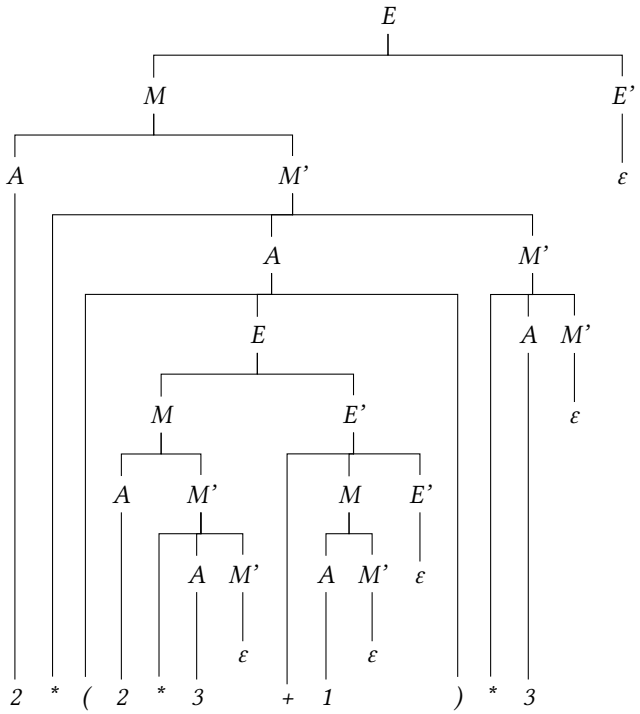
TD grammaires

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.
Correction :



□

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

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

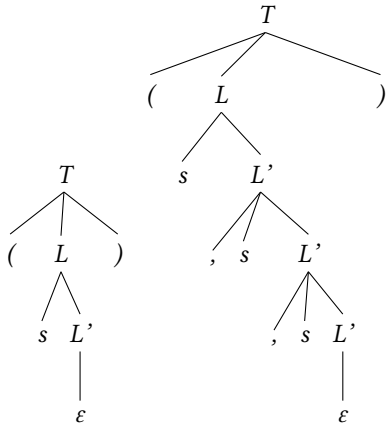
Plus précisément : un n-uplet est délimité par des parenthèses, et contient un certain nombre de chaînes de caractères séparées par des virgules. Les chaînes seront simplement réprésentées par un unique symbole `s`.

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, )`.

Correction :

1.

$$
\begin{array}{rcl}
T & ::= & \texttt{(}\ L\ \texttt{)} \\
L & ::= & \texttt{s}\ L' \\
L' & ::= & \varepsilon \\
 & | & \texttt{,}\ \texttt{s}\ L'
\end{array}
$$



2.

$$\begin{array}{rcl} T & ::= & (\ ) \\ & | & (\ L\ ) \end{array}$$

3. Une seule règle de la grammaire permet d'introduire le symbole , (virgule), et cette règle introduit nécessairement un symbole s *immédiatement après. On ne peut donc pas avoir une virgule suivie immédiatement par une parenthèse.*

□

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

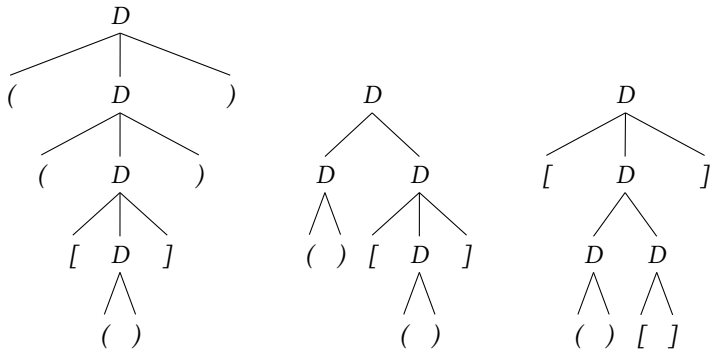
```
(()     // première parenthèse ( pas fermée
(()])   // première parenthèse ( fermée avec ]
[][]]   // dernière parenthèse ] sans ouvrante associée
```

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

Correction : Grammaire avec un unique non-terminal D, version qui n'autorise pas la séquence vide.

$$\begin{array}{rcl} D & ::= & () \\ & | & [] \\ & | & (D) \\ & | & [D] \\ & | & DD \end{array}$$

Trois dérivations.



□

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

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

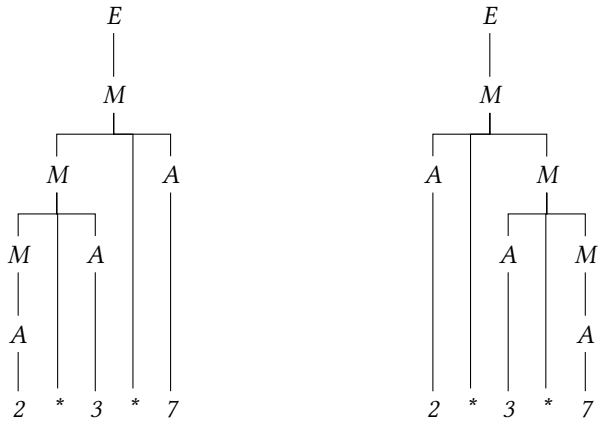
Grammaire G_4

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

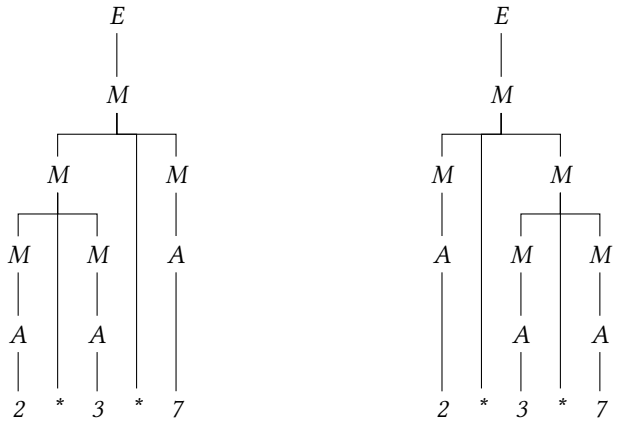
1. Deux de ces grammaires ne décrivent pas exactement le même langage que G_0. Lesquelles ? Quel langage décrivent-elles ?
2. Donner tous les arbres de dérivation possibles pour l'entrée 2 * 3 * 7 pour G_0 et les deux autres grammaires décrivant les expressions arithmétiques. En quoi l'approche de ces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

Correction :

1. *Les deux grammaires décrivant un langage différent sont G_2 et G_4 :*
   - *G_2 force l'utilisation de parenthèses à chaque enchaînement de deux additions ou multiplications. Elle accepte par exemple (1+2)+3 mais pas 1+2+3.*
   - *G_4 interdit les parenthèses superflues. Elle accepte par exemple 1+2+3 mais pas (1+2)+3.*
2. *Pour G_0 et G_3, on a à chaque fois un seul arbre de dérivation possible. Ci-dessous, l'arbre pour G_0 à gauche, et celui pour G_3 à droite.*



Pour G_1 en revanche, la grammaire permet les deux arbres de dérivation suivants.



Interprétation :

- *G_0 impose l'associativité à gauche : 2*3*7 est comprise comme (2*3)*7.*
- *G_3 impose l'associativité à droite : 2*3*7 est comprise comme 2*(3*7).*
- *G_1 est ambiguë et autorise les deux interprétations précédentes.*

3. *On peut ajouter une règle $E \to E$ - M dans G_0 ou G_1. Cela ne fonctionne en revanche pas pour G_3 (on n'admettrait pas l'entrée 1-2+3). La règle $E \to E$ - E ne convient pas, car autoriserait de mauvaises associativités (1-2+3 pouvant alors être comprise comme 1-(2+3)).*

□

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

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

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.

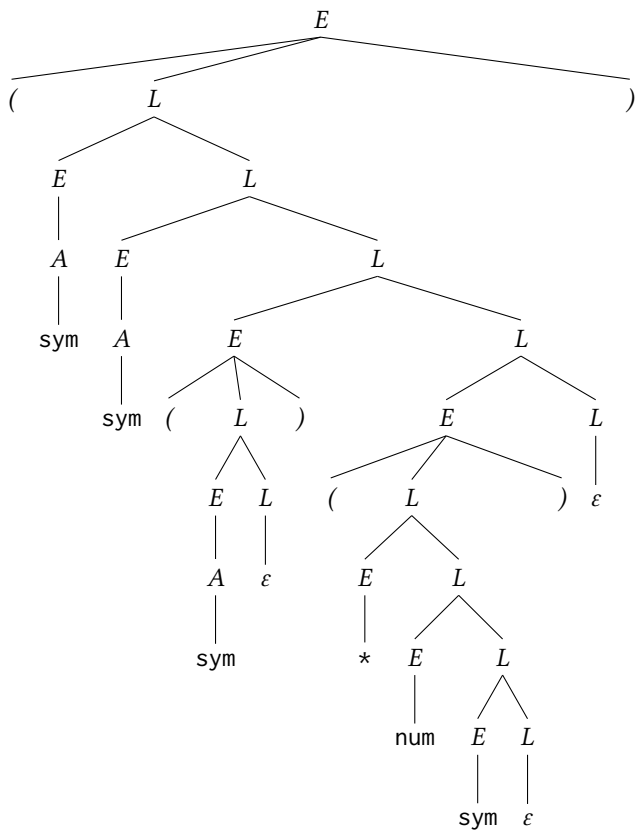
3. Définir un type caml `token` pour les symboles terminaux de cette grammaire, et une fonction `tokenize: string -> token list` qui convertit une chaîne de caractères en une liste de symboles.
4. Définir une fonction `validate: string -> bool` qui prend en paramètre une chaîne de caractère et renvoie `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 !*

Correction :

1.



2. La seule règle introduisant une parenthèse ouvrant introduit également une parenthèse fermante : il n'est pas possible de dériver une phrase avec moins de fermantes que d'ouvrantes.

3.

```
type token =
  | NUM of int
  | SYM of string
  | PLUS
```

```
  | STAR
  | LPAR
  | RPAR

let tokenize (s: string): token list =
  let rec loop i = match s.[i] with
    | '+' -> PLUS :: loop (i+1)
    | '*' -> STAR :: loop (i+1)
    | '(' -> LPAR :: loop (i+1)
    | ')' -> RPAR :: loop (i+1)
    | ' ' -> loop (i+1)
    | '0'..'9' -> let n, i' = num 0 i in NUM n :: loop i'
    | 'a'..'z' -> let i' = sym i in SYM (String.sub s i (i'-i)) :: loop i'
    | _ -> failwith "unknown character"
    | exception e -> []
  and num n i = match s.[i] with
    | '0'..'9' as c -> num (10*n + Char.code c - Char.code '0') (i+1)
    | _ -> n, i
    | exception e -> n, i
  and sym i = match s.[i] with
    | 'a'..'z' -> sym (i+1)
    | _ -> i
    | exception e -> i
  in
  loop 0
```

```
let rec validate_expr = function
  | (PLUS | STAR | NUM _ | SYM _) :: l -> l
  | LPAR :: l -> (match validate_list l with
                  | RPAR :: l' -> l'
                  | _ -> failwith "syntax error")
  | _ -> failwith "syntax error"

and validate_list l = match l with
  | RPAR :: _ -> l
  | _ -> l |> validate_expr |> validate_list

let validate_prog s =
  s |> tokenize |> validate_expr = []

let _ = validate_prog "(defun double (x) (* 2 x))"
```

4.

```
let rec parse_expr = function
  | NUM n :: l -> Num n, l
  | SYM s :: l -> Sym s, l
  | PLUS :: l -> Add, l
  | STAR :: l -> Mul, l
  | LPAR :: l -> (match parse_list l with
                  | li, RPAR :: l' -> List li, l'
                  | _ -> failwith "syntax error")
  | _ -> failwith "syntax error"

and parse_list l = match l with
  | RPAR :: _ -> [], l
  | _ -> let e, l' = parse_expr l in
         let li, l'' = parse_list l' in
         e :: li, l''

let parse_prog s =
  let e, l = parse_expr (tokenize s) in
  if l = [] then e else failwith "syntax error"

let _ = parse_prog "(defun double (x) (* 2 x))"
```

□