Sur la Construction d'une Fonction Récursive

jpr, L1 Info & Math (Oct 2014)

La **programmation par récurrence**[1] [⇔ *récursive*] est essentielle en Scheme puisque toutes les techniques répétitives passent par une récurrence dans ce langage, qui n'est au fond qu'une algèbre appliquée. Dans sa forme générale, le principe de récurrence pour démontrer une propriété P(n) portant sur un entier naturel n s'énonce ainsi :

- je commence par prouver que P(0) est vrai.
- je suppose que P(0), P(1), P(2), …, P(n-1) sont vrais [HR] et j'en déduis P(n)

Les matheux *prouvent* par récurrence tandis que les programmeurs *construisent* par récurrence. Ils construisent des fonctions mais aussi des données [nous verrons plus tard l'exemple typique des listes et des arbres] en utilisant abondamment la pensée récursive. Toute tentative de la négliger conduit à des marécages à la fois théoriques et pratiques… L'avantage d'une fonction récursive est qu'on peut raisonner sur elle [prouver qu'elle est correcte, calculer sa complexité, vérifier ses propriétés]… par récurrence, donc sans sortir de la mathématique usuelle.

Ceci dit, la récurrence demande quand même pas mal d'habitude et de pratique. Je vais prendre un exemple typique, celui des **nombres parfaits** [exo TD 5.2]. Un nombre entier $n \geq 2$ est *parfait* s'il est égal à la somme de ses diviseurs stricts, donc dans [1, n-1]. Par exemple $6 = 1+2+3$ est parfait tandis que $8 \neq 1+2+4$ ne l'est pas. En collant à la définition, je commence par programmer un prédicat[2] `(parfait? n)` retournant `true` si et seulement si n est parfait.

```
(define (parfait? n)       ; n entier
  (and (>= n 2) (= n (somdiv n))))
```

et je suis ramené au sous-problème de programmer la fonction `(somdiv n)` prenant un entier $n \geq 2$ et retournant la somme des diviseurs stricts de n. Je tente une récurrence brutale sur n. Supposons que je connaisse la somme des diviseurs de n-1, puis-je en déduire la somme des diviseurs de n ? Non bien entendu puique les diviseurs de n-1 n'ont rien à voir avec les diviseurs de n. Râté !

Puisque je ne peux pas *récurrer* sur n, je vais essayer de **GENERALISER LE PROBLEME** et de trouver une autre variable sur laquelle *récurrer*. Mon idée de base consiste à parcourir l'intervalle [1,n-1] avec une seconde variable k que je dois introduire. Ce qui me conduit à programmer une fonction `(somdiv-aux n k)` un peu plus générale, qui va retourner la somme des diviseurs de n dans l'intervalle [1,k]. Si je sais programmer cette fonction plus générale, j'aurai gagné puisqu'alors :

```
(define (somdiv n)             ; n ≥ 2
  (somdiv-aux n (- n 1)))      ; retourne la somme des diviseurs de n dans [1,n-1]
```

---

[1] http://fr.wikipedia.org/wiki/Raisonnement_par_recurrence

[2] Un prédicat est une fonction à valeur booléenne, donc *true* ou *false*.

Je suis donc encore ramené au **sous-problème**[3] de programmer la fonction `(somdiv-aux n k)`, par récurrence sur k cette fois, n restant fixe. La somme des diviseurs de n dans [1,k] n'est autre que la somme des diviseurs de n dans [1,k-1] à laquelle on ajoutera k si k est un diviseur de n. Le cas de base a lieu lorsque k = 1 :

```
(define (somdiv-aux n k)            ; somme des diviseurs de n dans [1,k]
  (cond ((= k 1) 1)
        ((= (modulo n k) 0) (+ k (somdiv-aux n (- k 1))))
        (else (somdiv-aux n (- k 1)))))
```

Donc je tiens mon prédicat `parfait?`, c'est parfait. Notez qu'il a une complexité [un coût] en O(n) opérations puisqu'on parcourt [1,n] en faisant une division chaque fois. Maintenant je cherche combien il y a de nombres parfaits jusqu'à 1000. Généralisation immédiate : jusqu'à n, puis je ferai n = 1000. Cette fois, une récurrence brutale sur n fonctionne immédiatement :

```
(define (nb-parfaits n)             ; nombre d'entiers parfaits dans [1,n]
  (cond ((<= n 1) 0)
        ((parfait? n) (+ 1 (nb-parfaits (- n 1))))
        (else (nb-parfaits (- n 1)))))
```

```
> (nb-parfaits 1000)   ; 0.1 sec          > (nb-parfaits 10000)   ; 15 sec !!
3                                         4
```

L'augmentation dramatique du temps de calcul tient au fait que la **complexité** de `nb-parfaits` est en 1+2+3+…+n = n(n+1)/2 = $O(n^2)$ cette fois !

Intéressons-nous au calcul de la *liste* [cours 6] de ces trois nombres parfaits mystérieux ≤ 1000. Dans la fonction précédente, je remplace l'addition + par un ajout dans une liste avec `cons` :

```
(define (liste-parfaits n)          ; liste des entiers parfaits de [1,n]
  (cond ((<= n 1) empty)
        ((parfait? n) (cons n (liste-parfaits (- n 1))))
        (else (liste-parfaits (- n 1)))))
```

```
> (liste-parfaits 1000)             ; complexité élevée… Le temps augmente vite !
(496 28 6)
```

Oui, on obtient la liste à l'envers ☹. On peut la redresser avec (un seul) `reverse`, ou bien opter pour une autre stratégie qui la construira directement à l'endroit ! En cas d'extrême fainéantise [et au risque de ralentir le programme], on programmera un *one-liner* [cours 9] avec `filter` :

```
(define (liste-parfaits n)          ; liste des entiers parfaits de [1,n]
  (filter parfait? (build-list n add1))      ; le style APL[4]…
```

[3] Cette technique de *se ramener systématiquement à des sous-problèmes* a fait ses preuves dans toutes les sciences !

[4] Sur le langage de programmation APL (dont le successeur est J) : http://www.afapl.asso.fr