ISIMA 1ère année — 3 février 2009
Durée : 2 heures

Documents autorisés

PROGRAMMATION FONCTIONNELLE

Pensez à expliciter en français ce que doivent faire vos fonctions.
Vous pourrez bien sûr écrire des fonctions intermédiaires.

Exercice 1 :
Que fait la fonction `tm` suivante, dont l'argument `L` est une liste de nombres :

```
(define tm (lambda (L)
   (map (lambda (x) (map (lambda (y) (* x y)) L)) L) )
```

Solution :
Cette fonction fait une table de multiplication des éléments de `L`, en retournant la liste contenant les résultats des multiplications de chaque élément de `L` par chaque élément de `L`.

Exercice 2 :
Vous pouvez répondre aux questions dans l'ordre inverse de leur numérotation si vous le souhaitez.

Les fonctions prédéfinies `length` et `member` pouront être utilisées sans avoir a être réécrites :
- L'expression fonctionnelle `(length L)` retourne la longueur (i.e. le nombre d'éléments) de la liste `L` ;
- L'expression fonctionnelle `(member x L)` retourne `L` si `x` est membre de la liste `L`, et `()` sinon.

Le but de ce problème est d'écrire une fonction `PGC` ayant comme arguments deux listes de nombres, `L1` et `L2`, non vides, et telle que l'évaluation de l'expression `(PGC L1 L2)` retourne une liste contenant la (ou les) Plus Grande(s) sous-liste(s) Commune(s) à `L1` et `L2`, ou retourne `()` si il n'y a pas de liste commune.
Exemple :
`(PGC '(1 2 3 4 5 6) '(5 2 3 1 3 4 5 2))` retourne `((3 4 5))` ;
`(PGC '(1 2 3 4 5 6) '(5 2 3 1 3 4 2))` retourne `((2 3) (3 4))`.

Soit la fonction `PGCIE` : elle a trois arguments, qui sont un entier `lg` strictement positif et deux listes `PL` et `GL` non vides telles que la longueur de la liste `PL` est inférieure ou égale à la longueur de la liste `GL` et l'évaluation de `(PGCIE lg PL GL)` retourne une liste contenant la (ou les) Plus Grande(s) sous-liste(s) Commune(s) à `PL` et `GL` de longueur Inférieure ou Egale à `lg`.
Exemples :
`(PGCIE 6 '(1 2 3 4 5 6) '(5 2 3 1 3 4 5 2))` retourne `((3 4 5))` ;
`(PGCIE 2 '(1 2 3 4 5 6) '(5 2 3 4 3 4 5 2))` retourne `((2 3) (3 4) (4 5))`.

Question 1 :
Ecrire la fonction `PGC` en utilisant les fonctions `PGCIE` et `length`.
Solution :

```
(define PGC (lambda (L1 L2)
   (let ((lgL1 (length L1)) (lgL2 (length L2)))
      (if (< lgL1 lgL2) (PGCIE lgL1 L1 L2)
                        (PGCIE lgL2 L2 L1) ) ) ))
```

Soit la fonction `SLCE` : elle a trois arguments, qui sont un entier `lg` strictement positif et deux listes `PL` et `GL` non vides telles que la longueur de la liste `PL` est inférieure ou égale à la longueur de la liste `GL`, et l'évaluation de l'expression `(SLCE lg PL GL)` retourne une liste contenant la (ou les) Sous Liste(s) Commune(s) à `PL` et `GL` de longueur Egale à `lg`.
Exemples :
`(SLCE 4 '(1 2 3 4 5 6) '(5 2 3 4 3 4 5 2))` retourne `()` ;
`(SLCE 3 '(1 2 3 4 5 6) '(5 2 3 4 3 4 5 2))` retourne `((2 3 4) (3 4 5))` ;

`(SLCE 2 '(1 2 3 4 5 6) '(5 2 3 4 3 4 5 2))` retourne `((2 3) (3 4) (4 5))`.

Question 2 :
Ecrire la fonction `PGCIE` en utilisant la fonction `SLCE`.
Solution :

```
(define PGCIE (lambda (lg PL GL)
   (if (> lg 0)
      (let ((SL (SLCE lg PL GL)))
         (if SL SL (PGCIE (- lg 1) PL GL)) )
      () ) ))
```

Soit la fonction test `SL?` : elle a comme arguments deux listes `L1` et `L2` non vides telles que la longueur de la liste `L1` est inférieure ou égale à la longueur de la liste `L2`, et l'évaluation de l'expression `(SL? L1 L2)` retourne la liste contenant la liste `L1` si la liste `L1` est une Sous Liste de la liste `L2` et retourne `()` sinon.
Exemples :
`(SL? '(2 3 4 5) '(5 2 3 4 3 4 5 2))` retourne `()` ;
`(SL? '(2 3 4) '(5 2 3 4 3 4 5 2))` retourne `((2 3 4))`.

Soit la fonction `LSL` : elle a deux arguments, qui sont un entier `lg` strictement positif et une liste `L` non vide, et l'évaluation de l'expression `(LSL lg L)` retourne la Liste des Sous Listes de `L` de longueur `lg`.
Exemple :
`(LSL 3 '(1 2 3 4 5 6))` retourne `((1 2 3) (2 3 4) (3 4 5) (4 5 6))`.

Question 3 :
Ecrire la fonction `SLCE` en utilisant la fonction prédéfinie `append-map` et les fonctions `SL?` et `LSL`.
Solution :

```
(define SLCE (lambda (lg PL GL)
   (append-map (lambda (SL) (SL? SL GL)) (LSL lg PL)) ))
```

Question 4 :
Ecrire la fonction `SL`.
Solution :

```
(define SL? (lambda (L1 L2)
   (if (member L1 (LSL (length L1) L2)) (list L1) ()) ))
```

Question 5 :
Ecrire la fonction `LSL`.
Solution :

```
(define LSL (lambda (lg L)
   (if (= lg (length L)) (list L)
      (let ((L+ (LSL (+ lg 1) L)))
         (cons (rcdr (car L+)) (map cdr L+)) ) ) ))
(define rcdr (lambda (L)
   (if (cdr L) (cons (car L) (rcdr (cdr L))) ()) ))
```

Exercice : Les ordinaux

La construction théorique des ordinaux, dont les entiers naturels font partie, est basée sur la théorie des ensembles. Les entiers naturels, en tant qu'ordinaux, sont donc représentés par des ensembles de la façon suivante : l'ordinal 0 est représenté par l'ensemble vide, et le successeur n^+ de tout ordinal n est représenté par l'ensemble $\{n\} \cup n$ (où n est donc un ensemble, et $\{n\}$ est l'ensemble qui contient n).

On peut ainsi construire des ordinaux, notés 0, 1, 2, 3, … de la manière suivante :

$$0 = \varnothing \; ; \; 1 = \{0\} \cup 0 = \{\varnothing\} \cup \varnothing = \{\varnothing\} \; ; \; 2 = \{1\} \cup 1 = \{\{\varnothing\}, \varnothing\} = \{1, 0\};$$

$3 = \{2\} \cup 2 = \{\{\{\varnothing\}, \varnothing\}, \{\varnothing\}, \varnothing\} = \{2, 1, 0\}$.

et en général $n = \{n^-, \ldots, 2, 1, 0\}$, en notant n^- le prédécesseur de n, c'est-à-dire l'ordinal tel que $(n^-)^+ = n$.

On convient de représenter l'ordinal 0 par `()`, et tout ordinal n non nul par une liste du type $(n^-, \ldots, 2, 1, 0)$. Ainsi par exemple l'ordinal 1 est représenté par `(())`, l'ordinal 2 par `( (()) () )`, l'ordinal 3 par `( ( (()) () ) (()) () )`, etc.

1°) Indiquer quelles sont les deux fonctions Scheme prédéfinies qui permettent, en leur passant un ordinal n non nul en argument, d'obtenir le prédécesseur n^- de n.

Solution :

Ces deux fonctions sont `car` et `cdr`.

2°) Ecrire une fonction `ordinal?` ayant comme argument une liste `L`, et telle que l'évaluation de l'expression `(ordinal? L)` retourne une valeur logique vraie si `L` représente un ordinal et une valeur logique fausse sinon.

Solution :

```
(define ordinal? (lambda (L)
   (or (not L)
      (and (equal? (car L) (cdr L)) (ordinal? (car L))) ) ))
```

3°) Ecrire une fonction `successeur` ayant comme argument une liste `O` représentant un ordinal, et telle que l'évaluation de l'expression `(successeur O)` retourne le successeur de cet ordinal. Par exemple, l'évaluation de l'expression `(successeur '(()) )` doit retourner `( (()) () )`.

Solution :

```
(define successeur (lambda (O) (cons O O)))
```

4°) Ecrire une fonction `ordinal` ayant comme argument un entier `n`, et telle que l'évaluation de l'expression `(ordinal n)` retourne l'ordinal correspondant à cet entier. Par exemple, l'évaluation de `(ordinal 2)` doit retourner `( (()) () )`.

Solution :

```
(define ordinal (lambda (n)
```

```
(if (> n 0) (successeur (ordinal (- n 1))) ()) ))
```

5°) En utilisant un schéma vu en cours, écrire une fonction `o+`, ayant comme arguments deux liste `n` et `m` représentants deux ordinaux *n* et *m* et telle que l'évaluation de l'expression `(o+ n m)` retourne la liste représentant l'ordinal *n* + *m*.

Solution :

On peut écrire `o+` en utilisant un schéma itératif en considérant `n` comme la liste traitée, `m` comme la variable d'accumulation, l'opération fait sur celle-ci à chaque "itération" état de prendre son successeur : en effet, on incrémentera `m` autant de fois qu'il y a d'éléments dans `n`, c'est-à-dire *n* fois. On remarque que le premier élément de `n` ne sert à rien ici, mais que, pour respecter la syntaxe du schéma itératif on doit quand même le mettre comme paramètre formel de la fonction passée en paramètre au schéma. On obtient :

```
(define o+ (lambda (n m) (SI n m (lambda (t m) (cons m m)))))
```

On peut aussi écrire la même chose avec un schéma récursif, l'interprétation étant différente : `n` est la liste à traiter, `m` le cas d'initialisation (en effet, quand `n` est vide, *n* = 0), et la fonction d'héritage entre `(car n)` et l'application de `o+` sur `(cdr n)` et `m`, qui retourne l'ordinal représentant *n* – 1 + *m* , et encore une fois l'application qui prend le successeur de cet ordinal (sans utiliser `(car n)` comme dans le schéma itératif). On obtient :

```
(define o+ (lambda (n m)
   (SR n m (lambda (t tr) (cons tr tr))))).
```

6°) En utilisant un schéma vu en cours, écrire une fonction `o-`, ayant comme arguments deux liste `n` et `m` représentants deux ordinaux *n* et *m* tels que *n* ≥ *m* et telle que l'évaluation de l'expression `(o- n m)` retourne la liste représentant l'ordinal *n* – *m*.

Solution :

Le principe est le même que dans la question précédente, sinon qu'il faut faire attention au fait que *m* ≤ *n*, et donc qu'il faut ici impérativement que ce soit `m` qui soit la liste à traiter (alors que dans la question précédente, le choix de `n` ou `m` est indifférent) et donc `n` la valeur d'intialisation ou la variable d'accumulation selon le schéma choisi. Ensuite, le raisonnement général étant que *n* – *m* = *n* si *m* = 0 et que *n* – *m* = (*n* – 1) – (*m* – 1) selon le point de vue du schéma itératif ou *n* – *m* = (*n* – 1 – *m*) – 1 selon le point de vue du schéma récursif, la fonction qui traite la variable d'accumulation ou le résultat remontant de l'appel récursif, en fonction du schéma est donc la fonction prédécesseur, soit `car` ou `cdr`, mais que l'on ne peut

pas écrire comme cela, car le schéma impose de prendre `(car m)` comme paramètre, même si, dans ce cas précis, on ne l'utilise pas. On obtient donc

```
(define o- (lambda (n m) (SI m n (lambda (t n) (car n)))))
```

ou bien

```
(define o- (lambda (n m) (SI m n (lambda (t tr) (car tr)))))
```