TD 3 : Parenthèses, ambiguïté, itération, clôture

Exercice 1 (Langage de Dyck)**.**

1. Montrer que le langage reconnu par S avec les règles
$$S \to SS \mid \alpha \mid \varepsilon \ ,$$
où $\alpha \in V^*$ est une chaîne mêlant potentiellement des symboles terminaux et non terminaux, est $L(S) = L(\alpha^*)$, et ce quelles que soient les règles permettant des dérivations depuis α.
2. Montrer que la grammaire
$$S \to SS \mid (S) \mid \varepsilon$$
sur l'alphabet terminal $\Sigma = \{(,)\}$ génère le langage de Dyck D_1^* des mots congrus à ε par la congruence engendrée par $() \equiv \varepsilon$.
3. Comment construire une grammaire algébrique qui reconnaît L^* où L est un langage algébrique (clôture par étoile) ? En déduire une grammaire non ambiguë pour D_1^*.

Exercice 2 (Lemme d'Ogden)**.**

1. Montrer que le langage $\{a^n b^n c^m d^m \mid n, m \geq 0\}$ est algébrique mais pas linéaire. On pourra considérer le mot $a^n b^n c^n d^n$ avec tous les d distingués.
2. Montrer que le langage $\{a^n b^n c^m \mid n, m \geq 0\} \cup \{a^n b^m c^m \mid n, m \geq 0\}$ est linéaire et inhéremment ambigu. On pourra appliquer le lemme aux mots $a^{n+n!} b^n c^n$ puis $a^n b^n c^{n+n!}$ et exhiber deux arbres de dérivations différents pour le mot $a^{n+n!} b^{n+n!} c^{n+n!}$.

Exercice 3 (Langages de programmation)**.**

1. La première utilisation des grammaires algébriques pour décrire la syntaxe d'un langage de programmation a été faite pour le langage Algol 60. Néanmoins, la modélisation par une grammaire algébrique seule n'est pas possible : le programme suivant

```
begin
  real x;
  y := z
end
```

n'est *sémantiquement* correct que si les identifiants `x`, `y` et `z` coïncident. Cependant un identifiant Algol 60 est une chaîne arbitrairement longue du langage rationnel $l(l+c)^*$ où l désigne n'importe quelle lettre majuscule ou minuscule de l'alphabet latin, et c n'importe quel chiffre de 0 à 9.

Montrer que ce niveau de correction ne peut pas être assuré par une grammaire algébrique seule.

2. Voici un extrait de la grammaire d'ALGOL 60 :

$$
\begin{aligned}
\langle \textit{statement} \rangle &\rightarrow \langle \textit{unconditional statement} \rangle \\
&\mid \langle \textit{conditional statement} \rangle \\
\langle \textit{unconditional statement} \rangle &\rightarrow \langle \textit{for statement} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \textit{conditional statement} \rangle &\rightarrow \langle \textit{if statement} \rangle \\
&\mid \langle \textit{if statement} \rangle \ \texttt{else} \ \langle \textit{statement} \rangle \\
\langle \textit{if statement} \rangle &\rightarrow \langle \textit{if clause} \rangle \ \langle \textit{unconditional statement} \rangle \\
\langle \textit{if clause} \rangle &\rightarrow \texttt{if} \ \langle \textit{boolean expression} \rangle \ \texttt{then}
\end{aligned}
$$

Comme vous pouvez l'observer, les fondateurs d'ALGOL 60 ont pris des précautions pour éviter l'ambiguïté de la construction `if/then/else`, en interdisant totalement d'emboîter des instructions `if/then` ! Comment pourrait-on modifier cette grammaire (et du coup le langage généré) pour permettre la dérivation de la phrase suivante sans introduire d'ambiguïté ?

```
if a > 1 then if b > 2 then c := 0 else c := 1
```

3. La grammaire d'ALGOL 60 comportait aussi les règles

$$
\begin{aligned}
\langle \textit{for statement} \rangle &\rightarrow \langle \textit{for clause} \rangle \ \langle \textit{statement} \rangle \\
\langle \textit{for clause} \rangle &\rightarrow \texttt{for} \ \langle \textit{variable} \rangle := \langle \textit{for list} \rangle \ \texttt{do} \\
\langle \textit{for list} \rangle &\rightarrow \langle \textit{for list element} \rangle \\
&\mid \langle \textit{for list} \rangle \ , \langle \textit{for list element} \rangle \\
\langle \textit{for list element} \rangle &\rightarrow \langle \textit{arithmetic expression} \rangle \\
&\mid \langle \textit{arithmetic expression} \rangle \ \texttt{step} \ \langle \textit{arithmetic expression} \rangle \\
&\quad \texttt{until} \ \langle \textit{arithmetic expression} \rangle
\end{aligned}
$$

Montrer que l'extrait de grammaire constitué des deux fragments est ambigu. Proposer une grammaire *équivalente* non ambiguë.

4. La grammaire fournie pour la norme ANSI du langage C est (presque) non ambiguë. Cependant, elle *surgénère* : elle permet des constructions syntaxiques qui n'ont pas de sémantique associée. On s'intéresse ici au cas des noms de types, qui suivent la

grammaire :

$$
\begin{aligned}
\langle \textit{type name} \rangle &\rightarrow \langle \textit{specifier qualifier list} \rangle \\
&\mid \langle \textit{specifier qualifier list} \rangle \ \langle \textit{abstract declarator} \rangle \\
\langle \textit{abstract declarator} \rangle &\rightarrow \langle \textit{pointer} \rangle \\
&\mid \langle \textit{direct abstract declarator} \rangle \\
&\mid \langle \textit{pointer} \rangle \ \langle \textit{direct abstract declarator} \rangle \\
\langle \textit{direct abstract declarator} \rangle &\rightarrow (\ \langle \textit{abstract declarator} \rangle \ ) \\
&\mid [\,] \\
&\mid [\ \langle \textit{constant expression} \rangle \ ] \\
&\mid \langle \textit{direct abstract declarator} \rangle \ [\,] \\
&\mid \langle \textit{direct abstract declarator} \rangle \ [\ \langle \textit{constant expression} \rangle \ ] \\
&\mid (\,) \\
&\mid (\ \langle \textit{parameter type list} \rangle \ ) \\
&\mid \langle \textit{direct abstract declarator} \rangle \ (\,) \\
&\mid \langle \textit{direct abstract declarator} \rangle \ (\ \langle \textit{parameter type list} \rangle \ ) \\
\langle \textit{pointer} \rangle &\rightarrow * \\
&\mid * \ \langle \textit{pointer} \rangle
\end{aligned}
$$

Ainsi, `int` est reconnu comme un ⟨*specifier qualifier list*⟩ et donc comme un nom de type (le nom de type « entier »). Sont aussi reconnus

- `int *` : le nom de type « pointeur sur entier »,
- `int *[3]` : le nom de type « tableau de trois pointeurs sur des entiers »,
- `int *(char)` : le nom de type « pointeur sur une fonction prenant un caractère en paramètre et retournant un entier »,
- `int (*[3]) (void)` : le nom de type « tableau de de trois pointeurs sur des fonctions sans paramètres qui retournent des entiers »,
- `int (*(void))(char)` : le nom de type « fonction sans paramètre retournant un pointeur sur une fonction des caractères vers les entiers ».

Un nom de type C est utilisé comme argument pour la fonction `sizeof` et pour les déclarations de paramètres lors d'une déclaration de fonction.

Pouvez-vous donner un exemple reconnu comme un nom de type par cette grammaire, mais qui n'est pas un nom de type C ? Proposer un fragment de grammaire qui évite cet écueil.

Exercice 4 (Langage naturel)**.**

1. Considérez la phrase

   Marie regarde un homme avec un télescope.

   Faites-en une analyse (comme en école primaire...) qui la subdivise en constituants (groupe nominal, groupe verbal, groupe prépositionnel, etc.). En déduire une grammaire très simple du français qui permet de dériver cette phrase. Votre grammaire reflète-t'elle bien l'ambiguïté syntaxique de la phrase ?

2. On considère les constructions de l'anglais de la forme « X or no X ». Par exemple, la phrase

   > The North Koreans were developing nuclear weapons anyway, Iraq war or no Iraq war.

   est un emploi de cette construction, tandis que la phrase incorrecte

   > * The North Koreans were developing nuclear weapons anyway, Iraq war or no IAEA inspections.

   en montre un emploi incorrect. En supposant (1) que l'anglais puisse être vu comme un langage sur un alphabet de mots et (2) que « X or no X » soit réellement un phénomème syntaxique et non sémantique, comment pourrait-on développer un argument démontrant que l'anglais n'est pas un langage algébrique ?

Exercice 5 (Clôture par intersection avec un rationnel)**.**

1. Soit G une grammaire algébrique et A un automate fini. Contruire G' algébrique telle que $L(G') = L(G) \cap L(A)$.
2. Modifier la construction pour calculer *au vol* si $L(G') = \emptyset$.