TD grammaires

Exercice 1 (Dérivation) Voici une grammaire pour les expressions arithmétiques :

$$
\begin{array}{rcl}
E & \to & M\,E' \\
E' & \to & +\,M\,E' \\
 & | & \varepsilon \\
M & \to & A\,M' \\
M' & \to & *\,A\,M' \\
 & | & \varepsilon \\
A & \to & (\,E\,) \\
 & | & \mathsf{n}
\end{array}
$$

Donner un arbre de dérivation pour l'entrée `2 * (2 * 3 + 1) * 3`. □

Exercice 2 (N-uplets) On souhaite proposer une grammaire pour représenter des n-uplets de la forme

```
(s, s, s, s)
```

Plus précisément : un n-uplet est délimité par des parenthèses, et contient un certain nombre de chaînes de caractères séparées par des virgules. Les chaînes seront simplement réprésentées par un unique symbole `s`.

1. Donner une grammaire décrivant de tels n-uplets, avec la règle supplémentaire qu'un n-uplet doit contenir au moins un élément, et des dérivations pour les phrases `(s)` et `(s, s, s)`.
2. Donner une variante de la grammaire précédente qui autorise le n-uplet vide, noté `()`.
3. Justifier qu'aucune de ces deux grammaires ne permet de dériver la phrase `(s, s, )`.

□

Exercice 3 (Langage de Dyck) On s'intéresse aux phrases « bien parenthésées » que l'on peut construire avec les deux paires de parenthèses `(/)` et `[/]`. On veut donc accepter les phrases

```
(([()]))
()[()]
[()[]]
```

mais rejeter les phrases

```
(()     // première parenthèse ( pas fermée
(()])   // première parenthèse ( fermée avec ]
[][]]   // dernière parenthèse ] sans ouvrante associée
```

Donner une grammaire caractérisant ces phrases bien parenthésées, et des dérivations pour les trois phrases valides données en exemple ci-dessus. □

Exercice 4 (Grammaires et associativité) Le cours proposait la grammaire G_0 suivante pour les expressions arithmétiques :

$$
\begin{array}{rcl}
E & \to & E + M \\
 & | & M \\
M & \to & M * A \\
 & | & A \\
A & \to & (\,E\,) \\
 & | & \mathsf{n}
\end{array}
$$

En voici plusieurs variantes :

Grammaire G_1

$$
\begin{array}{rcl}
E & \to & E + E \\
 & | & M \\
M & \to & M * M \\
 & | & A \\
A & \to & (\,E\,) \\
 & | & \mathsf{n}
\end{array}
$$

Grammaire G_2

$$
\begin{array}{rcl}
E & \to & M + M \\
 & | & M \\
M & \to & A * A \\
 & | & A \\
A & \to & (\,E\,) \\
 & | & \mathsf{n}
\end{array}
$$

Grammaire G_3

$$
\begin{array}{rcl}
E & \to & M + E \\
 & | & M \\
M & \to & A * M \\
 & | & A \\
A & \to & (\,E\,) \\
 & | & \mathsf{n}
\end{array}
$$

Grammaire G_4

$$
\begin{array}{rcl}
E & \to & E + M \\
 & | & M \\
M & \to & M * A \\
 & | & A * A \\
 & | & \mathsf{n} \\
A & \to & (\,E + M\,) \\
 & | & \mathsf{n}
\end{array}
$$

1. Deux de ces grammaires ne décrivent pas exactement le même langage que G_0. Lesquelles ? Quel langage décrivent-elles ?
2. Donner tous les arbres de dérivation possibles pour l'entrée `2 * 3 * 7` pour G_0 et les deux autres grammaires décrivant les expressions arithmétiques. En quoi l'approche de ces trois grammaires est-elle différente ?
3. Étendre ces grammaires pour qu'elles reconnaissent également les soustractions, si cela est possible.

□

Exercice 5 (Langage Inoxydable Simplement Parenthésé) Les règles suivantes définissent la grammaire du langage de programmation LISP.

$$
\begin{array}{lcl}
E & ::= & A \\
 & | & \texttt{(}\ L\ \texttt{)} \\
L & ::= & \varepsilon \\
 & | & E\,L \\
A & ::= & \texttt{sym} \mid \texttt{num} \mid \texttt{+} \mid \texttt{*}
\end{array}
$$

On a trois symboles non terminaux E (une expression, symbole de départ), L (une liste), et A (un atome). Les symboles terminaux sont les parenthèses `(` et `)`, un symbole `sym` désignant une séquence de caractères alphabétiques, un symbole `num` désignant un nombre, et quelques symboles d'opérateurs (on retient ici : `+` et `*`).

1. Donner une dérivation pour la phrase

```
(defun double (x) (* 2 x))
```

2. Justifier que la phrase

```
(let x 3
```

n'est pas dérivable.

3. Définir un type caml `token` pour les symboles terminaux de cette grammaire, et une fonction `tokenize: string -> token list` qui convertit une chaîne de caractères en une liste de symboles.
4. Définir une fonction `validate: string -> bool` qui prend en paramètre une chaîne de caractère et renvoie `true` si et seulement si cette chaîne représente un programme LISP bien formé. *Indication : dans un premier temps, on recommande d'opter pour la stratégie « fonctionnelle », avec des fonctions récursives* `validate_expr` *et* `validate_instr` *qui prennent en entrée une liste de mots, et qui renvoient la liste des mots qui n'ont pas encore été consommés.*
5. On propose le type suivant pour représenter les arbres syntaxes des expressions LISP.

```
type expr =
  | Num of int
  | Sym of string
  | Add
  | Mul
  | List of expr list
```

Définir une fonction `parse: s -> expr` qui prend en paramètre une chaîne de caractères supposée décrire une expression LISP et qui renvoie son arbre de syntaxe abstraite, ou échoue si l'expression source est mal formée. *Indication : dans un premier temps, on recommande comme ci-dessus la version fonctionnelle. Mais une fois que tout fonctionne, vous pouvez tenter une autre version !*

□