ENSEIRB-Matmeca – Filière informatique première année 2018/2019

Examen de programmation fonctionnelle

- Durée 2h, documents autorisés -

1. **[1 pt]** Expliquer quelles sont les stratégies de réductions en lambda-calcul et en scheme.
2. **[3 pts]** Proposer une suite de réductions pour les expressions suivantes et donner leur résultat.

```
(([lambda(i)
    (lambda(f) (f i))]
  1)
 [lambda(x) (* 3 x)])

(([lambda (x)
    [lambda (y) (list x y)]]
  '(1 2))
 '(3 4))

([lambda (f x y)
   (* (f x) (f y))]
 [lambda (x) (add1 x)]
 1
 2)
```

3. **[2 pts]** Voici quatre versions d'une fonction calculant le minimum d'une liste de nombres. Critiquer et comparer l'écriture de ces fonctions.

```
(define (minimum1 l)
  (cond ((null? l) (void))
        ((= (car l) (apply min l)) (car l))
        (else (minimum1 (cdr l)))))

(define (minimum2 l)
  (apply min l))

(define minimum3 ((curry apply) min))

(define (minimum4 l)
  (letrec ((min4 (lambda (m l)
                   (cond ((< (car l) m) (min4 (car l) (cdr l)))
                         (else (min4 m (cdr l)))))))
    (min4 (car l) (cdr l))))
```

4. **[2 pts]** Soient la fonction et la macro suivantes :

```
(define (=* x y)
  (set! x (* x y)))

(define-syntax-rule (=*! x y)
  (set! x (* x y)))
```

Soit la session suivante :

```
> (define a 1)
> (define b 1)
> (=* a 10)
> (=*! b 10)
```

Indiquer quels sont les résultats des expressions suivantes et donner deux arguments permettant de les expliquer :

```
> a
> b
```

5. **[8 pts]** Une permutation est une bijection d'un ensemble vers lui-même. On considère des ensembles correspondant à des suites ordonnées et on souhaite écrire un module de calcul de permutations. On se donne la fonction `transposition` qui représente une permutation échangeant les deux indices i et j de sorte que le nouvel élément en position i est celui qui était en position j et vice-versa.

```
(define (transposition i j)
  (lambda (x)
    (cond ((= x i) j)
          ((= x j) i)
          (else x))))
```

Voici un exemple d'utilisation.

```
> (define t23 (transposition 2 3))
> (t23 2)
3
> (t23 4)
4
```

- **[0.5 pt]** Écrire la fonction `empty-permutation` qui construit la permutation inopérante, c'est-à-dire renvoyant son argument inchangé.

  ```
  > (define p (empty-permutation))
  > (p 3)
  3
  > (p 1)
  1
  ```

- **[1 pt]** Écrire la fonction `cyclic-permutation` prenant un paramètre entier n et construisant la permutation circulaire sur n éléments, en décalant circulairement les éléments vers la droite. Par exemple, la liste des images des valeurs de la liste (1 2 3 4) est la liste décalée circulairement vers la droite (4 1 2 3). Ainsi sur cet exemple, la valeur à l'indice 2 est 2 dans la liste initiale et 1 dans la liste image.

  ```
  > (define p (cyclic-permutation 6))
  > (p 3)
  2
  > (p 4)
  3
  ```

- **[0.5 pt]** Remplacer le point d'interrogation dans l'expression suivante de façon à obtenir le résultat indiqué.

  ```
  > (? t23 '(0 1 2 3 4 5 6))
  '(0 1 3 2 4 5 6)
  ```

- **[1 pt]** Écrire la fonction `cycle` prenant en arguments une permutation et un indice i et renvoyant le cycle obtenu à partir de i, c'est-à-dire la suite des indices obtenus par applications successives de la permutation jusqu'à revenir sur i.

  ```
  > (cycle t23 1)
  '(1)
  > (cycle t23 2)
  '(2 3)
  > (cycle t23 3)
  '(3 2)
  ```

- **[3 pts]** On souhaite étendre ces permutations à des listes d'objets quelconques.
  - Quel est le résultat de l'expression suivante[1].

[1] La fonction `list-ref` appliquée à la liste l et l'entier n retourne l'élément de l en position n, en partant de 0

```
> ((compose (curry list-ref '(a b c d)) t23) 2)
```

– Soit la fonction `acces-permutation` suivante, où l est une liste d'objets quelconques et n l'indice d'un objet dans l :

```
 (define (acces-permutation l p i)
  (list-ref l (p i)))
> (acces-permutation '(a b c d e) t23 2)
'd
> (acces-permutation '(a b c d e) t23 1)
'b
```

Remplacer le point d'interrogation dans le corps de la fonction `make-acces-permutation` suivante de sorte qu'elle fonctionne comme indiqué. Quelles différences voyez-vous entre la fonction `acces-permutation` et la fonction `make-acces-permutation`?

```
(define (make-acces-permutation l p)
  (compose (? list-ref l) p))
> ((make-acces-permutation '(a b c d e) t23) 2)
'd
> ((make-acces-permutation '(a b c d e) t23) 1)
'b
```

– Écrire la fonction `liste-permute` prenant en arguments une liste d'objets de type quelconque et une permutation et renvoyant la liste réordonnée selon la permutation. On écrira cette fonction en utilisant la fonction `map` et `make-acces-permutation`. On pourra utilliser la fonction `range`[2].

```
> (list-permute '(a b c d) t23)
'(a b d c)
```

- `[2 pts]` Enfin, on souhaite pouvoir composer les permutations afin d'en construire de nouvelles à partir des constructeurs dont on dispose.

– Écrire une fonction `compose-permutation` permettant de composer un nombre quelconque de permutations de la façon suivante.

```
> (define p (compose-permutation t23 t23 t23))
> (p 2)
3
```

– La puissance n d'une permutation p est définie par la composition de p n fois. Écrire une fonction `puissance-permutat` prenant en arguments une permutation p et un entier n et qui calcule la permutation de p puissance n. On pourra utiliser la fonction `make-list`[3].

```
> (define t (puissance-permutation t23 4))
> (t 2)
2
```

6. **[4 pts]** Choisissez une séance de cours, entre la deuxième et la dernière.

   (a) Quels sont les concepts introduits dans cette séance?

   (b) Pourquoi ces concepts sont-ils introduits, que permettent-ils de comprendre et quelles compétences sont visées?

   (c) Comment cette séance s'articule-t-elle avec le reste du cours?

   (d) Ces concepts introduits dans cette séance s'articulent-ils avec d'autres concepts discutés dans d'autres enseignements de la formation? Si oui lesquels?

[2] La fonction `range` appliquée à un argument entier n construit la liste des entiers partant de zéro jusqu'à $n-1$

[3] La fonction `make-list` appliquée à un entier n et un objet o renvoie une liste de taille n dont tous les éléments sont l'objet o