Université de Lille

2026–2027
Licence Informatique

Programmation fonctionnelle

PF : TD 1

Échauffement exponentiel

Exercice 1 : Implémentations de l'exponentielle

Nous allons implémenter quatre versions différentes de l'exponentielle : la fonction `monExp` de type `Int -> Int -> Int` telle que `monExp a n` renvoie a^n.

Q 1.1 Version simple
En utilisant la relation entre a^n et a^{n-1} donner une implémentation récursive de `monExp`.

Q 1.2 Version récursive terminale
En utilisant une fonction intermédiaire (par exemple dans un clause `where ...`) qui manipule un accumulateur, donner une définition récursive terminale de `monExp`.

Q 1.3 Version rapide
Voici deux identités :

$$\begin{array}{lll} a^n & = a^{n/2} \times a^{n/2} & \text{si } n \text{ pair} \\ a^n & = a \times a^{n/2} \times a^{n/2} & \text{si } n \text{ impair} \end{array}$$

Utilisez-les pour écrire une dernière version de `monExp`. Pensez à utiliser des variables pour éviter les calculs redondants.

Vous pouvez utiliser la fonction `quotRem` de type `Int -> Int -> (Int,Int)` : `quotRem a b` calcule la paire `(q,r)` où `q` et `r` sont respectivement le quotient et le reste de la division euclidienne de `a` par `b`.

Q 1.4 Avec des fonctions du `Prelude`
En utilisant les fonctions suivantes :
- `product` de type `[Int] -> Int` : `product l` renvoie le produit des éléments de `l`
- `replicate` de type `Int -> Int -> [Int]` : `replicate n a` renvoie une liste `[a, ..., a]` qui contient `n` occurrences de `a`,

donnez une dernière version de la fonction `monExp`.

Compter les occurrences des éléments d'une liste

L'objectif des exercices qui suivent est d'écrire une fonction `compte` de type `[Int] -> [(Int, Int)]` qui transforme une liste d'entiers `l` en une *liste associative* qui a chaque entier présent dans `l` associe le nombre d'occurrences qu'il a dans `l`. L'ordre dans lequel les paires de la liste résultat apparaissent n'a pas d'importance.

Voici quelques exemples de calculs que `compte` pourrait effectuer :

```
>>> compte []
[]
>>> compte [0,1,1,3,0,2]
[(0,2), (1,2), (3,1), (2,1)]
```

Exercice 2 : Approche naïve

Nous allons commencer par implémenter un algorithme simple :
- nous allons parcourir la liste,
- pour chaque élément `a` de la liste :
  - nous allons compter le nombre `n` d'occurrences de `a` dans la suite de la liste,
  - nous allons ensuite éliminer cet élément de la suite de la liste,
  - *compter* récursivement les éléments de la liste ainsi obtenue,

— pour finalement ajouter en tête de la liste associative obtenue la paire `(a,n)`.

Cette description algorithmique fait apparaître la nécessité d'implémenter deux fonctions intermédiaires :

— `compterOcc` de type `Int -> [Int] -> Int` telle que `compterOcc a l` renvoie le nombre d'occurrences de `a` dans `l`.

— `eliminer a l` qui renvoie la liste obtenue à partir de `l` en éliminant les occurrences de `a`.

Q 2.1 Implémentation

— Écrire la fonction `compterOcc`.

— Écrire la fonction `eliminer`.

— Écrire la fonction `compter`.

Q 2.2 Fusionner `compterOcc` et `eliminer`

Vous devriez remarquer que les deux fonctions `compterOcc` et `eliminer` se ressemblent beaucoup. Au lieu de traverser deux fois la liste `l`, nous souhaiterions la traverser une seule fois et renvoyer une paire qui contient le nombre d'occurrences de l'élément `a` et la liste obtenue en éliminant `l`.

Écrivez une fonction `comptElim` de type `Int -> [Int] -> (Int, [Int])` telle que si `(n, el) = comptElim a l` alors `n` est le nombre d'occurrences de `a` dans `l` et `el` est la liste obtenue en éliminant les occurrences de `a` dans `l`.

Réécrivez `compter` en utilisant `comptElim`

Q 2.3 Complexité

Quel est l'ordre de grandeur de la complexité dans le pire des cas des implémentations de `compter` des deux questions précédentes ?

Exercice 3 : Trier puis compter

Afin d'obtenir un algorithme plus efficace, nous proposons de trier la liste, pour ensuite compter les occurrences de chaque élément présent dans la liste. Celle-ci étant triée, les occurrences de chaque élément seront consécutives dans la liste. Cela devrait permettre d'effectuer le comptage en temps linéaire. C'est le temps de tri qui est prépondérant avec cet approche. Nous allons utiliser un tri qui s'effectue en $\mathcal{O}(n\log(n))$ lorsque n est le nombre d'éléments de la lister à trier. La complexité de la fonction `compter` sera alors $\mathcal{O}(n\log(n))$.

Nous allons commencer par implémenter l'algorithme de tri fusion qui :

— découpe la liste en deux listes de longueurs égales (à au plus un élément près),

— trie récursivement chaque liste,

— puis fusionne les deux listes triées obtenues pour construire la liste triée.

Nous avons besoin d'implémenter trois fonctions :

— `decoupe` de type `[Int] -> ([Int], [Int])`,

— `fusion` de type `[Int] -> [Int] -> [Int]`,

— et `triFusion` de type `[Int] -> [Int]`.

Q 3.1 Implémentation de `decoupe`

Pour implémenter la fonction `decoupe`, nous allons chercher à renvoyer une paire de listes qui contiennent chacun un élément sur deux : la première les éléments d'indices pairs ; la seconde les éléments d'indices impairs.

Voici quelques exemples d'exécutions :

```
>>> decoupe []
([], [])
>>> decoupe [1]
([1], [])
>>> decoupe [8,4,1,5,7]
([8,1,7], [4,5])
```

Q 3.2 Implémentation de `fusion`

Implémentez la fonction `fusion` en prenant avantage du fait qu'elle prend en argument des listes déjà triées. Cette fonction doit s'exécuter en temps linéaire dans la taille des listes arguments.

Q 3.3 Implémentation de `triFusion`

Implémentez la fonction `triFusion`.

Q 3.4 Compter les occurrences

Implémentez la fonction `compteTri` de type `[Int] -> [(Int, Int)]` qui compte les occurrences des éléments d'une liste triée.

Indication : on pourra utiliser une fonction intermédiaire (dans une construction `where ...` ou une construction `let ... in ...`) `cptAux` de type `Int -> Int -> [Int] -> [(Int,Int)]` telle que `cptAux n cpt l` compte le nombre d'occurrences de `n` en tête de `l` à l'aide du paramètre `cpt`. La valeur de `n` n'est changée que lorsque `l` a en tête une valeur différente de `n` ; sinon, on incrémente `cpt`.

Implémenter ensuite la fonction `compter`.