TP8. Analyse lexicale et syntaxique en OCaml

Le but de ce TP est de construire une fonction `evalue : string -> int` prenant en entrée une chaîne de caractère représentant une expression arithmétique et renvoyant la valeur de cette expression. Ainsi, `evalue "((-75+9)* (7-9))"` renvoie `132`. Pour simplifier, une expression arithmétique ne pourra faire intervenir que des entiers, les opérateurs binaires `+, *, -, /` et l'opérateur unaire `-`.

Pour ce faire, on décompose le travail en trois étapes :

1. **L'analyse lexicale** qui consiste à découper la chaîne initiale en une liste de *lexèmes* (en anglais *tokens*).
2. **L'analyse syntaxique**, qui à partir de la liste de lexème précédemment obtenue, construit un arbre syntaxique permettant de savoir si la suite de lexèmes est bien formée.
3. **L'évaluation** permettra finalement d'associer une valeur à un arbre syntaxique.

I. L'analyse lexicale

Dans ce TP, on appelle chaîne d'expression arithmétique toute chaîne ne faisant intervenir que les caractères `'(', ')', '+' , '*', '-', '/'`, les chiffres et le caractère espace. Par exemple `"45 (+)7-"` est une chaîne d'expression arithmétique mais pas `"45*8*x"`.
L'objectif de cette partie est de construire une fonction qui décompose une chaîne d'expression arithmétique en une liste de lexèmes en ignorant les espaces. Si ce n'est pas possible elle renverra l'exception `Lexical_error` (à définir).
Les lexèmes sont définis par le type suivant :

```
type token = CONST of int | EOF | LPAR | RPAR | PLUS | TIMES | DIV
   | MINUS
```

Le lexème `LPAR` désigne une parenthèse ouvrante, le lexème `RPAR` une parenthèse fermante et le lexème `EOF` la fin de la chaîne analysée.

1. Écrire une fonction `get_number : string -> int -> token * int`. Elle prend en entrée une chaîne `s` et un entier `i` tel que `s.[i]` existe et est un chiffre. Elle renvoie le couple `(CONST n, j)` où `n` est le nombre commençant à la position `i` dans `s`, et `j` est l'indice du dernier chiffre de `n`.

2. Écrire une fonction `first_token : string -> int -> token * int` prenant en entrée une chaîne `s` et un indice `i` dans cette chaîne et renvoyant le premier lexème reconnu à partir de l'indice `i` dans `s` et l'indice `j` de `s` correspondant à l'indice du dernier caractère formant ledit lexème. Si aucun lexème ne peut être reconnu à partir de l'indice `i`, on lèvera l'exception `Lexical_error`.

3. En déduire une fonction `lexer : string -> token list` transformant une chaîne de caractères en la liste des lexèmes correspondante si la chaîne est une chaîne d'expression arithmétique et levant `Lexical_error` sinon.

II. L'analyse syntaxique

Maintenant que nous avons identifié la liste des lexèmes d'une chaîne d'expression arithmétique, on cherche à savoir si cette dernière est bien formée. Pour ce faire, on se dote de la grammaire G suivante, décrivant les formules arithmétiques syntaxiquement correctes (les variables sont S, E, B et O) :

$S \to E$ `EOF`
$E \to$ `CONST` n | `MINUS` E | `LPAR` B `RPAR`
$B \to E\ O\ E$
$O \to$ `PLUS` | `MINUS` | `TIMES` | `DIV`

L'objectif de cette partie est de construire une fonction qui associe à une liste de lexèmes provenant de l'analyse lexicale d'une chaîne d'expression arithmétique son arbre syntaxique selon cette grammaire si il existe et qui lève l'exception `Syntax_error` (à définir) sinon.

Un arbre syntaxique pour une expression arithmétique aura le type `ae` définit par :

```
type operation = Plus | Minus | Times | Div
type ae = Const of int | Bin of operation * ae * ae | Neg of ae
```

4. Écrire trois fonctions mutuellement récursives de signatures :

```
parser_E : token list -> ae * token list
parser_B : token list -> ae * token list
parser_O : token list -> operation * token list
```

La fonction `parser_E tl` renvoie l'arbre syntaxique du plus grand préfixe `p` de `tl` qui est un mot dérivé du non terminal E et le reste de la liste de lexèmes à analyser une fois que ceux utilisés pour former `p` ont été supprimés de `tl`. La fonction `parser_B` fait de même mais pour le non terminal B. La fonction `parser_O` fait de même sauf que le premier élément du couple renvoyé est un objet de type `operation`. Dans chaque cas, en cas d'impossibilité, `Syntax_error` sera levée.

5. En déduire une fonction `parser : token list -> ae` renvoyant l'arbre syntaxique associé à une liste de lexèmes si celle-ci forme un mot engendré par G et levant `Syntax_error` sinon.

III. L'évaluation

6. Écrire une fonction `evaluate : string -> int` permettant d'évaluer une chaîne à condition qu'il s'agisse d'une chaîne d'expression arithmétique correctement formée.