UNIVERSITÉ CAEN NORMANDIE

Département mathématiques et informatique
UFR des Sciences

Programmation Fonctionnelle
Licence 2 Informatique
L2 MATH
A. Ouali

TP2
"Récursion, filtrage et conditionnels"

1 Environnement de développement

1. Pour ce TP, travaillez dans le dossier ∼/Documents/L2/PF/tp2/. "∼" représente votre dossier personnel.
2. Voir **TP1** sur la préparation de votre environnement pour compiler et exécuter un code Haskell.

2 Concepts de base

2.1 Récursion : rire avec une liste de `Char`

Définir récursivement la fonction `(laugh n)` qui retourne la chaîne contenant `n` occurrences de l'interjection `"HA "`. Proposez une solution avec une expression conditionnelle et une autre avec les gardes.

```
> laugh 3  ==> "HA HA HA "
> laugh 1  ==> "HA "
```

2.2 Filtrage par motifs

1. Récrire la fonction récursive `(laugh n)` en utilisant pattern-matching.
2. Définir récursivement la fonction `(double xs)` qui "double" chaque chaîne de `xs`.

   ```
   > double ["je", "be", "gaye"] ==> "je je be be gaye gaye "
   > double ["stut", "te", "ring"]  ==> "stut stut te te ring ring "
   ```

3. Définir la fonction `(myZip xs ys)` qui calcule le couple d'éléments du même rang de deux listes `xs` et `ys` :

   ```
   > myZip [1, 2, 3] "abc"  ==> [(1, 'a'), (2, 'b'), (3, 'c')]
   > myZip [1, 2, 3] "abcdef"  ==> [(1, 'a'), (2, 'b'), (3, 'c')]
   > myZip [1, 2, 3] "ab"  ==> [(1, 'a'), (2, 'b')]
   ```

4. Définir la fonction `(split xs)` qui calcule le couple de listes formé des éléments de rangs pair et impair (utiliser un `let` ou un `where`).

   ```
   > split "azertyuiop"  ==> ("aetuo","zryip")
   > split [1..9]  ==> ([1,3,5,7,9],[2,4,6,8])
   ```

5. Définir la fonction réciproque `(unsplit xs ys)`.

   ```
   > unsplit ([1,3,5,7,9], [2,4,6,8])  ==> [1,2,3,4,5,6,7,8,9]
   > unsplit (split "azertyu")  ==> "azertyu"
   > split (unsplit ("aetuo","zryip"))  ==> ("aetuo","zryip")
   ```

3 Mutation

Lorsque les cellules se divisent, leur ADN se réplique également. Parfois, au cours de ce processus, des erreurs se produisent et des morceaux uniques d'ADN sont encodés avec des informations incorrectes. Nous lisons l'ADN en utilisant les lettres `C, A, G et T`. Deux séquences pourraient ressembler à ceci :

```
GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT
^ ^ ^  ^ ^    ^^
```

Si nous comparons deux séquence d'ADN et comptons les différences entre eux, nous pouvons voir combien d'erreurs se sont produites. C'est ce qu'on appelle la "distance de Hamming".

— Définir une fonction `hamming` en rajouter son type pour calculer la distance de Hamming entre deux séquence d'ADN. Noter que si les séquences ne sont pas fournies avec la même taille, on rajoute une distance de 1 sur les caractères en excédent.
```
Prelude> hamming "GAGCCTACTAACGGGAT" "CATCGTAATGACGGCCT"
7
```

4 Fonction Collatz

La fonction de Collatz est définie pour n entier, $n \geq 2$ par

$$collatz(n) = \begin{cases} 1 & \text{si } n = 2 \\ \frac{n}{2} & \text{si n est pair} \\ 3n+1 & \text{si n est impair} \end{cases}$$

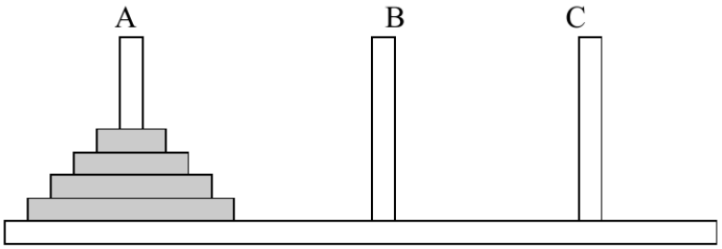
La conjecture de Collatz affirme que :

$$\forall n \geq 2, \exists k \in N^*, \underbrace{collatz(...(collatz(n))...)}_{k fois} = 1$$

1. Définir la fonction `(collatz n)`.
2. Définir récursivement la fonction `(nbCalls n)` qui calcule le nombre d'applications nécessaires de la fonction `collatz` pour atteindre la valeur `1` en partant de `n`.
   ```
   > nbCalls 16  ==> 4
   > nbCalls 3  ==> 7
   ```
3. De manière analogue, définir récursivement la fonction `(syracuse n)` qui construit, pour `n` donné, la liste des valeurs successives permettant d'atteindre `1`.
   ```
   > syracuse 16  ==> [16, 8, 4, 2]
   > syracuse 3  ==> [3, 10, 5, 16, 8, 4, 2]
   ```
4. En déduire une seconde définition de la fonction `(nbCalls n)`
5. Que se passerait-il si la conjecture de Collatz était fausse ?

5 Tour de Hanoi

On dispose de 3 broches, nommées A,B, et C, sur lesquelles on peut empiler des disques percés en leur centre. Les disques sont tous de taille différente. Initialement tous les disques sont sur A. On veut les faire passer sur B, en respectant la règle : *Un disque doit toujours reposer sur un disque plus grand.*



Une légende dit que les moines de certain monastère sont consacrés à cette tâche. Bien sûr, les broches sont de diamant et les disques de bronze. Et les disques sont au nombre de 64. L'achèvement de leur tâche annoncera la fin du monde... Sans attendre, nous allons réaliser un programme Haskell indiquant aux bons moines comment mener à bien leur tâche, pour un nombre n de disques, passé en paramètre.

La fonction `hanoi` prend en argument les noms des 3 broches (des strings) et retourne une liste de strings indiquant les mouvements à effectuer.

```
> hanoi "A" "B" "C" 3
["de A vers B","de A vers C","de B vers C","de A vers B","de C vers A",
"de C vers B","de A vers B"]
```

Définir les fonctions :

1. `bouge` *origine but* qui calcule la chaîne de origine vers but où origine et but sont des strings désignant des broches "A" "B" ou "C") donc dans notre exemple.

   ```
   > bouge "A" "B"
   "de A vers B"
   ```

2. En déduire la fonction récursive `hanoi` décrite ci-dessus. Quel est son type ?
3. Testez pour quelques valeurs croissantes de n et persuadez-vous que la fin du monde n'est pas pour demain...
4. Utilisez la fonction `hanoi` ou modifiez la pour calculer le nombre M(n) de déplacements effectués.

6 Compactification

Une manière très élémentaire de compacter une information consistant en une suite de nombres est d'opérer la transformation suivante.

Pour passer de L (liste donnée) à L' (compactée) on regarde les sous listes de nombres consécutifs égaux ; pour chacun de ces séquences, on produit 2 chiffres : la longueur de cette séquence et le chiffre qui y est répété : et on met ces couples bout à bout dans L', dans l'ordre des sous-listes correspondantes de L. Par exemple :

```
let l = [5,5,5,5,1,1,3,3,3,5,5,2,2,2,2,2]
> compacte l
[4,5,2,1,3,3,2,5,5,2]
```

1. Écrire une fonction Haskell qui prend en arguments un nombre x une liste non vide l et calcule le nombre d'occurrences de x, en tête de liste. Exemple :

   ```
   > nb_repetitions 5 l
   4
   > nb_repetitions 2 l
   0
   ```

2. En itérant cette fonction, définissez la fonction compacte.

3. Ecrire la fonction inverse decompacte :

```
> decompacte (compacte l)
[5,5,5,5,1,1,3,3,3,5,5,2,2,2,2,2]
```

4. On considère la suite Li de listes suivante :
   — L1 = [1]
   — On passe de Li à Li+1 en opérant une ≪ compactification ≫.
   Ecrire une fonction (suite n) qui calcule le n° terme de cette suite. Exemple :

```
> suite 1
[1]
> suite 2
[1,1]
> suite 3
[2,1]
> suite 6
[3,1,2,2,1,1]
> suite 7
[1,3,1,1,2,2,2,1]
```