(Programmation Scheme Avancée

Examen de Rattrapage L2/(I+M)

Université Nice Sophia-Antipolis, Juin 2017

LE POLYCOPIE DU COURS ET LE LIVRE DE COURS SONT LES SEULS DOCUMENTS AUTORISES. Soignez l'indentation, le parenthésage et la limpidité du code. Le correcteur est davantage intéressé par un code qui fonctionne que par de vagues idées. Répondez aux questions sur la copie d'examen qui vous est fournie. Ne joignez aucune autre feuille ni intercalaire. Et ne perdez pas de temps !

1 Une petite fonction dans divers styles [7 points]

a) Programmez de manière **fonctionnelle** – donc sans mutation – et **récursive enveloppée** la fonction `(take L k)` retournant la liste des k premiers éléments d'une liste `L`. Il est garanti que $0 \leq k \leq \text{length}(\texttt{L})$. Exemple :

```
> (take '(a b c d e f) 0)
()
> (take '(a b c d e f) 3)
(a b c)
```

b) Programmez-en une version `(take-it L k)` toujours **fonctionnelle** mais cette fois **itérative**, avec un accumulateur. Il est bien entendu hors de question d'utiliser une boucle `while`. Même exemple que ci-dessus.

c) Programmez-en une version `(take-imp L k)` cette fois **impérative**, avec une boucle `while` supposée exister, au format `(while test e1 e2 ...)`. Même exemple que ci-dessus.

d) Programmez maintenant une fonction `(gen-take L k)` dont le résultat est un **générateur** (donc une fonction d'arité 0) des k premiers éléments de `L`, un à la fois. On génèrera le symbole `'FAIL` après le k-ème élément. Suivant la philosophie des générateurs, il est bien entendu exclu de calculer d'abord la liste des k premiers éléments. *Rappel : un générateur est une fermeture.*

```
> (define foo (gen-take '(a b c d e f) 3))
> (foo)
a
> (foo)
b
> (foo)
c
> (foo)
FAIL
```

2 Une fonction `map` à mémoire des calculs effectués [3 points]

Un programmeur Scheme doit appliquer une seule fois la même fonction `f` sur tous les éléments d'une liste `L` comportant un très grand nombre d'éléments. La fonction `f` nécessite de lourds calculs,

et la liste `L` comprend beaucoup de répétitions. Il a l'idée d'implémenter une *mémo-fonction* comme en Python mais contrairement à ce dernier, il souhaite rester dans une optique récursive purement fonctionnelle, sans mutations. Pour cela il a l'idée de reprogrammer la primitive `map` sous la forme d'une fonction `(memo-map f L mem)` où `mem` est une A-liste de couples `((`x_i y_i`) ...)` exprimant que le calcul $y_i = f(x_i)$ a déjà été fait. Au départ cette mémoire est vide et se remplit au fur et à mesure des calculs. Programmez la fonction `(memo-map f L mem)` de manière **purement fonctionnelle, sans affectation**. Exemple :

```
> (define cpt 0)                    ; pour compter combien de fois on entre dans f
> (define (f x) (set! cpt (+ cpt 1)) (* 2 x))
> (define L (memo-map f '(1 1 0 8 1 0 1 8 8 0 1 0 0 8 1 1) '()))   ; mémoire vide
> L
(2 2 0 16 2 0 2 16 16 0 2 0 0 16 2 2)
> cpt
3                                   ; on n'a utilisé que 3 fois la fonction f
```

3 Macrologie [5 points]

a) Vous savez que `let` n'est que le mot-clé d'une macro et n'est pas une construction essentielle du langage SCHEME. Une expression `(let ((x e) ...) e1 e2 ...)` n'est qu'un *sucre syntaxique* masquant un appel de lambda-fonction. Les deux expressions ci-dessous sont équivalentes :

`(let ((x e) ...) e1 e2 ...)` $\Leftrightarrow$ `((lambda (x ...) e1 e2 ...) e ...)`

Programmez une **fonction** `let->lambda` prenant une liste `L` contenant une expression `let` et retournant une liste *équivalente* contenant un appel de lambda-fonction *(indication : ne raisonnez pas par récurrence, utilisez* `map` *!)*. On supposera que le mot `let` n'intervient qu'en tête de `L`. Exemple :

```
> (let->lambda '(let ((x 1) (y 2)) (print x) (+ x y)))
((lambda (x y) (print x) (+ x y)) 1 2)
```

Dans les trois questions suivantes, nous sommes dans un dialecte de SCHEME *qui ne possède pas les mots-clé* `and`*,* `or` *et* `not` *mais qui possède tout le reste. Gasp.*

b) Soient α et β deux expressions SCHEME quelconques. Ecrivez sur votre copie une expression équivalente[1] à l'expression `(and` α β`)` mais n'utilisant pas les mots `and`, `or` et `not`.

c) Pourquoi ne peut-on pas définir `and` avec le mot-clé `define` ?

d) Programmez néanmoins le mot-clé `and` sous la forme `(and e1 e2 ...)` d'arité ≥ 1 dans notre SCHEME appauvri.

4 Algorithmique : la structure de *liste bidirectionnelle* [6 points]

On souhaite implémenter en SCHEME un type de données classique de l'algorithmique, celui des **listes bidirectionnelles** – ou *bilistes* – dans lesquelles on puisse avancer ou reculer alors que dans une liste chaînée classique, on ne peut qu'avancer. Il y a deux manières d'y parvenir, soit en restant dans le cadre de la programmation purement fonctionnelle, soit en prenant un point de vue impératif avec des mutations.

1. *Equivalente* signifiant : donnant le même résultat en effectuant les mêmes calculs !

Prenons le point de vue purement fonctionnel

Il nous suffit de construire trois fonctions de base agissant sur une *biliste* `BL` :

- `(kar BL)`, qui retourne l'information couramment pointée.
- `(kdr BL)`, qui retourne un pointeur sur la cellule suivante **à droite** dans la *biliste*.
- `(retro-kdr BL)` qui retourne un pointeur sur la cellule précédente **à gauche**.

Pour exprimer que 4 est l'élément couramment pointé dans une *biliste* :

$$1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow \boxed{4} \leftrightarrow 5 \leftrightarrow 6$$

nous représenterons[2] cette *biliste* par un couple de deux listes usuelles `(3 2 1)` et `(4 5 6)`. Vous noterez que la portion arrière précédant le 4 est inversée ! Une *biliste* sera donc un couple formée d'une partie arrière et d'une partie avant. L'élément couramment pointé (le *focus*) sera le premier élément de la partie avant.

```
(define-struct biliste (ar av) #:transparent)
```

Implémentez les trois petites fonctions, où *Biliste** dénote l'ensemble des *bilistes* non vides :

`kar` : *Biliste** $\to$ *Elément*, `kdr` : *Biliste** $\to$ *Biliste*, `retro-kdr` : *Biliste** $\to$ *Biliste*

de sorte que la session suivante fonctionne comme prévu. On déclenchera bien entendu une *erreur* dans le cas où le `kar`, le `kdr` ou le `retro-kdr` n'existerait pas.

```
> (define BL (make-biliste '() '(1 2 3 4 5 6)))        ; [1] ↔ 2 ↔ 3 ↔ 4 ↔ 5 ↔ 6
> (retro-kdr BL)                                       ; je ne peux pas reculer !
ERROR : retro-kdr : Impossible !
> (define focus (kdr (kdr (kdr BL))))                  ; j'avance 3 fois
> focus                                                ; où suis-je ?
#(struct:biliste (3 2 1) (4 5 6))                      ; 1 ↔ 2 ↔ 3 ↔ [4] ↔ 5 ↔ 6
> (kar focus)                                          ; l'élément courant
4
> (kar (kdr focus))                                    ; l'élément suivant
5
> (kar (retro-kdr focus))                              ; l'élément précédent !
3
```

FIN)

2. Hors-sujet, mais notre technique utilise les *zippers* de G. Huet, voir la Wikipedia à *Zipper (data structure)*.