TP : Analyseur syntaxique

On a vu en cours que le moyen le plus efficace, algorithmiquement parlant, est de représenter les formules avec le type `formule`, qu'on avait défini *par

```
type binop = And | Or | Impl | Equiv

type formule =
  |Var of char
  |Not of formule
  |BinOp of formule * binop * formule
```

Cependant, si cette représentation est utile pour faire toutes sortes de calculs, elle n'est pas du tout user-friendly : dès que la formule devient un peu complexe, c'est à la fois impossible de la rentrer et/ou de la lire.

Un utilisateur voudrait pouvoir taper les formules comme des chaînes de caractères, par exemple

`"~~(p v ~p)".`

Nous allons donc dans ce TP programmer un analyseur syntaxique ; son objectif est double :

- vérifier qu'un chaîne de caractère représente bien une formule bien formée
- transformer cette chaîne de caractère en formule correspondante, et inversement

On pourra ainsi écrire une formule en langage naturel, la transformer en formule, faire les opérations voulues, puis afficher le résultat sous forme lisible.

1. Écrire une fonction `charlist_of_string:string -> char list` qui transforme une chaîne de caractères en liste de caractères.

On va maintenant définir les *lexèmes*, c'est-à-dire les unités lexicales de notre langage : les caractères qui représentent les variables, les parenthèses, les connecteurs.

```
type lexeme =
    |LexVar of char (* Variables *)
    |LexLeftPar (* Parenthèse ouvrante *)
    |LexRightPar (* Parenthèse fermante *)
    |LexNot (* Négation *)
    |LexBin of binop (* Opérateur binaire *)
    |LexAutre of char (* Cas d'un caractère inconnu *)
```

2. Écrire une fonction `lexemelist_of_charlist:char list -> lexeme list`.

Ceci achève l'analyse lexicale : on a transformé notre chaîne de caractères en liste de lexèmes. Reste à vérifier que la formule est valide : c'est la phase d'analyse syntaxique.

Pour cela, on va, étant donné une liste de lexème `s`, extraire le plus grand préfixe qui est une formule valide, et renvoie la formule associée à ce préfixe et le reste de la formule.

3. Montrer qu'une formule valide ne peut pas avoir de préfixe strict valide.

*. on simplifie un peu : les variables seront des lettres entre a et z au lieu de `string`

La formule sera donc valide si on peut extraire ce préfixe, et que le reste est vide. Dans les autres cas, la formule n'est pas valide.

On commence par des cas particuliers qui serviront plus tard

4. Écrire une fonction `analyse_binop : lexeme list -> binop * lexeme list` qui renvoie le couple α, s si la liste de lexème commence par l'opérateur binaire α et a pour queue s, et lève une exception sinon.
5. De la même façon, écrire une fonction similaire `analyse_rightpar : lexeme list -> lexeme list` qui vérifie que la liste commence par une parenthèse droite, et renvoie la queue de la liste. Sinon, on lève une exception.
6. Finalement, écrire une fonction `analyse : lexeme list -> formule * lexeme list` qui renvoie la formule associée au premier préfixe et le reste de la liste.
7. En déduire la fonction `formule_of_string : string -> formule` qui traduit une chaîne de caractère en formule si celle-ci est valide, et lève une exception sinon.

Il ne reste plus qu'à programmer la fonction inverse, ce qui ne devrait pas poser de problèmes :

8. Coder une fonction `string_of_formule : formule -> string`.
9. Vérifier que `string_of_formule (formule_of_string (`*`formule`*`))` est bien l'identité

Quelques applications :

10. Écrire une fonction `and_or_formule : formule -> formule` qui transforme une formule en une formule équivalente n'ayant que des connecteurs $\neg$, $\wedge$ et $\vee$.
11. Écrire une fonction `prenexe_formule : formule -> formule` prenant une formule n'ayant que des connecteurs $\neg$, $\wedge$ et $\vee$ et qui renvoie une formule prénexe, *i.e.* où les négations ne peuvent se trouver que devant une variable.
12. Écrire (et tester) une fonction prenant une formule quelconque sous forme de chaîne de caractères, et renvoyant une formule équivalente n'ayant que des $\neg$, $\wedge$, $\vee$ et sous forme prénexe.