Licence 3 d'Informatique — Programmation fonctionnelle — TP 3

Début : jeudi 28 septembre 2023

1 *Start*

Le petit parcours qui vous est proposé ci-après vise à vous familiariser avec les *paires* du langage Scheme, d'une part, avec la manipulation de *symboles*, d'autre part. Il se continue avec l'introduction des listes *quelconques* et quelques exercices de tri de listes linéaires. Commençons par mettre en évidence la différence entre l'écriture du point séparateur des éléments d'une paire et le point décimal. Évaluez les *six* expressions suivantes — qui se trouvent dans le fichier source d'amorce for-lc-3.scm, disponible, avec le présent fichier lc-3.pdf, sous le cours suivant de moodle :

Programmation fonctionnelle, scripts — XML

dans le répertoire :

Travaux pratiques Scheme 2023 > for-lc-3

Notez la différence de *look* que présentent les divers résultats :

```
(cons 0 1)                ⟹ ??
(cons 0.1 0.1)            ⟹ ??
(cons (cons 0.1 0.1) 1)   ⟹ ??
(cons 0.1 '())            ⟹ ??
(cons '() 0.1)            ⟹ ??
(cons '() '())            ⟹ ??
```

Voici quelques définitions de variables qui se trouvent elles aussi dans le fichier d'amorce for-lc-3.scm :

```
(define current-year 2023)
(define next-year (+ current-year 1))

(define academic-year
  (cons current-year next-year))

(define academic-year-0
  (cons current-year 'next-year))

(define academic-year-1
  (cons 'current-year 'next-year))

(define operations (cons + -))
(define symbols (cons '+ '-))
(define two-numbers (cons 0 1))

(define something
  (cons (cons 0.1 0.1) 1))
```

Pour chacune des évaluations reproduites ci-après, tentez d'abord de trouver la solution *par vous-même*, ensuite évaluez effectivement l'expression pour vérifier que vous avez trouvé le bon résultat. *Attention!!* certaines expressions peuvent produire des erreurs à l'évaluation : dans de tels cas, il est important de comprendre la raison de l'erreur [1].

⟹ Quelques exemples de ce même fichier d'amorce for-lc-3.scm illustrent l'emploi de la forme spéciale `quote` :

```
academic-year          ⟹ ??
(quote academic-year)  ⟹ ??
'academic-year         ⟹ ??
'undefined             ⟹ ??
undefined              ⟹ ??
+                      ⟹ ??
'+                     ⟹ ??
```

1. Même si les messages d'erreur de DrRacket deviennent assez rapidement explicites dès qu'on commence à pratiquer le langage, il est intéressant de les identifier et de les relier aux erreurs commises. Cela facilitera la recherche d'erreurs lorsque plus tard, vous verrez à nouveau les mêmes messages d'erreur.

```
(0 . 1)                 ⟹ ??
'(0 . 1)                ⟹ ??
(+ . 1)                 ⟹ ??
'(+ . 1)                ⟹ ??
```

À présent, voyons comme se comportent les fonctions `car` et `cdr` :

```
(car academic-year)     ⟹ ??
(cdr academic-year)     ⟹ ??
(car next-year)         ⟹ ??
academic-year-0         ⟹ ??
(cdr academic-year-0)   ⟹ ??
academic-year-1         ⟹ ??
```

Quelques mélanges :

```
((car operations)
 (car two-numbers)
 (cdr two-numbers))         ⟹ ??

((cdr symbols)
 (cdr two-numbers)
 (car two-numbers))         ⟹ ??

(cons (cdr symbols)
      (car two-numbers))    ⟹ ??

(cons (cdr operations)
      (cdr two-numbers))    ⟹ ??

((cdr operations) (cdr two-numbers))
                            ⟹ ??

(car something)             ⟹ ??
(cdr (car something))       ⟹ ??
(car (cdr something))       ⟹ ??
```

Familiarisez-vous aussi avec les diverses imbrications des fonctions `car` et `cdr`[2] — la variable `movie-title` a été définie dans le fichier d'amorce for-lc-3.scm — :

```
(define movie-title
  '((Mad . Max) beyond the
    (Thunderdome . ())))

(caar movie-title)      ⟹ ??
(cdar movie-title)      ⟹ ??
(cadr movie-title)      ⟹ ??
(cddr movie-title)      ⟹ ??
(caddr movie-title)     ⟹ ??
(cdddr movie-title)     ⟹ ??
(cadddr movie-title)    ⟹ ??
(cddddr movie-title)    ⟹ ??
```

2 Des paires aux listes

2.1 Paires de nombres réels

Dans cette section, les paires que nous allons manipuler représenteront des *vecteurs*, éléments de l'espace vectoriel $\mathbb{R}^2$.

⟹ Écrire la fonction `free-family?`, telle que l'évaluation de l'expression :

`(free-family?` v w`)`

retourne `#t` si la famille de vecteurs $\{v, w\}$ est libre, `#f` si elle est liée ; nous rappelons que si $\overrightarrow{v} = (v_1, v_2)$ et $\overrightarrow{w} = (w_1, w_2)$ — avec $v_1, v_2, w_1, w_2 \in \mathbb{R}$ — cette dernière proposition est alors équivalente au calcul suivant sur le déterminant constitué des coordonnées :

$$\begin{vmatrix} v_1 & w_1 \\ v_2 & w_2 \end{vmatrix} \neq 0$$

2.2 Passer aux listes

⟹ Trouvez l'écriture simplifiée des trois premières listes[3] données dans la figure 1 et le fichier d'amorce for-lc-3.scm. Vérifiez en évaluant les expressions que vos résultats sont justes.

⟹ Considérons les évaluations suivantes de la figure 1 : dans les expressions à gauche

2. Les imbrications à deux niveaux de ces fonctions :

`caar` `cadr` `cdar` `cddr`

appartiennent à la bibliothèque de base du langage Scheme, c'est-à-dire à `(scheme base)`. À partir de trois niveaux — et jusqu'à quatre niveaux — :

`caaar, caadr, ...,`
`caaaar, ..., cddddr`

elles ont toutes été rangées dans la bibliothèque prédéfinie `(scheme cxr)`. Vous n'avez pas à vous préoccuper de l'importation de cette bibliothèque, réalisée dans le fichier d'amorce for-lc-3.scm.

3. ... dont le contenu est librement inspiré de la chanson *Everybody Needs Somebody to Love*, popularisée par le groupe des *Rolling Stones* dans les années soixante. C'était il y a bien longtemps... mais à vrai dire, cette chanson a traversé pas mal d'années.

```
'(Sometimes . ((to . ()) . ((() . miss) . (now . ()))))                    ⟹ ??
'((Everybody . ()) . (() . wants) . (somebody . ()))                       ⟹ ??
'((Thats . ((() . ()) . why)) . ((() . I) . ((need . (you . baby)) . ()))) ⟹ ??

(***replace*** (***replace*** 'Aint 'nobody 'else 'around) '())
    ⟹ ((Aint nobody else around))

(***replace*** 'Someone (***replace*** 'to '(squeeze))) ⟹ (Someone (to squeeze))

(***replace*** (***replace*** 'I 'need 'you)
               (***replace*** 'you (***replace*** 'you '())))
    ⟹ ((I need you) (you you))

(***replace*** (***replace*** 'And 'I) (***replace*** '(need) '(you))
               (***replace*** 'you '(you)))
    ⟹ (And I need you you you)

(***replace*** 'I-m
               (***replace*** '((so glad)) (***replace*** 'to 'be 'here 'tonight)))
    ⟹ (I-m (so glad) to be here tonight)
```

Figure 1 : *Playing with lists.*

---

du symbole « ⟹ », remplacer chaque occurrence du symbole « ***replace*** » par « `cons` », « `list` », ou « `append` » de telle sorte que le résultat obtenu coïncide avec celui qui est affiché après le symbole « ⟹ ». Puis vérifier vos réponses.

3 Mini-base de données

Une base de données peut être vue comme une *collection d'informations* organisées par thèmes, et accessibles par des *clés*. Dans l'exemple illustratif ci-après, nous allons considérer une mini-base de données concernant quelques enregistrements de jazz. La manipulation de ces mini-bases de données nous montrera en outre comment mettre en œuvre une approche visant à séparer *représentation* et *abstraction*.

3.1 Types utilisés

Une donnée représentant l'enregistrement d'une pièce de jazz — que nous qualifierons par le type « *JAZZ-R* » dans toute la suite — consiste en :

- une **clé** identifiant l'enregistrement de façon univoque ;
- le **titre** de la pièce, donné sous la forme d'une chaîne de caractères ;
- l'(es) **auteur**(s) de cette pièce, donné(s) par une liste linéaire de symboles ;
- l'**année** où l'enregistrement a été réalisé ;
- l'**éditeur-propriétaire** de l'œuvre musicale, donné par un symbole ;
- une liste linéaire de symboles représentant des *compact discs* où a été repris cet enregistrement.

Certaines informations peuvent être inconnues, auquel cas nous représenterons cette situation par le symbole « `??` » et utiliserons, pour des raisons de lisibilité des programmes, les deux définitions `unknown` et `unknown?`, présentes dans le fichier d'amorce for-lc-3.scm (*cf.* figure 2). Remarquez que la fonction `unknown?` peut s'appliquer à une donnée tout à fait quelconque [4].

4. L'utilisation du *duo* `unknown`/`unknown?` permet en outre une modification des conventions sans danger

```
(define unknown '??)

(define (unknown? x)
  (eq? x unknown))

(define (mk-recording key-0 title-0 author-list-0 year-0 publisher-0 key-list-0)
  ;; SYMBOL × STRING × LIST-of[SYMBOL]^⋈ × INTEGER × SYMBOL^⋈ × LIST-of[SYMBOL]^⋈
  ;;      → JAZZ-R
  (list key-0 title-0 author-list-0 year-0 publisher-0 key-list-0))
```

Figure 2 : Représentation de l'information inconnue et constructeur du type *JAZZ-R*.

Les éléments autres que la clé seront appelés *éléments d'information* et désignés par le type « *INFORMATION-E* ». Nous allons définir une représentation pour les objets de type *JAZZ-R* ; plus précisément, nous allons convenir qu'ils sont représentés par des listes linéaires de la forme :

(⟨*key*$_0$⟩ ⟨*title*$_0$⟩ ⟨*author-list*$_0$⟩ ⟨*year*$_0$⟩ ⟨*publisher*$_0$⟩ ⟨*key-list*$_0$⟩)

Le constructeur `mk-recording`, retournant une donnée de type *JAZZ-R* à partir de ses informations, a été défini en figure 2 et dans le fichier d'amorce for-lc-3.scm — les types :

SYMBOL	*INTEGER*
STRING	*LIST-of*[...]

désignent respectivement les symboles, les chaînes de caractères, les entiers relatifs, les listes linéaires ; le signe « ...$^{\bowtie}$ » indiquant que l'information correspondante peut être inconnue — comme cela vous est montré dans les exemples de la figure 3.

⟹ Expliquez en quoi la réalisation du constructeur `mk-recording` donnée dans la figure 2 est différente — et bien meilleure — que la réalisation ci-après :

```
(define mk-recording list)
```

⟹ Donner les six sélecteurs suivants du type *JAZZ-R* — remarquez bien ce qui vous a été aussi précisé dans la figure 2 à propos du constructeur `mk-recording` : la clé, le titre et l'année de publication d'un enregistrement ne peuvent pas être des informations inconnues — :

— `key` : *JAZZ-R* → *SYMBOL*

— `title` : *JAZZ-R* → *STRING*

— `authors`, `cds` :

$$\textit{JAZZ-R} \rightarrow \textit{LIST-of}[\textit{SYMBOL}]^{\bowtie}$$

— `year` : *JAZZ-R* → *INTEGER*

— `publisher` : *JAZZ-R* → *SYMBOL*$^{\bowtie}$

retournant chaque élément d'information correspondant — éventuellement la valeur indiquant que l'information est inconnue — ; dans la suite, ces sélecteurs seront désignés par le type « *SELECTOR* ». Les exemples qui seront utilisés sont donnés dans la figure 3 et le fichier d'amorce for-lc-3.scm.

À partir de maintenant, les données de type *JAZZ-R* ne seront plus manipulées *qu'au travers des définitions précédentes*, conformément à la séparation entre *représentation* et *abstraction*.

Par contre, notez que les valeurs des deux variables `miles-davis-r` et `stan-getz-r` (*cf.* figure 3) étant « réellement » des listes linéaires, elles peuvent être manipulées à l'aide des fonctions usuelles sur les listes linéaires.

pour la maintenance du programme qui les utilise.

```
(define miles-davis-r
  (list (mk-recording 'd0 "On the Corner" unknown 1972 unknown '(c40))
        (mk-recording 'd1 "New-York Girl" unknown 1972 unknown '(c40))
        (mk-recording 'd2 "Thinkin' One Thing and Doin' Another" unknown 1972 unknown
                      '(c40))
        (mk-recording 'd3 "One and One" unknown 1972 unknown '(c40))
        (mk-recording 'd4 "Well You Needn't" '(Thelonius-Monk) 1954 'Blue-Ribbon-Music
                      '(c41))
        (mk-recording 'd5 "Love for Sale" '(Cole-Porter) 1958 'Chappell '(c41))
        (mk-recording 'd6 "Something Else" '(Miles-Davis) 1958 'EMI '(c41))
        (mk-recording 'd7 "Dear Old Stockholm" '(Stan-Getz P-Golly) 1952
                      'Windswept-Pacific-Music '(c41))
        (mk-recording 'd8 "Black Satin" unknown 1972 unknown '(c40))
        (mk-recording 'd9 "Mr. Freedom X" unknown 1972 unknown '(c40))
        (mk-recording 'd10 "Helen Butte" unknown 1972 unknown '(c40))
        (mk-recording 'd11 "Boplicity" '(Cleo-Henry) 1949 'Campbell '(c40))
        (mk-recording 'd12 "Ray's Idea" '(Ray-Bonner W-G-Fuller) 1953 'Bosworth '(c41))
        (mk-recording 'd13 "Yesterdays" '(Kern Harbach) 1952 'Universal '(c42))
        (mk-recording 'd14 "Vote for Miles" unknown 1972 unknown '(c40))
        (mk-recording 'd15 "Deception" '(Miles-Davis) 1950 'Sony '(c41))
        (mk-recording 'd16 "Israel" '(John-Carisi) 1949 'EMI '(c41))
        (mk-recording 'd17 "How Deep is the Ocean" '(Irving-Berlin) 1952 'EMI '(c41))
        (mk-recording 'd18 "Kelo" '(Jay-Jay-Johnson) 1953 'Kensington-Music '(c41))
        (mk-recording 'd19 "Woody 'n' You" '(Dizzy-Gillepsie) 1952 'Chappell '(c41))
        (mk-recording 'd20 "Chance It" '(Oscar-Petitford) 1952 'Orpheus-Music '(c41))))

(define stan-getz-r
  (list (mk-recording 'g30 "Autumn Leaves" '(Kosma Prevert) 1980 'Peter-Maurice-Co-Ltd
                      '(c44))
        (mk-recording 'g31 "Nature Boy" '(Abbon) 1980 'Chappell '(c44))))
```

Figure 3 : Mini-bases de données utilisées dans l'exercice de travaux pratiques.

---

4 Opérations sur les enregistrements

Dans toute la suite de cet énoncé, « *j-list* » va représenter une liste linéaire dont les éléments sont tous des objets de type *JAZZ-R*. Une telle liste linéaire sera désignée par le type « *JAZZ-MBD* ». Des exemples de telles listes — qu'on peut interpréter comme des « mini-bases de données » — sont bien sûr les variables `miles-davis-r` et `stan-getz-r` de la figure 3.

$\Longrightarrow$ Donner une fonction `unique-keys?` :

$$\textit{JAZZ-MBD} \rightarrow \textit{LOGICAL-VALUE}$$

— où « *LOGICAL-VALUE* » désigne une valeur logique —, telle que l'évaluation de l'expression (`unique-keys?` *j-list*) retourne la valeur `#t` si les clés des éléments de *j-list* sont toutes différentes deux à deux, `#f` sinon. Vérifier que vous obtenez bien :

```
(unique-keys? miles-davis-r) ⟹ #t
(unique-keys? stan-getz-r)   ⟹ #t
```

Indication Supposons que les n clés de la liste *j-list* soient $k_0, \ldots, k_n$. Il suffit de vérifier que :

$$k_0 \notin \{k_1, \ldots, k_n\}, k_1 \notin \{k_2, \ldots, k_n\}$$

et ainsi de suite. Nous vous rappelons qu'il est possible d'utiliser les fonctions prédéfinies `member` ou `memq` pour réaliser des tests d'appartenance.

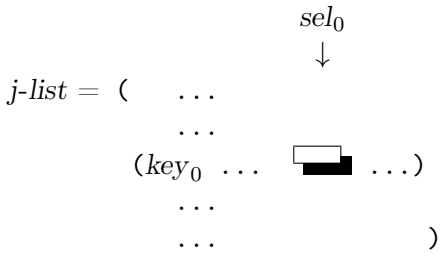

Figure 4 : Effet de `get-information-e`.

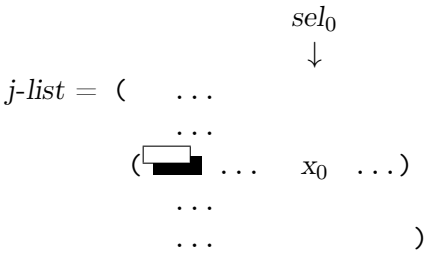

Figure 5 : Effet de `get-key`.

Enfin, vous pouvez employer la fonction prédéfinie `map` pour obtenir cette liste de clés à partir de la liste *j-list*. Rappelons que cette fonction `map` applique la fonction en premier argument — utilisable avec un argument formel — à chaque membre d'une liste linéaire [5], la liste des résultats étant retournée :

```
(map (lambda (x) (+ x 1)) '(28 9 2023))
  ⟹ (29 10 2024)
```

⟹ Écrire une fonction `get-information-e` :

$$SYMBOL \times SELECTOR \times JAZZ\text{-}MBD \rightarrow INFORMATION\text{-}E \cup \{\texttt{\#f}\}$$

telle que l'évaluation de l'expression :

`(get-information-e` key_0 sel_0 *j-list*`)`

cherche l'élément de *j-list* dont la clé est key_0 :

— si un tel élément existe, lui appliquer le sélecteur sel_0 (*cf.* figure 4) ;

— sinon le résultat est `#f`.

En déduire comment obtenir le titre de l'œuvre enregistrée par Miles Davis et dont la clé est `d14`. Puis utiliser la fonction `get-information-e` pour obtenir :

— les auteurs de l'œuvre enregistrée par Miles Davis et dont la clé est `d13` ;

— l'année où Stan Getz a enregistré l'œuvre dont la clé est `g30` ;

— l'éditeur de l'œuvre enregistrée par Miles Davis et dont la clé est `d14`.

Nous aurions pu utiliser la valeur `#f` — au lieu du symbole « `??` » — pour représenter une information inconnue. Mais quel serait l'inconvénient de ce choix ?

⟹ Écrire une fonction `get-key` :

$$INFORMATION\text{-}E \times SELECTOR \times JAZZ\text{-}MBD \rightarrow SYMBOL \cup \{\texttt{\#f}\}$$

qui réalise l'opération duale : l'évaluation de l'expression `(get-key` x_0 sel_0 *j-list*`)` recherche un élément j_0 de *j-list* tel que :

$$(sel_0\ j_0) \implies x_0$$

— si un tel élément j_0 existe, retourner la clé de j_0 (*cf.* figure 5) ;

— sinon la fonction `get-key` retourne `#f`.

Notes

— On utilisera la fonction `equal?` pour comparer les informations de *j-list* avec l'information x_0.

— Même s'il existe plusieurs éléments de *j-list* tels que l'application du sélecteur sel_0 retourne l'élément x_0, on s'arrêtera au premier élément trouvé.

5. En réalité, cette fonction `map` peut aussi s'employer avec un argument fonctionnel et *plusieurs* listes linéaires. Ce point sera abordé plus tard dans le cours.

Expliquer pourquoi les résultats retournés par la fonction `get-key` ne présentent pas d'ambiguïtés, contrairement à ce qui pourrait se produire pour ceux de la fonction `get-information-e`. Puis utiliser la fonction `get-key` pour obtenir la clé de *Kelo*, œuvre enregistrée par Miles Davis, puis de *Blue Moon*, parmi les enregistrements de Stan Getz.

$\Longrightarrow$ Étant donné :

— un prédicat [6] p_1? applicable à une donnée de type *JAZZ-MBD*, c'est-à-dire une fonction p_1? :

$$JAZZ\text{-}MBD \rightarrow LOGICAL\text{-}VALUE$$

— une fonction f_1, à un argument formel, applicable à une donnée de type :

$$JAZZ\text{-}MBD$$

écrire une fonction `those-that`, telle que l'évaluation de l'expression :

`(those-that` p_1? *j-list* f_1`)`

retourne la liste linéaire des éléments $f_1(j)$ tels que j soit un élément de la liste *j-list* qui vérifie le prédicat p_1?. Peu importe l'ordre dans lequel les éléments apparaissent dans la liste résultat. En déduire :

— la liste des noms des œuvres enregistrées par Miles Davis en 1972 ;

— la liste des clés des enregistrements de Stan Getz concernant les pièces ayant *plusieurs* auteurs ;

— la liste des clés des enregistrements de Miles Davis concernant les pièces ayant *un seul* auteur ;

— la liste des noms des pièces enregistrées par Miles Davis ou Stan Getz pour lesquelles l'éditeur est inconnu ;

— la liste des clés des œuvres enregistrées par Miles Davis dont l'éditeur est `EMI` ;

— la liste des noms des œuvres de Milea Davis qu'il a lui-même enregistrées ;

— la liste des clés des œuvres enregistrées par Stan Getz et reprises sur *plusieurs compact discs*.

$\Longrightarrow$ Écrire une fonction `top-years` :

$$JAZZ\text{-}MBD \rightarrow LIST\text{-}of[INTEGER]$$

telle que l'évaluation de l'expression :

`(top-years` *j-list*`)`

retourne une liste linéaire composée des années qui apparaissent le plus fréquemment dans les enregistrements de *j-list*. Si le résultat de la fonction `top-years` comporte *plusieurs* éléments, cela signifie que ces millésimes sont arrivés *ex æquo* en tête du *hit-parade* [7]. Si la liste *j-list* est vide, la fonction `top-years` retourne la liste vide. Exemples :

```
(top-years miles-davis-r) ⟹ (1972)
(top-years stan-getz-r)   ⟹ (1980)
```

Indication Vous pouvez passer par les étapes suivantes :

— d'abord, construire, en parcourant de proche en proche la liste *j-list*, une *liste d'associations a-list*, dans laquelle les éléments — associations — sont de la forme `(`y `.` n`)`, où y est une année qui apparaît dans les enregistrements et n est le nombre total de fois où cette année apparaît dans la liste *j-list* ;

— ensuite, déterminer, en parcourant cette fois la liste d'associations *a-list*, toutes les années qui sont en correspondance avec le nombre maximum d'occurrences, cette seconde phase peut être effectuée en *une seule passe* par une fonction récursive terminale utilisant *deux* arguments accumulateurs ;

vos gentils animateurs se feront un plaisir de vous guider pour des détails supplémentaires.

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

```
(define (mergesort l rel-2?)
  ;; La clôture lexicale permet aux fonctions locales d'accéder à la relation d'ordre rel-2?.
  ;;
  (define (merge-2-groups g0 g1)
    (cond ((null? g0) g1)
          ((null? g1) g0)
          (else (let ((first-0 (car g0))
                      (first-1 (car g1)))
                  (if (rel-2? first-0 first-1)
                      (cons first-0 (merge-2-groups (cdr g0) g1))
                      (cons first-1 (merge-2-groups g0 (cdr g1)))))))))
  ;;
  (define (merge-groups group-list)
    (if (or (null? group-list) (null? (cdr group-list)))
        group-list
        (cons (merge-2-groups (car group-list) (cadr group-list))
              (merge-groups (cddr group-list)))))
  ;;
  (define (make-groups l0)
    ;; l0 est une liste linéaire non vide.
    (let ((first (car l0))
          (rest (cdr l0)))
      (if (null? rest)
          (list (list first))
          (let ((next-groups (make-groups rest)))
            (if (rel-2? first (car rest))
                (cons (cons first (car next-groups)) (cdr next-groups))
                (cons (list first) next-groups))))))
  ;;
  (if (null? l)
      '()
      (let iter-merge-groups ((group-list (make-groups l)))
        (if (null? (cdr group-list))
            (car group-list)
            (iter-merge-groups (merge-groups group-list))))))

(define cp-list-example '((5 . 5) (6 . 0) (3 . 3) (6 . 1) (7 . 7) (1 . 1) (6 . 2)))
```

Figure 6 : Fonction de tri par fusion.

5 *Merge Sort*

Nous vous rappelons dans la figure 6 — et dans le fichier d'amorce for-lc-3.scm — la fonction de tri par fusion [8] telle qu'elle vous a été donnée en cours, c'est-à-dire que son deuxième argument est une relation d'ordre [9].

7. ... et dans ce cas, peu importe l'ordre dans lequel ces différentes années apparaissent dans la liste résultat.

8. D'autres méthodes sont le tri par sélection du minimum, le tri par insertion, le tri par segmentation, le tri par tas, le tri *Shell*, *etc.*

9. ... ou, à tout le moins, un *pré-ordre*. Rappelons qu'un **pré-ordre** est une relation réflexive et transitive, mais pas nécessairement antisymétrique. Un exemple simple de pré-ordre est la relation « divise » dans l'ensemble $\mathbb{Z}$ des entiers relatifs :

$$\forall p, q \in Z, p \mid q \wedge q \mid p \Rightarrow p = q \vee p = -q$$

Vous pouvez faire quelques essais pour en approfondir le fonctionnement.

$\Longrightarrow$ Utiliser la relation d'ordre suivante entre couples d'entiers naturels [10] pour trier la liste `cp-list-example`, donnée dans la figure 6 et dans le fichier for-lc-3.scm :

$$(x_1, y_1) \preceq (x_2, y_2) \overset{\text{déf}}{\Longleftrightarrow} x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 \leqslant y_2)$$

6 *Apotheosis*

6.1 *Adagio appassionato*

$\Longrightarrow$ Utilisez la fonction `mergesort` précédente pour trier les éléments de la liste linéaire `miles-davis-r` selon l'ordre croissant des années de ces enregistrements. Indépendamment de la méthode suivie pour trier, n'y a-t-il qu'une seule réponse à cette question ? ou en existe-t-il plusieurs ? et pourquoi ? Est-ce que votre version du tri par fusion est *stable* [11] ? Observez ce qui se produit en utilisant, pour comparer les années, un ordre *strict* [12] ($<$) et un ordre *large* [13] ($\leqslant$).

Indication La liste linéaire `miles-davis-r` est — comme vous l'avez vu dès la figure 3— assez longue et il est parfois difficile de s'y retrouver avec l'affichage « standard ». Si vous souhaitez un affichage ligne par ligne des éléments successifs d'une longue liste linéaire, vous pouvez utiliser la fonction :

10. L'ordre *lexicographique*, utilisé pour des chaînes de caractères, est en réalité une généralisation de cette relation.

11. Un tri est dit **stable** si l'ordre original d'apparition est maintenu pour les éléments considérés comme équivalents par la relation de pré-ordre.

12. Un ordre **strict** — ou relation binaire de **rangement** — sur un ensemble S est une relation binaire $\mathcal{R}$ *irréflexive*, antisymétrique et transitive, comme l'est la relation $<$ entre nombres. En outre, ne *pas* confondre une relation *irréflexive* — on dit aussi *antiréflexive* — :

$$\underline{\forall} x \in S, \neg(x \mathcal{R} x)$$

avec une relation binaire qui n'est *pas réflexive*, c'est-à-dire telle que $\underline{\exists} x \in S, \neg(x \mathcal{R} x)$.

13. C'est-à-dire une relation binaire réflexive, antisymétrique et transitive.

```
(define (pretty-writeln/return x)
  (if (and (pair? x) (list? x))
      (begin
        (write-char #\()
        (write (car x))
        (for-each (lambda (x0)
                    (newline)
                    (write-char #\space)
                    (write x0))
                  (cdr x))
        (write-char #\)))
      (write x))
  (newline)
  x)
```

Figure 7 : Affichage ligne par ligne.

`pretty-writeln/return`

donnée dans la figure 7 et — bien sûr — dans le fichier d'amorce for-lc-3.scm. Après l'affichage, cette fonction `pretty-writeln/return` passe à la ligne suivante et retourne son argument.

$\Longrightarrow$ Utilisez à nouveau la fonction `mergesort` pour trier les éléments de la liste linéaire `miles-davis-r`, mais cette fois d'abord selon l'ordre croissant des années des enregistrements, et selon les titres des pièces pour celles qui ont été enregistrées durant la même année.

Pour cette seconde phase de comparaisons, vous pouvez utiliser les fonctions prédéfinies `string-ci<?` ou `string-ci<=?`, qui comparent des chaînes de caractères suivant l'ordre lexicographique [14].

14. En outre, ces deux fonctions `string-ci<?` et `string-ci<=?` ne tiennent pas compte de la *casse*, c'est-à-dire de la distinction entre lettres majuscules et minuscules (« `-ci` » est mis pour « *case insensitive* »). Scheme fournit aussi une autre famille de fonctions de comparaison de chaînes de caractères, prenant cette différence en compte, par exemple, `string<?` et `string<=?`. Ces dernières fonctions appartiennent à la bibliothèque initiale `(scheme base)` de Scheme, alors que les fonctions précédentes, qui ignorent la casse, sont rangées dans la bibliothèque prédéfinie `(scheme char)`. Là encore, vous n'avez pas

$\Longrightarrow$ Nous allons à présent convenir que les symboles utilisés pour les noms des éditeurs-propriétaires sont ordonnés de la manière suivante :

$$\texttt{Blue-Ribbon-Music} \prec \texttt{Bosworth} \prec \texttt{Chappell} \prec \texttt{EMI} \prec \texttt{Kensington-Music} \prec \texttt{Sony} \prec \texttt{Universal} \prec \texttt{??} \tag{1}$$

et il vous est demandé d'écrire deux fonctions :

`publisher<=?`, `publisher<?` : $SYMBOL \times SYMBOL \rightarrow LOGICAL\text{-}VALUE$

réalisant la clôture antisymétrique et transitive de la relation précédente (1). Bien noter que la fonction `publisher<=?` (resp. `publisher<?`) est une relation d'ordre large (resp. strict). En outre, l'emploi de ces deux fonctions avec un symbole différent de ceux qui figurent dans (1) revient à remplacer ce symbole par la valeur « `??` ».

Indication Pour réaliser ces deux fonctions `publisher<=?` et `publisher<?`, une bonne idée est de construire une liste linéaire dont les éléments sont les symboles présents dans (1) — à l'exception du symbole « `??` » — et utiliser la fonction `position` vue dans le cours et rappelée dans la figure 8 ainsi que dans le fichier d'amorce for-lc-3.scm.

$\Longrightarrow$ Employez ces deux dernières définitions de fonctions pour trier les deux listes linéaires `miles-davis-r` et `stan-getz-r` selon les deux relations, stricte (`publisher<?`) et large (`publisher<=?`). Compte tenu des essais effectués précédemment avec les fonctions prédéfinies `string-ci<?` et `string-ci<=?`, vous devriez pouvoir deviner par vous-même laquelle de ces deux relations — `publisher<?` ou `publisher<=?` —, utilisée avec la fonction `mergesort`, permet d'effectuer des tris stables. Vérifiez que votre intuition est juste.

à vous préocupper de l'importation éventuelle de cette bibliothèque : elle est réalisée dans le fichier d'amorce for-lc-3.scm.

```
(define (position x l)
  (let thru ((l0 l)
             (current-position 0))
    (cond
     ((null? l0) #f)
     ((equal? (car l0) x)
      current-position)
     (else
      (thru (cdr l0)
            (+ current-position 1))))))
(define add-something
  (case-lambda
   ((x) (+ x 1))
   ((x y) (+ x y))))
```

Figure 8 : Les deux dernières définitions.

6.2 *Allegro furioso*

Une fonction de tri accomplissant un assez grand nombre d'appels à la relation d'ordre, cette dernière doit être aussi efficace que possible. De fait, beaucoup de tris utilisent un argument supplémentaire, la **clé** du tri, sur lequel porte la relation d'ordre. Donnez une nouvelle version du tri par fusion, `mergesort-plus`, sur le modèle suivant :

`(mergesort-plus` l $rel_2?$ k_1`)`

où l est la liste à trier, k_1 la clé à appliquer à chaque élément de l, $rel_2?$ la relation d'ordre entre clés qui est à utiliser. Adopter le *modus operandi* suivant, qui permet de ne pas recalculer la clé à chaque fois que deux éléments sont comparés :

— d'abord, nous construisons une nouvelle liste linéaire, formée des éléments d'origine associés aux clés correspondantes :

$((element_0 \ . \ key_0)$
$(element_1 \ . \ key_1)$
`...)`

— ensuite, nous appliquons l'algorithme de tri par fusion en considérant que la relation de pré-ordre est appliquée sur le « `cdr` » de chaque élément ;

— puis le résultat final se déduit de la liste triée précédente en prenant le « `car` » de chaque élément.

Indication Vous avez deux occasions supplémentaires d'utiliser la fonction prédéfine `map`, que nous avons déjà mentionnée au début du § 4.

$\Longrightarrow$ Utilisez cette nouvelle fonction de tri `mergesort-plus` pour trier plus efficacement les deux listes linéaires `miles-davis-r` et `stan-getz-r` selon les deux relations de pré-ordres fondées sur les noms des éditeurs-propriétaires et données précédemment.

6.3 *Andante serene*

Comme le rappelle le dernier exemple donné dans la figure 8, nous pouvons définir des fonctions dont quelques arguments sont optionnels — facultatifs — au moyen de la forme spéciale `case-lambda` [15].

$\Longrightarrow$ Retravaillez encore une fois cette fonction `mergesort-plus` et employer cette forme spéciale `case-lambda` pour en réaliser les utilisations suivantes :

— avec un seul argument : c'est la liste linéaire à trier, la relation de pré-ordre est par défaut `<=` et la clé est la fonction identité ;

— avec deux arguments : ce sont la liste à trier et la relation de pré-ordre, la clé par défaut est la fonction identité ;

— avec trois arguments, qui sont respectivement la liste à trier, la relation de pré-ordre et la clé.

Le mieux est d'utiliser des fonctions récursives locales au moyen de la forme spéciale `letrec`, et d'enrober le tout au moyen de la forme spéciale `case-lambda`. En ce qui concerne la fonction identité, vous pouvez la construire « à la main » ; vous pouvez aussi utiliser la fonction `values` avec un seul argument :

```
(values 'ok) ⟹ ok
```

Puis ré-essayez tous vos précédents exemples de tri en utilisant au mieux la dernière version de la fonction `mergesort-plus`.

15. Cette forme spéciale `case-lambda` appartient à la bibliothèque prédéfinie `(scheme case-lambda)`... importée lors du chargement de votre fichier d'amorce for-lc-3.scm.