TD6 - Grammaires

Exercice 1

Soit la grammaire non contextuelle $\mathcal{G} = (\mathcal{V}_N, \mathcal{V}_T, \mathcal{R}, S)$ définie par les règles de production suivantes.

$$\mathcal{R} = \{S \to TaT \mid \varepsilon, \quad T \to UU \mid c, \quad U \to TVT \mid c, \quad V \to a \mid b \mid \varepsilon\}$$

Question 1. Quelles sont les *symboles non terminaux*, les *symboles terminaux* et le *symbole initial* de $\mathcal{G}$?

Question 2. Répondre par *vrai* ou par *faux* aux questions suivantes.

- $a \in \mathcal{L}(\mathcal{G})$
- pour $k \in \mathbb{N}$, $a^k \in \mathcal{L}(\mathcal{G})$
- pour $k \in \mathbb{N}$, $b^k \in \mathcal{L}(\mathcal{G})$
- $cac \in \mathcal{L}(\mathcal{G})$
- $ccacc \in \mathcal{L}(\mathcal{G})$
- $cccaccc \in \mathcal{L}(\mathcal{G})$
- pour $m, n \in \mathbb{N}$, $c^m a c^n \in \mathcal{L}(\mathcal{G})$
- $\Sigma^* aa \Sigma^* \notin \mathcal{L}(\mathcal{G})$
- $(a \mid b)^* \notin \mathcal{L}(\mathcal{G})$
- $T \Rightarrow TVTU$
- $T \Rightarrow^* TVTU$
- $U \Rightarrow^* UUUU$
- $U \Rightarrow^* cbUU$
- $cccac$ est ambiguë
- $\mathcal{G}$ est ambiguë

Question 3. Quel est le *langage* généré par $\mathcal{G}$?

Exercice 2

Si w est un mot, w^R désigne le mot miroir de w. On note Arith l'ensemble des expressions arithmétiques valides. Construire une grammaire non contextuelle qui génère les langages suivants.

- $\mathcal{L}_1 = \{w \in \{a,b,c\}^* \wedge w = w^R\}$
- $\mathcal{L}_2 = \{w \in \{a,b,c\}^* \wedge w = xx^R\}$
- $\mathcal{L}_3 = \{w \in \{a,b,c\}^* \wedge w \in \mathcal{L}(a^*b^*c^*)\}$
- $\mathcal{L}_4 = \{w \in \{a,b,c\}^* \wedge w = a\Sigma^* a\}$
- $\mathcal{L}_5 = \{w \in \{a,b,c\}^* \wedge w$ débute et finit par le même symbole$\}$
- $\mathcal{L}_6 = \{w = a^i b^j c^k \wedge (i,j,k) \in \mathbb{N}^3 \wedge (i \neq j \vee j \neq k)\}$
- $\mathcal{L}_6 = \{w \in \{a,b\}^* \wedge \mid w \mid_a \geqslant \mid w \mid_b\}$
- $\mathcal{L}_6 = \{w \in \{(,),0,1,+,\times\}^* \wedge w \in$ Arith$\}$

Exercice 3

Soit $\mathcal{G} = (\mathcal{V}_N, \mathcal{V}_T, \mathcal{R}, S)$ avec $\mathcal{V} = \{S\}$, $\Sigma = \{a,b\}$ et $\mathcal{R} = \{S \to aSb \mid SS \mid \varepsilon\}$. Déterminer les premiers mots engendrés par cette grammaire. Quel langage est généré par cette grammaire ? Remplace a par (et b par) peut aider !

Exercice 4

Si $\Sigma = \{a,b,c\}$ et $e = (ab)^*(ca)^*$, construire une grammaire non contextuelle pour le langage $\mathcal{L}(e) \cup \{a^k b^k, k \in \mathbb{N}\}$.

Exercice 5

Si $\Sigma = \{a,b\}$, construire une grammaire non contextuelle qui génère le complément de $\{a^k b^k, k \in \mathbb{N}\}$.

Exercice 6

Montrer que les langages non contextuels sont clos par union, concaténation et étoile de Kleene.

Exercice 7

Si $\Sigma = \{$`[`, `]`, `;`, `1`$\}$, donner une grammaire reconnaissant les listes OCaml de `1`. Par exemple, `[]`, `[1]`, `[1;1]` sont des mots reconnus par cette grammaire.

Exercice 8

Le langage de Dyck est l'ensemble D des mots bien parenthésés sur $\Sigma = (,)$. Formellement, pour tout mot v du langage, le nombre de (et le nombre de) dans v sont égaux et pour tout préfixe u de v, le nombre de (est supérieur au nombre de).

Question 1. Montrer que le langage de Dyck n'est pas régulier.

Question 2. Proposer une grammaire reconnaissant le langage.

Question 3. Écrire un programme OCaml qui vérifie qu'une chaîne est un mot du langage de Dyck étendu, défini comme les mots bien parenthésés sur $\Sigma = \{[,], \{,\}, (,)\}$. On demande une fonction directe, pas une fonction simulant la grammaire de la question précédente.

Exercice 9

Proposer une CFG pour chacun des langages suivants. Cet exercice a déjà été recontré sous une autre forme. Il est conseillé de le chercher sans regarder la solution !

Question 1. $\mathcal{L}_0 = \{a^i b^j c^k \mid i \neq j \vee j \neq k\}$

Question 2. $\mathcal{L}_1 = \{w \in \{a,b\}^* \mid \ |w|_a \geqslant |w|_b\}$

Question 3. $\mathcal{L}_2 = \{w \in \{(,),0,1,+,*\}^* \mid w \text{ est une expression arithmétique valide}\}$

Exercice 10

Une expression arithmétique valide peut être *parsée*[1] de différentes manières. Proposer une CFG non ambiguë dans laquelle les arbres de dérivation[2] donnent la priorité à `*` par rapport à `+`. Par exemple, `2 + 3 * 4` est vue comme l'expression qui vaut `14` et non `20`.

Exercice 11

Prouver que les grammaires définies par les symboles S et B reconnaîssent le même langage mais que l'une est ambiguë et pas l'autre.

$$B \to (RB \mid \varepsilon \qquad\qquad S \to SS \mid (S) \mid \varepsilon$$
$$R \to) \mid (RR$$

Exercice 12

On considère la CFG suivante sur l'alphabet $\Sigma = \{x, y, +, -, *\}$.

$$E \to +EE \mid *EE \mid -EE \mid x \mid y$$

Question 1. Déterminer les dérivations gauches (*leftmost*) et droites (*rightmost*) ainsi qu'un arbre de dérivation pour le mot $+*-xyxy$.

Question 2. Prouver que cette grammaire est non-ambiguë.

Exercice 13

Toute CFG admet une grammaire équivalente[3] sous *forme normale de Chomsky*[4] et réciproquement.

Question 1. Lire l'article joint de Richard Cole pour comprendre la procédure de passage d'une CFG à une CNF. En faire une synthèse simple qui présente les règles de transformation.

Question 2. Transformer les règles de productions suivantes en CNF. S n'apparaît pas dans ces règles.

(i) $A \to bC$
(ii) $A \to Bc$
(iii) $A \to bc$
(iv) $A \to BCD$
(v) $A \to a_1 a_2 a_3$ avec $a_i \in \Sigma \cup \mathcal{V}$
(vi) $A \to a_1 \dots a_p$ avec $p \geqslant 3$
(vii) $A \to B$

Question 3. Proposer une CNF équivalente à la grammaire suivante.

$$S \to CSC \mid aB \qquad C \to B \mid S \qquad B \to b \mid \varepsilon$$

Exercice 14

Soit la CFG $\mathcal{G} = (\mathcal{V}_N, \mathcal{V}_T, \mathcal{R}, S)$ définie par les règles de production suivantes.

$$\mathcal{R} = \{S \to \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1\}$$

Montrer que le langage $\mathcal{L}(\mathcal{G})$ est l'ensemble des palindromes sur $\{0,1\}$.

1. Anglicisme qui peut être traduit en *analysée*.
2. On lit également *arbres de parsing*.
3. C'est-à-dire engendrant le même langage.
4. Notée CNF pour *Chomsky Normal Form*.

Exercice 15

Une CFG est dite *linéaire à droite* si chaque règle de production comporte au plus un *symbole non terminal* placé à droite du mot de $(\Sigma \cup \mathcal{V})^*$. Cela signifie que toute règle est de la forme $A \to wB$ ou $A \to w$ où A et B sont des symboles non terminaux, w est une suite de symboles terminaux ou le mot vide.

Question 1. Montrer que toute grammaire linéaire à droite engendre un langage régulier.
Indication. Construire un ε-NFA qui simule les dérivations à gauche.

Question 2. Montrer que tout langage régulier admet une grammaire linéaire à droite.
Indication. Partir d'un DFA et définir les symboles non terminaux à partir de ses états.

Exercice 16

On étudie un algorithme d'analyse syntaxique, appelé *analyse descendante*, qui s'applique à certaines grammaires non contextuelles. Pour une grammaire donnée, on note N l'ensemble de ses non terminaux (notés avec des majuscules) et T l'ensemble de ses terminaux (notés avec des minuscules). Les lettres grecques (α, β, γ, etc.) désignent des mots de $(N \cup T)^*$. Le mot vide est noté ε. Une règle de production de la grammaire est notée $X \to \gamma$. Une dérivation immédiate est notée $\alpha \Rightarrow \beta$, ce qui signifie qu'il existe une règle de production $X \to \gamma$ avec $\alpha = \alpha_1 X \alpha_2$ et $\beta = \alpha_1 \gamma \alpha_2$. On note $\Rightarrow^*$ la clôture réflexive transitive de la relation $\Rightarrow$, c'est-à- dire une dérivation en un nombre quelconque d'étapes, y compris zéro.

On prend en exemple une version simplifiée de la grammaire du langage de programmation LISP, avec un ensemble de quatre terminaux $T = \{\texttt{sym}, \texttt{(}, \texttt{)}, \texttt{\#}\}$, un ensemble de trois non terminaux $N = \{S, L, E\}$, dont le symbole initial S, et les cinq règles de production suivantes.

$$S \to L\texttt{\#} \qquad L \to \varepsilon \mid EL \qquad E \to \texttt{sym} \mid (L)$$

Appelons $\mathcal{G}$ cette grammaire.

Question 1. Pour un entier $n \in \mathbb{N}$ arbitraire, donner un mot de longueur au moins n engendré par cette grammaire et la dérivation à gauche correspondante. Pour réaliser l'analyse syntaxique de ce petit langage avec OCaml, on se donne un type `token` pour les symboles terminaux `sym`, `(`, `)` et `#` :

```
type token = Sym | Lpar | Rpar | Eof
```

Ainsi `Sym` représente le terminal `sym`, `Lpar` le terminal `(`, `Rpar` le terminal `)` et `Eof` le terminal `#`.

Notre objectif est d'écrire une fonction `accepts: token list -> bool` qui détermine si un mot, donné comme une liste de terminaux, appartient ou non au langage de cette grammaire. Pour cela, on va commencer par construire trois fonctions, une pour chaque non terminal de la grammaire, avec les types suivants.

```
val parseS: token list -> token list
val parseL: token list -> token list
val parseE: token list -> token list
```

La fonction `parseX` reconnaît un préfixe maximal de la liste passée en argument comme étant un mot dérivé de X et renvoie le reste de la liste. S'il n'y a pas de tel préfixe, alors la fonction lève l'exception `SyntaxError`, que l'on suppose définie.

Pour définir ces trois fonctions, on se donne une table à deux entrées appelée *table LL*. Cette table indique, pour un non terminal que l'on cherche à reconnaître et pour un terminal au début de la liste, la règle de production à utiliser. Voici cette table pour notre grammaire.

	`sym`	`(`	`)`	`#`
S	L#	L#		L#
L	EL	EL	ε	ε
E	`sym`	(L)		

La fonction `parseS` est donc définie en suivant la première ligne de cette table. Son code est donné ci-dessous.

```
let rec parseS l = match l with
  | (Sym | Lpar | Eof) :: _ ->
      (match parseL l with Eof :: q -> q | _ -> raise SyntaxError)
  | [] | Rpar :: _ -> raise SyntaxError
```

En particulier, une case vide dans la table est interprétée comme un échec de l'analyse.

Question 2. Donner le code des fonctions `parseL` et `parseE`.

Question 3. Donner le code de la fonction `accepts`.

Construction de la table. On va maintenant chercher à construire une telle table pour une grammaire arbitraire. On introduit pour cela une première notion : on dit qu'un non terminal X est nul, et on note $\text{Nul}(X)$, si le mot vide peut être dérivé de X, c'est-à-dire $X \Rightarrow^* \varepsilon$.

Question 4. Indiquer quels sont les symboles nuls de la grammaire $\mathcal{G}$ prise en exemple plus haut.

Calcul des symboles nuls. Pour déterminer $\text{Nul}(X)$, on propose l'algorithme suivant.

1. Initialement, on fixe $\text{Nul}(X)$ à `false` pour tout $X \in \mathbb{N}$.
2. Pour chaque non terminal X, on affecte la valeur true à $\text{Nul}(X)$ s'il existe une production $X \to \varepsilon$ ou une production $X \to X_1 X_2 \dots X_p$ avec X_i des non terminaux et $\text{Nul}(X_i)$ pour tout i.
3. Si l'étape 2 a modifié au moins une valeur $\text{Nul}(X)$ alors on recommence l'étape 2.

Question 5. Montrer que cet algorithme termine.

Question 6. Montrer que cet algorithme détermine bien la valeur de $\text{Nul}(X)$.

Les premiers et les suivants. On introduit deux autres notions sur la grammaire. Pour un non terminal $X \in \mathbb{N}$, on définit deux ensembles de terminaux.

- $\text{Premiers}(X)$ est l'ensemble des terminaux qui peuvent apparaître au début des mots dérivés depuis X.

$$\text{Premiers}(X) = \{t \in T \mid \exists \alpha \in (N \cup T)^* \quad X \Rightarrow^* t\alpha\}$$

- $\text{Suivants}(X)$ est l'ensemble des terminaux qui peuvent apparaître après X dans une dérivation.

$$\text{Suivants}(X) = \{t \in T \mid \exists \alpha, \beta \in (N \cup T)^* \quad S \Rightarrow^* \alpha X t \beta\}$$

Question 7. Donner les ensembles $\text{Premiers}(X)$ et $\text{Suivants}(X)$ pour la grammaire prise en exemple plus haut (soit six ensembles au total).

Construction de la table. On admet que l'on peut calculer les ensembles $\text{Premiers}(X)$ et $\text{Suivants}(X)$ pour toute grammaire. On étend Nul et Premiers sur tout mot de $(N \cup T)^*$ de la manière suivante.

$$\begin{aligned}
\text{Nul}(\varepsilon) &= \texttt{true} \\
\text{Nul}(X_1 X_2 \dots X_p) &= \texttt{true} \text{ si } \text{Nul}(X_i) = \texttt{true} \text{ pour tout } i \\
\text{Premiers}(\varepsilon) &= \varnothing \\
\text{Premiers}(t\alpha) &= \{t\} \\
\text{Premiers}(X\alpha) &= \text{Premiers}(\alpha) \text{ si } \text{Nul}(X) = \texttt{false} \\
\text{Premiers}(X\alpha) &= \text{Premiers}(X) \cup \text{Premiers}(\alpha) \text{ si } \text{Nul}(X) = \texttt{false}
\end{aligned}$$

où t est un terminal et X un non terminal.

On construit alors la table LL de la manière suivante : pour un non terminal $X \in N$ et un terminal $t \in T$, on indique la règle de production $X \to \gamma$ dans la case (X, t) de la table :

- si $t \in \text{Premiers}(\gamma)$,
- ou si $\text{Nul}(\gamma)$ et $t \in \text{Suivants}(X)$.

On peut alors se servir de cette table pour réaliser une analyse syntaxique dès lors que chaque case ne comporte qu'au plus une règle de production. *On pourra vérifier que l'on obtient bien la table donnée au début de cette partie avec les résultats obtenus aux questions précédentes.*

Un autre exemple. On considère une seconde grammaire $\mathcal{G}'$ sur les mêmes symboles terminaux, avec des non terminaux $\{S', L', E'\}$ et un symbole initial S'.

$$S' \to L'\texttt{\#} \qquad L' \to \varepsilon \mid L'E' \qquad E' \to \texttt{sym} \mid (L')$$

Question 8. Montrer que $\mathcal{G}'$ reconnaît le même langage que $\mathcal{G}$.

Question 9. Construire la table LL pour cette seconde grammaire. Permet-elle de coder un algorithme pour l'analyse syntaxique des mots générés par cette grammaire ?