Licence 3 d'Informatique — Programmation fonctionnelle — TP 2

Début : mardi 21 septembre 2021

Le but des exercices de cette deuxième séance de travaux pratiques en Scheme est d'achever de vous familiariser avec la programmation fonctionnelle au sens d'un calcul où les fonctions peuvent être des *arguments* et/ou des *résultats*. Ainsi, après un hors d'œuvre, le plat de résistance du menu ci-après peut être vu comme une fonction *génératrice de fonctions*. Le présent énoncé est accompagné d'un fichier source de démarrage for-lc-2.scm, reproduit *in extenso* à la figure 1 — les fonctions `first-mystery` et `second-mystery` seront utilisées respectivement aux §§ 3 & 4 — et disponible sous le cours :

Programmation fonctionnelle, scripts — XML

de moodle, dans le répertoire :

Travaux pratiques Scheme 2021–2022 > for-lc-2

Nous vous rappelons qu'il est possible d'écrire des fonctions récursives en Scheme — sur le modèle de la première version de la fonction factorielle donnée durant la séance précédente de travaux pratiques —; quant à l'avant-dernier exercice de cette seconde série vous montre un schéma récursif un peu plus compliqué, et le dernier est une introduction aux *couples* de Scheme.

1 *Preludio*

Dans tout ce § 1, nous appellerons **figure simple** une figure géométrique — à deux ou trois dimensions — qu'on peut définir par une *seule* information de longueur. Par exemple, un cercle est une figure simple (à deux dimensions), de même qu'une sphère (à trois dimensions), car on peut les définir uniquement à l'aide de la longueur du rayon. De même un carré, qu'on peut définir à l'aide de la longueur du côté. Par contre, un rectangle n'est *pas* une figure simple (deux informations de longueur sont nécessaires, sur les grand et petit côtés), une ellipse non plus (deux informations sont à fournir : les distances du centre à l'apogée et au périgée).

$\Longrightarrow$ Définir une fonction `simple-surface`, dont un appel quelconque est :

`(simple-surface` r f_1`)`

— où r est une longueur ($r \in \mathbb{R}$) et f_1 une fonction utilisable avec un argument formel —; cette fonction `simple-surface` est telle que l'évaluation des deux expressions suivantes :

```
(simple-surface r
                (lambda (p) (* pi p)))
    ⟹ ...
(simple-surface r
                (lambda (p)
                  (* 4 pi p))) ⟹ ...
```

retournent respectivement la surface d'un cercle et d'une sphère de rayon r ($r \in \mathbb{R}$) — rappelons que les formules de calcul respectives sont πr^2 et $4\pi r^2$; quant à la valeur de π, elle est donnée par la variable `pi`, définie dans le fichier for-lc-2.scm et la figure 1 selon la formule déjà utilisée durant la séance précédente. Dire également ce que

```
#!r7rs

(import (scheme base) (scheme inexact) (scheme write))

(define (writeln/return x)
  ;; Fonction très utile pour pouvoir afficher un résultat suivi d'un passage à la ligne, tout en pouvant
  ;; récupérer ce résultat dans une variable. Les trois expressions suivantes sont évaluées en séquence
  ;; et la valeur de la dernière est retournée.
  (write x)
  (newline)
  x)

(define pi (* 4 (atan 1)))

(define (first-mystery n)
  ;; Rappelons que les fonctions quotient et remainder retournent respectivement le quotient entier
  ;; et le reste d'une division euclidienne, à condition, bien sûr, que le diviseur ne soit pas nul.
  (if (< n 10) n (+ (first-mystery (quotient n 10)) (remainder n 10))))

(define (second-mystery n p)
  ;;
  (define (rec-mystery n0 p0)
    ;; Rappelons également qu'une forme spéciale define interne à une forme spéciale lambda est
    ;; équivalente à l'utilisation d'une forme spéciale letrec*.
    (if (zero? (remainder n0 p0)) (rec-mystery (quotient n0 p0) p0) n0))
  ;;
  ;; Principal call:
  (if (or (< p 2) (zero? n)) n (rec-mystery n p)))
```

Figure 1 : Le fichier for-lc-2.scm.

---

signifie le second argument de la fonction `simple-surface`.

$\Longrightarrow$ Comment pouvez-vous utiliser cette fonction `simple-surface` pour calculer la surface d'un carré ? et pour calculer la surface d'un cube ?

2 Conversion d'unités de mesure

2.1 Une réalisation *ad hoc* d'abord

$\Longrightarrow$ Donner une fonction `in-inches` telle que l'évaluation de l'expression :

(`in-inches` $miles_0$ $yards_0$ $feet_0$ $inches_0$)

retourne la valeur, uniquement donnée en *inches*, d'une longueur mesurée par $miles_0$ *miles*, $yards_0$ *yards*, $feet_0$ *feet*, $inches_0$ *inches*, d'après le système de mesure britannique — $miles_0, yards_0, feet_0, inches_0$ étant bien évidemment des nombres positifs. Revenant à notre *modus operandi*, nous ramenons la totalité de la longueur mesurée à une mesure n'utilisant que l'unité de mesure la plus petite. Voici quelles sont les correspondances :

$$\begin{aligned} 1\,&inch \\ 12\,&inches = 1\,foot \\ 3\,&feet = 1\,yard \\ 1760\,&yards = 1\,mile \end{aligned}$$

$\Longrightarrow$ Ensuite, écrivez une fonction :

`inches-to-meters`

qui, étant donné une longueur mesurée en *inches* (pouces), retourne sa valeur en mètres. L'équivalence est :

$$1\,\mathit{inch} = 25.4\,\mathrm{mm}$$

Puis utilisez les deux fonctions précédentes, `in-inches` et `inches-to-meters`, pour donner une fonction `british-to-metric`, telle que l'évaluation de l'expression :

`(british-to-metric` miles_0 yards_0 feet_0 inches_0`)`

retourne la valeur, en mètres, d'une longueur mesurée par miles_0 *miles*, yards_0 *yards*, feet_0 *feet*, inches_0 *inches*.

$\Longrightarrow$ Quel est le résultat de l'évaluation suivante ?

`(british-to-metric 3 61 2 10)` $\Longrightarrow$ `??`

2.2 Approche de la notion

Nous pourrions réaliser des fonctions semblables pour, étant donné des mesures données dans des unités non organisées selon le système décimal, les convertir en une mesure de type décimal. Nous utiliserions alors la table suivante pour les mesures de capacité :

$$\begin{aligned} 1\,\mathit{gill} & \\ 4\,\mathit{gills} &= 1\,\mathit{pint} \\ 2\,\mathit{pints} &= 1\,\mathit{quart} \\ 4\,\mathit{quarts} &= 1\,\mathit{gallon} \end{aligned}$$

$$1\,\mathit{gill} = 0.142\,\mathrm{l}$$

et, pour les mesures de masse, la table que voici :

$$\begin{aligned} 1\,\mathit{grain} & \\ 437.5\,\mathit{grains} &= 1\,\mathit{ounce} \\ 16\,\mathit{ounces} &= 1\,\mathit{pound} \\ 14\,\mathit{pounds} &= 1\,\mathit{stone} \end{aligned}$$

$$1\,\mathit{grain} = 0.065\,\mathrm{g}$$

Dans le cas d'une mesure de capacité (resp. de masse), le premier pas consiste à l'expliciter uniquement en *gills* (resp. *grains*), c'est-à-dire l'unité de mesure la plus fine, le second pas est la conversion proprement dite.

Le but de la suite de l'exercice est de définir des « super-fonctions » — des générateurs de fonctions de conversion, en fait — qui, étant donné de telles tables, nous permettent d'engendrer les fonctions de conversion correspondantes, et ceci sans avoir à réécrire à chaque fois des fonctions *ad hoc*. Par simplification, nous nous limiterons à quatre niveaux de mesures non décimales, comme c'est le cas dans les trois exemples — les mesures de longueurs, de capacité et de masse — présentés ci-dessus.

2.3 Fonction retournant une fonction

Nous allons donc considérer quatre unités de mesure $\mathit{unit}_1, \mathit{unit}_2, \mathit{unit}_3, \mathit{unit}_4$ s'appliquant à la même grandeur, et organisées comme suit :

$$\begin{aligned} 1\,\mathit{unit}_1 &= \mathit{nb}_1\,\mathit{unit}_2 \\ 1\,\mathit{unit}_2 &= \mathit{nb}_2\,\mathit{unit}_3 \\ 1\,\mathit{unit}_3 &= \mathit{nb}_3\,\mathit{unit}_4 \end{aligned}$$

— unit_1 est une mesure moins fine que unit_2, qui est une mesure moins fine que unit_3 et ainsi de suite — et l'unité *metric* est telle que :

$$1\,\mathit{unit}_4 = \mathit{metric}_0\,\mathit{metric}$$

où $\mathit{nb}_1, \mathit{nb}_2, \mathit{nb}_3, \mathit{metric}_0$ sont bien sûr des nombres réels.

$\Longrightarrow$ Donnez une fonction `to-metric` telle que l'évaluation de l'expression :

`(to-metric` nb_1 nb_2 nb_3 metric_0`)`

retourne une fonction qui :

$\Leftarrow$ appliquée à quatre arguments représentant respectivement des mesures dans les unités $\mathit{unit}_1, \mathit{unit}_2, \mathit{unit}_3, \mathit{unit}_4$,

$\Rightarrow$ retourne la même mesure globale, exprimée dans l'unité *metric*.

$\Longrightarrow$ Utilisez la fonction `to-metric` pour définir une nouvelle version de la fonction de conversion des longueurs :

`british-to-metric-v2`

et vérifiez que l'évaluation de l'expression :

`(british-to-metric-v2 3 61 2 10)`

retourne le résultat que vous aviez trouvé précédemment. Puis donnez la valeur en litres de la capacité suivante :

3 *gallons*, 3 *quarts*, 3 *pints*, 3 *gills*

et la valeur en grammes de la masse suivante :

2 *stones*, 10 *pounds*, 11 *ounces*, 320 *grains*

De même, donnez le montant en euros du prix suivant :

12 *pounds*, 3 *florins*, 1 *shilling*, 6 *pence*

en prenant £ 1 = 1.1715641 €, et en utilisant la table ci-après [1] :

$$\begin{aligned} & 1\ \textit{penny} \\ 12\ \textit{pence} &= 1\ \textit{shilling} \\ 2\ \textit{shillings} &= 1\ \textit{florin} \\ 10\ \textit{florins} &= 1\ \textit{pound} \end{aligned}$$

Comment utiliser la fonction `to-metric` pour donner la valeur en secondes de la durée suivante ?

1 jour 3 h 47 mn 48 s

3 *Scherzo nervoso*

$\Longrightarrow$ Examiner la fonction `first-mystery`, donnée dans le fichier for-lc-2.scm et dans la figure 1 : que calcule cette fonction ? Pour vous mettre sur la piste, vous pouvez essayer les évaluations suivantes :

```
(first-mystery 2021) ⟹ ??
(first-mystery 2009) ⟹ ??
```

$\Longrightarrow$ Utiliser cette fonction `first-mystery` pour l'écriture d'une fonction `one-digit`, telle que l'évaluation de l'expression :

`(one-digit` n`)`

— où n est un entier naturel — itère le calcul de la somme des chiffres du développement de n en base 10, jusqu'à ce que le résultat ne contienne plus qu'un seul chiffre, autrement dit, jusqu'à ce que le résultat appartienne à l'intervalle $[0, 10[$. Exemples :

```
(one-digit 2021) ⟹ 5
(one-digit 2009) ⟹ 2
```

4 *Andante doloroso*

$\Longrightarrow$ Considérer la fonction `second-mystery` du fichier for-lc-2.scm (*cf.* figure 1) et découvrir ce qu'elle calcule. Vous pouvez le cas échéant vous aider du Debugger de DrRacket pour en comprendre le fonctionnement pas à pas ou en demander une démonstration à votre gentil animateur.

$\Longrightarrow$ Y a-t-il moyen de simplifier l'écriture de la fonction `second-mystery` — plus exactement de la fonction locale `rec-mystery` — en utilisant le mécanisme de la clôture lexicale ?

Indication Observez bien toutes les occurrences du *dernier* argument de la fonction locale `rec-mystery`.

Les **nombres de Hamming** sont tous les nombres entiers naturels de la forme :

$$2^a \,.\, 3^b \,.\, 5^c \text{ avec } a, b, c \in \mathbb{N}$$

c'est-à-dire tous les nombres entiers naturels n'admettant pas de facteurs premiers autres que 2, 3 et 5. À titre d'exemples, 1 et 6 sont des nombres de Hamming, tandis que 14 ne l'est pas.

$\Longrightarrow$ Utiliser la fonction `second-mystery` précédente dans la définition d'une fonction de test :

`hamming-number?`

telle que l'évaluation de l'expression :

`(hamming-number?` n`)`

— avec n entier — retourne `#t` si n est un nombre de Hamming, `#f` sinon.

```
(hamming-number? 1)  ⟹ #t
(hamming-number? 6)  ⟹ #t
(hamming-number? 14) ⟹ #f
```

1. Il s'agit ici de l'ancien système monétaire britannique, en vigueur jusqu'en février 1971, date à laquelle la livre sterling (*pound sterling*) a été divisée en 100 *new pence*, tandis que le *shilling* disparaissait. Le *florin*, quant à lui, avait déjà disparu dès 1967. Quant au cours que nous avons considéré, c'est celui du mois de septembre 2021.

$$
\begin{aligned}
\texttt{alternate-digit-sum}(0, p?) &= 0 \\
\texttt{alternate-digit-sum}(\overline{d_n \cdots d_1 d_0}, p?) &= \\
&\begin{cases} \texttt{alternate-digit-sum}(\overline{d_n \cdots d_1}, \mathit{false}) + d_0 & \text{si } p? \text{ est vrai,} \\ \texttt{alternate-digit-sum}(\overline{d_n \cdots d_1}, \mathit{true}) - d_0 & \text{sinon.} \end{cases}
\end{aligned}
$$

$p?$ étant, bien sûr, une valeur logique. Exemples :

```
(alternate-digit-sum 154 #t)  ⟹ 0
(alternate-digit-sum 1749 #t) ⟹ 11
(alternate-digit-sum 1751 #t) ⟹ 2
```

Figure 2 : Spécification de la fonction `alternate-digit-sum`.

5 Divisibilité

On démontre qu'un nombre entier est divisible par 11 si et seulement si la somme alternée des chiffres de sa valeur absolue, itérée jusqu'à obtenir un nombre à un seul chiffre, est égale à zéro. Exemples :

$$
\begin{aligned}
154 &\longrightarrow 4 - 5 + 1 = 0 \qquad \textit{OK!} \\
-1749 &\longrightarrow 1749 \\
&\longrightarrow 9 - 4 + 7 - 1 = 11 \\
&\longrightarrow 1 - 1 = 0 \qquad \textit{OK!} \\
1751 &\longrightarrow 1 - 5 + 7 - 1 = 2 \qquad \text{non.}
\end{aligned}
$$

$\Longrightarrow$ Étant donné un nombre entier naturel dont l'écriture décimale est $\overline{d_n \cdots d_1 d_0}$, donner la fonction `alternate-digit-sum` décrite dans la figure 2.

Puis utiliser la fonction précédente pour l'écriture d'une fonction :

`divisible-by-11?`

telle que l'évaluation de l'expression :

`(divisible-by-11?` *i*`)`

— *i* étant un entier relatif — retourne `#t` si *i* est divisible par 11, `#f` sinon. Exemples :

```
(divisible-by-11? 154)   ⟹ #t
(divisible-by-11? -1749) ⟹ #t
(divisible-by-11? 1751)  ⟹ #f
```

6 *Postludio*

Nous rappelons l'existence de la fonction `cons` de Scheme pour assembler des *couples* [2] :

```
(cons 21 9) ⟹ (21 . 9)
```

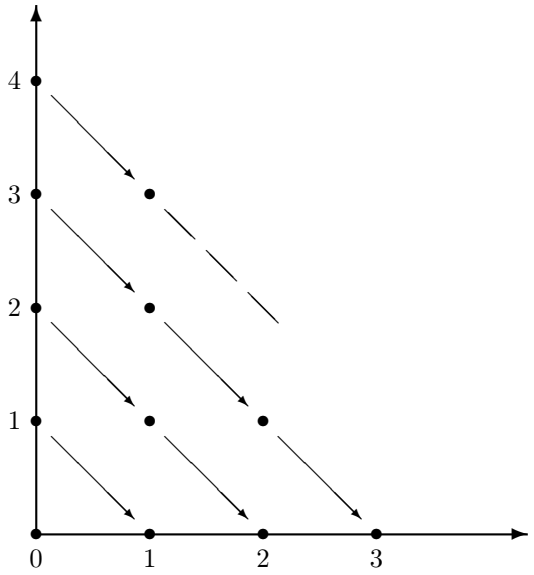
On démontre qu'il est possible de définir une bijection entre l'ensemble $\mathbb{N}$ des entiers naturels et l'ensemble $\mathbb{N} \times \mathbb{N}$ des couples d'entiers naturels. Autrement dit, nous pouvons *compter* les couples d'entiers naturels [3]. La fonction de comptage la plus utilisée consiste à se positionner sur les éléments successifs de l'axe des ordonnées, puis à considérer

2. *Paires*, selon la terminologie utilisée en Scheme. Mais il s'agit plutôt de couples, et non pas de paires au sens ensembliste, car l'ordre est important.

3. ... et par transitivité, on finit par montrer qu'on peut aussi compter les nombres rationnels de l'ensemble $\mathbb{Q} \subsetneqq \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$. Une bijection de $\mathbb{N}$ vers l'ensemble $\mathbb{Z}$ des entiers relatifs, qui utilise une technique de pliage, est :

$$
\begin{aligned}
\mathbb{N} &\to \mathbb{Z} \\
i &\mapsto \begin{cases} \dfrac{i}{2} & \text{si } i \text{ est pair,} \\ -\dfrac{i+1}{2} & \text{si } i \text{ est impair.} \end{cases}
\end{aligned}
$$

Par contre, cette propriété d'équipotence à $\mathbb{N}$ est fausse pour l'ensemble $\mathbb{R}$ des nombres réels. Ce dernier ensemble est en réalité équipotent à $\mathcal{P}(\mathbb{N})$, l'ensemble des parties de $\mathbb{N}$. En Théorie des ensembles, un nombre réel est défini par l'ensemble des nombres rationnels qui le minorent.



$$\begin{array}{lcll} & & & \texttt{diagonal}(0) = (0,0) \\ n > 0 \wedge \texttt{diagonal}(n-1) = (p,q) & \wedge \quad q = 0 & \Rightarrow & \texttt{diagonal}(n) = (0, p+1) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \wedge \quad q \neq 0 & \Rightarrow & \texttt{diagonal}(n) = (p+1, q-1) \end{array}$$

$n, p, q \in \mathbb{N}$.

Figure 3 : Comptage des couples d'entiers naturels.

---

la direction sud-est jusqu'à rencontre de l'axe des abscisses. Ce qui vous est montré dans la figure 3, et nous donne, dans l'ordre, la succession suivante de couples :

— d'abord le couple $(0, 0)$;

— puis les couples $(0, 1)$ et $(1, 0)$;

— puis $(0, 2), (1, 1), (2, 0)$;

— puis $(0, 3), (1, 2), (2, 1), (3, 0), (0, 4), \ldots$

Donner une fonction `diagonal`, telle que l'évaluation de l'expression `(diagonal` *i*`)` — où *i* est un entier naturel — retourne le couple n$^{\text{o}}$ *i* d'après la méthode que nous venons de décrire, le comptage commençant à zéro. La formule de récurrence vous est donnée dans la figure 3.