# BE de programmation fonctionnelle

3h : avec documents

Année 2022-2023

## 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 les fichiers `conway.ml` et `expression.ml`. Les contrats et tests de certaines fonctions sont fournis dans les fichiers `conwayTests.txt`, `expressionArbreBinaire.txt` et `expressionArbreNaire.txt`.
— Les noms et types des fonctions doivent être respectés.
— Pour tester dans `utop` vous devez ouvrir le module Be (**open** Be;;) puis ceux que vous voulez tester, par exemple Conway.
— 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 exercices ont peu de dépendance.

⚠ ⚠ ⚠ **Rendu du BE** ⚠ ⚠ ⚠

— Vous ne pourrez faire qu'un seul dépôt (modification impossible).
— Utilisez le script `genererRendu.sh` pour générer un fichier `<votre login>.tar` compression d'un répertoire `<votre login>` contenant les fichiers `conway.ml`, `expression.ml` et les fichiers dune.
— Avant de déposer le tar, vérifiez qu'il contient bien les fichiers souhaités.
— URL de dépôt selon vos groupes

| | |
|---|---|
| A1 | `https://pigeonfiles.com/u/nl4dr94zbe1-zrqdc9f0s2` |
| A2 | `https://pigeonfiles.com/u/ypiadgxknfkciby4oav5pe` |
| B1 | `https://pigeonfiles.com/u/vdj33fddosrxutp4mx3v6v` |
| B2 | `https://pigeonfiles.com/u/06jp-18ulh34mvkir446x5` |
| L1 | `https://pigeonfiles.com/u/d18vjoojrxu9z78xfb0u46` |
| L2 | `https://pigeonfiles.com/u/cm_4r19vo4mvemtpok3_og` |
| L3 | `https://pigeonfiles.com/u/d876aburj3jvaj64ufb4fo` |
| L4 | `https://pigeonfiles.com/u/cw70ikrl7_fabcukr779z2` |
| M1 | `https://pigeonfiles.com/u/_s_i1qmjibo75x74szyxw-` |
| M2 | `https://pigeonfiles.com/u/o98iu7mr3_j3h95yfl-osb` |
| 3A | `https://pigeonfiles.com/u/29oi8td_g5zwrkmrnz-qg_` |

# 1 Suite de Conway – `conway.ml`

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. Ses tests sont fournis dans* `conwayTests.txt`.
2. *Écrire la fonction* max_max, *qui renvoie la valeur maximale d'une liste de listes d'entiers (max des max). Ses tests sont fournis dans* `conwayTests.txt`.

▷ **Exercice 2** *Nous représenterons 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. Ses tests sont fournis dans* `conwayTests.txt`.

   *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. Ses tests sont fournis dans* `conwayTests.txt`.
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.*
4. *Indiquer dans la partie "Remarque" du fichier, ce que vous pensez de cette technique de vérification de la propriété.*

# 2 Expressions – `expression.ml`

## 2.1 Expression

Nous souhaitons représenter les expressions mathématiques 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.

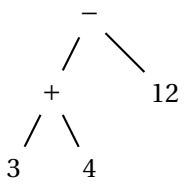
Nous allons proposer deux représentations possibles. L'une à base d'arbres binaires et l'autre à base d'arbres n-aires.

▷ **Exercice 3** *Dans l'interface* Expression*, écrire le contrat de la fonction* eval *qui permet d'évaluer la valeur d'une expression.*

## 2.2 Arbres binaires

Le première représentation des expressions est à l'aide d'un arbre binaire, 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 :

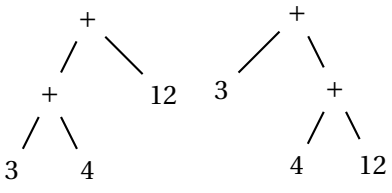



▷ **Exercice 4**

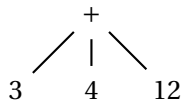
1. *Écrire un module* ExpAvecArbreBinaire *conforme à l'interface* Expression *et dont le contenu sera (dans un premier temps) le contenu du fichier* `expressionArbreBinaire.txt`.
2. *La fonction* eval *calcule la valeur d'une expression représentée par un arbre binaire dont le type est fourni. Par exemple, appelée avec l'arbre précédent en paramètre, elle renverra* $-5$.
   — *Écrire les tests de la fonction* eval.
   — *Écrire le corps de la fonction* eval.

## 2.3 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 5**

1. *Écrire un module* ExpAvecArbreNaire *conforme à l'interface* Expression *et dont le contenu sera (dans un premier temps) le contenu du fichier* `expressionArbreNaire.txt`.
2. *Écrire la fonction* bienformee *qui vérifie qu'un arbre n-aire représente bien une expression, 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.*
3. *Écrire la fonction* eval_bienformee *qui calcule la valeur d'une expression n-aire représentée par un arbre n-aire bien formé.*
4. *Définir l'exception* Malformee.
5. *Écrire la fonction* eval *qui lève l'exception* Malformee *si l'arbre n-aire passé en paramètre est mal formé et calcule la valeur de l'expression sinon.*