ENSEIRB-Matmeca – Filière informatique première année 2014/2015

Examen de programmation fonctionnelle

Durée 2h, documents autorisés

1. **[4pts]** Soit les définitions suivantes

```
(define a 0)
(define b 10)
(define c 3)
```

   Quels sont les résultats des expressions suivantes?

   - (cons a (cons b (cons c '())))
   - (list a b c)
   - (list 'a b c)
   - '(a b c)
   - (make-list 3 a)
   - (cons 1 '(a b))
   - (cons '(a b) 1)
   - (list 1 '(a b))
   - (list '(a b) 1)
   - (append 1 '(a b))
   - (append '(a b) 1)
   - (append '(1) '(a b))

2. **[10pts]** L'objectif de cet exercice est de réaliser un type `expression`, permettant l'évaluation d'expressions contenant éventuellement des symboles auxquels seront associées des valeurs par l'intermédiaire d'un environnement passé comme paramètre à la fonction d'évaluation.

   Une expression est définie récursivement comme étant

   - soit un symbole;
   - soit un nombre;
   - soit une liste dont le premier élément est un opérateur légal, et chacun des autres éléments est une expression.

   On suppose définie dans l'environnement global, une liste `liste-operateurs` contenant tous les opérateurs légaux :

```
(define liste-operateurs (list + * / - cos sin))
```

   (a) Définir les prédicats `expression?` et `operateur?`

   (b) Définir une fonction `creer-exp` permettant de construire des expressions :

```
> (creer-exp + 1 2 3)
'(#<procedure:+> 1 2 3)
```

   (c) On envisage d'utiliser des `environnements` définis sous forme de listes d'associations :

```
(define un-environnement '((a . 3) (x . -5) (y . 0)))
```

Définir une fonction `valeur`, prenant en arguments un symbole et un environnement, et retournant la valeur attribuée à ce symbole si celui-ci est bien défini, soit levant une exception au moyen de la forme `raise`.

```
> (valeur 'x un-environnement)
-5
> (valeur 't un-environnement)
uncaught exception: "Symbol t is undefined"
```

(d) Définir une fonction `evaluation`, qui prend en arguments une expression et un environnement, et calculer la valeur de l'expression dans l'environnement;

```
> (evaluation (creer-exp * 'x (creer-exp + 1 'a)) un-environnement)
-20
```

(e) Définir une fonction `application`, qui prend comme arguments un opérateur, une liste d'arguments (qui sont eux-mêmes des expressions), et un environnement, et applique l'opérateur à la liste des valeurs des arguments dans l'environnement.

```
>  (application + (list 1 2 (creer-exp * 'x 2)) un-environnement)
-7
```

(f) Supposons maintenant vouloir enrichir nos expressions à l'aide d'expressions "let". Une expression peut alors être, en plus des possibilités précédentes :

- soit une liste à 4 éléments dont le premier élément est le symbole `'let`, le deuxième élément un symbole quelconque et les 3ème et 4ème éléments des expressions.

Par exemple, l'`expression` suivante :

```
> '(let x 2 (+ x x))
4
```

... représente l'expression Scheme :

```
> (let ([x 2]) (+ x x))
4
```

Réécrire la fonction `evaluation` pour permettre d'évaluer ce type d'expressions.

Expliquez la notion de portée d'une variable en prenant comme exemple votre fonction.

(g) Tel qu'il est proposé, notre type `expression` est peu lisible :

```
> (creer-exp + 'x 1)
'(#<procedure:+> x 1)
```

Proposer une solution alternative conduisant à :

```
guile> (creer-exp + 'x 1)
'(+ x 1)
```

3. On souhaite une fonctionnelle prenant en paramètres une fonction unaire `f` et un entier positif `n` et réalisant l'itération de `f` `n` fois.

(a) On écrira deux versions de cette fonction, `itere1` non récursive terminale et `iterer2` récursive terminale.

```
> (define 4+ (iterer1 add1 4))
> (4+ 0)
4
> (define 3+ (iterer2 add1 3))
> (3+ 0)
3
```

On pourra s'inspirer de la fonctionnelle de composition n-aire pour écrire ces fonctions.

(b) On comparera les deux versions, en terme d'écriture, de terminaison de la boucle récursive et d'efficacité.