# BE de programmation fonctionnelle
# Session 2

3h : avec documents

Année 2021-2022

## Préambule

- Le code rendu doit impérativement compiler. Pour cela, les fonctions non implémentées peuvent être remplacées par un code quelconque, par exemple **let** suivant = **fun** _ -> **assert** false .
- Vous devez tout écrire dans le fichier `be.ml`.
- Les noms et types des fonctions doivent être respectés (tests automatiques).
- Pour tester dans `utop` vous devez ouvrir le module Be (**open** Be;;).
- La non utilisation d'itérateur sera pénalisée ainsi que l'utilisation inutile d'accumulateurs.
- Vous pouvez utiliser toutes les fonctions définies dans le module List d'OCaml `https://v2.ocaml.org/api/List.html`.
- Les quatre exercices sont presque indépendants : seul 2.3 dépend de 1.2.

## 1 Suite de Conway

Extrait de Wikipédia :

"La suite de Conway est une suite mathématique inventée en 1986 par le mathématicien John Horton Conway, initialement sous le nom de *suite audioactive*. Elle est également connue sous le nom anglais de Look and Say (*regarde et dis*). Dans cette suite, un terme se détermine en annonçant les chiffres formant le terme précédent.

Le premier terme de la suite de Conway est posé comme égal à 1. Chaque terme de la suite se construit en annonçant le terme précédent, c'est-à-dire en indiquant combien de fois chacun de ses chiffres se répète.

Concrètement :

$X_0 = 1$

Ce terme comporte simplement un « 1 ». Par conséquent, le terme suivant est :

$X_1 = 11$

Celui-ci est composé de deux « 1 » :

$X_2 = 21$

En poursuivant le procédé :

$X_3 = 1211$, $X_4 = 111221$, $X_5 = 312211$, $X_6 = 13112221$, ...

Et ainsi de suite.

Il est possible de généraliser le procédé en prenant un terme initial différent de 1."

▷ **Exercice 1** *Préambule*

1. *Écrire la fonction* max*, qui renvoie la valeur maximale d'une liste d'entiers.*
2. *Écrire la fonction* max_max*, qui renvoie la valeur maximale d'une liste de listes d'entiers (max des max).*

▷ **Exercice 2** *Nous représenteront un terme de la suite de Conway par une liste d'entier, par exemple $X_5$ sera représenté par la liste [3 ;1 ;2 ;2 ;1 ;1].*

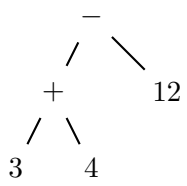
1. *Écrire la fonction* suivant*, qui pour une liste d'entiers construit le terme suivant en annonçant les chiffres formant cette liste.*
   *Aide :*
   — *suivant 1 = 11 , suivant 2 = 12 , suivant 3 = 13 , . . .*
   — *suivant **12**232 = 11221312 (appel récursif : suivant 2232 = 221312)*
   — *suivant **22**232 = 321312 (appel récursif : suivant 2232 = 221312)*
2. *Écrire la fonction* suite*, qui calcule la suite de Conway d'une taille donnée passée en paramètre et avec le premier terme donné en paramètre.*
3. *Écrire une série de tests ppx (`let%test ...`), qui permet de tester si la propriété "Aucun terme de la suite ne comporte un chiffre supérieur à 3" est vraie pour la suite dont le premier terme est 1.*

# 2 Expressions

## 2.1 Arbres binaires

Nous pouvons représenter les expressions binaires à l'aide d'un arbre, où l'opérateur (addition, soustraction, multiplication ou division) est indiqué dans les nœuds et où les nombres sont indiqués dans les feuilles.

Par exemple, l'expression $((3 + 4) - 12$ sera représentée par l'arbre :

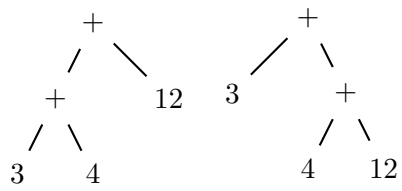



▷ **Exercice 3** *La fonction* eval *calcule la valeur d'une expression binaire représentée par un arbre binaire dont le type est fourni dans le fichier* `be.ml`*. Par exemple, appelée avec l'arbre précédent en paramètre, elle renverra $-5$.*

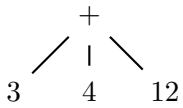
— *Écrire le contrat de la fonction* eval.
— *Écrire les tests de la fonction* eval.
— *Écrire le corps de la fonction* eval.

## 2.2 Arbres n-aires

Les opérateurs $+$ et $*$ sont associatifs gauche et droit donc les expressions suivantes sont équivalentes :




Nous proposons alors de les représenter par un arbre n-aire :




Bien sûr une telle représentation n'est pas possible pour les opérateurs de soustraction ou division.

▷ **Exercice 4**

- *Écrire la fonction* bienformee *qui vérifie qu'un arbre n-aire (dont le type est fourni dans le fichier* `be.ml`*) représente bien une expression n-aire, c'est-à-dire que les opérateurs d'addition et multiplication ont* ***au moins*** *deux opérandes et que les opérateurs de division et soustraction ont* ***exactement*** *deux opérandes.*
- *Écrire la fonction* eval_bienformee *qui calcule la valeur d'une expression n-aire représentée par un arbre n-aire bien formé.*
- *Définir l'exception* Malformee.
- *Écrire la fonction* eval_n *qui lève l'exception* Malformee *si l'arbre n-aire passé en paramètre est mal formé et calcule la valeur de l'expression sinon.*