TP8 - Analyseur lexical et syntaxique

Dans ce TP, les programmes sont écrits en OCaml.

L'objet de ce TP est de reconnaître qu'une chaîne de caractères OCaml contient une formule logique stricte et de construire une valeur du type permettant de représenter les formules.

Le programme ci-dessous donne le type des formules logiques. On appelle une telle structure de donnée un *arbre de syntaxe abstraite*. Par opposition aux arbres de dérivation syntaxique, l'arbre de syntaxe abstraite représente une forme idéale de l'entrée [1], où tous les éléments purement syntaxiques ont été supprimés [2]. Par exemple, notre type `fmla` ne contient pas de cas correspondant aux parenthèses. Le cas des variables est également simplifié : le caractère `x` n'est pas stocké, car il n'apporte aucune information ; seul le numéro de la variable permet d'identifier cette dernière.

Programme 1 – type des formules strictes

```
type binop = And | Or | Imp

type fmla =
  | False
  | True
  | Var of int
  | Not of fmla
  | Bin of binop * fmla * fmla
```

```
type token =
  | TRUE | FALSE
  | VAR of int
  | LPAR | RPAR
  | NOT
  | OR | AND | IMP
  | EOF
```

La conversion de la chaîne de caratères en objet de type `fmla` se fait en deux étapes : une *analyse lexicale* suivie d'une *analyse syntaxique*.

Analyse lexicale

Elle consiste à découper la chaîne de caractères (ou le fichier) en une liste de *lexèmes*, groupes de caractères qui forment une unité. Ces lexèmes [3] sont utilisés ensuite comme *symboles terminaux pour la grammaire du langage*. L'analyse lexicale peut donc se résumer à une fonction `lexer: string -> token list` où le type `token` est celui défini dans le programme 1. Pour notre langage très simple, on a quasiment une correspondance entre les éléments du type `fmla` et ceux du type `token`. Des différences subsistent cependant. Ainsi, le type `token` contient deux valeurs permettant de représenter les parenthèses ouvrantes ou fermantes ainsi qu'un lexème spécial `EOF` représentant la fin de l'entrée. Une dernière remarque est que c'est dans cette phase que les caractères *inutiles* sont reconnus et ignorés : commentaires, espaces, retours à la ligne, etc.

Question 1. Proposer une fonction `lexer` qui renvoie une liste de `token` après analyse lexicale d'une chaîne de caractères. Vous prendrez toutes les initiatives nécessaires pour définir les règles de construction de la liste. En particulier, vous avez l'entière responsabilité de la définition des symboles utilisés dans la chaîne de caractères pour caractériser chaque opérateur binaire, chaque variable logique, etc.

Analyse syntaxique

Étant donnés une grammaire pour notre langage et un mot constitué d'une suite de lexèmes, l'analyse syntaxique permet de construire son *arbre de syntaxe abstraite*. Elle est définie par la fonction `parser: token list -> fmla`. Pour reconnaître la structure d'une formule à partir de la liste de ses lexèmes, on adopte la grammaire suivante.

$$S \to F\ \texttt{EOF} \qquad F \to \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{VAR(n)} \mid \texttt{NOT}\ F \mid \texttt{LPAR}\ B\ \texttt{RPAR} \qquad B \to F\ O\ F \qquad O \to \texttt{AND} \mid \texttt{OR} \mid \texttt{IMP}$$

Question 2. Proposer une fonction `parseS` qui reçoit une liste de lexèmes et qui renvoie un objet de type `fmla`.

Exemple. Si `f_str = "( (x1 \\/ x2)-> (~ x3 \\/ x4))"` où `\\/`, `/\\`, `->`, `~` représentent respectivement le *ou*, le *et*, l'*implication* et la *négation* logiques, l'appel `lexer f_str` renvoie la liste de `token` suivante.

```
[LPAR; LPAR; VAR 1; OR; VAR 2; RPAR; IMP; LPAR; NOT; VAR 3; OR; VAR 4; RPAR; RPAR; EOF]
```

L'appel à `parser` sur cette liste renvoie alors la formule logique sous la forme suivante.

```
Bin (Imp, Bin (Or, Var 1, Var 2), Bin (Or, Not (Var 3), Var 4))
```

1. Formule, terme, programme.
2. Parenthèses, accolades, séparateurs comme la virgule ou le point-virgule, etc.
3. *Token* ou jetons en anglais.