Programmation fonctionnelle

6 novembre 2023

Exercice 1 : Interpréteur `ghci`

Les directives de `ghci` sont listées dans `https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/ghci-commands.html`.

(a) À l'aide de la directive `:type`, examinez les types des expressions suivantes.

1. `1`
2. `1 :: Int`
3. `1 :: Integer`
4. `1 :: Float`
5. `1 :: Double`
6. `1.0`
7. `1.0 :: Int`
8. `1.0 :: Integer`
9. `1.0 :: Float`
10. `1.0 :: Double`

(b) Évaluer les expressions suivantes dans l'interpréteur.

1. `21 + 2`
2. `(+) 21 2`
3. `21+2*3`
4. `(+) 21 (2*3)`
5. `(+) 21 ((*) 2 3)`
6. `1 == 2`
7. `1 /= 2`
8. `1 + (-1)`
9. `div 11 2`
10. `11 `div` 2`
11. `42 `mod` 2 == 0`
12. `even 42`
13. `odd 42`
14. `pred 42`
15. `succ 42`

(c) À l'aide de la directive `:type`, examinez les types des expressions suivantes.

1. `'a'`
2. `"abc"`
3. `[]`
4. `['a']`
5. `['a', 'b']`

6. `["abc"]`
7. `["abc", "def"]`
8. `['a', "abc"]`
9. `('a', 'b')`
10. `("abc", "def")`
11. `('a', "abc")`
12. `('a', "abc", 1)`
13. `[odd, even]`
14. `[odd, even, (==) 1]`
15. `[odd, even, mod]`
16. `(odd, even, mod)`

Exercice 2 : Liaisons locales

Dans le paradigme fonctionnel, la notion de variable (et donc d'affectation) n'existe pas. Pour nommer des expressions, on utilise le mécanisme de liaison. La liaison n'est pas une affectation mais elle en a certaines propriétés. Pour réaliser une liaison locale, on utilise la syntaxe suivante :

```
let nom = expr1 in expr2
```

Ceci est une expression et possède donc une valeur : c'est la valeur de `expr2` dans laquelle toutes les occurrences (libres) de nom sont remplacées par `expr1`. Par exemple :

```
λ > let x = 1 in x + 2
3
λ > let x = 1 in (x + 2, x + 3)
(3,4)
```

Il est bien sûr possible de faire plusieurs liaisons dans une même expression.

```
λ > let x = 1; y = 2 in x + y
3
λ > let x = 1 in let y = x + 2 in x + y
4
λ > let x = 1; y = x + 2 in x + y
4
λ > let x = 1 in let y = x + 2; z = x + 3 in x + y + z
8
λ > let x = 1; y = x + 2; z = x + 3 in x + y + z
8
```

(a) Déterminer les résultats des expressions suivantes.

```
let x = 1 in let x = 2 in x
let x = 1; y = x in let x = 3 in x + y
let x = 1; y = x; z = x + y in let x = 3; y = 2*x in (x, y, z)
let x = y + 1; y = x + 2 in (x, y)

let x = x*x in x
let x = x*x in let x = 1 in x
let x = 1 in let x = x*x in x
```

(b) Considérons un cylindre de hauteur h et un rayon r. Commencer par calculer l'aire d du disque qui forme la base du cylindre (c.-à-d. πr^2) en une seule expression et en utilisant une liaison locale pour r.

(c) Écrire une expression permettant de calculer le triplet (p, s, v) où p est le périmètre de la base (c.-à-d. $2\pi r$), s est la surface totale (c.-à-d. $2d + ph$), et v son volume (c.-à-d. dh). Bien entendu, les expressions apparaissant dans plusieurs calculs différents ne doivent être calculées qu'une seule fois.

Exercice 3 : Expressions conditionnelles

La structure de contrôle conditionnelle en haskell possède la syntaxe suivante :

```
if expr1 then expr2 else expr3
```

Attention, il s'agit d'une expression, elle a donc un type et une valeur. L'expression `expr1` est de type `Bool` et les expressions `expr2` et `expr3` doivent être du même type (qui est également le type de l'expression complète).

```
λ > if True then "true message" else "false message"
"true message"
λ > if False then "true message" else "false message"
"false message"
λ > :type if True then "true message" else "false message"
if True then "true message" else "false message" :: String
λ > :type if False then "true message" else "false message"
if False then "true message" else "false message" :: String
```

(a) Calculer le maximum entre les entiers x et y (dans une liaison locale).

(b) Calculer le maximum entre les entiers x, y et z (dans une liaison locale).

(c) Étant donné un entier x, en utilisant une expression conditionnelle, donnez le code permettant d'obtenir $\lceil x/2 \rceil$. Rappelons que $\lceil \ldots \rceil$ est la fonction « *partie entière supérieure* ».

(d) Étant donnés trois entiers x, y et z, calculer l'expression

$$\begin{cases} \min(x,y)^2 + 1 & \text{si } z \text{ est divisible par } 3 \\ \min(x,y)^2 & \text{sinon} \end{cases}$$

Exercice 4 : Premières fonctions

Pour le chargement de code dans `ghci`, le faut utiliser les directives `:load` et `:reload`.

Conseil : pour cet exercice, lorsque c'est possible, vous avez intérêt à essayer d'écrire plusieurs versions de fonctions demandées, en essayant d'éviter d'utiliser `if` et avec ou sans gardes.

(a) Écrire la fonction

```
myAverage :: Float -> Float -> Float -> Float
```

qui calcule la moyenne de 3 entiers.

(b) Écrire les fonctions

```
myMin2 :: Int -> Int -> Int
```

et

```
myMax2 :: Int -> Int -> Int
```

qui calculent le minimum et le maximum de deux entiers. Écrire ensuite les fonctions

```
myMin3 :: Int -> Int -> Int -> Int
```

et

```
myMax3 :: Int -> Int -> Int -> Int
```

qui calculent le minimum et le maximum de trois entiers.

(c) En tirant partie de $x + 0 = x$ et $x + y = (x + 1) + (y - 1)$, écrire la fonction

```
myAdd :: Int -> Int -> Int
```

qui calcule la somme $x + y$. Nous supposerons dans cet exercice que $y \geqslant 0$. Essayer sans utiliser les opérateur $+$ et $-$

(d) En tirant partie de $x \times 0 = 0$ et $x \times y = x \times (y - 1) + x$, écrire la fonction

```
myMult :: Int -> Int -> Int
```

qui calcule le produit $x \times y$. Nous supposerons dans cet exercice que $y \geqslant 0$. Vous écrirez ensuite une version de `myMult` en utilisant la fonction `myAdd`.

(e) Écrire la fonction

```
myFact :: Int -> Int
```

qui calcule la factorielle $n!$. Nous supposerons dans cet exercice que $n \geqslant 0$.

(f) Si vous essayez avec 10 puis 100, vous aurez sûrement :

```
λ > myFact 10
3628800
λ > myFact 100
0
```

Écrire une nouvelle version de la fonction `myFact` pour résoudre ce problème.

(g) Écrire la fonction

```
myGCDEuclid :: Int -> Int -> Int
```

qui calcule le plus grand diviseur commun de x et y en utilisant l'algorithme d'Euclide : Étant donnés deux entiers x et y positifs,
- si $x = y$ alors le plus grand diviseur de x et y est x.
- si $x > y$ alors le plus grand diviseur commun de x et de y est égal au plus grand diviseur commun de $x - y$ et y.

(h) Écrire la fonction

```
myGCDEuclidean :: Int -> Int -> Int
```

qui calcule le plus grand diviseur commun de x et y en utilisant une version différente de l'algorithme d'Euclide : Étant donnés deux entiers x et y positifs,
- si $y = 0$ alors le plus grand diviseur de x et y est x.
- sinon, le plus grand diviseur commun de x et de y est égal au plus grand diviseur commun de y et du reste de la division entière de x par y.

(i) Écrire la fonction (attention au typage)

```
myAverage2 :: Int -> Int -> Int -> Float
```

qui calcule la moyenne de 3 entiers.