Licence 3 d'Informatique — Programmation fonctionnelle — TP 4

Début : jeudi 19 octobre 2023

Cette quatrième série d'exercices de travaux pratiques vise d'abord à donner quelques exemples d'utilisation des valeurs multiples. Nous proposons ensuite des exercices de révision des listes, en particulier à propos des listes d'**associations**. Comme de coutume, le présent énoncé est accompagné d'un fichier source de démarrage for-lc-4.scm, disponible, avec le présent fichier lc-4.pdf, sous le cours « Programmation fonctionnelle, scripts — XML » de moodle, dans le répertoire « Travaux pratiques Scheme 2023 > for-lc-4 ».

1 *Intrada*

$\Longrightarrow$ Définir une fonction `complement`, telle que l'évaluation de l'expression `(complement` p_1?`)` — où p_1? est un prédicat [1] à un argument formel — retourne un prédicat à un argument formel qui retourne `#t` (resp. `#f`) lorsque p_1? retourne `#f` (resp. une valeur logique *vrai*). Exemples :

```
((complement odd?) 2023)  ⟹ #f
((complement zero?) 2023) ⟹ #t
```

(rappelons que `odd?` est la fonction de test des nombres entiers relatifs pairs).

$\Longrightarrow$ Donner à présent la fonction `remove`, telle que l'évaluation de l'expression `(remove` p_1? l`)` retourne, dans une liste et selon l'ordre d'apparition, les éléments de la liste linéaire l, *sauf* ceux qui satisfont le prédicat p_1? ; exemples :

```
(remove even? '(19 21 10 2023 2024 2025)) ⟹ (19 21 2023 2025)
(remove zero? '(19 21 10 2023 2024 2025)) ⟹ (19 21 10 2023 2024 2025)
```

Indication Le fichier for-lc-4.scm vous fournit la fonction `retain` (*cf.* figure 1), vue en cours. Vous pouvez l'utiliser, ainsi que la fonction précédente `complement`, pour l'écriture de la fonction `remove`.

2 *Andante doloroso* : valeurs multiples

Nous rappelons que la fonction `values` de Scheme permet de retourner *plusieurs* valeurs (éventuellement *zéro* valeur), la *production* de n valeurs ($n \in \mathbb{N}$) pouvant être *consommée* par une fonction acceptant n arguments, la jonction entre le producteur et le consommateur étant effectuée par la fonction `call-with-values`. Scheme fournit également des formes spéciales facilitant la manipulation de valeurs multiples : `let-values` et `let*-values`, analogues aux formes spéciales `let` et `let*` pour les valeurs « simples ». Rappelons également que les deux expressions

1. C'est-à -dire une fonction dont le résultat est une valeur logique.

```
#!r7rs

(import (scheme base) (scheme cxr) (scheme write))

(define (writeln/return x)
  (write x)
  (newline)
  x)

(define (retain p1? l)
  (if (null? l)
      '()
      (let ((first (car l))
            (more (retain p1? (cdr l))))
        (if (p1? first) (cons first more) more))))

(call-with-values (lambda () (values 'L 3)) cons) ⟹ (L . 3) ; Ces exemples sont donnés
                                                             ; dans le fichier for-lc-4.scm.

(let ((a 'sure)
      (b 'here)
      (x 'little)
      (y 'players))
  (let-values (((a b) (values x y))
               ((x y) (values a b)))
    (list a b x y))) ⟹ (little players here sure)

(let ((a 'sure)
      (b 'here)
      (x 'little)
      (y 'players))
  (let*-values (((a b) (values x y))
                ((x y) (values a b)))
    (list a b x y))) ⟹ (little players little players)

(let-values (((first) (car '(very little players)))
             ((the-quotient the-remainder) (truncate/ 2000 9)))
  (cons the-quotient (cons the-remainder first))) ⟹ (222 2 . very)

(define (take n l)
  (if (zero? n) '() (cons (car l) (take (- n 1) (cdr l)))))

(define (drop n l)
  (if (zero? n) l (drop (- n 1) (cdr l))))

(define (take-while p1? l)
  (if (null? l)
      '()
      (let ((first (car l)))
        (if (p1? first) (cons first (take-while p1? (cdr l))) '()))))

(define (drop-while p1? l)
  (if (or (null? l) (not (p1? (car l)))) l (drop-while p1? (cdr l))))
```

Figure 1 : Fonction `retain` et utilisation de valeurs multiples.

```
(define-syntax write-values
  (syntax-rules ()
    ((write-values the-values) (let ((nothing-string "*nothing*")
                                     (and-string " *and* "))
                                 (call-with-values (lambda () the-values)
                                   (lambda arg-list
                                     (if (null? arg-list)
                                         (display nothing-string)
                                         (begin
                                           (write (car arg-list))
                                           (for-each (lambda (arg)
                                                       (display and-string)
                                                       (write arg))
                                                     (cdr arg-list))))
                                           (newline)
                                           #t)))))
```

Figure 2 : Écriture de valeurs multiples.

E et `(values` E`)` — E étant une expression quelconque — sont synonymes [2]. Enfin, mentionnons que la présentation de plusieurs valeurs alors qu'une seule est attendue n'est pas définie dans la norme de Scheme [3] : par exemple, la norme ne précise pas quel est le résultat de l'expression suivante :

```
(let ((x (values 'I 'dunno)))
  x) ⟹ ???
```

Vous pouvez vérifier qu'elle provoque une erreur de l'interprète DrRacket [4].

La figure 1 donne quelques utilisations des formes utilisées pour les valeurs multiples de Scheme. Nous ajoutons à ces exemples la définition de `write-values`, donnée dans le fichier for-lc-4.scm et dans la figure 2, permettant l'affichage de valeurs multiples, après quoi la valeur retournée est `#t` [5]. Dans la suite de ce § 2, « n » désigne un entier naturel, « l » une liste linéaire et « p_1? » désigne un prédicat à un argument formel. Pour mémoire, nous rappelons les fonctions `take`, `drop`, `take-while` et `drop-while`, vues en cours et données dans la figure 1 :

— l'évaluation de l'expression `(take` n l`)` retient dans une liste linéaire les l premiers éléments de la liste l ; cette fonction ne s'emploie que si la liste linéaire l contient au moins n éléments ;

— l'évaluation de l'expression `(drop` n l`)` retourne une liste linéaire composée des éléments de l et apparaissant dans le même ordre, sauf les n premiers qui sont omis ; comme la fonction précédente, elle ne s'emploie que si la liste linéaire l contient au moins n éléments ;

2. C'est pourquoi la fonction `values` avec un seul argument peut être vue comme la fonction identité, comme nous l'avions fait remarquer dans l'énoncé précédent de travaux pratiques.

3. Ce qui signifie que les interprètes de Scheme ont toute latitude sur ce point.

4. . . . mais tous les interprètes de Scheme ne réagissent pas ainsi ; par exemple, l'interprète bigloo lie la variable à la première valeur produite — et au nombre `0` si cette première valeur n'existe pas — : ici, `x` serait liée au symbole `I`.

5. Il ne s'agit pas d'une fonction, mais d'une *macro*. L'écriture de *macros* en Scheme dépasse du cadre de l'unité *Programmation fonctionnelle et scripts* enseignée en L3.

— l'évaluation de l'expression `(take-while` p_1? l`)` retient dans le résultat les éléments placés au début de la liste linéaire l tant qu'ils satisfont le prédicat p_1? ; appliquée à la liste vide, la fonction `take-while` retourne la liste vide ;

— l'évaluation de l'expression `(drop-while` p_1? l`)` retourne dans une liste linéaire le premier élément de l qui ne satisfait pas le prédicat p_1?, ainsi que tous ceux qui le suivent ; appliquée à la liste vide, la fonction `drop-while` retourne la liste vide.

Exemples :

```
(take 2 '(19 21 10 2023 2024 2025))          ⟹ (19 21)
(drop 3 '(19 21 10 2023 2024 2025))          ⟹ (2023 2024 2025)
(take-while odd? '())                        ⟹ ()
(take-while odd? '(19 21 10 2023 2024 2025)) ⟹ (19 21)
(drop-while odd? '())                        ⟹ ()
(drop-while odd? '(19 21 10 2023 2024 2025)) ⟹ (10 2023 2024 2025)
```

⟹ Définir les trois fonctions suivantes :

— `split-at`, telle que l'évaluation de l'expression `(split-at` n l`)` retourne deux valeurs : la liste linéaire composée des n premiers éléments de l, puis la liste linéaire des éléments de l qui suivent les n premiers ; cette fonction `split-at` ne s'emploie que si la liste l contient au moins n éléments ; par exemple :

```
(split-at 3 '(19 21 10 2023 2024 2025)) ⟹ (19 21 10)     (2023 2024 2025)
```

une réalisation de cette fonction `split-at` pourrait être :

```
(define (split-at n l)
  (values (take n l) (drop n l)))
```

mais il s'agit vraiment d'une version de petit joueur, car le début de la liste linéaire `l` est parcouru deux fois ; il vous est demandé d'écrire une version récursive de la fonction `split-at`, n'effectuant que le strict minimum de parcours de la liste linéaire ;

— `span`, telle que l'évaluation de l'expression `(span` p_1? l`)` retourne deux valeurs : la liste linéaire des premiers éléments de l, tant qu'ils satisfont le prédicat p_1?, puis la portion de la liste linéaire l qui commence par le premier élément de l qui ne satisfait pas le prédicat p_1? ; appliquée à la liste vide, la fonction `span` retourne deux valeurs toutes deux égales à la liste vide ; exemples :

```
(span odd? '())                          ⟹ ()        ()
(span odd? '(19 21 10 2023 2024 2025)) ⟹ (19 21)  (10 2023 2024 2025)
```

de façon analogue, une réalisation de petit joueur pour cette fonction `span` pourrait être :

```
(define (span p1? l)
  (values (take-while p1? l) (drop-while p1? l)))
```

mais pour les mêmes raisons que précédemment, il vous est demandé d'écrire une version récursive de la fonction `span`, n'effectuant que le strict minimum de parcours de la liste linéaire ;

— `partition`, telle que l'évaluation de l'expression (`partition` p_1? l) retourne deux valeurs qui sont des listes linéaires : la première (resp. seconde) liste groupe les éléments de l qui satisfont (resp. ne satisfont pas) le prédicat p_1? ; les élément de ces deux listes sont donnés suivant leur ordre d'apparition dans la liste l originale ; exemples :

```
(partition odd? '(19 21 10 2023 2024 2025))
        ⟹ (19 21 2023 2025) (10 2024)
(partition zero? '(19 21 10 2023 2024 2025))
        ⟹ () (19 21 10 2023 2024 2025)
```

là aussi, une réalisation de petit joueur pour cette fonction `partition` pourrait être :

```
(define (partition p1? l)
  (values (retain p1? l) (remove p1? l)))
```

mais là encore, il vous est demandé d'écrire une efficace n'effectuant qu'un seul parcours de la liste linéaire ;

— `partition-tr`, telle que l'évaluation de l'expression (`partition-tr` p_1? l) retourne elle aussi deux valeurs qui sont des listes linéaires : la première (resp. seconde) liste groupe les éléments de l qui satisfont (resp. ne satisfont pas) le prédicat p_1? ; par contre :

  ⋆ l'ordre dans lequel apparaissent les éléments de ces deux listes n'a pas d'importance ;

  ⋆ il est demandé, pour cette fonction `partition-tr`, une réalisation efficace utilisant une récursivité terminale.

⟹ Soient f_1, g_1, h_1 trois fonctions utilisables avec un argument formel, donner la définition d'une fonction `map-3-functions`, telle que l'évaluation de l'expression :

(`map-3-functions` f_1 g_1 h_1 l)

retourne trois valeurs : la liste des résultats des applications successives de f_1 aux membres de l, puis même chose en utilisant les fonctions g_1 et h_1. Il est en outre demandé de ne parcourir la liste linéaire l qu'une seule fois. Exemple :

```
(map-3-functions (lambda (x) (+ x 1)) (lambda (x) (+ x 2)) (lambda (x) (* x 2))
                 '(19 21 10 2023 2024 2025)) ⟹
        (20 22 11 2024 2025 2026) (21 23 12 2025 2026 2027)
        (38 42 20 4046 4048 4050)
```

3 Mélanges

Pour cet exercice, nous allons à nouveau considérer des listes d'informations telles que `miles-davis-r` et `stan-getz-r` — de type *JAZZ-MBD* —, déjà manipulées durant les exercices précédents de travaux pratiques.

⟹ Définir une fonction `extract-max-author-nb/max-year`, telle que l'évaluation de l'expression (`extract-max-author-nb/max-year` *j-list*) — où *j-list* est une liste linéaire d'informations, de type *JAZZ-MBD* — retourne deux valeurs : le plus grand nombre d'auteurs pour une pièce de *j-list*, et l'année qui correspond à l'enregistrement le plus récent de *j-list*. Exemples :

```
(extract-max-author-nb/max-year miles-davis-r) ⟹ 4  1993
(extract-max-author-nb/max-year stan-getz-r)   ⟹ 1  1994
```

Pour le calcul de la première valeur, nous conviendrons qu'une pièce dont les auteurs sont inconnus équivaut à une pièce *sans* auteur. Nous conviendrons également que les deux valeurs retournées sont égales au nombre `0` si cette fonction `extract-max-author-nb/max-year` est appliquée à la liste vide.

Indication Vous pouvez utiliser une récursivité terminale, fondée sur une fonction récursive locale admettant *deux* arguments accumulateurs.

4 Listes d'associations

Nous rappelons qu'une **liste d'associations** est une liste linéaire de paires appelées **associations**, dans lesquelles l'élément de tête joue un rôle particulier : c'est la **clé** de l'association. Remarquons au passage que les listes `miles-davis-r` et `stan-getz-r` peuvent parfaitement être considérées comme des listes d'associations. Rechercher une clé *key* dans une liste d'associations *alist* s'effectue au moyen des fonctions `assq` et `assoc`. La première (resp. seconde) utilise la fonction `eq?` (resp. `equal?`) pour comparer la clé *key* aux clés successives de la liste *alist*. En cas de succès, on s'arrête à la première occurrence trouvée pour *key* et c'est l'association tout entière qui est retournée :

```
(assq 'd1 miles-davis-r) ⟹ (d1 "New-York Girl" ?? 1972 ?? '(c40))
```

Si *key* n'est pas une clé de la liste *alist*, les fonctions `assq` et `assoc` retournent la valeur logique `#f` :

```
(assq 'd39 miles-davis-r) ⟹ #f
```

⟹ Définir une fonction `assq-split`, telle que l'évaluation de l'expression `(assq-split` *x alist*`)` — où *x* est une donnée quelconque et *alist* une liste d'associations — retourne *deux* valeurs :

- la première association de *alist* dont la clé est *x*, ou la valeur `#f` si une telle association n'existe pas ;
- une liste linéaire formée de tous les éléments de *alist* différents de l'association retournée en première valeur.

Notez bien que :

- la fonction `assq-split` utilise la fonction `eq?` pour comparer la valeur *x* aux clés successives de la liste d'associations *alist* ;
- peu importe l'ordre d'apparition des éléments de la liste linéaire retournée en seconde valeur.

Exemples de réalisations possibles :

```
(assq-split 'L '((M . 2) (L . 3) (Nautilus . -20000)))
    ⟹ (L . 3)   ((M . 2) (Nautilus . -20000))
(assq-split 'Titanic '((M . 2) (L . 3) (Nautilus . -20000)))
    ⟹ #f        ((M . 2) (L . 3) (Nautilus . -20000))
```

Indication On pourra adopter une récursivité terminale utilisant une fonction locale avec un argument accumulateur.

$\Longrightarrow$ Écrire une fonction `merge-alists`, telle que l'évaluation de l'expression :

(`merge-alists` *alist* $alist_0$)

— où *alist* et $alist_0$ sont des listes d'associations dont les clés sont des symboles — retourne deux valeurs :

- une liste d'associations composée des associations de *alist* dont la clé n'appartient pas à $alist_0$, des associations de $alist_0$ dont la clé n'appartient pas à *alist*, et des associations *identiques* dont la clé est à la fois une clé de *alist* et de $alist_0$; par « association identique », nous entendons que les valeurs associées à de telles clés sont les mêmes — au sens de la fonction `equal?` — dans *alist* et $alist_0$;
- une liste linéaire des clés communes à *alist* et $alist_0$, mais les valeurs associées sont différentes dans les deux lists ;

l'ordre d'apparition des éléments dans ces deux listes linéaires retournées par notre fonction `merge-alists` étant indifférent. Voici un exemple de réalisation possible :

```
(merge-alists '((players . little) (L . 3) (here . ufc))
              '((L . 3) (there . ubfc) (players . master)))
    ⟹ ((there . ubfc) (L . 3) (here . ufc))   (players)
```

Indication On pourra là aussi utiliser une récursivité terminale, fondée sur une fonction locale avec plusieurs arguments accumulateurs. La fonction précédente, `assq-split`, pourrait s'avérer, elle aussi, très utile.

5 Révisions

$\Longrightarrow$ Soient l_0 et l_1 deux listes linéaires quelconques, donner les deux fonctions suivantes.

- `included-into?`, telle que l'évaluation de l'expression (`included-into?` l_0 l_1) retourne `#t` si les éléments de l_0 appartiennent à l_1, `#f` sinon. On utilisera la fonction `equal?` pour comparer les éléments de l_0 à ceux de l_1. Exemples :

```
(included-into? '(19 10 2023) '(2023 19 21 10)) ⟹ #t
(included-into? '(19 10 2023) '(2023 20 21 10)) ⟹ #f
```

  Indication On pourra utiliser la fonction prédéfinie `member` de Scheme.

- `same-elts?`, telle que l'évaluation de l'expression (`same-elts?` l_0 l_1) retourne `#t` si les listes linéaires l_0 et l_1 sont formées des mêmes éléments, les ordres d'apparition étant indifférents. Comme précédemment, on utilisera la fonction `equal?` pour comparer les éléments de l_0 à ceux de l_1. Exemples :

```
(same-elts? '(19 10 2023) '(2023 19 10)) ⟹ #t
(same-elts? '(19 10 2023) '(2023 20 10)) ⟹ #f
```

  Indication Penser à utiliser deux fois la fonction précédente `included-into?`.

$\Longrightarrow$ Soient *l* une liste linéaire quelconque et p_1? un prédicat à un argument formel, écrire une fonction `find-tail`, telle que l'évaluation de l'expression `(find-tail` p_1? *l*`)` retourne la portion de la liste *l* qui commence par le premier élément satisfaisant le prédicat p_1?. Si aucun élément de *l* ne satisfait ce prédicat, la fonction `find-tail` retourne `#f`. Exemples :

```
(find-tail even? '(19 21 10 2023)) ⟹ (10 2023)
(find-tail zero? '(19 21 10 2023)) ⟹ #f
```

(rappelons que `even?` est la fonction de test des entiers relatifs pairs).

$\Longrightarrow$ Donner deux fonctions `memq-rd` et `member-rd` — le suffixe « `-rd` » étant mis pour « *redefined* » —, redéfinissant respectivement les fonctions prédéfinies `memq` et `member` à partir de la fonction `find-tail` précédente.

$\Longrightarrow$ Utiliser également la fonction `find-tail` précédente pour la définition d'une nouvelle fonction `assq-ff`, telle que l'évaluation de l'expression `(assq-ff` *x alist*`)` — où *x* est une donnée quelconque et *alist* une liste d'associations — retourne la portion de la liste *alist* qui commence par la première association dont la clé est *x*. Si une telle association n'existe pas, la fonction `assq-ff` retourne `#f`. Les comparaisons des diverses clés de *alist* avec l'élément *x* s'effectueront au moyen de la fonction `eq?`. Exemples :

```
(assq-ff 'L '((M . 2) (L . 3) (Nautilus . -20000)))
        ⟹ ((L . 3) (Nautilus . -20000))
(assq-ff 'Titanic '((M . 2) (L . 3) (Nautilus . -20000)))   ⟹ #f
```