

# PROGRAMMATION FONCTIONNELLE

## ANALYSE LEXICALE

Dans le TD précédent, pour entrer l'expression représentant  $a \cdot x^2 + b \cdot x + c$ , vous deviez employer la notation barbare suivante :

```
Operation (Plus, Operation (Fois, Variable "a", Operation (Fois, Variable "x",
Variable "x") ), Operation (Plus, Operation (Fois, Variable "b", Variable "x"),
Variable "c") )
```

L'*analyse syntaxique* est un processus qui permet d'écrire plutôt quelque chose comme expression "a \* x \* x + b \* x + c" – et de se débrouiller pour que OCaml transforme ceci en l'arbre précédent. Conceptuellement, l'analyse syntaxique d'un langage est la même chose que l'analyse grammaticale en français.

Cette semaine, nous allons commencer par voir la première phase de l'analyse syntaxique : l'*analyse lexicale*, c'est-à-dire apprendre à OCaml à reconnaître les mots.

Pour ce faire, vous allez employer le module `Exp`, que je vous ai placé, avec sa documentation, dans votre répertoire d'échange.

## Logistique

**Exercice 1.** Écrire une fonction qui transforme une chaîne de caractères en une liste des caractères de cette chaîne.

Exemple :

```
# charlist_of_string "bouh !"
- : char list = ['b'; 'o'; 'u'; 'h'; ' '; ' ']

# charlist_of_string "a * x * x + b * x + c";;
- : char list = ['a'; ' '; '*'; ' '; 'x'; ' '; '*'; 'x'; ' '; '+'; 'b'; ' '; '*';
' '; 'x'; ' '; '+'; 'c']
```

Pour ce faire, vous aurez besoin d'utiliser les fonctions `String.length` et `String.get`, définies toutes les deux dans le module `String`, et dont je vous laisse chercher la définition.

## Analyse lexicale

En français, la phase d'analyse lexicale est celle où le lecteur reconnaît les différents symboles (lettres, espaces, ponctuation...) et rassemble les lettres en mots. Pour le langage d'expressions arithmétiques, il va s'agir de reconnaître les opérateurs, de rassembler les lettres en noms de variables et les chiffres en nombre. Le terme linguistique pour désigner espaces, ponctuation et mots est *lexèmes*.

Pour les exercices qui suivent, les lexèmes seront les éléments de type `Exp.instruction`.

**Exercice 2.** Compléter la fonction suivante de manière à ce qu'elle gère les parenthèses, les lettres et les opérateurs et qu'elle soulève `Caractere_inconnu c` si elle rencontre un caractère `c` qui n'est ni un chiffre, ni une lettre, ni un opérateur, ni une espace, ni une parenthèse).

```
let rec lexemelist_of_charlist = function
| ' ' :: t          -> lexemelist_of_charlist t
                        (*On ignore les espaces.          *)
| ('0'..'9' as n) :: t -> (LChiffres (int_of_chiffre n) ) ::
                        (lexemelist_of_charlist t)
```

Exemples d'utilisation de la fonction :

```
# lexemelist_of_charlist (charlist_of_string "bouh !");;
Exception: Exp.Caractere_inconnu '!' .

# lexemelist_of_charlist (charlist_of_string "a * x * x + b * x + c");;
- : Exp.instruction list = [Var "a"; Op Fois;
                           Var "x"; Op Fois;
                           Var "x"; Op Plus;
                           Var "b"; Op Fois;
                           Var "x"; Op Plus; Var "c"]

# lexemelist_of_charlist (charlist_of_string "xx * 1024");;
- : Exp.instruction list = [Var "x"; Var "x"; Op Fois;
                           Const 1; Const 0; Const 2; Const 4]
```

Maintenant que nous avons classé les symboles, nous pouvons commencer à rassembler les chiffres consécutifs, histoire d'en faire un jour des nombres, et de même les lettres consécutives, histoire d'en faire un jour des noms de variables.

**Exercice 3.** \* Compléter la fonction suivante, de manière à ce que, dans une `instruction list`, elle rassemble les chiffres consécutifs et un seul `Const` et qui rassemble les lettres consécutives en un seul `Var`.

```
let rec rassemble_lexemes = fonction
  | (Const a)::(Const b)::t -> Const (ajoute_chiffres a b)::(rassemble_lexemes t)
```

Exemple :

```
# rassemble_lexemes (lexemelist_of_charlist (charlist_of_string "xx * 1024"));;
- : lexeme list = [Var "xx"; Op Fois ; Const 1024]
```

**Exercice 4.** \*/<sub>2</sub> Écrire une fonction qui transforme une liste de caractères en une chaîne composée des mêmes caractères (dans l'ordre).

Exemple :

```
# string_of_charlist ['b'; 'o'; 'u'; 'h'; ' ' ; '!']
- : string = "bouh !"
```

**Exercice 5.** \*\*\*/<sub>2</sub> Écrire une fonction qui simplifie autant que possible une `expression`, en calculant les résultats des opérations sur les constantes lorsque celle-ci ne contient pas de variables. À vous de choisir ce que fait cette fonction lorsqu'il y a des variables dans l'`expression` – documentez vos choix.

Vous pourrez vous inspirer de ce qui a été fait pour appliquer les lois de Morgan, au cours de l'un des TDs précédents.

Exemple :

```
# evaluate_expression (Operation(Div, Constante 1024, Constante 256));;
- : expression = Constante 4
# evaluate_expression (Operation(Div, Variable "x", Operation (Div, Constante 1024, Constante 256)));;
- : expression = Operation(Div, Variable "x", Constante 4)
```