TP 2 : Ordre Supérieur

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Exercice I : Relation de récurrence

Soit u_n définie par :

$$\begin{cases} u_0 & = & 42 \\ u_{n+1} & = & 3u_n + 4 \end{cases}$$

Q1 – Donnez la définition de la fonction `u : int -> int` telle que `(u n)` donne $u_{\mathtt{n}}$ en supposant `n` positif.

Q2 – On peut généraliser ce type de calcul. Si f est la fonction unaire $x \mapsto 3x + 4$ alors $u_n = f^n(42)$, où f^n est l'itération de la fonction f, n fois. Intuitivement, $f^n(a) = f(f(\ldots f(a) \ldots))$. On définit f^n de la manière suivante :

$$\begin{cases} f^0(x) & = & x \\ f^{n+1} & = & f(f^n(x)) \end{cases}$$

Pour obtenir un mécanisme général de ce genre de calcul, on définit une *fonction d'ordre supérieur*, disons $\mathcal{I}$ qui prend en argument un entier, indice de l'itération, la fonction f à itérer et la valeur a du cas de base et qui vérifie les équations suivantes :

$$\begin{cases} \mathcal{I}(0, f, a) & = & a \\ \mathcal{I}(n+1, f, a) & = & f(\mathcal{I}(n, f, a)) \end{cases}$$

En suivant ce schéma récursif, donnez la définition de la fonction `iter : int -> (int -> int) -> int -> int` telle que `(iter n f a)` donne $\mathtt{f}^{\mathtt{n}}(\mathtt{a})$, pour $\mathtt{n} \geq 0$.

Q3 – Vérifiez que les appels `(u 10)` et `(iter 42 f 10)`, où f est une fonction anonyme qui à x associe $(3x + 4)$, donnent le meme resultat

Exercice II : Fusion de listes

On se propose d'implémenter la fonction `flatten` qui aplatit une liste de liste au sens suivant :

si `l = [ [`$a_{1,1}$`;`$\cdots$`;`$a_{n_1,1}$`] ; ... ; [`$a_{1,m}$`;...;`$a_{n_m,m}$` ]]`

alors `flatten l = [`$a_{1,1}$`;`a_{2,n_1}`;...;`$a_{n_1,1}$`;`$a_{1,2}$`;...;`$a_{n_2,2}$`;...;`$a_{1,m}$`;...;`$a_{n_m,m}$`]`

Exemple
```
flatten [[1;2;3];[4;5;6];[];[7;8]] = [1;2;3;4;5;6;7;8]
flatten [[];[];[];[]] = []
```

Q1 – Donner la signature de la fonction `flatten`

Q2 – Proposer une implantation de la fonction `flatten`

Q3 – Écrivez une fonction `revert_and_push` qui prend en argument deux listes, retourne (au sens des listes) la première et la met en tête de la seconde.

Exemple
```
revert_and_push [3;2;1] [4;5;6] = [1;2;3;4;5;6]
```

EXERCICE III : Permutations d'une liste

Q1 – Écrivez d'abord une fonction `insert` qui prend deux arguments : un élément `x` à insérer et une liste dans laquelle insérer x. Cette fonction devra retourner une liste de listes où x à été inséré à toutes les positions possibles.

Exemple
```
insert 0 [] = [[0]]
insert 0 [1;2;3] = [[0;1;2;3];[1;0;2;3];[1;2;0;3];[1;2;3;0]]
```

Q2 – Nous allons maintenant utiliser cette fonction d'insertion pour écrire notre fonction `perm`. Pour cela, nous allons parcourir la liste et pour chaque élément, l'insérer l'élément à toutes les positions des permutations du reste de la liste (vous pourrez réutiliser la fonction `flatten` de l'exercice précèdent).

Exemple
```
perm [0;1;2] = [[0;1;2]; [1;0;2]; [1;2;0]; [0;2;1]; [2;0;1]; [2;1;0]]
```

EXERCICE IV : Fonction inverse

Q1 – Définissez une fonction `equiv` qui prend une liste d'éléments l, deux fonctions f et g, et qui vérifie que f et g sont équivalentes pour toutes les entrées de l, *i.e.* $\forall x \in l, f(x) = g(x)$.

On désire avoir un mécanisme qui permet à partir d'une fonction f d'obtenir sa réciproque $g = f^{-1}$.

Q2 – Définissez une fonction `inverse` qui prend une liste $\{e_1, e_2, \ldots, e_n\}$ d'éléments et une fonction f à inverser. Elle devra retourner une fonction anonyme unaire qui vérifiera si son argument est égal à une des images $\{(f\ e_1), (f\ e_2), \ldots, (f\ e_n)\}$ et retourner l'antécedent correspondant (e_i) dans ce cas. On pourra lever une erreur si la fonction est appellée sur un élement dont l'antécédent n'est pas dans la liste.

Exemple

```
(inverse [0;1;2] succ) 1 = 0
(inverse [0;1;2] succ) 3 = 2
(inverse [0;1;2] succ) 4 = error "entry not in co-domain"
```

Q3 – Définissez une fonction `square` qui associe à un entier x son carré, et une fonction `mysqrt` qui sera l'inverse de `square` sur la liste des 10 premiers entiers.

Q4 – Vérifier que votre fonction `mysqrt` est équivalente à la primitive `sqrt` déjà définie dans le langage sur une liste donnée de carrés parfaits (*e.g.* `[1, 4, 9, 16]`)