Correction PFON 2014

Kévin “Chewie” Sztern

Haskell

1. Expliquer :

```
g :: Int -> [ Double ]
g i = 1 / fromIntegral (i)^2 : g (i + 1)
```

Cette fonction construit la suite (sous forme de list) des 1/n^2, en commençant au rang i.

Exemple pour i = 42 :

```
[ 1/42^2, 1/43^2, 1/44^2 ...]
```

2. Que valent (g 1 !! 0) et (g 1 !! 1) ?

- g 1 !! 0 = 1/1^2 = 1.0
- g 1 !! 1 = 1/2^2 = 0.25.

3. Pourquoi que ça marche bien en Haskell ?

Parce que Haskell est lazy (paresseux pour les mangeurs de camembert) : les valeurs ne sont évaluées que quand on en a besoin, donc on peut manipuler une liste infinie tant qu'on demande pas à l'évaluer dans son entièreté.

4. Expliquer le principe de fonctionnement de ce machin:

```
f :: Double -> [Double] -> Double -> Double
f x y e
    | head y < e = x
    | otherwise = f (x + head y) (tail y) e
```

La fonction accumule dans x la somme des éléments de la liste y jusqu'à tomber sur un élément plus petit que e (notre epsilon).

5. Comment ça s'appelle ce type de branchement ?

Ça s'appelle des gardes, mon couillon.

6. Au vu de tout ça, on définit approx comme ça:

```haskell
approx :: Double -> Double
approx e = f 0 (g 1) e
```

En déduire la méthode d'approximation mathématique utilisée.

C'est le problème de Bales à l'envers: on approxime pi^2/6 par la somme infinie de la suite des 1/n^2, jusqu'à une précision de e (genre e = 0.001).

Lisp, take 1

1. DIfférence entre Lisp et Haskell mise en évidence ?

J'ai du mal à deviner ce qu'il veut entendre. Trucs obvious: pas de pattern matching à base de gardes, pas d'information de typage parce que c'est typé dynamiquement (on aurait pu les omettre en Haskell though), les opérateurs sont préfixes, y'a des parenthèses partout...

L'histoire du lazy vs eager est développé sur les questions suivantes, donc j'en parle pas là.

2. Soit la fonction g en lisp:

```lisp
(defun g (i)
    (cons (/ 1 (* i i)) (g (1+ i))))
```

Que vaut (g 1) ?

Ça vaut un joli stack overflow.

3. Pourquoi ?

Parce que lisp est eager, contrairement à Haskell: on évalue dès qu'on peut, donc on part en récursion infinie.

4. Par conséquent, est-ce que notre merde marche ?

À ton avis, banane ?

Lisp, take 2

1. Qu'est-ce qu'une fonction d'ordre supérieur ?

C'est une fonction qui est fonction d'une autre fonction (ainsi font, font, ction...), ou qui renvoie une fonction.

(Si vous me piquez cette blague au partiel je vous tue)

2. Que représente ce machin ?

```
(lambda (x) (* 2 x))
```

C'est une fonction anonyme (lambda quoi) qui prend un nombre et le multiplie par 2.

3. Que vaut donc:

```
((lambda (x) (* 2 x)) 4)
```

C'est l'application de la fonction sur 4, donc ça vaut 2 * 4 = 8.

4. Mais il dit "graisse". On redéfinit g comme ça:

```
(defun g (i)
    (lambda (x)
        (cond ((= x 0)
                (/ 1 (* i i)))
              ((= x 1)
                (g (1+ i))))))
```

Que vaut (g 1) ?

Ça renvoie une fonction qui prend un x, et selon sa valeur (0 ou 1), renvoie respectivement soit 1, soit (g 2).

5. 2 caractéristiques de lisp qui permettent de faire ça ?

hm.. lambda fonctions et closures ?

Petit rappel: une closure, c'est quand on renvoie une fonction qui se sert d'une variable temporaire.

Par exemple:

```
(defun create-adder (i)
    (lambda (x) (x + i)))
```

La fonction qu'on renvoie se sert de i, qui est censé disparaitre à la fin de create-adder. Résultat, la durée de vie de i est étendue à celle de la fonction qu'on a crée. Ça permet de faire des trucs marrants.

6. Soient les fonctions suivantes:

```
(defun fcar (f)
    (funcall f 0))

(defun fcdr (f)
    (funcall f 1))
```

Que valent (fcar (g 1)) et (fcar (fcdr (g 1))) ?

Bon, comme vous l'avez compris, le but de tout ce bordel c'est de simuler les listes infinies en manipulant des fonctions qui renvoient potentiellement le terme suivant de la "liste". Ces fonctions s'appellent comme ça par référence aux fonctions car et cdr, qui renvoient respectivement la tete et la queue d'une vraie liste. Donc ces fonctions font le meme comportement pour notre pseudo liste, en appelant nos lambdas avec le paramètre kivabien pour soit renvoyer le terme courant, soit renvoyer la lambda qui nous filera le terme suivant.

En conclusion:

- (fcar (g 1)) renvoie la tete de notre "liste", cad 1
- (fcar (fcdr (g 1))) renvoie le terme juste après, cad 0.25

7. En conclusion, comment on réécrit f ?

On change juste car par fcar et cdr par fcdr, olol c'est magique.

```
(defun f (x y e)
    (cond ((< (fcar y) e) x)
    (t (f (+ x (fcar y)) (cdr y) e))))
```

Bonus !

Approximer notre valeur jusqu'au rang i de la suite, en une ligne.

```
superApprox i = sum $ take i [1 / x^2 | x <- [1..]]
```

51 caractères, on va dire que ça passe.