TP 4

Analyses lexicale et syntaxique

MPI/MPI*, lycée Faidherbe

Résumé

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

Cela se fait en 3 étapes

Analyse lexicale On découpe la chaîne initiale en une liste de *lexèmes* qui forment les symboles terminaux pour la grammaire G du langage étudié.

Analyse syntaxique À l'aide de cette grammaire, l'analyse syntaxique permet d'associer à une liste de lexèmes (donc à un mot de $L(G)$) son arbre syntaxique, unique si G est non ambiguë.

Évaluation On calcule la valeur associée à l'arbre.

I Analyse lexicale

On part d'une chaîne de caractères ne faisant intervenir que les caractères `'('`, `')'`, `'+'` , `'*'`, `'-'`, `'/'`, les chiffres et le caractère espace : une *chaîne d'expression arithmétique*.
Par exemple `"45 (+)7-"` est une chaîne d'expression arithmétique mais pas `"45*8*x"`.
On veut découper une chaîne d'expression arithmétique en lexèmes (token en anglais), chaque lexème étant constitué d'un ensemble de caractères formant une unité. C'est dans cette étape que les caractères 7 et 5 accolés seront reconnus comme formant l'unité lexicale 75.
Dans notre cas, les lexèmes seront donnés par le type suivant :

```
type token = CONST of int | EOF | PARG | PARD | PLUS | FOIS | DIV
   | MOINS
```

`PARG` et `PARG` désignent les parenthèses ouvrantes et fermantes.
et `EOF` désigne la fin de la chaîne analysée.
À la fin de cette étape, on disposera d'une fonction `lexer : string -> token list` telle que `lexer "((-75+9) * (7- 9))"` renvoie la liste de lexèmes :

```
[PARG; PARG; MOINS; CONST 75; PLUS; CONST 9; PARD; FOIS; PARG;
   CONST 7; MOINS; CONST 9; PARD; PARD; EOF]
```

Si la décomposition n'est pas possible la fonction lèvera une l'exception

```
exception Lexical_error
```

Une telle fonction `lexer` s'appelle un *analyseur lexical* (ou *lexeur*).

Question 1 Écrire une fonction `get_number : string -> int -> token*int`. Elle prend en entrée une chaîne `s` et un entier `i` tel que la lettre à l'indice `i` dans `s` 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 l'indice du dernier chiffre de `n` est `j`.

Question 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`.

Question 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.

N'oubliez pas de tester vos fonctions.

II Analyse syntaxique

Les lexèmes utilisés sont les symboles terminaux, l'ensemble Σ, pour la grammaire $G = (\Sigma, V, P, S)$ engendrant le langage des expressions arithmétiques correctement formées.
L'ensemble des variables est $V = \{S, E, B, O\}$ et les règles sont

- $S \to E\,\texttt{EOF}$,
- $E \to \texttt{CONST}\,n | \texttt{MOINS}\,E | \texttt{PARG}\,B\,\texttt{PARD}$,
- $B \to E\,O\,E$,
- $O \to \texttt{PLUS} | \texttt{MOINS} | \texttt{FOIS} | \texttt{DIV}$

On admet que cette grammaire est non ambiguë.
La règle $E \to \texttt{CONST}\,n$ signifie qu'il existe une règle permettant de dériver E en $\texttt{CONST}\,n$ pour chaque entier n. Comme il n'existe qu'un nombre fini d'entiers représentable sur machine, il y a un nombre fini de règles de ce type et la grammaire G a bien un nombre fini de règles.
On souhaite construire, non l'arbre de dérivation, mais l'arbre syntaxique associé ; il est plus simple, débarrassé des parenthèses et les variables sont remplacées par les opérations.
Son type est donné par

```
type op = Plus | Moins| Fois | Div
type arbreSyntaxe = Const of int
                  | Bin of arbreSyntaxe * op * arbreSyntaxe
                  | Neg of arbreSyntaxe
```

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.

Question 4 Écrire trois fonctions mutuellement récursives de signatures respectives

- `parserE : token list -> arbreSyntaxe * token list`
- `parserB : token list -> arbreSyntaxe * token list`
- `parserO : token list -> op * token list`

La fonction `parserE suite` renvoie l'arbre syntaxique du plus grand préfixe p de `suite` 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 `suite`.
La fonction `parserB` fait de même mais pour le non terminal `B`.
La fonction `parserO` 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.

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

Question 6 Conclure à l'évaluation d'un formule arithmétique : `evaluation : string -> int`.

III Bonus

Question 7 Modifier la grammaire et les fonctions pour permettre une lecture et évaluation des formules arithmétiques tenant compte de la priorité de la multiplication sur l'addition.

Pour l'écriture des formules booléennes, on propose l'utilisation des caractères suivants

Sémantique	$\wedge$	$\vee$	$\Rightarrow$	$\Leftrightarrow$	$\neg$	x_{42}	$\bot$	$\top$
Symboles	`&`	`|`	`->`	`<->`	`-`	`x42`	`F`	`V`

Question 8 Reprendre l'étude pour la lecture des formules booléennes.