DEUG $1^{ere}$ année
MIAS 1 et MASS 1 – Programmation fonctionnelle

TP n° 10 : Fonctions d'ordre supérieur

**Exercice 1 :** Implantation de la méthode de Newton

La méthode de Newton permet de trouver les racines d'une fonction dérivable. Elle dit que si $x_0$ est une approximation d'une racine d'une fonction $f$ , alors une meilleure approximation est donnée par : $x_0 - \frac{f(x_0)}{f'(x_0)}$.

1. Spécifier et écrire une fonction **derive** qui renvoie fonction qui est une approximation de la dérivée d'une fonction $f$ donnée pour un nombre $dx$ donné (nombre très petit).
   On rappelle, que pour $dx$ petit, une approximation du nombre dérivé $f'(x)$ d'une fonction $f$ est : $\frac{f(x+dx)-f(x)}{dx}$.
2. Spécifier et écrire un prédicat **assez-proche?**, qui teste si un nombre donné $x_0$ est une racine approchée convenable d'une fonction $f$ donnée, relativement à une erreur fixée.
3. Spécifier et écrire une fonction **ameliore**, qui étant donnée $x_0$, une approximation d'une racine d'une fonction $f$, calcule une meilleure approximation selon le critère de Newton.
4. Spécifier et ecrire une fonction **newton**, qui, étant donnée une fonction $f$, calcule une solution approchée de l'équation $f(x) = 0$, en prenant *init* comme première approximation.

**Rappel :** Utilisation des fonctions **apply** et **map**

**apply** et **map** sont des fonctions d'ordre supérieure prédéfinies dans Scheme. Elles permettent d'appliquer une fonction à une liste de paramètres.

La fonction **apply** prend comme paramètres une fonction $f$ et une liste $l$ et renvoie le résultat de la fonction en prenant comme valeurs de ses paramètres les éléments de $l$.
Si $f : D_1, D_2, \ldots, D_n \to A$ alors $l$ doit être une liste de longeur $n$ dont les types des éléments correspondent aux types des arguments de la fonction, c'est à dire une liste de la forme $(e_1\ e_2\ \ldots\ e_n)$ avec $e_1 \in D_1$, $e_2 \in D_2$ etc...
L'appel `(apply f '(e1 e2 ... en))` est equivalent à `(f e1 e2 ... en)`
Par exemple l'évaluation de `(apply + '(1 2 3))` rend le résultat 6.

La fonction **map** permet de distribuer l'appel d'une fonction sur plusieurs arguments et renvoie la liste des résultats.
Pour une fonction $f : D \to A$ et une liste $l = (e_1 \ldots e_k)$ dont tous les éléments sont du type $D$, l'appel `(map f '(e1 ... ek))` applique la fonction à chacun des `ei` et renvoie la liste $(f(e_1)f(e_2)\ldots f(e_k))$.
Par exemple, l'évaluation de `(map car '((1 2)(3)(7 8 9)))` rend comme résultat la liste `(1 3 7)`.

Si la fonction $f$ est d'arité $n$, c'est à dire $f : D_1, D_2, \ldots D_n \to A$, alors la fonction **map** prend en arguments $n$ listes $l_1, l_2, \ldots, l_n$, les éléments de chaque liste $l_i$ étant tous du type $D_i$ correpondant.

**Exercice 2 :**

1. Étant données une fonction $f : \mathbb{N}^p \to \mathbb{N}$ et une fonction $g : \mathbb{N} \to \mathbb{N}$, définir en Scheme la fonction dont la spécification est :

$$\begin{array}{lccc} \text{combine :} & \textit{Fonction} & \textit{Fonction} & \to & \textit{Fonction} \\ & f : \mathbb{N}^p \to \mathbb{N}, & g : \mathbb{N} \to \mathbb{N} & \mapsto & \mathbb{N}^p \to \mathbb{N} \\ & & & & x_1 \ldots x_p \mapsto f(g(x_1), \ldots, g(x_p)) \end{array}$$

2. On peut représenter en Scheme un vecteur de nombres sous la forme d'une liste de nombres et une matrice sous la forme d'une liste de vecteurs (une liste de listes de nombres).
   a) Définir une fonction `(produit-scalaire v w)`.
   b) En utilisant la fonction `produit-scalaire` et à l'aide de la fonction d'ordre supérieure **map**, définir une fonction `(matrice-fois-vecteur m v)`.

**Exercice 3 :** Un générateur de fonctions récursives simples.

Lorsqu'on définit une fonction récursive, on doit se poser 4 questions :

1. Quel est le test d'arrêt ?
2. Quelle est la valeur rendue lorsque la récursivité s'arrête, c'est à dire la solution triviale?
3. Quel calcul doit-on effectuer sur le résultat de l'appel récursif de la fonction?
4. Quelle opération effectuer sur l'argument pour faire tendre celui-ci vers le test d'arrêt?

Dans cet exercice, on veut définir une fonction `(genrec arret valarret calcul decrois)` qui prend comm ramètres 4 fonctions, correspondants aux 4 questions ci-dessus et rend la fonction récursive correspondant paramètres sont :

1. `arret` : une lambda-fonction à un argument qui correspond au test d'arrêt de la fonction récursive génér
2. `valarret` : une lambda-fonction à un argument (celui de la fonction générée) et qui rend la solution triv savoir une constante ou bien le résultat d'un calcul sur son argument.
3. `calcul` : une lambda-fonction à deux arguments : le premier est l'argument de la fonction qu'on génère, le est le résultat des appels récursifs.
4. `decrois` : une lambda-fonction qui fait décroître la taille du problème.

Exemples d'utilisations de la fonction `genrec` :

```
(define factorielle
   (genrec (lambda (x) (= x 0))
           (lambda (x) 1)
           (lambda (x r) (* x r))
           (lambda (x) (- x 1))) )

(define fois2
   (genrec (lambda (l) (null? l))
           (lambda (l) '())
           (lambda (l r) (cons (* 2 (car l)) r))
           (lambda (l) (cdr l))) )
```

En utilisant la fonction `genrec`, générer les fonctions :

- `longueur` qui rend le nombre d'élément d'un liste plate
- `renverse` qui renverse une liste plate (on utilisera la fonction prédéfinie append)
- `nbatt` qui rend le nombre d'atomes d'une liste, en comptant à tous les niveaux