Programmation fonctionnelle avancée

Examen final

L2 – Université Nice Sophia Antipolis

21 décembre 2017

Durée : 1h30.

- Aucun document ni aucune machine ne sont autorisés.
- Les téléphones doivent être rangés.
- Les réponses doivent être écrites lisiblement.
- L'indentation du code doit être non ambiguë : chaque bloc sera indenté de 4 cases. En cas d'ambiguïté possible, l'interprétation la moins favorable sera retenue.
- Les questions sont indépendantes : une question peut être sautée, et une fonction `f` demandée à une question peut être utilisée pour définir une fonction g dans une question ultérieure même si la solution pour `f` n'a pas été trouvée.
- Les réponses sont à rédiger dans le langage Racket, ou Lisp, ou Scheme.

Exercice 1 (Répétition générale, 4 points)

1. *Définissez une fonction* `(repeat-until p)` *qui prend en argument une fonction sans argument* `p` *et qui l'appelle jusqu'à ce qu'elle renvoie* `#t`*. Par exemple*

```
(repeat-until
 (lambda ()
   (let ((i (random 2)))
     (printf "~a" i)
     (equal? i 1))))
```

   affiche (au hasard) 1, ou 01, ou 001, ou 0001, etc.

2. *Définissez sans utiliser le* `while` *une macro* `(repeat p until condition)` *qui évalue* p*, au moins une fois, jusqu'à ce que la condition soit vraie. Par exemple*

```
(define i 0)
(repeat
  (begin (set! i (random 2)) (printf "~a" n))
  until (equal? i 1))
```

   affiche (au hasard) 1, ou 01, ou 001, ou 0001, etc.
   Indication : on rappelle la définition d'une macro (si condition alors p1 sinon p2) :

```
(define-syntax si
  (syntax-rules (alors sinon)
    ((si condition alors p1 sinon p2) (if condition p1 p2))))
```

□

Exercice 2 (Des piles à la pelle, 6 points)

Le but de cet exercice est d'explorer différentes implémentations d'une pile en Racket. On rappele qu'une pile est un type de données abstrait équipé de deux opérations, empiler une valeur v, et dépiler qui renvoie la dernière valeur empilée qui n'a pas encore dépilée. Si la pile est vide, dépiler renvoie `#<void>`

1. *Définissez les fonctions* `empiler!` *et* `depiler!` *qui modifient une variable globale* `PILE` *contenant une liste dont vous écrirez aussi la définition. Par exemple, si la pile est vide, on a l'interaction suivante au toplevel*

```
> (empiler! 1)
> (empiler! 2)
> (list (depiler!) (empiler! 3) (depiler!) (depiler!) (depiler!))
'(2 #<void> 3 1 #<void>)
```

Code à compléter sur votre copie

```
(define PILE ...)
(define (empiler! x) ...)
(define (depiler!) ...)
```

2. *Définissez la constante* `pile-vide` *et les fonctions* `empiler` *et* `depiler` *qui implémentent une pile persistante :*
   — `(empiler pile v)` *renvoie la pile* `pile2` *obtenue en empilant* `v` *sur* `pile`, sans modifier `pile`.
   — `(depiler pile)` *renvoie la liste à deux éléments* `(pile2 v)` *où* `pile2` *et* `v` *sont respectivement la pile obtenue en dépilant et la valeur dépilée. Si la pile* `pile` *est vide, la fonction renvoie* `#f`.
   Par exemple :

```
(define p1 (empiler (empiler pile-vide 1) 2))
(define p2 (car (depiler p1)))
(define v1 (cadr (depiler p1)))
(define p3 (car (depiler p2)))
(define v2 (cadr (depiler p2)))
(printf "~a ~a ~a ~a ~a ~a" v1 v2 p1 p2 p3 (depiler p3))
```

affiche 2 1 (2 1) (1) () #f

Code à compléter sur votre copie

```
(define pile-vide ...)
(define (empiler p x) ...)
(define (depiler p) ...)
```

3. *Définissez la pile comme un « objet soft » (une clôture) avec deux méthodes* `'empiler` *et* `'depiler`. *Par exemple,*

```
(define P1 (nouvelle-pile))
(define P2 (nouvelle-pile))
(P1 'empiler 1)
(P1 'empiler 2)
(P2 'empiler 3)
(printf "~a ~a ~a ~a" (P1 'depiler) (P1 'depiler) (P2 'depiler)
  (P2 'depiler))
```

affiche `2 1 3 #<void>`.

Code à compléter sur votre copie

```
(define (nouvelle-pile)
  (let ...
    (define (self method . args)
      (case method
        [(empiler) ...]
        [(depiler) ...]))
    self))
```

□

Exercice 3 (Générateurs, 3 points)

On rappelle qu'un générateur est une fonction sans argument dont le résultat dépend du nombre d'appels antérieurs – le générateur égrenne une suite infinie.

1. *Écrivez une fonction* `(constant-gen v)` *qui renvoie un générateur qui égrenne la suite* $v, v, v, \ldots$ *Par exemple :*

```
(define g (constant-gen 1))
(printf "~a" (list (g) (g) (g)))
```

affiche `(1 1 1)`

2. *Écrivez une fonction* `(counter-gen)` *qui renvoie un générateur qui égrenne la suite* $0, 1, 2, \ldots$ *des entiers. Par exemple :*

```
(define c1 (counter-gen))
(define c2 (counter-gen))
(printf "~a" (list (c1) (c2) (c1) (c1) (c2)))
```

affiche `(0 0 1 2 1)`

3. *Que va afficher le code ci-dessous ? Justifiez.*

```
(define (gen-map f gen) (lambda () (f (gen))))
(define c1 (counter-gen))
(define c2 (gen-map sqr c1))
(printf "~a" (list (c1) (c2) (c1)))
```

□

Exercice 4 (Générateurs persistants, 7 points)

Un générateur persistant `pgen` *est une fonction sans argument qui renvoie une liste* `(pgen2 v)` *contenant un autre générateur persistant et une valeur. Le générateur persistant* `pgen2` *est celui à appeler pour continuer à égrenner la suite infinie des valeurs. Par exemple, en supposant que la fonction* `(counter-pgen)` *renvoie un nouveau générateur persistant qui égrenne la suite* $0, 1, 2, \ldots$*,*

```
(define pgen1 (counter-pgen))
(define v1 (cadr (pgen1)))
(define pgen2 (car (pgen1)))
(define v2 (cadr (pgen2)))
(printf "~a ~a" v1 v2)
```

affiche `0 1`.

1. *Écrivez une fonction* `(counter-pgen)` *qui renvoie un nouveau générateur persistant qui égrenne* $0, 1, 2, \ldots$

2. *Écrivez une fonction* `(pgen-print pgen n)` *qui affiche les* n *premiers éléments de la suite égrenée par le générateur persistant pgen sous la forme* $[v_1, v_2, \ldots]$.
   Par exemple `(pgen-print (counter-pgen) 3)` *affiche* `[0,1,2,...]`

3. *Écrivez une fonction* `(pgen-map f pgen)` *qui prend en argument une fonction* `f` *à un seul argument et un générateur persistant* `pgen` *qui égrenne la suite* $v_1, v_2, \ldots$ *et qui renvoie un nouveau générateur persistant qui égrenne* $f(v_1), f(v_2), \ldots$

4. *Qu'affiche le code ci-dessous ?*

```
(define pc1 (counter-pgen))
(define pc2 (pgen-map sqr pc1))
(pgen-print pc1 3)
(pgen-print pc2 3)
```

5. *Un générateur (persistant) aléatoire est un générateur qui égrenne une suite de 0 et de 1 tirée au hasard. Écrivez une fonction* `(make-random-pgen)` *qui renvoie un nouveau générateur persistant aléatoire.* ***Attention.*** *Deux générateurs persistants aléatoires créés par* `(make-random-pgen)` *égrennent des suites infinies qui sont différentes presque sûrement, mais la suite infinie égrennée par un générateur aléatoire est fixée une fois pour toute. Par exemple,*

```
(define rand1 (make-random-pgen))
(define rand2 (make-random-pgen))
(pgen-print rand1 5)
(pgen-print rand2 5)
(pgen-print rand1 5)
```

   affichera quelque chose comme

```
[0,1,1,1,0,...]
[1,1,0,0,1,...]
[0,1,1,1,0,...]
```

   Rappel la fonction `(random n)` *tire au hasard un nouveau nombre entre* 0 *et* $n-1$ *à chaque appel.*

□