INF431

Grammaires
CORRIGÉ
Version: 1131:2574M

Introduction

L'objectif de ce sujet est de vous faire travailler sur la notion de grammaire qui est à la base des langages informatiques, et sur l'analyse syntaxique.

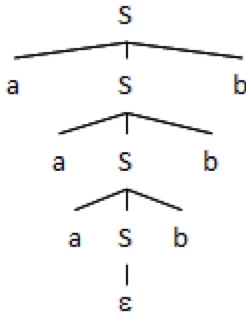
1 Pour commencer…

On considère la grammaire sur l'alphabet de symboles terminaux $\{a, b\}$, ayant comme unique symbole non-terminal le symbole de départ S, et composée des deux productions suivantes :

$$S \longrightarrow \epsilon \qquad S \longrightarrow aSb$$

Question 1 Montrer que S peut se récrire en $aaabbb$ en donnant l'arbre de production. ◇

Solution. L'arbre de production :



□

On considère maintenant les langages d'alphabet $\{a, b\}$ qui vérifient les deux propriétés suivantes :

1. Le mot vide ϵ est dans le langage.
2. Si W est dans le langage alors aWb y est aussi.

Question 2 Donner trois langages distincts, dont le plus petit langage, qui satisfont ces deux propriétés. Montrer que la grammaire de la question 1 engendre exactement le plus petit langage. ◇

Solution. Voici 3 langages qui satisfont ces deux propriétés $(a|b)^*$, a^*b^* et le plus petit possible $\{a^nb^n \mid n \geq 0\}$.

Par récurrence directe sur n, on vérifie que la grammaire engendre tous les mots de la forme a^nb^n. Pour montrer que la grammaire ne produit pas d'autre mot, on effectue cette fois-ci une induction sur un arbre de production comme suit. Le seul arbre de production de taille 1 est $S \to \epsilon$ qui produit le mot $\epsilon = a^0b^0$. Supposons que l'on a un arbre de production non réduit à l'application de la première production. Sa racine a donc été produite par la seconde production. Elle a trois nœuds fils étiquetés dans l'ordre par a, S, et b. Par hypothèse, S permet de dériver un mot de la forme a^nb^n pour un certain n. L'arbre produit donc le mot $a^{n+1}b^{n+1}$. □

```
{   "menu":
    {
        "id": "file",
        "value": "File",
        "popup":
        {
            "menuitem":
                [
                    { "value": "New", "onclick": "CreateNewDoc()" },
                    { "value": "Open", "onclick": "OpenDoc()" },
                    { "value": "Close", "onclick": "CloseDoc()" }
                ]
        }
    } }
```

FIGURE 1 – Exemple d'objet JSON

2 JavaScript Object Notation

JSON (JavaScript Object Notation) est un format de données textuel et générique. Il permet de représenter de l'information structurée. Un objet JSON est de trois types possibles (voir figure 1) :

- Une valeur terminale : un booléen, un nombre, une chaîne de caractères ou la constante `null` ;
- Un tableau d'objets, éventuellement vide, de la forme $[objet_1, \ldots, objet_k]$;
- Une liste associative : une liste de paires (nom, objet), éventuellement vide, de la forme $\{nom_1 : objet_1, \ldots, nom_k : objet_k\}$;

Question 3 Écrire la grammaire de ce langage, en supposant que l'on dispose des symboles terminaux `bool` (pour les booléens), `num` (pour les nombres), `string` (pour les chaînes de caractères), `null`, `lbrace` (« { »), `rbrace` (« } »), `lbrack` (« [»), `rbrack` («] »), `comma` (« , ») et `colon` (« : »).

Solution. Voici une grammaire de JSON :

Object	$\rightarrow$	TerminalValue | AssociativeList | Array
TerminalValue	$\rightarrow$	`bool` | `num` | `string` | `null`
AssociativeList	$\rightarrow$	`lbrace` `rbrace` | `lbrace` Pairs `rbrace`
Pairs	$\rightarrow$	Pairs `comma` Pair | Pair
Pair	$\rightarrow$	`string` `colon` Object
Array	$\rightarrow$	`lbrack` `rbrack` | `lbrack` Objects `rbrack`
Objects	$\rightarrow$	Objects `comma` Object | Object

□

3 Ambiguïté

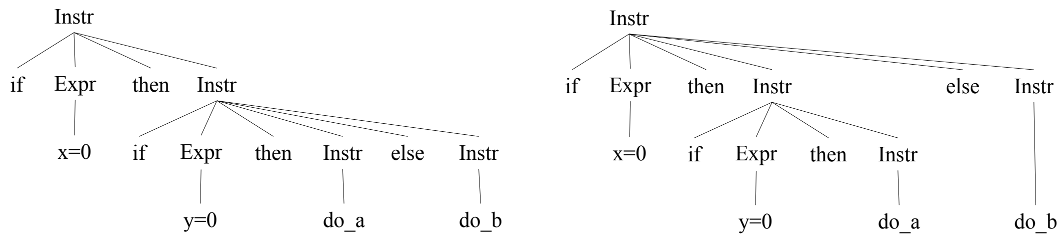
Une grammaire est ambiguë si elle permet d'engendrer un même mot (sur l'alphabet des symboles non-terminaux) de deux façons différentes. L'ambiguïté est à éviter si possible car elle rend les analyseurs non déterministes et donc non efficaces.

Comme nous allons le voir sur un exemple, il est parfois possible de transformer une grammaire ambiguë en une grammaire non-ambiguë équivalente. Considerer la grammaire suivante, qui est un fragment de la quasi-totalité des langages de programmation :

Instr	$\rightarrow$	`if` Expr `then` Instr
Instr	$\rightarrow$	`if` Expr `then` Instr `else` Instr
Instr	$\rightarrow$	`do_a` | `do_b`
Expr	$\rightarrow$	`x=0` | `y=0`

Question 4 Donner deux arbres de production pour une même expression. ◇

Solution. L'expression “if x=0 then if y=0 then do_a else do_b” admet les deux arbres de production suivants :



Le problème essentiel vient de la précédence, c'est à dire ici des règles de priorité. Dans le cas des langages de programmation, le else se rapporte en général au dernier if rencontré.

Question 5 Proposer une grammaire non-ambiguë pour ce langage. Vérifier qu'elle engendre le même langage. Argumenter, sans le prouver, que cette nouvelle grammaire est non-ambiguë. ◇

Solution. Pour lever l'ambiguïté, on va écrire une grammaire qui permet de dériver un if-then-else uniquement en rattachant le else au if le plus proche. On introduit pour cela un nouveau symbole non-terminal Instr_full qui correspond à une instruction à laquelle il est impossible de lui rattacher un potentiel else qui suivrait, On modifie alors la production du symbole non-terminal Instr en faisant intervenir Instr_full.

Voici la grammaire correspondante :

Instr	$\to$	if Expr then Instr
Instr	$\to$	if Expr then Instr_full else Instr
Instr	$\to$	do_a | do_b
Instr_full	$\to$	if Expr then Instr_full else Instr_full
Instr_full	$\to$	do_a | do_b
Expr	$\to$	x=0 | y=0

□

4 Algorithme d'analyse syntaxique CYK

En 1967, l'algorithme Cocke-Younger-Kasami (CYK) à base de programmation dynamique a été découvert. Un de ses avantages est qu'il fonctionne pour toute grammaire. Sa complexité est cubique en la taille de la séquence à analyser, et linéaire en le nombre de productions de la grammaire (si celle-ci est en forme normale de Chomsky).

La première étape de l'algorithme CYK consiste à normaliser les grammaires, c'est-à-dire à les transformer en des grammaires équivalentes mais sous une forme contrainte, dite normale de Chomsky. Elle consiste à n'avoir que des productions de l'une des trois formes suivantes :

— $S \to \epsilon$, où S est le symbole non-terminal initial de la grammaire.

— $X \to YZ$, où Y et Z sont des symboles non-terminaux.

— $X \to a$, où a est un symbole terminal.

De plus, si la production $S \to \epsilon$ est présente, alors S ne peut apparaître dans le membre droit d'aucune production.

Question 6 Mettre la grammaire suivante sous forme normale de Chomsky :

$$S \to TbT \qquad T \to TaT \mid ca$$

Solution. Voici une forme normale possible :

$$S \to TX \qquad X \to BT \qquad B \to b$$
$$T \to TY \mid CA \qquad Y \to AT \qquad A \to a \qquad C \to c$$

Nous supposons maintenant donnée une grammaire $\mathcal{G}$ sous forme normale de Chomsky constituée de g productions.

Question 7 Montrer que $\mathcal{G}$ reconnait le mot vide ϵ si et seulement si $S \to \epsilon$ est une production de $\mathcal{G}$. Dans ce cas, donner une grammaire sous forme normale de Chomsky qui engendre le même langage que $\mathcal{G}$ privé du mot vide ϵ avec $(g-1)$ productions. ◇

Solution. Si $S \to \epsilon$ est une production de $\mathcal{G}$, alors par définition ϵ est reconnu par $\mathcal{G}$. Inversement, si ϵ est reconnu par $\mathcal{G}$, alors il doit exister une dérivation produisant ϵ, et en donc en particulier une production menant à ϵ. A cause de la normalisation de Chomsky, cette production ne peut être que $S \to \epsilon$.

Définissons $\mathcal{G}'$ comme la grammaire $\mathcal{G}$ privée de la production $S \to \epsilon$. Donc tout mot reconnu par $\mathcal{G}'$ est reconnu par $\mathcal{G}$. Inversement, soit $w \neq \epsilon$ un mot reconnu par $\mathcal{G}$. Puisque $S \to \epsilon$ est une production présente dans $\mathcal{G}$, le symbole non-terminal S n'apparait dans le membre droit d'aucune production de $\mathcal{G}$. Ainsi, toute dérivation de $S \to^* w$ dans $\mathcal{G}$ n'utilise pas la production $S \to \epsilon$, et est donc bien une dérivation dans $\mathcal{G}'$. □

On suppose dorénavant que $S \to \epsilon$ n'est pas une production. Fixons une séquence $w := w_1 w_2 \dots w_n$ de n symboles terminaux, avec $n \geq 1$. L'analyse syntaxique consiste à décider si $S \to^* w$, c'est-à-dire si la séquence w peut se dériver de S par une série de productions de $\mathcal{G}$.

Vous avez vu en cours qu'il était possible de ramener ce problème de décision à la résolution d'un système d'équations booléennes positives. Lorsque la grammaire est sous forme normale de Chomsky, la résolution du système peut s'effectuer par programmation dynamique en considérant uniquement les variables booléennes $H_{i,k,X}$, pour chaque intervalle $[i,k[$ et non-terminaux X, avec $1 \leq i < k \leq n+1$, de sorte que $H_{i,k,X}$ est vraie si et seulement si $X \to^* w_i w_{i+1} \dots w_{k-1}$.

Question 8 Donner un algorithme pour décider en temps $\mathrm{O}(g \times n^3)$ si $S \to^* w$ dans $\mathcal{G}$. ◇

Solution. Le booléen $H_{i,k,X}$ est vrai si et seulement si une des deux conditions suivantes est remplie :

1. $k = i+1$ et $X \to w_i$ est une production de $\mathcal{G}$;
2. $k \geq i+2$, et il existe $j \in]i,k[$ et une production $X \to YZ$ tels que $H_{i,j,Y}$ et $H_{j,k,Z}$ sont vrais.

La réponse au problème complet est donc $H_{1,n+1,S}$. La première condition permet d'initialiser toutes les variables $H_{i,i+1,X}$, qui sont au nombre de n pour chaque non-terminal X. Il s'agit donc ensuite de calculer chaque variable $H_{i,k,X}$ par programmation dynamique à l'aide de la deuxième condition.

Plus précisément, la deuxième condition mène, comme vu en cours, à la récurrence suivante qui considère toutes les productions de la forme $X \to YZ$ et toutes les façons de découper l'intervalle $]i,j[$ en deux :

$$H_{i,k,X} = \bigvee_{\{X \to YZ\} \in \mathcal{G}} \bigvee_{j=i+1}^{k-1} (H_{i,j,Y} \wedge H_{j,k,Z}).$$

Puisqu'aucune production mène à ϵ, les symboles Y et Z consomment chacun au moins un symbole terminal, et l'indice j varie bien de $i+1$ à $k+1$.

Le tout peut donc être résolu par programmation dynamique comme suit :

```
Pour i ← 1,...,n et chaque non−terminal X
  H[i,i+1,X] ← [(X → w[i]) ∈ G]
Pour Δ ← 2,...,n
    Pour i ← 1,...,n+1−Δ et chaque non−terminal X
      k ← i+Δ , H_{i,k,X} ← faux
      Pour j ← i+1,...,k−1 et chaque production X → YZ de G
        H_{i,k,X} = H_{i,j,x} ∨ (H_{i,j,Y} ∧ H_{j,k,Z})
```

Soit ℓ_X le nombre de productions associées au non terminal X. Alors, la complexité de la procédure ci-dessus est au plus $\sum_X \ell_X \times n^3$. Mais le nombre g de productions de $\mathcal{G}$ satisfait $g = \sum_X \ell_X$. Donc la complexité finale est au plus $\mathrm{O}(g \times n^3)$. □