Partie II. Analyse descendante

Ce problème utilise le langage OCaml.

Dans ce problème, 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 $(N \cup T)^\star$. 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 $\Rightarrow^\star$ 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 :

$$\begin{array}{rcl} S & \to & L\ \texttt{\#} \\ L & \to & \varepsilon \\ & \mid & E\ L \\ E & \to & \texttt{sym} \\ & \mid & \texttt{(}\ L\ \texttt{)} \end{array}$$

Appelons G cette grammaire.

Question 11. 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.

Correction : Il suffit de considérer le mot

$$\underbrace{\texttt{sym sym ... sym}}_{n}\ \texttt{\#}$$

Sa dérivation à gauche est

$$\begin{array}{rcl} S & \Rightarrow & L\ \texttt{\#} \\ & \Rightarrow & E\ L\ \texttt{\#} \\ & \Rightarrow & \texttt{sym}\ L\ \texttt{\#} \\ & \Rightarrow & \texttt{sym}\ E\ L\ \texttt{\#} \\ & \Rightarrow & \texttt{sym sym}\ L\ \texttt{\#} \\ & \vdots & \\ & \Rightarrow & \texttt{sym ... sym}\ L\ \texttt{\#} \\ & \Rightarrow & \texttt{sym ... sym \#} \end{array}$$

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 `parse`X 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 et voici son code :

```
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 12. Donner le code des fonctions `parseL` et `parseE`.

Correction :

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

```
| Lpar :: q ->
    (match parseL q with Rpar :: r -> r | _ -> raise SyntaxError)
| [] | (Rpar | Eof) :: _ ->
    raise SyntaxError
```

Question 13. Donner le code de la fonction `accepts`.

Correction : Il faut rattraper l'exception `SyntaxError`, mais il faut également vérifier que tous les terminaux ont bien été consommés.

```
let accepts l =
  try parseS l = [] with SyntaxError -> false
```

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 $\textsc{Nul}(X)$, si le mot vide peut être dérivé de X, c'est-à-dire $X \Rightarrow^\star \varepsilon$.

Question 14. Indiquer quels sont les symboles nuls de la grammaire G prise en exemple plus haut.

Correction : Le symbole L est trivialement nul, car on a la règle de production $L \to \varepsilon$.

Les symbole S et E ne sont pas nuls, car tout mot dérivé de S contient au moins le terminal `#`, et de même tout mot dérivé de E contient soit le terminal `sym`, soit le terminal `(`.

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

1. Initialement, on fixe $\textsc{Nul}(X)$ à `false` pour tout $X \in N$.
2. Pour chaque non terminal X, on affecte la valeur `true` à $\textsc{Nul}(X)$ 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 $\textsc{Nul}(X_i)$ pour tout i.
3. Si l'étape 2 a modifié au moins une valeur $\textsc{Nul}(X)$, alors on recommence l'étape 2.

Question 15. Montrer que cet algorithme termine.

Correction : La valeur de $\textsc{Nul}(X)$ n'évolue que dans le sens `false` vers `true`. Dès lors, le nombre de valeurs `false` ne fait que diminuer. S'il ne change pas, l'algorithme s'arrête. Sinon, il diminue strictement et constitue donc un variant de l'algorithme.

En particulier, le nombre d'étapes est borné par le nombre de non terminaux de la grammaire.

Question 16. Montrer que cet algorithme détermine bien la valeur de $\textsc{Nul}(X)$.

Correction : D'une part, on montre par récurrence sur le nombre d'étapes de l'algorithme que, si on obtient $\textsc{Nul}(X) = \texttt{true}$, alors effectivement $X \Rightarrow^\star \varepsilon$. C'est clair initialement, car $\textsc{Nul}(X) = \texttt{false}$ pour tout X. Et si $\textsc{Nul}(X)$ devient `true`, alors soit $X \to \varepsilon$, soit $X \to X_1 X_2 \dots X_p$ avec $X_i \Rightarrow^\star \varepsilon$ par hypothèse de récurrence.

D'autre part, on montre par récurrence sur le nombre d'étapes de la dérivation $X \Rightarrow^\star \varepsilon$ qu'on obtient bien $\textsc{Nul}(X) = \texttt{true}$ par l'algorithme. Si $X \Rightarrow \varepsilon$, c'est qu'il existe une production $X \to \varepsilon$ et on aura bien $\textsc{Nul}(X) = \texttt{true}$ par l'algorithme. Sinon, $X \Rightarrow X_1 X_2 \dots X_p$ avec $X_i \Rightarrow^\star \varepsilon$ et par hypothèse de récurrence, on aura $\textsc{Nul}(X_i) = \texttt{true}$ par l'algorithme, pour tout i, et donc $\textsc{Nul}(X) = \texttt{true}$.

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

- $\textsc{Premiers}(X)$ est l'ensemble des terminaux qui peuvent apparaître au début des mots dérivés depuis X, c'est-à-dire $\textsc{Premiers}(X) = \{t \in T \mid \exists \alpha \in (N \cup T)^\star.\, X \Rightarrow^\star t\alpha\}$;
- $\textsc{Suivants}(X)$ est l'ensemble des terminaux qui peuvent apparaître après X dans une dérivation, c'est-à-dire $\textsc{Suivants}(X) = \{t \in T \mid \exists \alpha, \beta \in (N \cup T)^\star.\, S \Rightarrow^\star \alpha X t \beta\}$.

Question 17. Donner les ensembles $\textsc{Premiers}(X)$ et $\textsc{Suivants}(X)$ pour la grammaire prise en exemple plus haut (soit six ensembles au total).

Correction :

$$
\begin{aligned}
\textsc{Premiers}(S) &= \{\texttt{sym}, \texttt{(}, \#\} \\
\textsc{Premiers}(L) &= \{\texttt{sym}, \texttt{(}\} \\
\textsc{Premiers}(E) &= \{\texttt{sym}, \texttt{(}\} \\
\textsc{Suivants}(S) &= \{\} \\
\textsc{Suivants}(L) &= \{\texttt{)}, \#\} \\
\textsc{Suivants}(E) &= \{\texttt{sym}, \texttt{(}, \texttt{)}, \#\}
\end{aligned}
$$

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

la manière suivante :

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

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 $t \in \textsc{Premiers}(\gamma)$,
— ou si $\textsc{Nul}(\gamma)$ et $t \in \textsc{Suivants}(X)$.
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 14 et 17.*

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

$$\begin{aligned}
S' &\to L'\ \texttt{\#} \\
L' &\to \varepsilon \\
&\mid L'\ E' \\
E' &\to \texttt{sym} \\
&\mid \texttt{(}\ L'\ \texttt{)}
\end{aligned}$$

Question 18. Montrer que G' reconnaît le même langage que G.

Correction : On comprend que G reconnaît les listes par la gauche et G' par la droite, le reste étant identique. On va montrer la double inclusion des langages.

On commence par un lemme : si $L \Rightarrow^\star w \in T^\star \backslash \{\varepsilon\}$ alors $w = w'u$ avec $L \Rightarrow^\star w'$ et $E \Rightarrow^\star u$. On le prouve par récurrence sur la longueur de la dérivation. La dérivation commence nécessairement par $L \Rightarrow EL$ car $w \neq \varepsilon$. Ensuite, elle se poursuit avec $E \Rightarrow^\star w_1$ et $L \Rightarrow^\star w_2$ et $w = w_1 w_2$. Par hypothèse de récurrence, $w_2 = w_3 u$ avec $L \Rightarrow^\star w_3$ et $E \Rightarrow^\star u$. Il vient donc $L \Rightarrow EL \Rightarrow^\star w_1 w_3$ et on conclut en posant $w = w_1 w_3$.

Montrons maintenant que $X \Rightarrow^\star w$ implique $X' \Rightarrow^\star w$ pour chaque des trois non terminaux. On procède par récurrence sur la longueur de la dérivation $X \Rightarrow^\star w$.
— pour $X = S$: la dérivation est $S \Rightarrow L\texttt{\#}$ avec $L \Rightarrow^\star w'$ et $w = w'\texttt{\#}$. Par HR, $L' \Rightarrow^\star w'$ et donc $S' \Rightarrow L'\texttt{\#} \Rightarrow^\star w'\texttt{\#}$.

— pour $X = L$: Si $L \Rightarrow^\star \varepsilon$ alors $L' \Rightarrow^\star \varepsilon$. Sinon, on peut appliquer le lemme et $w = w'u$ avec $L \Rightarrow^\star w'$ et $E \Rightarrow^\star u$. Par HR, on a $L' \Rightarrow^\star w'$ et $E' \Rightarrow^\star u$, et donc $L' \Rightarrow L'E' \Rightarrow^\star w'u = w$.
— pour $X = E$, on a deux cas. Si $E \Rightarrow^\star \texttt{sym}$ alors $E' \Rightarrow \texttt{sym}$ également. Si $E \Rightarrow \texttt{(}L\texttt{)} \Rightarrow^\star \texttt{(}w'\texttt{)}$ alors par HR $L' \Rightarrow^\star w'$ et donc $E' \Rightarrow \texttt{(}L'\texttt{)} \Rightarrow^\star \texttt{(}w'\texttt{)} = w$.

L'autre inclusion se prouve de façon tout à fait similaire (avec un lemme similaire).

Question 19. Construire la table LL pour cette seconde grammaire. Permet-elle de coder un algorithme pour l'analyse syntaxique des mots générés par cette grammaire ?

Correction : Pour cette grammaire, on trouve

$$\begin{aligned}
\textsc{Premiers}(S') &= \{\texttt{sym}, \texttt{(}, \texttt{\#}\} \\
\textsc{Premiers}(L') &= \{\texttt{sym}, \texttt{(}\} \\
\textsc{Premiers}(E') &= \{\texttt{sym}, \texttt{(}\} \\
\textsc{Suivants}(S') &= \{\} \\
\textsc{Suivants}(L') &= \{\texttt{sym}, \texttt{(}, \texttt{)}, \texttt{\#}\} \\
\textsc{Suivants}(E') &= \{\texttt{sym}, \texttt{(}, \texttt{)}, \texttt{\#}\}
\end{aligned}$$

et on obtient du coup la table suivante :

	sym	(	)	#
S'	$L'\texttt{\#}$	$L'\texttt{\#}$		$L'\texttt{\#}$
L'	$\varepsilon / L'E'$	$\varepsilon / L'E'$	ε	ε
E'	sym	$\texttt{(}L\texttt{)}$		

Il y a deux cases dans la table où deux choix apparaissent, ce qui ne permet pas l'analyse.
