ENSEIRB-Matmeca – Filière informatique première année 2015/2016

Examen de programmation fonctionnelle

Durée 2h, documents autorisés

1. Stratégie d'évaluation

   - Quelle est la stratégie d'évaluation du langage scheme?
   - Soit l'expression suivante : `(/ (* (sqr (+ 1 3)) (+ 1 3)) (- 10 2))` Décomposer cette expression en sous-expressions. Un ordre d'évaluation de ces sous-expressions peut-il être donné ou bien l'évaluation de certaines d'entre elles est-elle parallélisable? Donner une suite de reécritures de l'expression dont chaque pas correspond à l'évaluation d'une (et une seule) des sous-expressions et correspondant à la stratégie d'évaluation de scheme.
   - Cette stratégie d'évaluation est-elle toujours efficace?

2. Étant donnée une fonction `g` déjà implémentée, considérons la fonction C ci-dessous. Cette fonction fait-elle un effet de bord (si oui lequel) ?

```
int f(int i, int j, int k)
{
  int r1 = i*j;
  int r2 = g(k);
  if (r1 > r2)
  {
     int r3 = g(r1);
     return r3+1;
  else return r2-1;
}
```

   Ecrire une fonction scheme fonctionnelle pure effectuant le même calcul que la fonction `f`, étant donnée une fonction `g` déjà implémentée en scheme et faisant le même calcul de la fonction `g` écrite en C. .

3. Considérons la fonction C ci-dessous. Cette fonction fait-elle un effet de bord (si oui lequel) ?

```
int ff(int n)
{
  int r=0;
  for (int i=1; i<=n; i++)
    r+=i + r;
  return r;
}
```

   Ecrire une fonction scheme fonctionnelle pure effectuant le même calcul que la fonction C ci-dessous.

4. Compléter les expressions suivantes de façon à ce qu'elles soient correctes et qu'elles produisent le résultat demandé, en remplaçant le point d'interrogation par une expression

```
> (? * '(1 2 3))
6

> (? * '(1 2 3) '(1 2 3))
'(1 4 9)

> (map + '(1 2) (3 4) ?)
```

```
'(4 8)

> (foldl * ? '(1 2 3))
6
```

5. Au vu des exemples vus en cours et en TD, critiquer les expressions suivantes et proposer une meilleure formulation :

```
(if (= a 1) #f #t)

(/ (+ (sqrt x) y) (- (sqrt x) y))

(define (belongs e l)
  (if ((null? l) #f)
      ((equal? e (car l)) l)
      (else (belongs e (cdr l)))))
```

6. Critiquer cette fonction

```
(define (inverser-liste l)
  (if (null? l)
      (append (inverser-liste (cdr l)) (car l))))
```

7. Soit une liste comportant des éléments atomiques ou bien des listes, on souhaite parcourir les éléments de la liste pour effectuer des traitements sur tous les éléments atomiques. On va considérer deux spécifications inductives de la liste, soit par une liste soit par un arbre pour écrire une fonction de deux façons différentes.

   (a) Considérons la définition suivante sous forme de grammaire :

   $$\text{liste} ::= \text{null} \mid (\text{atome} \ . \ \text{liste}) \mid (\text{liste} \ . \ \text{liste})$$

   Ecrire une fonction `compter-les-elements` qui compte le nombre d'éléments atomiques d'une liste dont le type répond à la spécification ci-dessus. Ecrire d'abord la spécification de la fonction en détaillant les résultats correspondant aux différents cas à considérer.
   Exemple : `(compter-les-elements '(1 2 (3) ((4 5) (6 7)) 8)) -> 8`

   (b) Considérons la définition suivante : Un arbre est
   - soit l'arbre vide
   - soit une feuille contenant un entier
   - soit un noeud avec un fils droit et un fils gauche qui sont des arbres non vides

   Écrire la définition formelle sous forme de grammaire de cette définition. Puis écrire une fonction qui compte le nombre de feuilles d'un arbre dont le type répond à cette spécification. Ecrire d'abord la spécification de la fonction en détaillant les résultats correspondant aux différents cas à considérer.

8. On se donne la fonction suivante :

```
(define (multiplier* ll)
  (cond ((null? l) 1)
        ((not (pair? (car l))) (* (car l) (multiplier* (cdr l)))
        (else (* (multiplier* (car l)) (multiplier* (cdr l)))))))
```

   Donner une spécification de cette fonction en détaillant le type du paramètre et les résultats correspondant aux différents cas considérés.

   Écrire une autre version de cette fonction en utilisant les formes `map` et `foldr` pour effectuer les deux boucles récursives. On pourra écrire une fonction auxiliaire.