TP1. Interprète – let- Récursivité

Remarques générales :

Déroulement des TP

- Avant de démarrer les TP, un répertoire CS212 doit être créé sous le répertoire home (Linux)

- A chaque séance de TP, un répertoire correspondant (TP1, TP2 ou TP3) sera créé sous le répertoire CS212.

- La séance de TP se déroule par défaut sous le répertoire correspondant (tout changement de répertoire courant sera indiqué explicitement).

Evaluation des TP

Chaque sujet de TP est à traiter sur 2 séances de 1h30 chacune. Un CR de TP sera rendu 5 jours au plus tard après la deuxième séance. Ce CR doit se présenter sous la forme **d'un unique fichier pdf** comportant les éléments suivants :

- Une description de vos observations (pour les exercices d'observation),

- les programmes sources et les captures d'écran montrant les résultats d'exécution pour les exercices demandant l'écriture de programmes Haskell.

- Ce fichier sera téléchargé sur l'espace Travaux de Chamilo.

Objectifs du TP1 :

1. Prise en main de l'interprète `ghci`
2. Utilisation de la commande `let`
3. Introduction à la récursivité
4. Définitions de fonctions

Exercice 1

1. L'environnement `ghci` permet l'évaluation et l'interprétation des expressions Haskell. Sur les machines de l'Esisar, cet environnement est déjà installé. Sur vos machines personnelles, vous pouvez installer cet environnement à partir du lien suivant : https://www.haskell.org/platform/

2. Lancez l'exécution de `ghci` en utilisant la commande `ghci` sous Linux.

3. Tapez la commande :

```
> :?
```

Observez le résultat affiché.

4. Evaluez les expressions suivantes et observez les résultats obtenus. A chaque fois que cela est adéquat, analysez en fonction du résultat l'application des règles de précédence des opérateurs arithmétiques :

```
> 30 * 12
> pi
> 4 ^ 2
> (+) 4 6
> (-) 5 2
> div 19 2
> 19 `div` 2
> 19/2
> ((3 * 4) / 12) + 36
> True && False
> True && 1
> False || False
> not False
> not (True || True)
> reverse "Hello World"
> reverse [1 .. 5]
> odd 15
```

```
> even 17

> succ 7

> 3 == 3

> 3 /= 3

> 3 /= 6

> 5 * 4 ^ 2

> 4 ^ 5 ^ 2

> div 19 (succ 2)

> pred (succ 9)

> :type 9

> :type "Ceci est mon premier TP Haskell"
```

Exercice 2

Donner les résultats des suites de commandes suivantes:

```
> let x = 67
> let y = x * 2
> let a = 12
> div y a


> let as = [1 .. 10]
> let bs = ['a', 'b', 'c', 'd', 'e']
> zip as bs

> let h = head [1 .. 10]
> let t = tail [1 .. 10]
> h
> t

> let f = (+3)
> let r = f 7
> r

> let addOne = map (+1)
> addOne [1 .. 5]
```

Exercice 3

1. En utilisant le mot-clé `let`, définissez une fonction `double` qui permet de dupliquer un argument numérique. Testez la fonction `double` sur les arguments suivants : 5, 6, 9.23

2. En utilisant un éditeur de texte, écrivez un programme `exo3.hs` qui contient la fonction `double`. Cette fonction permet de dupliquer un argument numérique. Testez la fonction `double` avec les arguments suivants : 5, 6, 9.23

3. Modifiez le fichier `exo3.hs` de sorte à rajouter la définition de la fonction `quadruple` qui permet de quadrupler un argument numérique. Définir la fonction `quadruple` en utilisant la fonction `double`.

Testez la fonction `quadruple` avec les mêmes arguments que ci-dessus.

Exercice 4

On considère un système dont les trois couleurs primaires sont le rouge, le bleu et le jaune. A partir de ces couleurs, on peut facilement obtenir d'autres couleurs en appliquant des mélanges. Par exemple, le vert s'obtient avec le mélange bleu et jaune, l'orange s'obtient en mélangeant le rouge et le jaune alors que le violet s'obtient en mélangeant le rouge et le bleu.

1. Dans un fichier de noms `couleurs.hs`, définissez un nouveau type énuméré `Couleur` qui comprend toutes les couleurs citées ci-dessus.

2. Définir une fonction **`primaire :: Couleur → Bool`**, qui indique si une couleur est primaire ou pas.

3. Définissez la fonction `defcl` dont le prototype est :

```
defcl :: Couleur → String
```

Pour chaque couleur du type `Couleur`, la fonction `defcl` affiche un message spécifique. Pour les couleurs de base, ce message indique simplement la couleur en question. Pour les couleurs secondaires, ce message indique le mélange des couleurs de base qui a permis d'obtenir la couleur secondaire correspondante.

4. Testez votre programme pour toutes les valeurs du type `Couleur`.

Exercice 5

1. Dans un fichier de nom `puissance.hs`, définissez la fonction `puissance` en utilisant la description récursive ci-dessous :

```
puissance a 0 = 1
puissance a n = a * (puissance a (n-1))
```

Testez cette fonction avec les expressions suivantes :

```
> puissance 7 1

> puissance 6 0

> puissance 16 2
```

Sur le même modèle que la fonction puissance ci-dessus, définir de manière récursive les fonctions suivantes :

- Fonction Factorielle
- somme des n premiers entiers
- somme des carrés des n premiers entiers
- Fonction Fibonacci

Ces fonctions seront définies dans des fichiers de noms `factorielle.hs`, `somme.hs`, `sommeCarres.hs`, `fibonacci.hs` respectivement.

Exercice 6

1. Quel est l'effet de la définition "let z = (>=) 3"? Pour le savoir, évaluez z sur des entiers...

La fonction `map`, de type `(a -> b) -> [a] -> [b]`, prend une fonction `f` transformant des éléments de type `a` en des éléments de type `b`, et une liste `l` d'objets de type `a` ; elle renvoie une liste d'objets de type `b`, par application de la fonction `f` aux éléments de la liste `l`.

2. Pouvez-vous prédire (ou expliquer) la valeur de l'expression

```
> map z [1, 2, 3, 4, 5]
```

Donnez à `ghci` la définition: `let app f = map f [1, 2, 3, 4, 5]`

3. Comment appeler `app` pour obtenir la même valeur qu'à la question précédente ?

4. Faites un appel à `app` qui illustre l'incrémentation de 1 de chaque élément de la liste.

5. Faites un appel à `app` qui illustre l'ajout de 3 à chaque élément de la liste.

6. Définissez une fonction, et faites un appel à `app` qui élève chaque élément de la liste au carré.

Exercice 7

1. Ecrire une fonction `repeter` qui prend en arguments un entier `n` et un caractère `c` et qui renvoie la chaîne obtenue en répétant le caractère `c` `n` fois de suite.

2. Définir la fonction `etoiles` qui prend en argument un entier `n` et une chaîne et qui entoure la chaîne de `n` étoiles avant et après.

3. Définir la fonction `slashes` qui entoure une chaîne avec deux « slashes ».

4. En utilisant la composition de fonctions avec l'opérateur (.) écrire une fonction `commentaireC` qui prend une chaîne et qui en fait un commentaire C en rajoutant 12 étoiles de part et d'autre de la chaîne et en entourant le tout de slashes. Utiliser pour cela les résultats des questions précédentes.

Exercice 8

1. Ecrire une fonction `echange` de type `(a->a->a) -> (a->a->a)`, qui échange l'ordre des paramètres de la fonction passée en argument.

2. Utiliser `echange` et la fonction puissance (^) pour définir la fonction `cube`.

Remarque : il n'est pas demandé d'utiliser une définition de fonction à paramètre.

3. Définir de même la fonction `moinsDeux` qui retire 2 à un nombre.