Master 1 à distance — PFA — Devoir n° 2

À rendre au plus tard vendredi 16 décembre 2016

0 Notes préliminaires

Nous rappelons qu'on peut implanter en Scheme un *stream* par une paire dont le premier élément est la tête du *stream*, et le second une valeur promise permettant de calculer par évaluation retardée les éléments suivants. Il est d'usage de manipuler les *streams* au travers des fonctions rappelées ci-après :

```
(define head car)
(define (tail s)
  (force (cdr s)))
```

Nous rappelons aussi que l'évaluation de l'une des valeurs promises peut retourner à nouveau un *stream*, mais aussi la liste vide auquel cas le *stream* est fini : il est d'usage de représenter cette situation et le test qui y est associé par :

```
(define end-of-stream '())
(define end-of-stream? null?)
```

Toutes ces définitions vues dans le cours vous sont données dans le fichier for-hw-2.scm. Nous vous rappelons qu'il vous suffit d'insérer les définitions de ce fichier comme suit dans votre devoir :

```
#lang scheme

(require "for-hw-2.scm")

...
```

Ce fichier for-hw-2.scm contient également une fonction `see-firsts`, vue dans les exercices du cours, et qui permet de tester les premiers éléments d'un *stream* fini ou infini. Enfin, terminons ces rappels en mentionnant que par analogie avec les éléments d'une liste linéaire, les éléments successifs d'un *stream* sont numérotés à partir de zéro.

Les deux exercices §§ 2 & 3 sont indépendants.

1 *Intrada*

$\Longrightarrow$ Soit *year* un entier relatif, donner une fonction `leap?`, telle que l'évaluation de l'expression `(leap?` *year*`)` retourne `#t` si l'année *year* est bissextile, `#f` sinon. Nous rappelons qu'une année est bissextile :

— si elle est divisible par 4 et pas par 100 ;

— ou si elle est divisible par 400.

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

2 Première série

2.1 Programmation de *streams*

$\Longrightarrow$ Définir la variable `beat-5` dont la valeur est le *stream* infini composé des éléments suivants [1] :

`#t  #f  #t  #f  #f  #t  #f  #t  #f  #f  #t  #f  ...`

(observez la périodicité de 5 en 5).

$\Longrightarrow$ Soit x une donnée quelconque, nous nous proposons d'écrire une fonction :

`make-constant-stream`

telle que l'évaluation de l'expression `(make-constant-stream` x`)` retourne un *stream* infini dont tous les éléments sont égaux à x. Deux réalisations différentes de cette fonction vous sont proposées dans le fichier for-hw-2.scm — `make-constant-stream-v0` et `make-constant-stream-v1` — :

```
(define (make-constant-stream-v0 x)
  (cons x (delay (make-constant-stream-v0 x))))

(define (make-constant-stream-v1 x)
  (letrec ((this (cons x (delay this))))
    this))
```

L'une d'entre elles est cependant bien meilleure que l'autre. De laquelle s'agit-il ? Et pourquoi est-ce le cas ?

$\Longrightarrow$ Définir la variable `switch` dont la valeur est le *stream* composé des éléments suivants :

`#t  #f  #t  #f  #t  #f  ...`

$\Longrightarrow$ Étant donné un entier naturel n, donner la fonction `stay-at-false`, telle que l'évaluation de l'expression `(stay-at-false` n`)` retourne un *stream* dont les éléments sont composés comme suit :

`#t` $\underbrace{\texttt{\#f} \ \ldots \ \texttt{\#f}}_{n \text{ éléments}}$ `#t` $\underbrace{\texttt{\#f} \ \ldots \ \texttt{\#f}}_{n \text{ éléments}}$ `...`

Généralisons à présent la définition précédente.

$\Longrightarrow$ Soient a et b deux données quelconques, et soient n_a et n_b deux entiers naturels, définir la fonction `two-elements`, telle que l'évaluation de l'expression `(two-elements` a b n_a n_b`)` retourne le *stream* suivant :

$\underbrace{a \ldots a}_{n_a \text{ éléments}}$ $\underbrace{b \ldots b}_{n_b \text{ éléments}}$ $\underbrace{a \ldots a}_{n_a \text{ éléments}}$ $\underbrace{b \ldots b}_{n_b \text{ éléments}}$ $\ldots$

1. En anglais « musical », « *beat 5* » signifie « battre une mesure à cinq temps ».

2.2 D'autres opérations sur les *streams*

$\Longrightarrow$ Soient n un entier naturel et s un *stream*, donner la fonction `truncate-stream`, telle que l'évaluation de l'expression `(truncate-stream` s n`)` retourne un *stream* fini composé des n premiers éléments de s. Si s est fini et possède m éléments avec $m < n$, le résultat est un *stream* fini composé uniquement des m éléments de s.

$\Longrightarrow$ Soit s un *stream*, définir la fonction `loop-on-stream` telle que l'évaluation de l'expression :

`(loop-on-stream` s`)`

retourne un *stream* infini :

— composé des éléments de s si s est infini,

— composé des éléments n$^{\text{os}}$ $0, 1, \ldots, m-1, 0, 1, \ldots, m-1, 0, \ldots$ de s si s est fini avec m éléments.

3 Seconde série

$\Longrightarrow$ Écrire une fonction `list->infinite-stream`, telle que l'évaluation de l'expression :

`(list->infinite-stream` l`)`

— où l est une liste linéaire — retourne un *stream* infini composé des éléments successifs de l, bouclant indéfiniment. Si la liste l est vide, le résultat est la fin de *stream*. Par exemple, l'évaluation de l'expression `(list->infinite-stream '(#t #f #t #f #f))` retourne un *stream* identique à la valeur de la variable `beat-5` du § 2.

$\Longrightarrow$ Donner une fonction `mk-day-name-stream-from`, telle que l'évaluation de l'expression :

`(mk-day-name-stream-from` *day-name*`)`

— où *day-name* est un symbole représentant le nom d'un jour de la semaine — retourne un *stream* infini dont l'élément de tête est *day-name* et les éléments suivants les noms des jours qui suivent, indéfiniment. Exemple :

```
(define day-name-stream-from-tuesday (mk-day-name-stream-from 'Tuesday))
(head day-name-stream-from-tuesday)          ⟹ Tuesday
(head (tail day-name-stream-from-tuesday))   ⟹ Wednesday
...
```

Indication On pourra utiliser la fonction `list->infinite-stream` précédente.

$\Longrightarrow$ Définir une fonction `parallel-advance`, telle que l'évaluation de l'expression

`(parallel-advance` s_0 s_1`)`

— où s_0 et s_1 sont deux *streams* quelconques — retourne un *stream* dont l'élément de tête est une paire constituée des deux éléments de tête de s_0 et s_1, puis une paire constituée des deux éléments suivants de s_0 et s_1, et ainsi de suite jusqu'à ce que l'un des deux *streams* touche à sa fin. Le résultat est un *stream* infini si s_0 et s_1 sont tous deux des *streams* infinis.

4 Troisième série : une variante du calendrier

$\Longrightarrow$ Soit n un entier relatif, définir la fonction `iota-stream`, telle que l'évaluation de l'expression `(iota-stream` n`)` retourne le *stream* fini composé des nombres de `1` à n. Si $n < 1$, le *stream* résultat est vide.

$\Longrightarrow$ Soient s un *stream* — fini ou infini —, et f_1 une fonction utilisable avec un argument. Donner la fonction `map-on-stream`, telle que l'évaluation de l'expression :

`(map-on-stream` f_1 s`)`

retourne le *stream* dont les éléments successifs sont les résultats des applications de f_1 aux éléments successifs de s.

$\Longrightarrow$ Utiliser les fonctions `iota-stream` et `map-on-stream` précédentes pour définir un *stream* `december-stream`, dont les éléments successifs sont les listes linéaires suivantes :

```
(1 December) (2 December) ... (31 December)
```

$\Longrightarrow$ Soient s_0 et s_1 deux *streams*, définir la fonction `stream-append`, telle que l'évaluation de l'expression `(stream-append` s_0 s_1`)` retourne un *stream* formé des éléments du *stream* s_0, suivis de ceux du *stream* s_1 si s_0 est fini.

$\Longrightarrow$ Soit *stream-list* une liste linéaire de *streams*, donner une fonction `iterate-stream-append`, telle que l'évaluation de l'expression `(iterate-stream-append` *stream-list*`)` réalise l'itération de l'opération précédente de concaténation de *streams*. Autrement dit, elle retourne la fin de *stream* si *stream-list* est vide, et sinon, le *stream* composé :

- des éléments du premier *stream* de *stream-list*,
- éventuellement suivis de ceux du second si ce second *stream* existe et que le premier *stream* est fini,
- éventuellement suivis de ceux du troisième *stream* si ce troisième *stream* existe et que les deux premiers *streams* sont finis,
- ...

$\Longrightarrow$ Utiliser les fonctions précédentes pour réaliser la fonction `a-calendar`, telle que l'évaluation de l'expression `(a-calendar` *year*`)` — où *year* est un entier relatif — retourne un *stream* fini dont les éléments successifs, formant un calendrier pour l'année *year*, sont de la forme :

```
(1 January) (2 January) ... (31 December)
```

en rappelant que les mois de janvier, mars, avril, mai, juin, juillet, août, septembre, octobre, novembre, décembre ont respectivement 31, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 jours. Quant au mois de février, il a 29 ou 28 jours, selon que l'année est bissextile ou non.

Indication On pourra aussi utiliser la fonction `leap?` du § 1.

$\Longrightarrow$ Donner à présent une fonction `a-complete-calendar`, telle que l'évaluation de l'expression :

`(a-complete-calendar` *year day-name*`)`

— où *year* est un entier relatif et *day-name* un symbole représentant le nom d'un jour de la semaine — retourne un *stream* fini dont les éléments successifs forment un calendrier pour l'année *year*, avec la mention des jours successifs de la semaine, le jour de départ étant *day-name* :

```
(Saturday 1 January) (Sunday 2 January) ... (Sunday 31 December)        (1)
```

Vous pouvez vérifier que l'évaluation de l'expression `(a-complete-calendar 2016 'Saturday)` retourne bien le calendrier pour l'année 2016, c'est-à-dire les éléments successifs donnés en (1).

Indication On pourra utiliser les fonctions `mk-day-name-stream-from` et `parallel-advance` du § 3, ainsi que la fonction `calendar` précédente.