Lambda-calcul et programmation fonctionnelle : Devoir 1

Due : le dimanche 23 janvier 2022 23 :59 CET

Credit cards, photo by Lotus Head, CC BY-SA 3.0 https://www.freeimages.com/photographer/LotusHead-31750

La plupart des sites de payement en ligne vérifient le numéro de la carte de payement saisie avant de réaliser la transaction. Ceci permet d'éviter de démarrer une transaction avec un numéro de carte incorrect (avec des erreurs de saisie). Cette vérification est souvent faite avec une méthode appelée « algorithme de Luhn », crée par Hans Peter Luhn (`https://fr.wikipedia.org/wiki/Formule_de_Luhn`).

L'algorithme procède en trois étapes.

1. Multiplie par deux un chiffre sur deux, en commençant par l'avant dernier et en se déplaçant de droite à gauche. Si le double d'un chiffre est supérieur à neuf (par exemple `2 * 8 = 16`), alors il faut le ramener à un chiffre entre `1` et `9`. Pour cela, on soustrait `9` au double. Avec le même exemple, `16 - 9 = 7`.
2. La somme de tous les chiffres obtenus est effectuée.
3. Le résultat est divisé par `10`. Si le reste de la division est égal à zéro, alors le nombre original est valide.

Dans ce TP, vous devez implémenter cet algorithme en écrivant en Haskell les fonctions spécifiées dans les exercices qui suivent. Pour chaque fonction spécifiée ci-dessous, vous devez également implémenter un test. Le test est implémenté sous forme d'une liste de Booléens, chacun correspondant à un cas de test. Un cas de test correspond à l'appel de la fonction comparé au résultat espéré. Par exemple, pour la fonction `toDigitsRev` de l'exercice 1, le test sera :

```
testToDigitsRev :: [Bool]
testToDigitsRev =
    [ toDigitsRev 1234 == [4,3,2,1],
      toDigitsRev 0 == [],
      toDigitsRev (-17) == [] ]
```

Certains cas de test sont suggérés dans l'énoncé de chaque exercice. Vous devez ajouter d'autres cas de test pour vous assurer que la fonction a été implémentée correctement.

Exercice 1. Écrivez la fonction :

```
toDigitsRev :: Integer -> [Integer]
```

qui converti un nombre positif dans une liste de nombres, mais avec les chiffres inversés. Pour les nombres négatifs et zéro, la fonction retourne la liste vide. Exemples :

```
toDigitsRev 1234 == [4,3,2,1]
toDigitsRev 0 == []
toDigitsRev (-17) == []
```

Exercice 2. Une fois le nombre est converti en liste, nous devons doubler un chiffre sur deux. Écrivez la fonction :

```
doubleEveryOther :: [Integer] -> [Integer]
```

qui double un nombre sur deux de la liste, en commençant par le second de la liste. Exemples :

```
doubleEveryOther [8,7,6,5] == [8,14,6,10]
doubleEveryOther [1,2,3] = [1,4,3]
```

Exercice 3. La sortie de `doubleEveryOther` contient un mix des nombres avec un et deux chiffres. Écrivez la fonction :

```
rem9 :: Integer -> Integer
```

qui, étant donné un entier, si'il est supérieur à `9`, le ramene à un chiffre entre `1` et `9`. Pour cela, on soustrait `9` au double. Pour les nombres négatifs, la fonction retourne zéro. Exemples :

```
rem9 (-12) == 0
rem9 3 == 3
rem9 12 == 3
rem9 18 == 9
```

Exercice 4. Écrivez la fonction :

```
sumDigits :: [Integer] -> Integer
```

qui calcule la somme de touts les nombres d'une liste de chiffres. Exemples :

```
sumDigits [1,1,8,1,0,4] == 15
sumDigits [1,2,3,4] == 10
```

Exercice 5. Écrivez la fonction :

```
validate :: Integer -> Bool
```

qui vérifie si un entier peut être un nombre de carte de payement valide. Ceci utilisera toutes les fonctions précédentes. En particulier, n'oubliez pas que les chiffres doublés supérieurs à `10` doivent être remplacés par des nombres à un chiffre. Exemples :

```
not validate 4012888888881881
not validate 4012888888881882
validate 4012888888881892
```

Exercice 6. Écrivez la fonction :

```
test :: [Bool] -> Bool
```

qui reçoit les résultats d'un test et qui retourne vrai si tous les résultats du test sont vrai et faux sinon. Exemple :

```
test [True, True, True] == True
test [True, False ] == False
```

Écrivez la fonction :

```
testAll :: [Char]
```

qui retourne la chaine de caractères `"Success!"` si tous les résultats des tests de toutes les fonctions du devoir sont vraies et `"Fail!"`, sinon.