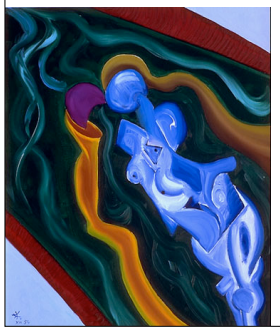
*OPTION PF2 - Automne 2016*
*Jean-Paul Roy*
*L2-Info&Maths, Faculté des Sciences*

http://deptinfo.unice.fr/~roy

*Cours n°12*
*(enfin presque…)*

# La Programmation paresseuse



## (streams, lazy programming)



## Evaluation retardée : les glaçons

• Un **glaçon** (*thunk* en anglais) est une **fermeture**, valeur d'une fonction sans paramètre (`lambda () ...`).

• Idée : plutôt que lancer immédiatement l'évaluation d'une expression expr, on peut la **geler** en construisant un glaçon :

```
(define (glacer expr)
  (lambda () expr))
```

• On pourra plus tard effectuer le calcul en dégelant le glaçon :

```
(define (dégeler glaçon)
  (glaçon))
```

• Exemple :

```
> (define g (glacer (+ 2 3)))
> g
#<procedure>
> (dégeler g)
5
```



• L'idée est intéressante mais l'implémentation est mauvaise puisque Scheme fonctionne en **appel par valeur** : expr a été calculée lors de la construction du glaçon ! On a simplement gelé sa **valeur** et non l'expression elle-même !

• Le remède est facile : une **macro**, pour ne pas évaluer l'argument.

```
(define-syntax glacer
  (syntax-rules ()
    ((glacer expr) (lambda () expr))))

(define (degeler glaçon)
  (glaçon))
```

```
> (define g (glacer (+ 2 3)))
> (degeler g)
5
> (degeler g)        ; un second dégel
5
```

• Hum. Le glaçon a-t-il été (inutilement) dégelé 2 fois ?...



• Le glaçon a-t-il été (inutilement) dégelé 2 fois ? Vérifions :

```
> (define g (glacer (begin (printf "dégel !") (+ 2 3))))
> (degeler g)
dégel !
5
> (degeler g)
dégel !
5
```

• Il a donc bien été dégelé 2 fois ! Mémorisons le fait qu'il a déjà été dégelé la première fois, et conservons la valeur trouvée :

```
(define-syntax glacer
  (syntax-rules ()
    ((glacer expr) (let ((degel? #f) (val '?))
                     (lambda () (if degel? val
                                    (begin (set! val expr)
                                           (set! degel? #t)
                                           val)))))))
```

# Application aux générateurs

• Soit à résoudre un problème ayant plusieurs solutions [peut-être une infinité !]. Plutôt que générer la liste de **toutes** les solutions, on génère un couple (sol,g) formée de la première solution et d'un glaçon qui n'est autre que la **promesse de continuer le calcul** si besoin. On retarde donc le calcul des autres solutions, si la première n'est pas satisfaisante. IDEE FORTE !

• Exemple simpliste : soit à trouver les éléments d'une liste `L` vérifiant un prédicat `p?`. Convenons de retourner le symbole `*echec*` s'il n'y a plus d'autre solution.

*blocage de la récursion !*

```
(define (solution p? L)
  (cond ((null? L) '*echec*)
        ((p? (car L)) (cons (car L)
                            (glacer (solution p? (cdr L)))))
        (else (solution p? (cdr L)))))
```



• La fonction `solution` retourne la première solution (ou `*echec*`) ainsi qu'un glaçon contenant la poursuite du calcul :

```
> (define s (solution integer? '(2 et 3 font 5)))
> s
(2 . #<procedure)
```

*la solution courante* — *un glaçon (λ ( ) ...) sur la solution suivante*

• Et pour continuer le calcul de la solution suivante :

```
> (define s2 (suivante s1))
> s2
(3 . #<procedure>)
> (suivante s2)
(5 . #<procedure>)
> (suivante (suivante s2))
*echec*
```

```
(define (suivante sol)
  (if (equal? sol '*echec*)
      '*echec*
      (degeler (cdr sol))))
```



# Les listes paresseuses

• Les idées précédentes conduisent à un style de programmation où il n'est plus déraisonnable de commencer à traiter une liste avant de l'avoir complètement construite. Surtout si elle est *infinie* !!!

• Après tout, c'est l'analogue d'un *pipe* en Unix :

```
$ cat streams.rkt | grep define
```

• Nous allons développer une technique d'**évaluation paresseuse** [*lazy evaluation*]. En particulier, nous étudierons les **listes paresseuses** [*lazy lists*] implémentées sous forme de **flots** [*streams*]. Elles nous permettront au passage de travailler avec des listes *infinies*. Voir la section 17.4.4 de PCPS. Le langage *lazy* en grandeur nature se nomme **Haskell**, il est purement fonctionnel (PCPS § 17.4.1).

• *Une autre exploitation de l'idée de glaçon est la délicate théorie des* ***continuations****, très avancée en Scheme. PCPS chap. 17 et L3-Info.*



• Soit à calculer le second nombre premier d'un gros intervalle, par ex. `[2,50000]`. Utilisons naïvement la primitive `(filter pred? L)` :

```
(define (interval a b)        ; a ≤ b entiers
  (build-list (+ b (- a) 1) (λ (i) (+ a i))))
```

```
> (second (filter premier? (interval 2 50000)))
[Time = 266 ms]     ; slow !
3
```

• Correct, mais **très inefficace** car il faut d'abord construire le gros intervalle, puis tous ses nombres premiers, avant d'extraire le second ! Pourtant, ce style de programmation d'ordre supérieur est *cool*...

• On veut en conserver l'élégance, tout en entrelaçant la production de l'intervalle et le calcul des nombres premiers !

• Euréka : **pourquoi CONS devrait-il évaluer ses deux arguments ?**

# Les flots [streams]

| Listes | Streams |
|---|---|
| empty | sempty |
| (car (cons x L)) == x | (scar (scons x S)) == x |
| (cdr (cons x L)) == L | (scdr (scons x S)) == S |

• La **cons**truction d'un flot gèlera [*retardera*] l'évaluation de son cdr :

```
(scons x s) <==> (delay (cons x (delay s)))
```

• En vertu de l'ordre d'évaluation (appel par valeur), scons ne peut pas être une fonction, ce sera donc une macro :

```
(define-syntax scons
  (syntax-rules ()
    ((scons x s) (delay (cons x (delay s))))))
```



# Le pb du second nombre premier [revisité]

• Ecrivons l'analogue flot de (interval a b) de la page 8 :

```
(define (sinterval a b)
  (if (> a b)
      sempty
      (scons a (sinterval (+ a 1) b))))
```

```
> (define INT (sinterval 0 +inf.0))    ; !!!
> (time (scar (scdr (sfilter premier? INT))))
[Time = 1 ms]  ; fast !
3
```

*cf page 14*

• Vous voyez pourquoi ?

```
> (time (sinterval 1 100000000000000000000))
[Time = 0 ms]
#<promise>
```

*ce n'est qu'une promesse !*



• La *macro* primitive (delay expr) de Racket se contente de geler l'expression expr pour produire une **promesse** p. Pour dégeler une promesse p, on utilise la *fonction* primitive (force p).

• D'après la définition de stream-cons, un flot n'est qu'une promesse de dégeler la construction d'un doublet contenant une valeur et une promesse ! Pour obtenir le premier élément d'un flot, on dégèle donc le doublet :

```
(define (scar s)
  (car (force s)))
```

• Le reste du flot est une promesse qu'il faut forcer [dégeler] :

```
(define (scdr s)
  (force (cdr (force s))))
```

• Le flot vide est simplement implémenté par la liste vide :

```
(define sempty'())
```

```
(define (sempty? s)
  (empty? (force s)))
```



# L'élément numéro k d'un flot

• Les flots s'utilisent un peu comme les listes, pour représenter une suite (*infinie ?*) de données. On peut donc programmer sur les flots la plupart des fonctions classiques sur les listes. Exemple :

```
(define (sref S k)        ; élément numéro k du flot
  (if (= k 0)
      (scar S)
      (sref (scdr S) (- k 1))))
```

• Je peux calculer le flot de TOUS les nombres premiers, puis extraire un élément du flot :

```
> (define PRIME (sfilter premier? INT))   ; Time = 0 ms
> (sref PRIME 1000)                       ; Time = 46 ms
7879
> (sref PRIME 1000)                       ; Time = 1 ms
7879
```

# Affichage [du début] d'un flot

• Un flot étant en général très long [en général **infini** !], on n'affichera par défaut que les 10 premiers éléments du flot :

```
(define (sprint s . Largs)        ; 10 éléments par défaut
  (define n (if (null? Largs) 10 (car Largs)))
  (printf "[")
  (do ((i 0 (+ i 1))
       (s s (scdr s)))
      ((>= i n) (printf "...]"))
    (printf "~a," (scar s))))
```

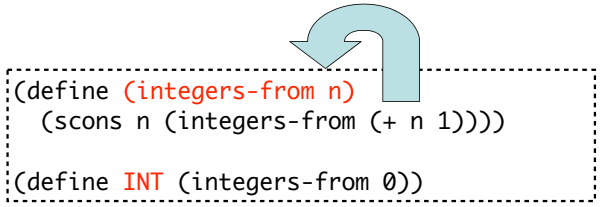
```
> (sprint PRIME)
[2,3,5,7,11,13,17,19,23,29,…]
> (sprint PRIME 40)
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,
79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,
163,167,173,…]
```



# Des flots infinis !

• Des mathématiques [l'ensemble des entiers] à l'informatique [un serveur], des objets et processus infinis sont couramment à l'oeuvre.

• Utilisons **l'évaluation retardée** pour implémenter l'idée d'infini :



```
(define (integers-from n)
  (scons n (integers-from (+ n 1))))

(define INT (integers-from 0))
```

*Une récursivité sans cas d'arrêt !!!*

• Le flot `INT` est infini, mais à tout instant nous n'examinons qu'une portion finie de ce flot ! Le fond de l'histoire, là encore, c'est que `INT` est une **promesse** et tient dans un seul doublet !!!

```
> INT
#<promise>
```



# L'ordre supérieur : map, filter…

• On peut appliquer [paresseusement !] une fonction f à tous les éléments d'un flot S, même infini :

```
(define (smap f S)
  (if (sempty? S)        ; inutile si le flot S est infini !
      S
      (scons (f (scar S))
             (smap f (scdr S)))))
```

```
> (sprint (smap (lambda (x) (* x 2)) INT))
[0,2,4,6,8,10,12,14,16,18,...]
```

```
(define (sfilter f S)
  …)
```

*en TP*

```
> (sprint (sfilter (λ (x) (zero? (modulo x 7))) INT))
[0,7,14,21,28,35,42,49,56,63,...]
```



• Le **flot des nombres de Fibonacci**, dans le même style. Programmons `(fibonacci a b)` retournant la suite infinie [a,b,…] des nombres de Fibonacci :

```
(define (fibonacci a b)
  (scons a (fibonacci b (+ a b))))

(define FIBS (fibonacci 0 1))
```

```
> (sprint FIBS)
[0,1,1,2,3,5,8,13,21,34,…]
```

• Le flot de **TOUS** les nombres premiers :

```
(define PRIME (sfilter premier? INT))
```

```
> (sprint PRIME)
[2,3,5,7,11,13,17,19,23,29,…]
```

# Le crible d'Eratosthènes

• On part du flot **infini** F des entiers ≥ 2 que l'on souhaite "cribler".

```
      2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
...
```

etc…

*Voir PCPS…*

• Construire ainsi le flot de tous les nombres premiers !



• On part d'un flot S = (scons p R) dont le premier élément est un nombre premier p. On construit un nouveau flot commençant par p et dont le reste est le crible du flot R privé des multiples de p.

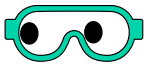
```
(define ERATO
  (define (crible S)
    (let ((p (scar S)))
      (stream-cons p
        (crible (sfilter
                  (lambda (n) (not (zero? (modulo n p))))
                  (scdr S))))))
  (crible (integers-from 2)))

> (sprint ERATO)
[2,3,5,7,11,13,17,19,23,29,…]
> (sref ERATO 100)            ; le 100-ème nombre premier !
547
```



# Définitions implicites de flots

• Encore plus fort ! Il est possible de définir un flot implicitement. Exemple : un flot infini ne contenant que des nombres 1 :

```
(define ONE (scons 1 ONE))
```



```
> (sprint ONE)
[1,1,1,1,1,1,1,1,1,1,…]
```

• D'où une autre construction des entiers naturels :

```
  0 1 2 3 4 5 6 ...      INT
+ 1 1 1 1 1 1 1 ...      ONE
------------------      ---
0 1 2 3 4 5 6 7 ...      INT
```

```
(define INT
  (scons 0 (smap2 + INT ONE)))
```

*binaire, cf TP…*

```
> (sprint INT)
[0,1,2,3,4,5,6,7,8,9,…]
```



# Itération ou flot de données ?

• Considérons le processus itératif de calcul de `(sqrt x)` par améliorations successives, par exemple avec la méthode de Newton :

```
(define (ameliore x approx)
  (moyenne approx (/ x approx)))
```

*cf PF1…*

• On peut générer les approximations dans un flot implicite :

```
(define (stream-sqrt x)          ; flot des approximations de √x
  (define (ameliore approx)
    (* 0.5 (+ approx (/ x approx))))
  (define approximations
     (scons 1.0 (smap ameliore approximations)))
  approximations)
```

```
> (sprint (stream-sqrt 2))
[1.0,1.5,1.4166666666666665,1.4142156862745097,1.41421356237468
99,1.414213562373095,1.414213562373095,1.414213562373095,1.4142
13562373095,1.414213562373095,…]
```

# Application aux séries numériques (formelles)

$$\sum_{n\geq 0} a_n x^n = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

• Les **séries** interviennent naturellement en théorie du signal, et se prêtent bien à une modélisation sous forme de flots !

• Se donner une série revient en effet à se donner le **flot infini de ses coefficients** $\{a_0, a_1, a_2, a_3, \ldots\}$ tout comme se donner un polynôme revenait à se donner la liste finie de ses coefficients.

• On ne se pose pas de problème de convergence (**série formelle**) !



• Un **polynôme** peut être vu comme une série particulière :

```
(define (poly->serie L)    ; L = coeffs en puissances croissantes
  (define (copy L)         ; transformation en flot infini
    (if (null? L)
        ZERO               ; la série nulle [0,0,0,…]
        (scons (car L) (copy (cdr L)))))
  (copy L))
```

```
> (sprint (poly->serie '(1 2 3)))
{1,2,3,0,0,0,0,0,0,0,...}
```

$1 + 2x + 3x^2$

• Une série peut être intégrée terme à terme : $\int (a_0 + a_1 x + a_2 x^2 + \ldots) dx = \frac{a_0}{1} x + \frac{a_1}{2} x^2 + \frac{a_2}{3} x^3 + \ldots$

```
(define (int-serie S)
  (define (aux S i)
     (scons (/ (scar S) i)
            (aux (scdr S) (+ i 1))))
  (aux S 1))
```



• Exemple : la série exponentielle $E = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = 1 + \int E$

• On peut donc la définir de manière implicite :

```
(define EXPO (scons 1 (int-serie EXPO)))
```

```
> (sprint EXPO)
[1,1,1/2,1/6,1/24,1/120,1/720,1/5040,1/40320,1/362880,…]
```

• Pour les séries de sin(x) et cos(x), on peut remarquer que :

sin'(x) = cos(x)  et  cos'(x) = -sin(x)

```
(define COSINUS (scons 1  ???))
(define SINUS (scons 0  ???))
```

*cf TP...*

```
> (sprint COSINUS)
[1,0,-1/2,0,1/24,0,-1/720,0,1/40320,0,…]
> (sprint SINUS)
[0,1,0,-1/6,0,1/120,0,-1/5040,0,1/362880,…]
```



• Reste à implémenter les opérations sur les séries :

```
(define (serie+ S1 S2) (smap2 + S1 S2))
(define (serie- S1 S2) (smap2 - S1 S2))
```

*smap2, cf TP...*

• Hélas, le produit de deux séries ne se fait pas terme à terme !

$$(a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots) \times (b_0 + b_1 x + b_2 x^2 + b_3 x^3 + \cdots)$$
$$= a_0 b_0 + a_0 (b_1 x + b_2 x^2 + \ldots) + (a_1 x + a_2 x^2 + \ldots) \times (b_0 + b_1 x + b_2 x^2 + \ldots)$$

```
(define (serie* S1 S2)
  (scons (* (scar S1) (scar S2))
         (serie+ ??? (serie* ???  ???))))
```

• On peut par exemple [cf TP] vérifier l'identité trigonométrique :

$$\cos^2 x + \sin^2 x = 1$$

en calculant la série associée au membre gauche...

## • INVERSION d'une série formelle de premier terme 1.

• Soit $S = \sum_{n\geq 0} a_n x^n$ une série formelle. Elle est **inversible** si :

il existe une série formelle $T = \sum_{n\geq 0} b_n x^n$ telle que S * T = 1

• Maths : S est inversible si et seulement si son terme constant $a_0$ est inversible. Quitte à la diviser par $a_0$, on peut donc supposer $a_0 = 1$ :

S = 1 + R  où R est la série restante après le premier terme

• On cherche à résoudre S * T = 1, soit (1 + R) * T = 1, ou encore T + R * T = 1, donc [de manière implicite] : T = 1 - R * T

```
(define (inv-serie S)          ; avec S = 1 + a1x + a2x^2 + ...
  (scons 1 (smap - (serie* (scdr S)
                           (inv-serie S)))))
```

black magic !



• Retrouvons par exemple le développement bien connu :

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \ldots$$

```
> (sprint (inv-serie (poly->serie '(1 -1))))
[1,1,1,1,1,1,1,1,1,1,…]
```

• Connaissant `(serie* S1 S2)` et `(inv-serie S)`, il est immédiat d'en déduire la division de deux séries `(serie/ S1 S2)` pourvu que le terme constant de `S2` soit ≠ `0` :

```
(define (serie/ S1 S2)
  (let ((constant (scar S2)))
    (if (zero? constant)
        (error "serie/ : dénominateur non inversible!"))
        (serie*  ???   ???))))    ; cf TP

> (define TANGENTE (serie/ SINUS COSINUS))
> (sprint TANGENTE)
[0,1,0,1/3,0,2/15,0,17/315,0,62/2835,…]
```

$\tan x = x + \frac{x^3}{3} + \frac{2x^5}{15} + \ldots$

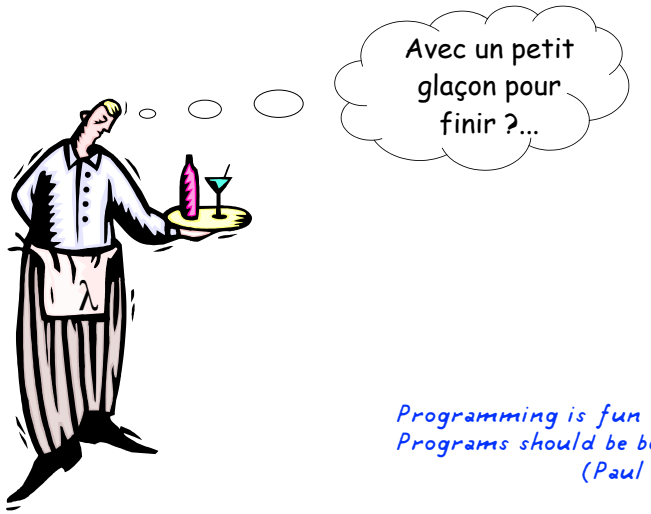
*Elle est pas belle, la vie ?...*



## Une implémentation de delay et force

```
(define-syntax stream-cons          ; le constructeur de flots
  (syntax-rules ()
    ((stream-cons x S) ($delay (cons x ($delay S))))))

(define-syntax $delay               ; pour geler une expression
  (syntax-rules ()
    (($delay expr) (make-promise (lambda () expr)))))

(define (make-promise thunk)        ; prend un glaçon...
    (let ((result-ready? #f) (result #f))
      (lambda ()                    ; ... et rend un glaçon
        (if result-ready?           ; à mémoire !
            result
            (let ((x (thunk)))
              (begin (set! result-ready? #t)
                     (set! result x)
                     result)))))))

(define ($force g)                  ; pour dégeler un glaçon
  (g))
```






*Programming is fun*
*Programs should be beautiful.*
*(Paul Graham)*

http://www.apmep.fr/IMG/pdf/16_Roy_Hamming.pdf