ENSEIRB-Matmeca – Filière informatique première année 2017/2018

Examen de programmation fonctionnelle

Durée 2h, documents autorisés

1. **[6 pts]** Expliquer en quoi consiste la β-réduction et le rôle qu'elle joue dans l'évaluation des expressions en scheme. Quelle stratégie d'évaluation est utilisée dans le langage scheme?

   - Donner un exemple de suite de β-réductions de l'expression : `(* (/ (add1 (+ 1 3)) (* 6 2 4)) (- 12 22))` Combien y-a-t-il de suites possibles satisfaisant la stratégie d'évaluation du langage scheme?
   - Indiquer une suite de réductions possibles et le résultat de chacune des expressions suivantes :

     ```
     ((lambda (f) (f 1)) (lambda (x) (sqr x)))
     (((lambda (x) (lambda (y) (cons x y))) 1) '(a b))
     (((lambda (f x) (lambda (y) (f (+ x y)))) (lambda(x) x) 1) 2)
     ```

2. **[4 pts]** Voici plusieurs versions d'un prédicat renvoyant vrai en résultat si la liste de nombres qui lui est fournie en paramètre est ordonnée au sens large et faux sinon. Critiquer l'écriture de chacune de ces fonctions. Proposez une autre version qui soit explicitement récursive et qui utilise des opérateurs booléens plutôt qu'une forme conditionnelle telle que `if` ou `cond`.

   ```
   (define (order0? l)
     (cond ((null? l) #t)
           ((= (car l) (apply min l)) (order0? (cdr l)))
           (else #f)))

   (define (order1? l)
     (foldl < #t l))

   (define (order2? l)
     (apply <= l))

   (define order3? <=)

   (define (order4? l)
     (map < l))
   ```

3. On définit un module `lens` destiné à faciliter l'accès en lecture à des structures de données quelconques et permettre des transformations fonctionnelles des contenus. On donne la fonction `make-lens` suivante :

   ```
   (define (make-lens getter setter)
     (cons setter getter))
   ```

   Ce module pourra être utilisé comme suit :

   ```
   > (define car-lens (make-lens car set-car))
   > (define cdr-lens (make-lens cdr set-cdr))
   ```

   avec :

```
(define (set-car p v)
    (cons v (cdr p)))

(define (set-cdr p v)
    (cons (car p) v))
```

Et permettre l'utilisation des fonctions d'accès en lecture et transformation :

```
> (lens-view car-lens (cons 1 2))
1
> (lens-set car-lens (cons 1 2) 'x)
'(x . 2)
```

- **[2 pts]** Proposer une écriture des fonctions `lens-view` et `lens-set`
- **[1 pts]** Écrire la fonction `lens-transform` qui permet de transformer le contenu d'une structure de données par l'application d'une fonction donnée en paramètre :

  ```
  >  (lens-transform car-lens (cons 1 2) add1)
  '(2 . 2)
  ```

- On souhaite composer les `lens` en accès lecture et écriture afin de pouvoir écrire :

  ```
  > (define cdar-lens (lens-compose cdr-lens car-lens))
  > (define tree '((1 . 0) (2 . 3)))
  > (lens-view cdar-lens tree)
  0
  >  (lens-set cdar-lens tree 'x)
  '((1 . x) (2 . 3))
  ```

  - **[1 pts]** Soit la variable `tree` définie ci-dessus, remplacer dans l'expression suivante les points d'interrogation par les fonctions `car` ou `cdr` afin d'obtenir le résultat voulu :

    ```
    > (? (? tree))
    0
    ```

  - **[1 pts]** Soient `f` et `g` les fonctions choisies pour le remplacement dans l'expression ci-dessus, proposer une façon de les combiner afin de construire une fonction `h` comme suit :

    ```
    > (define h (? f g))
    ```

    qui permettrait d'écrire directement :

    ```
    > (h tree)
    0
    ```

  - **[1 pts]** Remplacer dans l'expression suivante les points d'interrogation par les fonctions `cons`, `car` ou `cdr` afin d'obtenir le résultat voulu :

    ```
    > (? (? (? (? tree)) 'x) (? tree))
    '((1 . x) (2 . 3))
    ```

  - **[2 pts]** Quel sont les résultats des expressions suivantes ?

    ```
    > (lens-set cdr-lens (lens-view car-lens tree) 10))
    > (lens-set car-lens tree 10)
    > (lens-set car-lens tree (lens-set cdr-lens (lens-view car-lens tree) 10))
    ```

  - **[2 pts]** Écrire la fonction `lens-compose`