Grammaires

CS410 - Langages et Compilation

Julien Henry
Catherine Oriat

Grenoble-INP Esisar

2012-2013

Summary

Grammaires hors-contexte

Les grammaires hors contexte permettent de définir la classe des *langages hors-contexte*.

Définition: (Grammaire hors-contexte)

Une grammaire hors-contexte est un quadruplet $G = < V_T, V_N, S, R >$, *où :*

- V_T *est un vocabulaire, appelé vocabulaire terminal ;*
- V_N *est un vocabulaire, appelé vocabulaire non terminal, et tel que* $V_N \cap V_T = \emptyset$ *;*
- $S \in V_N$ *est appelé axiome (ou source) de la grammaire.*
- R *est un ensemble de règles de la forme* $A \to w$ *avec* $A \in V_N$ *et* $w \in (V_N \cup V_T)^*$

Langages Réguliers $\subset$ Langages Hors Contexte

Exemple

On définit la grammaire suivante sur le vocabulaire $V_T = \{(,), a, b, \cdots, z\}$:

$$R = \begin{cases} S & \rightarrow & \varepsilon \\ S & \rightarrow & (S) \\ S & \rightarrow & LSL \\ L & \rightarrow & a|b|\cdots|z \end{cases}$$

$V_N = \{S, L\}$

Relation de Dérivation

Définition: (Relation de dérivation)

- *Soit $x, y \in V^*$. On dit que x dérive vers y, noté $x \rightarrow_R y$ si et seulement si il existe une règle $A \rightarrow w$ et deux chaînes $\alpha, \beta \in V^*$ telles que $x = \alpha A \beta$ et $y = \alpha w \beta$.*
- *On note $\rightarrow_R^*$ la fermeture transitive de $\rightarrow_R$. $x \rightarrow_R^* y$ si et seulement si il existe $x_1, \cdots, x_k$, tels que $x \rightarrow_R x_1 \rightarrow_R \cdots \rightarrow_R x_k \rightarrow_R y$.*

Exemple

$$R = \left\{ \begin{array}{lcl} S & \rightarrow & \varepsilon \\ S & \rightarrow & (S) \\ S & \rightarrow & LSL \\ L & \rightarrow & a|b|\cdots|z \end{array} \right.$$

- $(abS) \rightarrow_R (ab(S))$
- $(ab(S)) \rightarrow_R (ab(LSL))$
- $(ab(S)) \rightarrow_R^* (ab(ab()a))$

Langage engendré

Définition: (Langage engendré par une grammaire hors contexte)

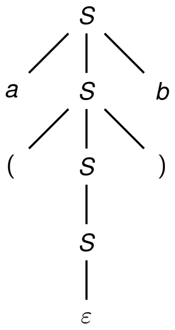
- *Le langage engendré par une grammaire* $G =< V_T, V_N, S, R >$ *est* $L(G) = \{x \in V_T^*, S \rightarrow_R^* x\}$
- *Deux grammaires* G_1, G_2 *sont équivalentes ssi* $L(G_1) = L(G_2)$.

$$R = \begin{cases} S & \rightarrow & \varepsilon \\ S & \rightarrow & (S) \\ S & \rightarrow & LSL \\ L & \rightarrow & a|b|\cdots|z \end{cases}$$

Cette grammaire définit le langage des expressions sur V qui sont bien parenthésées.

Arbre de dérivation

Exemple : un arbre de dérivation de la chaîne $a()b$, pour la grammaire précédente :



Grammaire Ambigüe

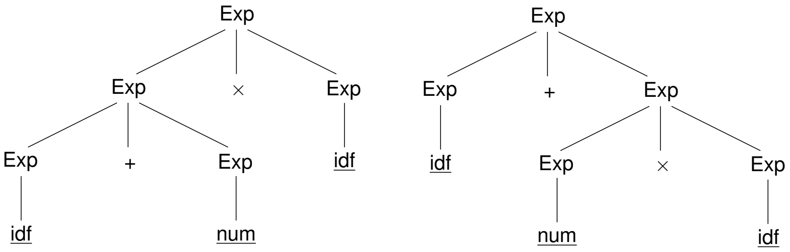
Définition: (Grammaire ambigüe)

Une grammaire est ambigüe si et seulement si il existe une chaîne qui admet plusieurs arbres de dérivation.

Exemple : Expressions arithmétiques

$$\begin{cases} Exp & \rightarrow & (Exp) \\ Exp & \rightarrow & Exp + Exp \\ Exp & \rightarrow & Exp - Exp \\ Exp & \rightarrow & Exp * Exp \\ Exp & \rightarrow & \underline{num} \\ Exp & \rightarrow & \underline{idf} \end{cases}$$

Chaîne *idf* + *num* $\times$ *idf* :



Arbres de dérivation

Arbres abstraits

Élimination des ambiguïtés

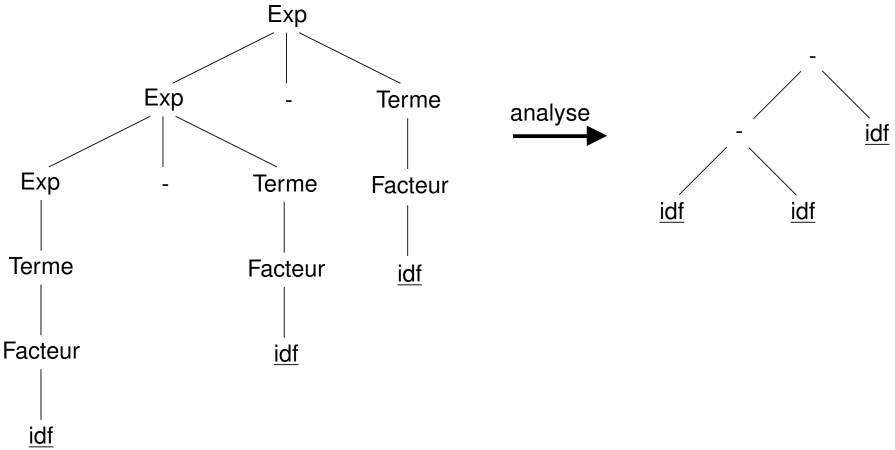
1. Choix des priorités : L'opérateur $\times$ est prioritaire sur $+$ et $-$.

$$\begin{array}{lcl} Exp & \rightarrow & Terme \\ Terme & \rightarrow & Terme * Terme | Facteur \\ Facteur & \rightarrow & \underline{idf} | \underline{num} | (Exp) \end{array}$$

2. Choix d'associativités : On décide que les opérateurs sont associatifs à gauche.

$$\begin{array}{lcl} Exp & \rightarrow & Exp + Terme | Exp - Terme | Terme \\ Terme & \rightarrow & Terme * Facteur | Facteur \\ Facteur & \rightarrow & \underline{idf} | \underline{num} | (Exp) \end{array}$$

Chaîne *idf* – *idf* – *idf* :



Langages de programmation

Les langages de programmation peuvent être caractérisés par une grammaire hors contexte. Un programme écrit dans un langage de programmation est syntaxiquement correct si on peut lui associer un arbre de dérivation.

Prog	$\rightarrow$	<u>`program`</u> Liste_idf <u>`begin`</u> Liste_inst <u>`end`</u> ;
Liste_idf	$\rightarrow$	ε | Liste_idf <u>`idf`</u> ;
Liste_inst	$\rightarrow$	ε | Inst ; Liste_Inst
Inst	$\rightarrow$	<u>`idf`</u> := Exp ;
Exp	$\rightarrow$	Exp + Terme | Exp - Terme | Terme
Terme	$\rightarrow$	<u>`idf`</u> | <u>`num`</u> | (Exp)

Exemple

Prog	$\rightarrow$	program Liste_idf begin Liste_inst end ;
Liste_idf	$\rightarrow$	ε | Liste_idf idf ;
Liste_inst	$\rightarrow$	ε | Inst ; Liste_Inst
Inst	$\rightarrow$	idf := Exp ;
Exp	$\rightarrow$	Exp + Terme | Exp - Terme | Terme
Terme	$\rightarrow$	idf | num | (Exp)

Le programme suivant est syntaxiquement correct :

```
program
    A; B; A;
begin
    A := 1;
    Total := A + B;
end
```

Summary

Exemple

On prend le langage suivant :

$$\text{Exp} \rightarrow \text{Exp + Exp} \mid \text{Exp - Exp} \mid \underline{\texttt{num}}$$

On peut facilement calculer la valeur d'une telle expression par induction sur les chaînes de la grammaire :

- Si $Exp \rightarrow \underline{\texttt{num}}$, alors $val(Exp) = num$
- Si $Exp \rightarrow Exp_1 + Exp_2$, alors $val(Exp) = val(Exp_1) + val(Exp_2)$
- Si $Exp \rightarrow Exp_1 - Exp_2$, alors $val(Exp) = val(Exp_1) - val(Exp_2)$

On peut calculer des propriétés sur l'arbre abstrait grâce à des grammaires attribuées.

Grammaire Attribuée

Définition: (Grammaire Attribuée)

Une grammaire attribuée est une grammaire Hors-Contexte $G(V_T, V_N, S, R)$, à laquelle on associe à chaque élément de V_T et de V_N des attributs.

Il existe 2 types d'attributs :

- attribut synthétisé (noté $\uparrow^{attr}$) : attribut dont la valeur est transmise du fils vers le père dans l'arbre de dérivation
- attribut hérité (noté $\downarrow_{attr}$) : attribut dont la valeur est transmise du père vers le fils dans l'arbre de dérivation

Exemple

Pour déterminer la valeur de l'expression, on utilise des attributs synthétisés :

$$\begin{array}{lcl}
\text{Exp} \uparrow^{val} & \to & \text{Exp} \uparrow^{val_1} + \text{Exp} \uparrow^{val_2} \\
 & & val = val_1 + val_2 \\
 & & \\
\text{Exp} \uparrow^{val} & \to & \text{Exp} \uparrow^{val_1} - \text{Exp} \uparrow^{val_2} \\
 & & val = val_1 - val_2 \\
 & & \\
\text{Exp} \uparrow^{val} & \to & \underline{\texttt{num}} \\
 & & val = num
\end{array}$$

Summary

Problème de la reconnaissance

Soit une grammaire $G = (V_T, V_N, S, R)$. On note $L(G)$ le langage engendré par G. Le problème de la reconnaissance consiste à répondre à la question :

$$\text{Est-ce que } p \in V_T^* \in L(G)\,?$$

Pour répondre à cette question, on a deux possibilités :

- Analyse ascendante : faire correspondre la chaîne à des parties droites de règles, et remonter vers l'axiome.
- Analyse descendante : Partir de l'axiome et obtenir la chaîne à dériver en appliquant les règles.

Exemple

On considère la grammaire G suivante, avec $V_T = \{a, b, c\}$ et $S = P$:

$$\begin{array}{lcl} \mathrm{P} & \rightarrow & \mathrm{L}b \\ \mathrm{L} & \rightarrow & a \mid a\mathrm{L} \mid c\mathrm{L} \end{array}$$

On cherche à déterminer si $acaab \in L(G)$, c'est à dire si $P \rightarrow_R^* acaab$.
Il faut donc faire un parcours de l'arbre des dérivations possibles.

Grammaires LL(1)

Définition: (Grammaires LL)

- *La classe des grammaires LL(1) définit les grammaires dont il est possible de répondre au problème de la reconnaissance de facon déterministe par une analyse descendante en connaissant uniquement le premier symbole de la chaîne.*
- *D'une manière générale, la classe des grammaires LL(k) est l'ensemble des grammaires dont le problème de reconnaissance se résout de façon déterministe par une analyse descendante en connaissant les k premiers symboles de la chaîne.*

Exemple

$$\begin{array}{lcl} S & \rightarrow & Ab \\ S & \rightarrow & a \\ A & \rightarrow & aaS \\ A & \rightarrow & b \end{array}$$

Cette grammaire est elle LL(1) ?

$$\begin{array}{lcl} S & \rightarrow & Ab \\ S & \rightarrow & \varepsilon \\ A & \rightarrow & aaS \\ A & \rightarrow & b \end{array}$$

Cette grammaire est elle LL(1) ?

Ensembles des symboles directeurs

Soit $G = (V_T, V_N, S, R)$ une grammaire. On définit :

- soit $\alpha \in (V_T \cup V_N)^*$, le prédicat **Vide**$(\alpha)$ est vrai si et seulement si ε peut être dérivé de α, c'est à dire $\alpha \rightarrow^* \varepsilon$.
- soit $\alpha \in (V_T \cup V_N)^*$, **Premier**$(\alpha)$ est l'ensemble des symboles terminaux qui peuvent débuter une dérivation de α.

$$\textbf{Premier}(\alpha) = \{a \in V_T / \exists \beta \in V_T^*, \alpha \rightarrow^* a\beta\}$$

- Soit $X \in V_N$. **Suivant**(X) est l'ensemble des symboles terminaux qui peuvent suivre une occurence de X dans une dérivation de S par G.

$$\textbf{Suivant}(X) = \{a \in V_T / \exists (\alpha, \beta) \in {V_T^*}^2, S \rightarrow^* \alpha X a \beta\}$$

- soit $X \rightarrow u$ une règle de R, avec $u \in (V_T \cup V_N)^*$. **Directeur**$(X \rightarrow u)$ est l'ensemble des symboles terminaux qui peuvent débuter une dérivation par cette règle.

$$\textbf{Directeur}(X \rightarrow u) = \textbf{Premier}(u) \cup (\text{ si } \textbf{Vide}(u) \text{ alors } \textbf{Suivant}(X), \text{ sinon } \emptyset)$$

Calcul de Premier et Suivant

Premier(u) : les u qui nous intéressent sont les parties droites de règles (voir définition Directeur)

- Si $u = \varepsilon$, alors **Premier**$(u) = \emptyset$,
- Si $u \in V_T$, alors **Premier**$(u) = u$,
- Si $u \in V_N$, alors **Premier**$(u) = \bigcup_{u \to v \in R}$ **Premier**(v)
- Si $u = u_1 u_2 \cdots u_n$ avec les $u_i \in (V_T \cup V_N)$, alors **Premier**$(u) =$ **Premier**$(u_1) \cup$ (si **Vide**(u_1) alors **Premier**$(u_2 \cdots u_n)$, sinon $\emptyset$)

Suivant(X) : On regarde toutes les parties droites de règles contenant le symbole X, c'est à dire les règles de la forme $u \to \alpha X \beta$ (α et β possiblement vides)

- **Suivant**$(X) = \bigcup_{u \to \alpha X \beta \in R}$ **Premier**$(\beta) \cup$ (sinon si **Vide**(β) alors **Suivant**(u) sinon $\emptyset$)

Exemple

$$
\begin{aligned}
Z &\rightarrow S\#\\
S &\rightarrow Xa\\
S &\rightarrow \varepsilon\\
X &\rightarrow Xb\\
X &\rightarrow Y\\
Y &\rightarrow aX\\
Y &\rightarrow \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
Premier(S\#) &= Premier(S) \cup \{\#\}\\
Premier(Xa) &= Premier(X) \cup \{a\}\\
Premier(Xb) &= Premier(X) \cup \{b\}\\
Premier(X) &= Premier(Xb) \cup Premier(Y)\\
&= \{b\} \cup Premier(Y)\\
Premier(Y) &= Premier(aX) = \{a\}\\
Premier(S) &= Premier(S\#) \cup Premier(Xa)\\
&= Premier(X) \cup \{\#, a\}
\end{aligned}
$$

$$
\begin{aligned}
Suivant(Z) &= \emptyset\\
Suivant(S) &= \{\#\}\\
Suivant(X) &= \{a\} \cup \{b\} \cup Suivant(Y)\\
Suivant(Y) &= Suivant(X)
\end{aligned}
$$

Exemple

$$
\begin{array}{lcl}
Z \rightarrow S\# & \quad & Directeur(Z \rightarrow S\#) = \{a, b, \#\} \\
S \rightarrow Xa & & Directeur(S \rightarrow Xa) = \{a, b\} \\
S \rightarrow \varepsilon & & Directeur(S \rightarrow \varepsilon) = \{\#\} \\
X \rightarrow Xb & & Directeur(X \rightarrow Xb) = \{a, b\} \\
X \rightarrow Y & & Directeur(X \rightarrow Y) = \{a, b\} \\
Y \rightarrow aX & & Directeur(Y \rightarrow aX) = \{a\} \\
Y \rightarrow \varepsilon & & Directeur(Y \rightarrow \varepsilon) = \{a, b\}
\end{array}
$$

CNS pour qu'une grammaire soit LL(1)

Théorème:

Une grammaire est LL(1) si et seulement si pour tout couple de règles $(X \to u_1, X \to u_2)$, *on a*

$$\textit{\textbf{Directeur}}(X \to u_1) \cap \textit{\textbf{Directeur}}(X \to u_1) = \emptyset$$

Transformation de grammaires

Le problème de construire une grammaire LL(1) équivalente à une grammaire Hors contexte est *indécidable* : Il n'existe pas d'algorithme général pour rendre une grammaire LL(1). On peut cependant utiliser des techniques qui fonctionnent généralement en pratique.

Suppression de la récursivité à gauche

Un non terminal A est récursif à gauche lorsqu'il existe une dérivation de la forme $A \rightarrow A\alpha$.
Dans ce cas, on introduit un nouveau non terminal A' et on remplace les règles de départ :

$$\begin{array}{rcl} A & \rightarrow & A\alpha_1|A\alpha_2|\cdots|A\alpha_n \\ A & \rightarrow & \beta_1|\beta_2|\cdots|\beta_m \end{array}$$

par :

$$\begin{array}{rcl} A & \rightarrow & \beta_1 A'|\beta_2 A'|\cdots|\beta_m A' \\ A' & \rightarrow & \varepsilon|\alpha_1 A'|\alpha_2 A'|\cdots|\alpha_n A' \end{array}$$

Exemple

On considère la grammaire suivante, récursive à gauche :

$$\begin{array}{lcl} S & \rightarrow & A\# \\ A & \rightarrow & Aa \\ A & \rightarrow & b \end{array}$$

Après suppression de la récursivité à gauche, on obtient :

$$\begin{array}{lcl} S & \rightarrow & A\# \\ A & \rightarrow & bA' \\ A' & \rightarrow & aA' \\ A' & \rightarrow & \varepsilon \end{array}$$

Factorisation

Si une grammaire possède deux règles de la forme

$$\begin{array}{lcl} A & \rightarrow & \omega\alpha_1 \\ A & \rightarrow & \omega\alpha_2 \end{array}$$

On transforme ces deux règles et on crée un nouveau symbole non terminal :

$$\begin{array}{lcl} A & \rightarrow & \omega A' \\ A' & \rightarrow & \alpha_1 \\ A' & \rightarrow & \alpha_2 \end{array}$$

Substitution

Si une grammaire possède des règles de la forme

$$\begin{array}{lcl} A & \rightarrow & B|C \\ B & \rightarrow & \omega X|\alpha_1 \\ C & \rightarrow & \omega Y|\alpha_2 \end{array}$$

On transforme la grammaire en substituant B et C pour obtenir :

$$A \rightarrow \omega X|\alpha_1|\omega Y|\alpha_2$$

Ensuite, on factorise et on obtient :

$$\begin{array}{lcl} A & \rightarrow & \omega A'|\alpha_1|\alpha_2 \\ A' & \rightarrow & X|Y \end{array}$$