Programmation Fonctionnelle I, Printemps 2017 – TD2

http://deptinfo.unice.fr/~roy

*Rappel : la **signature** d'une fonction est la donnée de son domaine de départ et de son domaine d'arrivée, ce que les matheux dénotent par une flèche comme E x F → G.*

Exercice 2.1 a) Comment s'écrirait en Scheme la fonction anonyme : $(x,y) \mapsto x + \sqrt{y-1}$?
b) Comment calculeriez-vous sa valeur en `(x, y) = (1, 0.6)`, sans lui donner de nom ? Cette valeur est-elle réelle ?
c) Nommez `h` cette fonction, de deux manières différentes.

Exercice 2.2 a) Quelle est la différence entre les mots « paramètre » et « argument » pour une fonction ?
b) Quelle est l'arité de la fonction `expt` ? de la fonction `cos` ? de la fonction `max` ?
c) Donnez l'exemple d'une fonction anonyme d'arité 3 [*ternaire*].
d) Donnez l'exemple d'une fonction dont la signature soit : `Number → Fonction`
e) Donnez l'exemple d'une fonction dont la signature soit : `Number x Fonction → Number`
f) Donnez l'exemple d'une fonction dont la signature soit : `Fonction x Fonction → Number`
g) Donnez l'exemple d'une fonction dont la signature soit : `Fonction x Fonction → Fonction`

Exercice 2.3 Comment pouvez-vous améliorer l'infiniment pauvre qualité de la définition ci-dessous ?

```
(define (truc x y)
(if (ln (x) > y);                    ln est le logarithme népérien
(x + y)
(ln (x))))
```

Exercice 2.4 a) Définissez en Scheme la fonction `f(x,y) = x + |y|` dans les réels, sans utiliser la fonction *valeur absolue* qui se nomme `abs`, mais un `if`.
b) Sauriez-vous mettre x en facteur dans la solution à la question précédente [de sorte que la lettre x n'apparaisse plus qu'une fois dans la solution] ?
c) Définissez la fonction `g(x,y) = Max(x,y)` dans les réels, avec un `if`, sans utiliser la fonction *maximum* qui se nomme `max`.
d) Même question que c) mais sans `max`, sans `if` et sans `cond`, en utilisant cette fois la *valeur absolue* ?

Exercice 2.5 Donnez une définition possible de la fonction `foo` de sorte que `((foo 3 4) 5)` ait pour valeur 8.
Posez-vous la question : quelle est une signature possible de `foo` ?...

Exercice 2.6 Le tirage de **nombres aléatoires** est très fréquent en programmation. La fonction primitive `random` retourne un nombre aléatoire et s'utilise sous l'une des deux formes suivantes :

```
(random)      ; retourne un nombre réel inexact aléatoire de ]0,1[, par exemple #i0.3458671190171411
(random n)    ; retourne un entier exact aléatoire de [0,n-1] si n est un entier exact ≥ 1
```

a) Quelles sont les valeurs possibles de `(random 5)` ?
b) Définissez une fonction `(randomInt a b)` prenant deux entiers `a` ≤ `b` et retournant un entier aléatoire exact de `[a,b]`.

N.B. *Si vous avez besoin de grands entiers aléatoires [des dizaines de chiffres], utilisez `(srandom n)` au lieu de `(random n)`.*

Exercice 2.7 Définissez une **structure** de balle, avec trois champs : la position p [un *posn*] et les deux composantes de la vitesse dx et dy. Quelles sont les 5 fonctions automatiquement définies par Racket ? Construisez une balle située en (5,8) de vitesses dx=0.5 et dy=4. Demandez à voir la somme x + dx.

Exercice 2.8 *http://mitpress.mit.edu/sicp (full text online, bottom of section 1.1.6, exercise 1.5).*
Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using **applicative-order evaluation strategy** *[cours 1 page 18]*. He defines the following two procedures :

```
(define (p)
  (p))

(define (test x y)
  (if (= x 0) 0 y))
```

Then he evaluates the expression `(test 0 (p))`. What behavior will Ben observe with an interpreter that uses applicative-order evaluation ?

Programmation Fonctionnelle I, Printemps 2017 – TP2

http://deptinfo.unice.fr/~roy

Exercice 2.1 Programmez la fonction `(einstein u v)` : loi d'addition des vitesses en **relativité restreinte**, avec la vitesse de la lumière c ≈ 300000 km s^{-1} que l'on définira à part :

$$(u,v) \mapsto \frac{u+v}{1+\frac{uv}{c^2}}$$

A.N. Rien ne vaut un problème concret : un voyageur court à 250000 km.s^{-1} dans le couloir d'un train, dans le sens de la marche. Et le train roule à 270000 km.s^{-1}. Quelle est la vitesse du voyageur par-rapport à la voie ?... Attention, en Physique, un résultat doit être un nombre inexact "à virgule", pas un rationnel p/q !

Exercice 2.2 Ecrire une fonction `(stirling n)` retournant pour `n` grand une valeur approchée de `n!` en utilisant la **formule de Stirling**. Comparez avec la vraie factorielle pour `n=5 [5! = 120]` puis `n=150` [le quotient des deux tend vers 1 lorsque n tend vers +∞]. *Mais ne prenez pas n trop grand, le plus grand nombre inexact est environ `1.17976e+308`.*

$$n! \approx \sqrt{2n\pi}\left(\frac{n}{e}\right)^n$$

Exercice 2.3 *Relisez l'exercice 2.6 du TD, puis :*

a) Définissez une fonction `(rand-impair)` retournant un entier aléatoire impair de [0,100].

b) Donnez une expression dont la valeur soit un nombre réel inexact aléatoire de]15,40[.

c) Donnez une expression dont le résultat soit un nombre rationnel exact aléatoire de]10,20[.

d) Définissez une fonction `(cannes)` retournant aléatoirement `3 ou 5`.

e) Définissez une fonction `(monte-carlo)` retournant aléatoirement `2`, `3` ou `5`.

f) Définissez une fonction `(las-vegas)` retournant `2, 3 ou 5` mais de manière biaisée : `2` avec la probabilité `2/7`, `3` avec la probabilité `1/7` et `5` avec la probabilité `4/7`.

Exercice 2.4 Reprenons la **dérivation numérique approchée** [cours 2 page 16] où la fonction `(derivee f x)` retournait une valeur approchée de `f'(x)`.

a) Rappelez comment on utilise cette fonction pour obtenir une approximation de la *dérivée première* log'(2).

b) Un peu plus délicat, montrez comment l'on peut obtenir une approximation de la *dérivée seconde* log"(2). Vous devez obtenir un résultat voisin de –1/4 ≈ -0.25 n'est-ce pas ?

c) Et encore un peu plus intellectuel. La fonction `dérivée` ci-dessus retournait un nombre réel. Or les matheux parlent de *la fonction dérivée* tout court. Sauriez-vous programmer la fonction `(deriv f)` retournant la fonction `f'` ? Testez votre définition en résolvant à nouveau la question a) avec la fonction `deriv`. Puis faites de même avec la question b).

Exercice 2.5 Donnez une définition possible de la fonction `bar` de sorte que la valeur de `((bar 5))` soit égale à 10.

Exercices Supplémentaires

Exercice 2.6 En traitement du signal, le *lissage d'une fonction* est une technique très utilisée. Si f est une fonction et h une quantité proche de 0 [par exemple 0.01], la version lissée de f est une nouvelle fonction dont la valeur en `x` est.la moyenne entre `f(x-h)`, `f(x)` et `f(x+h)`. Définissez la fonction `(lissage f h)` retournant la fonction lissée de f. Pour h = 0.01, que vaut la version lissée de la valeur absolue en x = 0 ?

Exercice 2.7 a) Programmez *par récurrence* sur n ≥ 0 la fonction `(u n)` retournant le terme u_n de la suite (u_n) étudiée dans le cours 2 pp 13-14. Comparez la valeur de `(u 8)` avec la valeur attendue $\sqrt{2}$ [convergence rapide !]. Complétez le schéma :

```
(define (u n)       ; {entiers ≥ 0} → réel
  (if (= n 0)
      ...
      ...))
```

b) Le nombre d'opérations effectuées par la fonction *u* pour calculer u_n est-il proportionnel à n ?

Exercice 2.8 *Exercice plus théorique.* Jusqu'à présent, nous n'avons pas de fonction `(couple a b)` permettant de construire un *couple* de deux valeurs, ce que les mathématiciens notent *(a,b)*. Une *théorie des couples* est la donnée de trois fonctions :

`(couple a b),  (proj1 c),  (proj2 c)`

vérifiant les propriétés : `(proj1 (couple a b)) ≡ a` et `(proj2 (couple a b)) ≡ b`

Aux alentours de 1940, les théoriciens du *lambda-calcul*[1] ont proposé la définition suivante pour la fonction `couple` qui exprime qu'un couple est une fonction (oui, cela peut vous paraître bizarre mais pourquoi pas ?) :

```
(define (couple a b)
  (lambda (c)
    (if c a b)))
```

Pouvez-vous définir les fonctions `proj1` et `proj2` de sorte que les axiomes des couples soient satisfaits ?

```
> (define c (couple 2 3))
> c
#<procedure>
> (proj1 c)
2
> (proj2 c)
3
```

N.B. La conclusion provisoire que vous pourriez en tirer, dans la droite ligne du λ-calcul, pourrait être quelque chose comme : « C'est fou ce que l'on peut faire avec des fonctions !... ». ☺

Vous avez installé le teachpack 'valrose.rkt' et vous utilisez la fonction `(show expr)`. Bravo ! Si telle ou telle fonction vous semble ré-utilisable, n'hésitez pas à la déposer dans ce teachpack et n'oubliez pas de la rajouter à la liste `(provide …)`. Il faudra ré-installer le teachpack après toute modification !

[1] Le *lambda-calcul* est une théorie née aux alentours de 1940, notamment due à Alonzo Church, et faisant reposer les mathématiques non pas sur la théorie des ensembles mais sur la théorie des fonctions anonymes [les λ-fonctions]. Les entiers eux-mêmes sont définis comme étant des λ-fonctions ! Le λ-calcul joue un rôle essentiel en *théorie de la programmation*. Le langage Scheme est en quelque sorte un λ-calcul appliqué.

Annexe : Python vs Scheme - Séance 2

http://deptinfo.unice.fr/~roy

Le langage Scheme fait ses choux gras du concept de fonction, et de la lambda-notation. Python dispose lui aussi des lambda-fonctions [fonctions anonymes], utiles par exemple dans les tris ou dans les interfaces graphiques :

PYTHON	*SCHEME*
`lambda x,y : 2 * x + y`	`(lambda (x y) (+ (* 2 x) y))`
`>>> (lambda x,y : x + y + 1)(2,3)` `6`	`> ((lambda (x y) (+ x y 1)) 2 3)` `6`
`>>> lambda x : x + 1` `<function <lambda> at 0x1024e7200>`	`> (lambda (x) (+ x 1))` `#<procedure>`

Bizarrement les lambda-fonctions de Python ont des limitations. Leur contenu ne peut qu'être réduit à une seule expression. En Scheme elles peuvent s'étendre sur plusieurs lignes avec des variables locales, etc.

```
(lambda (x)
  (local [(define c (sin x))]   ; pour ne pas calculer (sin x) deux fois !
    (+ c (sqrt c))))
```

Si le langage Scheme est incollable sur les lambda-fonctions, c'est qu'il est un descendant de l'inventeur de la chose : le langage **LISP**, qui tire son origine à la fois d'une théorie [le λ-calcul] et d'une pratique naissante [l'Intelligence Artificielle], à la fin des années 1950. Avec **FORTRAN** (1955) qui est impératif, ce sont les deux grands ancêtres des langages de programmation.

Une fonction en Python comme en Scheme peut prendre des fonctions en arguments et retourner une fonction en résultat. Il y a deux manières de procéder, comme le montre le tableau ci-dessous :

PYTHON	*SCHEME*

`def compose(f,g) :`
`  return lambda x : f(g(x))`
 	
`def compose(f,g) :`
`  def res(x) : return f(g(x))`
`  return res`
	`(define ($compose f g)  ; version 1`
	`  (lambda (x) (f (g x))))`
	
	`(define ($compose f g)  ; version 2`
	`  (local [(define (res x) (f (g x)))]`
	`  res))`

`>>> from math import sqrt`
`>>> h = compose(sqrt,sqrt)`
`>>> h(4)`
`1.4142135623730951`
	`> (define h ($compose sqrt sqrt))`
	`> (h 4)`
	`#i1.4142135623730951  ;  $\sqrt[4]{4} = \sqrt{2}$`

Bref, Python fait ce qu'il peut pour incorporer une certaine dose de **programmation fonctionnelle**. Hélas, limitant les λ-expressions et interdisant toute efficacité aux fonctions définies par récurrence, il échoue à un certain moment. Mais ses tentatives sont bien sympathiques tout de même... Une bonne connaissance de Scheme renforcera votre souplesse en Python !