EPITA ING1 2014 S2 PFON

Didier Verna

Documents et calculatrice interdits

Toute réponse non justifiée sera comptée comme nulle.
Il est préférable de ne rien répondre que de tenter d'inventer d'importe quoi.
Toute tentative de bluff sera sanctionnée par un malus.

Dans ce partiel, on se propose d'implémenter un programme fonctionnnel pur d'approximation de $\frac{\pi^2}{6}$ avec une précision arbitraire, car dans la vie, il est évident que cela sert tous les jours.

1 Haskell

1. Expliquez les deux lignes de code Haskell suivantes (soyez synthétiques dans vos explications) :

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

2. On rappelle que l'opérateur d'indexation est `!!` en Haskell. Que valent les expressions `(g 1 !! 0)` et `(g 1 !! 1)` ?
3. Quelle caractéristique fondamentale d'Haskell permet à ce type de code de fonctionner correctement ?
4. Expliquez le principe de fonctionnement du code Haskell ci-dessous (quelques lignes suffisent).

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

5. Comment s'appelle cette forme de branchement conditionnel ?
6. Compte tenu de ce qui précède, on défini notre fonction d'approximation de $\frac{\pi^2}{6}$ de la manière suivante :

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

Déduisez-en la méthode d'approximation mathématique utilisée.

2 Lisp, take I

Nous allons maintenant développer une version sémantiquement équivalente en Common Lisp, principalement parce que la nature adore les parenthèses, et nous aussi.

1. Les équivalents Lispiens de `approx` et `f` sont donnés ci-dessous :

```
(defun approx (e)
  (f 0 (g 1) e))

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

   Quelle différence fondamentale entre les deux langages est mise en évidence ici (comparez avec les versions Haskelliennes) ?

2. Le programme ci-dessous fourni à son tour l'équivalent Lispien de `g`.

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

   Que vaut l'expression `(g 1)` ?

3. Quelle caractéristique fondamentale de Lisp justifie votre réponse à la question précédente ?
4. Par conséquent, `approx` peut-elle fonctionner correctement (de la même manière qu'en Haskell) ?

3 Lisp, take II

Je ne sais plus si je vous l'ai déjà dit, mais en Lisp, on peut tout faire. Il est donc bien entendu possible de s'affranchir du problème mis en évidence dans la section 2. C'est ce que nous allons faire maintenant. Pour ceux qui n'ont pas déjà quitté l'amphi, notez bien que la solution proposée ici ne nécessite *même pas* de changer la technique d'évaluation utilisée par le langage. La solution proposé se base uniquement sur les aspects fonctionnels du langage.

1. Définissez succintement ce que l'on appelle une fonction d'ordre supérieur.
2. Que représente l'expression suivante en Lisp ?

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

3. Que vaut par conséquent l'expression suivante ?

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

4. Mais je digresse (« gresse »). Revenons à nos moutons. Nous proposons de redéfinir `g` de la manière suivante :

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

   Que vaut l'expression `(g 1)` ?

5. Quelles sont les deux caractéristiques des languages fonctionnels qui nous permettent d'écrire une telle expression ?

6. Supposons maintenant que nous définissions les fonctions suivantes (ce n'est pas pour vous embrouiller, hein, c'est pour vous aider) :

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

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

   Que valent les expressions `(fcar (g 1))` et `(fcar (fcdr (g 1)))` ?

7. En fonction de ce qui précède, indiquez les 3 modifications à apporter à `f` pour obtenir un programme sémantiquement équivalent à la version Haskellienne, mais qui fonctionne correctement cette fois-ci.

4 Bonus

Comme cette année, je n'ai pas arrêté de délirer tout au long du partiel, une fois n'est pas coutume, la question bonus n'est pas une blague...

Seriez-vous capable d'ecrire en une seule ligne de code (en fait, il vous faut moins de 50 caractères) la fonction Haskell `superApprox` qui calcule l'approximation de $\frac{\pi^2}{6}$ au rang `i` de la suite mathématique utilisée dans le partiel ?

Le prototype de `superApprox` est donc le suivant :

```
superApprox :: Int -> Double
```

Indication : vous pouvez partir de la liste (infinie) des entiers qui est notée `[1 .. ]`.

Bon. Sur ce, moi, je suis fatigué d'écrire des partiels.
Là, je vais aller me coucher.
Allez, ciao.