

## 11 Les streams et les séquences infinies

### 11.1 Exercices simples

Pour tous les exercices dans cette section, utilisez une structure de données abstraite pour les streams. L'interface de cette structure est fournie ci-dessous:

```
(define head car)

(define (tail stream)
  (force (cdr stream)))

(define-macro cons-stream
  ; delay evaluation until needed
  (lambda (a b)
    `(cons ,a (delay ,b))))
```

**Remarque.** Utilisez le langage Pretty Big (Assez Gros Scheme).

**Exercice 11.1** Implémentez les fonctions *(nth-stream n s)* et *(nth-value n s)* pour calculer le *n*-ième stream et la *n*-ième valeur dans le stream *s*. Par exemple,

```
>(define (allIntegersFrom n)
  (cons-stream n (allIntegersFrom (+ 1 n))))
> (nth-stream 5 (allIntegersFrom 10))
(15 . #<struct:promise>)
> (nth-value 5 (allIntegersFrom 10))
15
```

**Solution 11.1** Voici les solutions:

```
(define (nth-stream n s)
  (if (= n 0)
      s
      (nth-stream (- n 1) (tail s))))

(define (nth-value n s)
  (head (nth-stream n s)))
```

**Exercice 11.2** Implémentez la fonction *(print-stream s)* pour afficher toutes les valeurs d'un stream fini. Implémentez la fonction *(print-stream-to s n)* pour afficher les *n* premières valeurs d'un stream (fini ou infini). Par exemple

```
> (print-stream-to 20 (allIntegersFrom 10))
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29]
```

**Solution 11.2** Voici la solution:

```

(define (print-stream stream)
  (display "["
    (for-each (lambda (x) (display x) (display " ")) stream)
    (display "]"
      (newline))

(define (print-stream-to n stream)
  (define (print-iter n s)
    (if (> n 0)
      (begin
        (display (head s))
        (display " ")
        (print-iter (- n 1) (tail s))))
    (begin
      (display "["
        (print-iter n stream)
        (display "]"
          (newline)))

```

**Exercice 11.3** Implémentez les fonctions *accumulate*, *map*, *filter* et *for-each* pour les streams.

Conseil. La définition de ces fonctions pour les streams est presque la même que pour les listes.

**Solution 11.3** Voici les solutions:

```

(define (accumulate combiner nullvalue stream)
  (if (null? stream)
      nullvalue
      (combiner (head stream) (accumulate combiner nullvalue (tail stream)))))

(define (map-stream op stream)
  (if (null? stream)
      ()
      (cons-stream (op (head stream)) (map-stream op (tail stream)))))

(define (filter-stream predicate stream)
  (cond
    ((null? stream) ())
    ((predicate (head stream))
     (cons-stream (head stream) (filter-stream predicate (tail stream))))
    (else
     (filter-stream predicate (tail stream)))))

(define (for-each op stream)
  (if (not (null? stream))
      (begin

```

```
(op (head stream))
(for-each op (tail stream))))))
```

**Exercice 11.4** Implémentez un stream infini qui représente une séquence d'approximations de  $e = 2.71828182845904590\dots$  en utilisant la formule de Taylor:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots$$

Par exemple,

```
> (print-stream-to 5 estream)
[1.0 2.0 2.5 2.6666666666666665 2.7083333333333335]
```

**Solution 11.4** Voici une solution très inefficace. (Pourquoi?):

```
(define (calc-e n)
  (accumulate + 1
    (map-stream (lambda (x) (/ 1 (fac x))) (interval 1 n))))

(define estream
  (letrec
    ((iter (lambda (index)
      (cons-stream (calc-e index) (iter (+ 1 index))))))
    (iter 0)))
```

Voici une solution qui est beaucoup plus efficace! (Pourquoi?)

```
(define estream2
  (letrec
    ((iter (lambda (n factor term)
      (cons-stream factor
        (iter (+ 1 n)
          (+ factor (/ term n))
          (/ term n))))))
    (iter 1 1 1)))
```

**Exercice 11.5** Utilisez *filter* pour obtenir un stream infini qui contient tous les nombres entiers qui ne sont pas divisibles par 2, 3 ou 5.

**Solution 11.5** Voici la solution:

```
> (define (div? n) (lambda (x) (= (modulo x n) 0)))
> (define (predicat x) (not (or ((div? 3) x) ((div? 5) x) ((div? 2) x))))
> (print-stream-to 20
  (filter-stream predicat allIntegers))
[1 7 11 13 17 19 23 29 31 37 41 43 47 49 53 59 61 67 71 73]
```

Et voici une autre solution en utilisant une composition de 3 filtres:

```
> (define (notdiv? n) (lambda (x) (not (= (modulo x n) 0))))
> (print-stream-to 20
```

```
(filter-stream (notdiv? 5)
  (filter-stream (notdiv? 3)
    (filter-stream (notdiv? 2)
      allIntegers))))
```

## 11.2 Combining streams

Pour tous les exercices dans cette section, vous pouvez utiliser les fonctions suivantes:

```
(define (combine-streams combiner s1 s2)
  (cons-stream (combiner (head s1) (head s2))
    (combine-streams combiner (tail s1) (tail s2))))
```

```
(define (add-streams s1 s2)
  (combine-streams + s1 s2))
```

```
(define (multiply-streams s1 s2)
  (combine-streams * s1 s2))
```

```
(define (divide-streams s1 s2)
  (combine-streams / s1 s2))
```

**Exercice 11.6** *Prédisez le résultat de l'affichage pour les expressions suivantes:*

1. (define ones (cons-stream 1 ones))  
 (print-stream-to 10 ones)
2. (print-stream-to 10  
 (add-streams (tail allIntegers) allIntegers))
3. (define (addrec stream)  
 (cons-stream (head stream)  
 (addrec (add-streams (tail stream) stream))))  
 (print-stream-to 10 (addrec ones))
4. (define (addrec2 stream)  
 (cons-stream (head stream)  
 (addrec2 (add-streams stream stream))))  
 (print-stream-to 10 (addrec2 allIntegers))

**Solution 11.6** *Voici les solutions:*

1. [1 1 1 1 1 1 1 1 1 1 ]
2. [3 5 7 9 11 13 15 17 19 21 ]
3. [1 2 4 8 16 32 64 128 256 512 ]
4. [1 2 4 8 16 32 64 128 256 512 ]

**Exercice 11.7** Utilisez la fonction `add-streams` pour la création d'une séquence infinie de nombres entiers, où chaque nombre à la position  $n$  doit être égal à la somme des  $n$  premiers nombres entiers.

À titre d'exemple, voici les 10 premières valeurs de cette séquence infinie:  
[0 1 3 6 10 15 21 28 36 45 ...]

**Solution 11.7** Voici la solution:

```
(define (addrec stream)
  (cons-stream (- (head stream) 1)
               (add-streams stream (addrec stream))))

(print-stream-to 10 (addrec allIntegers))
[0 1 3 6 10 15 21 28 36 45 ]
```

**Exercice 11.8** Utilisez la fonction `multiply-streams` pour la création d'une séquence infinie de nombres entiers, où chaque nombre à la position  $n$  doit être égal au factoriel de  $n$ .

À titre d'exemple, voici les 10 premières valeurs de cette séquence infinie:  
[1 1 2 6 24 120 720 5040 40320 362880 ...]

**Solution 11.8** Voici la solution:

```
(define (facrec stream)
  (cons-stream (head stream)
               (multiply-streams stream (facrec stream))))

(define facstream (facrec allIntegers))
(print-stream-to 10 facstream)
[1 1 2 6 24 120 720 5040 40320 362880 ]
```

**Exercice 11.9** Refaites l'exercice 12.4 pour créer un stream infini qui représente une séquence d'approximations de  $e$ . Créez ce stream à partir du résultat de l'exercice précédent, en utilisant une combinaison des opérateurs `add-streams` et `divide-streams`.

**Solution 11.9** Voici la solution:

```
(define inversefacstream (divide-streams ones facstream))
(define estream
  (cons-stream 1
               (add-streams (tail inversefacstream) estream)))
(print-stream-to 7 estream)
[ 1.0 2.0 2.5 2.66666666 2.70833333 2.71666666 2.71805555 ]
```

## 12 Programmation orientée objet

**Exercice 12.1** *Implementez une classe orientée objet en Scheme comme expliqué pendant le cours. La classe doit créer des objets qui se comportent comme montré ci-dessous:*

```
>(define number1 (new-number 5))
> (send number1 ' (positive))
#t
> (send number1 ' (decreasewith 10))
-5
> (send number1 ' (show))
-5
> (send number1 ' (negative))
#t
> (send number1 ' (increasewith 5))
0
> (send number1 ' (negative))
#f
```

**Solution 12.1** *Le code Scheme ci-dessous:*

```
(define (new-number number)
  (define (positive) (>= number 0))
  (define (negative) (< number 0))
  (define (decreasewith no)
    (set! number (- number no))
    number)
  (define (increasewith no)
    (set! number (+ number no))
    number)
  (define (show) (display number))
  (define (dispatch m)
    (cond ((eq? m 'positive) positive)
          ((eq? m 'negative) negative)
          ((eq? m 'decreasewith) decreasewith)
          ((eq? m 'increasewith) increasewith)
          ((eq? m 'show) show)
          (else (error "unknown request" m)
                )))
  dispatch)

(define (send object msg)
  (if (null? (cdr msg))
      ((object (car msg)))
      ((object (car msg)) (cadr msg))))
```

```
(define number1 (new-number 5))  
(define number2 (new-number 10))
```