'un mot de **L(G)**.

- On dit qu'un langage L est LR(k) s'il existe une grammaire LR(k) qui engendre L.
- Tout langage LR(k) est aussi LR(1).
- On appelle LR un langage qui est LR(1).
- Un langage est LR si et seulement si il est algébrique et déterministe.

L'analyse LR(k)

Pour décider si un mot *omega* appartient au langage défini par une grammaire LR, on utilise un automate à pile déterministe particulier (analyseur LR) présenté sous forme d'une table d'analyse.

- Une table LR(k) est de dimension k+1.
- Dans la pratique, on n'utilise que les tables LR(1), qui sont des tables à double entrée.

Remarque

La classe des langages LL est strictement incluse dans la classe des langages LR :

$$\text{langages rationnels} \subsetneq \text{langages LL} \subsetneq \text{langages LR}$$

où langages LR = langages algébriques déterministes

$$\text{langages LR} \subsetneq \text{langages algébriques}$$

L'analyse LR(1)

- C'est une analyse avec une inspection d'un symbole terminal en avance sur le symbole courant traité.
- Dans la pratique ce sont les grammaires les plus utilisées.
- Le but de l'analyse LL(1) est de pouvoir (toujours) connaître la règle de production à appliquer et ceci en se basant sur le terminal suivant non encore traité de la phrase en cours d'examen.
- Deux concepts nouveaux se révèlent nécessaires pour faciliter l'analyse LL(1) :
  - PREMIER (FIRST)
  - SUIVANT (FOLLOW)

Thank you for your attention!