

La Programmation paresseuse



(streams, lazy
programming)

1

• 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 ?...

3

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 (degeler glaçon)
  (glaçon))
```

• Exemple :

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

2

• 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)))))))
```

4

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. IDÉE 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.

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

*blocage de la
récursion !*

5

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.

7

- 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))))
```

6

- 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 ?



8

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,...]
```

13

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 (empty? 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 (lambda (x) (zero? (modulo x 7))) INT))
[0,7,14,21,28,35,42,49,56,63,...]
```

14

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>
```

15

- 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,...]
```

16

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 !

17

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

```

```
(set! INT
(scons 0 (smap2 + INT ONE)))
```

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

→ binaire, cf TP...

18

Itération ou flot ?

- Considérons le processus itératif de calcul de $(\text{sqrt } x)$ par améliorations successives, par exemple avec la méthode de Newton :

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

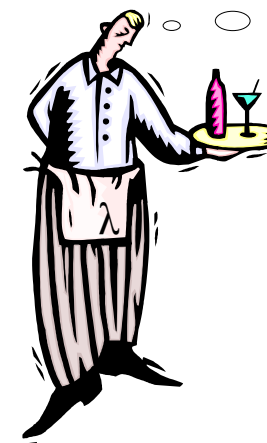
cf PFI...

- On peut générer les approximations dans un flot implicite :

```
(define (stream-sqrt x) ; flot des approximations de  $\sqrt{x}$ 
(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,...]
```

19



Avec un petit
glaçon pour
finir ?...

Good luck...

20