Chapitre 16 : Grammaires non contextuelles

23 janvier

Le chapitre sur les langages réguliers nous a permis de travailler sur une classe de langages dont l'expressivité permet par exemple de faire de la recherche de motifs (préfixes, suffixes, facteurs...). L'expressivité de la classe des langages réguliers est cependant limitée et on a facilement mis en évidence des langages qui ne correspondent pas à cette classe. Il faut avoir en tête qu'un des objectifs majeurs de la théorie des langages est de permettre l'analyse automatique d'un code produit dans un langage de programmation donné. Si les langages réguliers permettent de faire ce qu'on appelle l'analyse lexicale c'est-à-dire la reconnaissance des différentes chaines de caractères utilisées par le programmeur : elle permet notamment de reconnaitre les "mots clés" du langage comme `for`,`while`,`struct`...les commentaires, une variable (d'où des règles) ou encore une constante numérique. Les langages réguliers ne permettront cependant pas de comprendre la structure de votre code afin de l'interpréter en une suites d'instructions machines élémentaires. Ce sont les grammaires non contextuelles qui vont permettre cette interprétation de la structure du code que l'on appellera l'analyse syntaxique.

1 Un premier exemple

Nous allons commencer par une question qui apparait naturellement dans l'analyse d'un programme : l'interprétation du parenthésage.

Nous allons donc considérer le langage des expressions bien parenthésées sur l'alphabet suivant : $\Sigma = \{(,)\}$.

Une expression bien parenthésée est définie inductivement de la manière suivante :

- ϵ est bien parenthésée.
- Si e et e' sont bien parenthésées alors $e.e'$ l'est aussi.
- Si e est bien parenthésée alors (e) l'est aussi.

Les grammaires vont d'ailleurs parfaitement correspondre à l'expressivité de formules que l'on peut définir inductivement.

Voici la description de la grammaire que nous allons considérer :

$$S \to \epsilon | SS | (S)$$

On remarquera au passage que le lemme de l'étoile nous permet de montrer que ce langage n'est pas régulier.

2 DÉFINITIONS

Définition 1. *Une grammaire non contextuelle est un 4-uplet $\mathcal{G} = \{\mathcal{V}, \Sigma, \mathcal{R}, S\}$ où*
- *$\mathcal{V}$ est un ensemble de symboles appelés symboles non terminaux.*
- *Σ est un ensemble de symboles appelés symboles terminaux. On aura $\Sigma \cap \mathcal{V} = \emptyset$.*
- *S est un élément de $\mathcal{V}$ appelé symbole initial.*
- *$\mathcal{R} \subset \mathcal{V} \times (\Sigma \cup \mathcal{V})^*$ est appelé ensemble de règles de production. C'est un ensemble fini.*

Dans l'exemple de la partie précédente :

Définition 2. *Soit $\mathcal{G} = \{\mathcal{V}, \Sigma, \mathcal{R}, S\}$ une grammaire non contextuelle. Soit $u = u_1 X u_2 \in (\Sigma \cup \mathcal{V})^* \mathcal{V} (\Sigma \cup \mathcal{V})^*$. On dit que u se dérive directement en v s'il existe une règle de la forme $X \to s$ telle que $v = u_1 s u_2$. On note alors $u \Rightarrow v$.*

Définition 3. *Soit $\mathcal{G} = \{\mathcal{V}, \Sigma, \mathcal{R}, S\}$ une grammaire non contextuelle. Soit $u = u_1 X u_2 \in (\Sigma \cup \mathcal{V})^* \mathcal{V} (\Sigma \cup \mathcal{V})^*$. On dit que u se dérive immédiatement à gauche (resp à droite) en v si $u \Rightarrow v$ et $u_1 \in \Sigma^*$ (resp $u_2 \in \Sigma^*$). Autrement dit, on a choisi d'appliquer une règle de production au symbole non terminal le plus à gauche (resp le plus à droite).*

Exemple 1. *Considérons : $S \to SGa|b$; $G \to GSb|a$.*

On remarque qu'il y a des choix à faire.

Définition 4. *Soit $\mathcal{G} = \{\mathcal{V}, \Sigma, \mathcal{R}, S\}$ une grammaire non contextuelle.*
On note $\Rightarrow^*$ la cloture réflexive transitive de $\Rightarrow$. On dit que u se dérive en v quand $u \Rightarrow^* v$ c'est-à-dire qu'il existe une suite $u_0 = u, \dots u_n = v$ d'éléments de $(\Sigma \cup \mathcal{V})^*$ telle que pour tout i dans $\{0, \dots, n-1\}, u_i \Rightarrow u_{i+1}$. Une suite de dérivations immédiates est appelée une dérivation. Si toutes les dérivations immédiates utilisées sont gauches (resp. droites) on dit que la dérivation est gauche (resp. droite).

Définition 5. *Soit $\mathcal{G} = \{\mathcal{V}, \Sigma, \mathcal{R}, S\}$ une grammaire non contextuelle. Le langage engendré par cette grammaire est l'ensemble des mots $u \in \Sigma^*$ pour lesquelles il existe une dérivation depuis le symbole initial S.*
$\mathcal{L}(\mathcal{G}) = \{u \in \Sigma^* : S \Rightarrow^* u\}$.

Définition 6. *Un langage engendré par une grammaire non contextuelle est un langage non contextuel.*

Pourquoi parle-t-on de grammaire "non contextuelle" ?

3 DES EXEMPLES

Nous allons décrire les grammaires qui correspondent aux langages suivants :

1. $L = \{a^n b^n : n \in \mathbb{N}\}$.
2. L'ensemble des palindromes sur l'alphabet $\{a, b\}$ (exemple de preuve formelle).
3. L'ensemble des formules logiques du calcul propositionnel.
4. Considérons la grammaire suivante : $E \to I|E + E|E * E|(E)$; $I \to a|b|Ia|Ib|I0|I1$. Montrer que $(aa10 + bbb) * (a + b0)$ est dans le langage engendré par cette grammaire de symbole initial E.
5. Considérons maintenant la grammaire suivante :
   - $\mathcal{V} = \{S, I, E, V, N, C\}$.
   - $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, print, =, +, (,), ;, x, y\}$.
   - $\mathcal{R}$:

- $S \to I; S|\epsilon$
- $I \to V = E|print(E)$
- $E \to E + E|N$
- $V \to x|y$
- $N \to CN|C$
- $C \to 0|1|2|3|4|5|6|7|8|9$

Montrer que le mot $x = 42;$ appartient au langage engendré par la grammaire.

4 $Rec \subset CFL$

Théorème 1. *L'ensemble des langages réguliers est inclus dans l'ensemble des langages non contextuels.*

5 Arbre de dérivation

Le plus souvent, un mot généré par une grammaire admet plusieurs dérivations distinctes. Si l'on considère la grammaire $S \to \neg S \mid (S) \mid S \vee S \mid S \wedge S \mid a \mid b \mid c$, par exemple, les trois dérivations suivantes correspondent au même mot $a \wedge b \vee c$:

— $S \Rightarrow S \wedge S \Rightarrow a \wedge S \Rightarrow a \wedge S \vee S \Rightarrow a \wedge b \vee S \Rightarrow a \wedge b \vee c$
— $S \Rightarrow S \wedge S \Rightarrow S \wedge S \vee S \Rightarrow S \wedge S \vee c \Rightarrow S \wedge b \vee c \Rightarrow a \wedge b \vee c$
— $S \Rightarrow S \vee S \Rightarrow S \vee c \Rightarrow S \wedge S \vee c \Rightarrow a \wedge S \vee c \Rightarrow a \wedge b \vee c$

En un certain sens, que nous allons préciser, les deux premières dérivations ne sont " pas vraiment différentes " : depuis $S \wedge S$, on peut commencer par remplacer le premier S et obtenir $a \wedge S$ ou commencer par le deuxième et obtenir $S \wedge S \vee S$, mais " ça commute ". La troisième dérivation, en revanche, est complètement différente, parce qu'elle ne correspond pas au même *arbre de dérivation*.

Définition 7. *Soit $\mathcal{G} = \{\mathcal{V}, \Sigma, \mathcal{R}, S\}$ une grammaire non contextuelle. On va associer à chaque dérivation dans $\mathcal{G}$ un arbre de la manière suivante :*

— *La racine de l'arbre est S.*
— *Les noeuds internes sont des élements de $\mathcal{V}$.*
— *Les feuilles sont des éléments de $\Sigma \cup \{\epsilon\}$.*
— *toute feuille d'étiquette ϵ est fille unique.*
— *Chaque noeud interne X qui a pour fils $u_1, u_2, \ldots, u_n$ est tel que $X \to u_1 ... u_n$ est une règle de la grammaire.*
— *La concaténation des feuilles de gauche à droite est le mot engendré par la dérivation.*

Dessinons les arbres correpondant aux trois exemples précédents.

Exemple 2. $S \to S + T|T$
$T \to T \times F|F$
$F \to (S)|a$

Construire un arbre pour $(a + a + a) \times (a + a)$.

On veut maintenant formaliser le fait que deux dérivations soient ou non *réellement différentes*, ce qui signifie qu'elles ne correspondent pas au même arbre de dérivation.

Proposition 1. *Soient $\mathcal{G}$ une grammaire et $u \in \mathcal{L}(\mathcal{G})$. Il y a une bijection entre les dérivations gauches de u et les arbres de dérivation de u.*

Remarque 1. *De même, il y a une bijection entre les dérivations droites de u et ses arbres de dérivation.*

Démonstration 1. *À un arbre de dérivation T (dont la racine est un symbole $\alpha \in \Sigma \cup V$ quelconque) on associe la dérivation $d(T)$ définie comme suit :*

- *si T est une feuille, alors $d(T) = \alpha$ (dérivation de longueur nulle) ;*
- *sinon, $T = (X; \alpha_1, \dots, \alpha_k)$ avec $X \to \alpha_1 \dots \alpha_k$ une règle de G et $\alpha_1, \dots, \alpha_k \in \Sigma \cup V$. On a alors des dérivations $d(\alpha_i) : \alpha_1 \Rightarrow^* u_i$, que l'on " concatène " : $d(T) = X \Rightarrow \alpha_1 \dots \alpha_k \Rightarrow^* u_1\alpha_2 \dots \alpha_k \Rightarrow^* \dots \Rightarrow^* u_1 \dots u_k$.*

Autrement dit, on applique les règles dans l'ordre du parcours préfixe de l'arbre. On remarque que :

- *en prenant T un arbre de dérivation de u, on obtient bien une dérivation de u (la racine de T est S, et les feuilles lues dans l'ordre donnent u) ;*
- *il s'agit bien, par construction, d'une dérivation gauche.*

Inversement, à une dérivation gauche $S \Rightarrow^* \alpha \in (\Sigma \cup V)^*$, on associe un arbre de dérivation partiel, dont les feuilles sont étiquetées par des symboles quelconques et donnent α quand on les lit de gauche à droite, comme suit :

- *si la dérivation est de longueur nulle, alors on prend l'arbre réduit à S ;*
- *sinon, la dérivation est de la forme $S \Rightarrow^* uX\gamma \Rightarrow u\beta\gamma$ avec $u \in \Sigma^*$, $\beta, \gamma \in (\Sigma \cup V)^*$ et $X \to \beta$ dans $\mathcal{R}$. On prend alors l'arbre de dérivation associé à $S \Rightarrow^* uX\gamma$ et l'on remplace la feuille X la plus à gauche par $(X; \beta_1, \dots, \beta_k)$.*

Ces deux applications sont réciproques l'une de l'autre.

Définition 8. *Soient $\mathcal{G}$ une grammaire et $u \in \mathcal{L}(\mathcal{G})$.*

- *On dit que u est* ambigu pour *$\mathcal{G}$ s'il existe plusieurs arbres de dérivation distincts pour u.*
- *On dit que la grammaire $\mathcal{G}$ est* ambiguë *s'il existe au moins un mot u ambigu pour $\mathcal{G}$.*

Remarque 2. *L'ambiguïté est ici une propriété de la **grammaire**, et non du **langage**.*

Définition 9. *Deux grammaires $\mathcal{G}$ et $\mathcal{G}'$ sont dites* faiblement équivalentes *si $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$.*

Remarques 1.

Le faiblement *vient du fait qu'on ne demande que l'égalité des langages générés, et rien sur les dérivations ou les arbres syntaxiques.*

Il n'y a pas vraiment de manière canonique de définir une équivalence forte.

Une grammaire ambiguë peut être faiblement équivalente à une grammaire non ambiguë ou non.

Théorème 2. *Soient $\mathcal{G}$ une grammaire et $u \in \mathcal{L}(\mathcal{G})$. Les propositions suivantes sont équivalentes :*

1. *u est ambigu pour $\mathcal{G}$;*
2. *u admet plusieurs dérivations gauches pour $\mathcal{G}$;*
3. *u admet plusieurs dérivations droites pour $\mathcal{G}$.*

Exemple 3. *Considérons $E \to E + E | E \times E | (E) | a$ de symbole initial E. Montrer que cette grammaire est ambigue.*

Exemple 4. $S \to N_p V_p$
$N_p \to C_n | C_n P_p$
$C_n \to AN$
$P_p \to PC_n$
$C_v \to V | V N_p$
$V_p \to C_v | C_v P_p$
$A \to$ *un* | *le* | *la*
$V \to$ *joue* | *voie* | *touche*
$N \to$ *chien* | *chat* | *balle*
$P \to$ *avec*

Considérons la phrase : "le chien touche le chat avec la balle".

Exemple 5. *Un exemple classique d'ambiguïté dans les grammaires est celui du* dangling else. *En C, comme dans beaucoup de langages, la clause* `else` *est optionnelle dans un* if/then/else.

Les deux morceaux de code suivants sont syntaxiquement corrects :

```
if (n >= 0)
  if (n <= 10)
    printf("0 <= n <= 10\n");
  else
    printf("n > 10\n");
```

```
if (n >= 0)
  if (n <= 10)
    printf("0 <= n <= 10\n");
else
  printf("n < 0\n");
```

Intuitivement, ils correspondent aux deux arbres syntaxiques suivants :

Le problème est que, à l'exception du contenu du deuxième `printf`*, il s'agit en fait de* deux fois le même morceau de code *: l'indentation est ignorée par le compilateur en C. Pour lever l'ambiguïté, il y a trois solutions :*

- *garder une grammaire ambiguë, mais préciser le choix qui sera fait lors de l'analyse syntaxique (en l'occurrence, c'est le premier arbre qui sera produit) ;*
- *réécrire la grammaire pour supprimer l'ambiguïté, tout en gardant la même syntaxe pour le langage de programmation (on remplace la grammaire par une grammaire faiblement équivalente) ;*
- *modifier la syntaxe du langage (les règles d'indentation de Python, par exemple, évitent ce problème).*

Exemple 6. *Voici un exemple de grammaire permettant de visualiser le problème :*
$I \rightarrow$ **if** (E)*then*I
$I \rightarrow$ **if** (E)*then*I **else** I
$I \rightarrow E;$
$E \rightarrow V = E|E$**<=**$E|E$**=>**$E|N$
$N \rightarrow CN|C|V$
$C \rightarrow 0|1|2|3|4|5|6|7|8|9$
$V \rightarrow x|y|z$

Considérons le mot suivant :
`if (x>=0) then if (y <=5) then z=2; else z=3;`

6 Analyse syntaxique

Il s'agit de la théorie algorithmique qui consiste à construire un arbre d'analyse d'un mot dans une grammaire donnée. C'est une des étapes fondamentales de la compilation qui permet de construire l'arbre qui capture la structure de votre code. Les techniques sont hors-programme. On attend cependant que la grammaire qui définit un langage de programmation ne soit pas ambigue.

Exemple ad-hoc d'algo permettant de générer un arbre de dérivation pour un mot reconnu par la grammaire suivante :

Exemple 7. $S \rightarrow TS|c$
$T \rightarrow aSb$

```
type arb = A|B|C|S of arb*arb | T of arb*arb*arb;;

let rec genereS l = match l with
    |'c'::t-> C,t
    |h::t-> (let a1,l1 = genereT l in
             let a2,l2 = genereS l1 in
             (S(a1,a2)),l2)
    |_->failwith "Syntax Error"

and genereT l = match l with
    |'a'::t-> (let a1,l1 = genereS t in match l1 with
                |'b'::s-> (T(A,a1,B)),s
                |_->failwith "Syntax Error")
    |_->failwith "Syntax Error";;
```

Exemple 8. $S \rightarrow T + S|T$
$T \rightarrow F \times F|F$
$F \rightarrow (S)|int$

Ce qui suit est hors programme mais très classique.

7 Clotures

Montrons que l'ensemble des langages non contextuels est stable par union, concaténation et étoile de Kleene. On va montrer qu'on n'a pas la stabilité par intersection en considérant $L_1 = \{a^n b^n c^k : (n,k) \in \mathbb{N}^2\}$ et $L_2 = \{a^k b^n c^n : (n,k) \in \mathbb{N}^2\}$.

8 Forme normale de Chomsky

Le résultat suivant permet de restreindre la forme des règles de dérivation utilisées dans une grammaire. D'un point de vue algorithmique c'est un point central car il permet de considérer les règles dans cette forme uniquement et non pas aussi générales que la définition d'une grammaire ne l'autorise.

Théorème 3. *Toute grammaire qui ne génère pas le mot vide est faiblement équivalente à une grammaire dont toutes la règles sont de la forme $X \to AB$ ou $X \to a$ où A et B sont des symboles non terminaux distincts du symbole initial et a est un symbole terminal. On appelle la grammaire dont les règles sont de cette forme, forme normale de Chomsky.*

Voici le principe de la transformation (en 5 étapes) :

1. Si nécessaire (si S apparait dans un des termes de droite), on introduit un nouveau symbole non terminal S_0 qui sera le nouveau symbole initial et la règle $S_0 \to S$ où S est l'ancien symbole initial.
2. Elimination des règles de la forme $A \to \epsilon$: pour toute règle de cette forme, on va considérer toutes les règles où A apparait dans le terme de droite. Chacune de ces règles sera de la forme $X \to \alpha A \beta$ et on ajoutera les règles $X \to \alpha\beta$. Attention, si on a plusieurs occurences de A alors il faut ajouter des règles qui suppriment chaque combinaison possible de A. Par exemple si on a $X \to aAbAcA$ on ajoutera $X \to abc$, $X \to aAbAc$,$X \to aAbc$,$X \to abAcA$,$X \to abAc$, $X \to aAbcA$,$X \to abAc$. En procédant ainsi, on peut créer de nouvelles règles de la forme que l'on est en train d'éliminer, dans ce cas on recommence en évitant de faire réapparaitre des formes déjà éliminées jusqu'à ne plus avoir de telles règles du tout.
3. Elimination des règles de la forme $A \to B$: pour chacune de ces règles, on ajoute pour chaque règle de la forme $B \to \alpha$ la règle $A \to \alpha$. On répète sans recréer de règles déjà éliminées jusqu'à ne plus en avoir.
4. Remplacement des longues productions : pour chaque production dont le terme de droite est de longueur au moins 3 de la forme $X \to \alpha_1 \dots \alpha_n$, on crée de nouveaux symboles non terminaux et les règles $X \to \alpha_1 N_1$, $N_1 \to \alpha_2 N_2, \dots, N_{n-2} \to \alpha_{n-1}\alpha_n$.
5. Pour chaque terminal a qui reste à droite d'une règle sans être seul, on introduit un nouveau non terminal X_a qui le remplace et une règle $X_a \to a$.

Appliquer cette méthode aux grammaires suivantes :

1. $S \to AbA; A \to Aa|\epsilon$.
2. $S \to aXbX; X \to aY|bY|\epsilon; Y \to X|c$.