TP 1 : Différences

©2023 Ghiles Ziat
ghiles.ziat@epita.fr

Exercice I : Premier Programme

Q1 – Créez deux programmes *hello.hs* et *hello.lisp* qui affichent sur la sortie standard la chaîne de caractère "Hello world"

Q2 – Compilez et exécutez les deux programmes.

Remarque :

- Pour compiler un (ou plusieurs) fichier(s) source(s) Haskell, il faut lancer dans un terminal la commande `ghc file.hs`. On peut ensuite lancer l'exécutable produit en faisant `./file`
- Pour compiler et évaluer un (ou plusieurs) fichier(s) source(s) lisp, il faut lancer dans un terminal la commande `sbcl --script file.lisp`.
- En haskell, **il faut qu'un 'main' soit défini** comme le point d'entrée du programme
- Vous pourrez utiliser la fonction `putStrLn` en Haskell et la fonction `write-line` en lisp pour les affichages

Exercice II : Récursion

On désire avoir une fonction `sumDigits` qui calcule la somme des chiffres d'un nombre. Par exemple, `sumDigits 1234` doit retourner 10 (4+3+2+1). Une façon *impérative* de **faire** ce calcul est donnée par :

```
sumDigits(n){
  sum = 0;
  while (n != 0) do
    sum = sum + (n % 10);
    n = n/10;
  done;
  return sum
}
```

Une façon *fonctionnelle* de **définir** ce calcul est de dire que la somme des chiffres d'un entier n est égale à n si $n < 10$, sinon, c'est égal au chiffre des unités de n auquel on ajoute la somme des chiffres

restants de n (Notez que la définition est naturellement récursive).

Q1 – Écrivez une fonction récursive avec ce comportement (on pourra utiliser les fonctions `div` et `rem` en Haskell, ainsi que les fonctions `floor` et `rem` en lisp pour la division euclidienne son reste)

Q2 – Un nombre est divisible par 3 si

- il est plus petit que 10 et que c'est 0, 3, 6 ou 9
- il est plus grand ou egal à 10, et la somme de ses décimales est elle même divisible par 3

Définissez une fonction récursive qui implémente cette méthode de calcul.

Exercice III : Logique

On rappelle ici les tables de vérité de certains opérateurs booléens classiques (et, ou et non). Vrai (resp. faux) est identifié par $\top$ (resp. $\bot$) :

$\wedge$	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

$\vee$	$\top$	$\bot$
$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$

$\neg$	
$\top$	$\bot$
$\bot$	$\top$

On rappelle de plus que l'implication ($\Rightarrow$), l'équivalence ($\Leftrightarrow$) et le xor ($\oplus$) sont définis par :

- $A \Rightarrow B \equiv (\neg A) \vee B$
- $A \Leftrightarrow B \equiv A \Rightarrow B \wedge B \Rightarrow A$
- $A \oplus B \equiv (A \vee B) \wedge \neg(A \wedge B)$

Q1 – Définissez les opérateurs `andOp` et `orOp`, de type `bool -> bool -> bool` en utilisant un *if-then-else*.

Q2 – Définissez les opérateurs `imply`, `equiv` et `xor` effectuant respectivement les opérations ($\Rightarrow$, $\Leftrightarrow$, $\oplus$).

Q3 – Comparez le résultat des appels en Haskell et en Lisp (les appels sont écrits en Haskell, vous utiliserez la valeur lispienne `t` au lieu de `True`) :

- `orOp True (error "failure")`
- `orOp (error "failure") True`

Que constatez vous ?

Exercice IV : Arithmétique

La conjecture de Syracuse [1] est un problème non résolu en mathématiques qui demande si la répétition de deux opérations arithmétiques simples finira par transformer chaque entier positif en 1.

1. `https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse`

Q1 – Considérons l'opération suivante sur un nombre entier strictement positif : s'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1. Écrivez une fonction qui implémente ce comportement.

Q2 – En répétant l'opération, on obtient une suite d'entiers strictement positifs dont chacun ne dépend que de son prédécesseur. Écrivez une fonction qui construit cette liste jusqu'à tomber sur la valeur 1.

Q3 – Testez votre fonction et affichez son résultat sur l'entrée 10.

Exercice V : Listes

Le tri fusion [2] est un algorithme de tri par comparaison stable qui consiste à diviser une liste en sous-listes, à trier séparément ces sous-listes et à fusionner leur résultat.

Q1 – Définissez une fonction qui divise une liste en deux sous-listes de tailles égales (à un près). Vous aurez très probablement besoin de stocker le résultat d'un appel récursif dans une variable [3]. Pour cela, vous pourrez utiliser un `let binding` qui associe des valeurs à des noms en utilisant soit la syntaxe lispienne `(let ((a (+ 20 1)) (b 2)) (* a b))` ou l'Haskellienne `let a = 20 + 1 in let b = 2 in a*b`, qui déclare deux variables "a" et "b" et qui les utilise dans l'expression "a*b"

Q2 – Définissez une fonction qui à partir de deux listes triées, construit une liste triée comportant les éléments issus de ces deux listes.

Q3 – Implémentez la fonction de tri suivante :

- si la liste est vide ou contient un seul élément, alors elle est déjà triée.
- sinon,
  - diviser la liste en deux sous liste de même taille (à un près),
  - trier les deux sous listes séparément,
  - fusionner les sous-listes triées en maintenant l'ordre.

2. `https://fr.wikipedia.org/wiki/Tri_fusion`
3. Notons qu'en programmation fonctionnelle, le terme variable est utilisé au sens mathématique du terme. Les variables étant immutables, leur contenu ne ... varie pas