Comprendre la Comprehension

- Comprehension sur les listes
  - exemples avec l'extension de syntaxe CamlP4
  - coder la comprehension à l'aide de concat, map et fold
  - correction du codage
- Comprehension sur d'autres structures: découverte de return et bind

Définir des fonctions par comprehension

En mathématique, on a l'habitude de définir des ensembles à partir de propriétés que leur éléments satisfont. Par exemple

$$PythagoreanTriples(n) = \{(a, b, c) \mid a^2 + b^2 = c^2,\ a, b, c \leq n\}$$

Ce type de définition est très concise; elle décrit l'objet d'intérêt, pas le moyen de le construire.

Comprehension en programmation

Le langage de programmation SETL (voir un exemple sur `http://setl.org/setl/setl-server.html`) introduit des définitions par comprehension, comme

```
{p in {2 .. n} | forall i in {2 .. floor sqrt p}
               | p mod i /= 0}
```

Dans les langages de programmation fonctionnels, on dispose d'une syntaxe, dite aussi *par comprehension*, qui permet d'écrire des définitions dans ce même style.

On verra ici comment l'utiliser, les pièges à éviter, et comment obtenir les mêmes résultats sans extensions de syntaxe.

List comprehension en OCaml avec Camlp4

La distribution standard de OCaml vient avec un outil, Camlp4, un "Pre-Processor-Pretty-Printer for OCaml", qui permet d'écrire et utiliser des extensions de syntaxe.
On peut charger ces extensions de syntaxe dans l'interpréteur; par exemple:

```
#load "dynlink.cma";;

#load "camlp4o.cma";;
|| Camlp4 Parsing version 4.00.1
||
```

charge le parseur standard Camlp4 pour OCaml (camlp4o), qui a besoin de la librairie dynlink.
Ensuite, nous allons charger une extension de syntaxe qui ajoute la *compréhension des listes*:

```
#load "Camlp4Parsers/Camlp4ListComprehension.cmo";;
```

List comprehension en OCaml

Avec cette extension, nous pouvons écrire la fonction qui calcule les triples Pythagoréennes presque comme dans la définition mathématique:

```
let rec range i j = if i > j then [] else i :: range (i+1) j;;
|| val range : int -> int -> int list = <fun>

let triples n =
 [ (a,b,c) | a <- range 1 n; b <- range 1 n; c <- range 1 n;
             a*a + b*b = c*c ];;
|| val triples : int -> (int * int * int) list = <fun>
```

Le calcul nous donne un résultat correct ...

```
triples 10;;
|| - : (int * int * int) list = [(3, 4, 5); (4, 3, 5); (6, 8, 10); (8, 6, 1
|| 0)]
```

... mais rédondant.

List comprehension en OCaml

Nous pouvons améliorer notre fonction pour produire les triples Pythagoréennes sans répétitions:

```
let triples_ordered n =
 [ (a,b,c) | a <- range 1 n; b <- range a n; c <- range b n;
             a*a + b*b = c*c ];;
|| val triples_ordered : int -> (int * int * int) list = <fun>


triples_ordered 10;;
|| - : (int * int * int) list = [(3, 4, 5); (6, 8, 10)]
```


List comprehension: partition d'un entier

Voyons un autre exemple... on veut trouver toutes les façons de partitionner un entier en deux:

$$Partition(n) = \{(a, b) \mid a + b = n, a \leq b\}$$

On peut écrire:

```
let partition n = [ (a,b) | a <- range 1 n; b <- range 1 n;
                            a+b=n; a<=b ];;
|| val partition : int -> (int * int) list = <fun>
```

Le calcul nous donne un résultat correct ...

```
partition 10;;
|| - : (int * int) list = [(1, 9); (2, 8); (3, 7); (4, 6); (5, 5)]
```

... mais on n'est pas très efficaces, essayez partition 10000!

List comprehension: partition efficace d'un entier

Il vaut bien mieux écrire:

```
let partition_better n =
 [ (a,b) | a <- range 1 n; b <- range a n; a+b=n];;
```

Et même:

```
let partition_efficient n =
 [ (a,b) | a <- range 1 (n/2); b <- [n-a]];;
```

Le calcul nous donne le même résultat mais c'est bien plus efficace, essayez partition 10000!
Et au passage, il vaut aussi mieux de reecrire range comme suit (pourquoi?):

```
(* tail recursive version of range *)
let range i j =
  let rec aux acc n = if n < i then acc else aux (n::acc) (n-1)
  in aux [] j;;
```

Traduction de la comprehension des listes

Ouvrons le capot

List comprehension sous le capot

Avec l'instruction

camlp4o Camlp4ListComprehension.cmo list_comprehension_example.ml

on peut obtenir la traduction en OCaml de la fonction partition écrite avec la comprehension des listes; ell'est nettement moins lisible:

```
let partition n =
 List.concat
  (List.map
    (fun a ->
      List.map (fun b -> (a, b))
       (List.filter (fun b -> a <= b)
         (List.filter (fun b -> (a + b) = n) (range 1 n))))
    (range 1 n))
;;
```

Effectivement, on peut tout faire avec concat, map et filter !

La list comprehension: la syntaxe

Pour comprenre comment la comprehension de listes peut être traduite, il faut d'abord fixer la syntaxe qui est permise

Syntaxe

$$\begin{array}{rcl} comp & ::= & [\ e \mid qseq\ ] \\ qseq & ::= & q \mid q; qseq \\ q & ::= & x \leftarrow e \mid f \end{array}$$

Donc une expression de comprehension de liste a la forme $[\ e \mid qseq\ ]$ où e est une expression OCaml, et $qseq$ est une séquence de *qualifiers*, qui sont soit des *générateurs* $x \leftarrow e$, soit des *filtres* f, c'est à dire des expressions booléennes.

La séquence de qualifiers doit commencer avec un générateur, les variables qui apparaissent dans les générateurs se comportent comme des lieurs, et l'expression e à droite de la flèche doit produire une liste.

La list comprehension: la sémantique

On donne maintenant la sémantique[1] qu'on entend pour ces expressions:

Sémantique

$$\begin{array}{lll} (pass) & [\ e \mid true;\ Q\ ] & = [\ e \mid Q\ ] \\ (fail) & [\ e \mid false;\ Q\ ] & = [\ ] \\ (end) & [\ e \mid\ ] & = [\ e\ ] \\ (cut) & [\ e \mid x \leftarrow [\ ];\ Q\ ] & = [\ ] \\ (enumerate) & [\ e \mid x \leftarrow e_1 :: e_r;\ Q\ ] & = \\ & \quad (let\ x\ = e_1\ in\ [\ e \mid Q\ ])@[\ e \mid x \leftarrow e_r; Q\ ] & \end{array}$$

[1] Ceci est informel, mais on pourrait être très précis.

Traduction de la list comprehension

Il y a plusieurs façons de traduire une expression avec la comprehension.
Celle utilisée par l'extension CamlP4 est la suivante:

Traduction dans CamlP4

$$
\begin{array}{lcl}
T[\![[\ e\ |x \leftarrow e']]\!] & = & \text{List.map (}\textbf{fun}\ \text{x}{->}\ \text{e)}\ \text{e'} \\
T[\![[\ e\ |x \leftarrow e'; y \leftarrow e''; \ Q\ ]]\!] & = & \\
\quad (\text{List.concat}\ (\text{List.map (}\textbf{fun}\ \text{x} -> T[\![[\ e\ |y \leftarrow e''; \ Q\ ]]\!])\ e')) & & \\
T[\![[\ e\ |x \leftarrow e'; f; \ Q\ ]]\!] & = & \\
\quad T[\![[\ e\ |x \leftarrow (\text{List.}\ \text{filter}\ \ (\textbf{fun}\ \text{x} -> \text{f})\ \text{e'}); \ Q\ ]]\!] & &
\end{array}
$$

On peut vérifier qu'elle respecte la sémantique. Essayez d'appliquer cette traduction à la main, et comparez avec la sortie de OCamlP4.

Optimisations

On peut observer que le motif

List.concat(List.map f e)

dans cette traduction n'est pas optimal:

- on utilise List.map pour construire une liste
- et on la consomme tout de suite avec List.concat

On peut utiliser à la place, comme c'etait fait en Haskell,

```
let rec concatMap f = function
  | [] -> []
  | a::r -> (f a) @ (concatMap f r);;
```

Ou en utilisant *fold_right*

```
let concatMap f l =
  List.fold_right (fun x acc -> (f x)@acc) l [] ;;
```


Une traduction uniforme

Vers bind et return

Une autre approche

Il existe une traduction plus uniforme qui utilise concatMap:

Traduction uniforme

$$
\begin{array}{lcl}
T[\![[\ e\ |x \leftarrow e']]\!] & = & \text{concatMap (}\textbf{fun}\ \text{x}{->}\ \text{[e])}\ \text{e'} \\
T[\![[\ e\ |x \leftarrow e'; y \leftarrow e''; \ Q\ ]]\!] & = & \\
\quad \text{concatMap (}\textbf{fun}\ \text{x} -> T[\![[\ e\ |y \leftarrow e''; \ Q\ ]]\!])\ e' & & \\
T[\![[\ e\ |x \leftarrow e'; f; \ Q\ ]]\!] & = & \\
\quad \text{concatMap (}\textbf{fun}\ \text{x} -> \textbf{if}\ \text{f}\ \textbf{then}\ T[\![[\ e\ |\ Q\ ]]\!]\ \textbf{else}\ [])\ \text{e'} & &
\end{array}
$$

Ell'est un peu moins efficace, sur le cas de base, que la traduction vue plus haut, mais globalement très raisonnable.
Son grand avantage est qu'elle ouvre la porte à une programmation par comprehension *sans extensions de syntaxe spécifiques*!

Analyse des ingrédients

Pour notre traduction, nous avons eu besoin seulement de quelques ingrédients:

```
let bind l f = concatMap f l
let return = fun x -> [x]
let zero = []
;;
|| val bind : 'a list -> ('a -> 'b list) -> 'b list = <fun>
|| val return : 'a -> 'a list = <fun>
|| val zero : 'a list = []
```

Notez le type de ces fonctions: elles vont nous suffire pour faire de la comprehension sans extensions de syntaxe.

Un module List bien spécifique

Mettons tout cela ensemble:

```
module ListMonad =
  struct
    type 'a t = 'a list
    let return x = [x]
    let bind l f = List.fold_right (fun x acc -> (f x)@acc) l []
    let zero = []
    let ( >>= ) l f  = bind l f
  end;;
|| module ListMonad :
||   sig
||     type 'a t = 'a list
||     val return : 'a -> 'a list
||     val bind : 'a list -> ('a -> 'b list) -> 'b list
||     val zero : 'a list
||     val ( >>= ) : 'a list -> ('a -> 'b list) -> 'b list
||   end
```


Revisitons nos exemples I

Voyons comment écrire nos exemples avec ça:

```
open ListMonad;;


(* echauffement *)

(* [ (x,y) | x <- range 1 3; y <- range 1 3 ] *)

[1;2;3] >>= fun x ->
[1;2;3] >>= fun y ->
return (x,y);;
|| - : (int * int) list =
|| [(1, 1); (1, 2); (1, 3); (2, 1); (2, 2); (2, 3); (3, 1); (3
||
```


Revisitons nos exemples II

```
(* [ (x,y) | x <- range 1 3;
           y <- range 1 3;
           x+y > 3 ] *)

[1;2;3] >>= fun x ->
[1;2;3] >>= fun y ->
if x+y > 3 then return (x,y) else zero;;
|| - : (int * int) list = [(1, 3); (2, 2); (2, 3); (3, 1);
```

Revisitons nos exemples III

```
(* Pythagorean triples :
   [ (a,b,c) | a <- range 1 n;
               b <- range a n;
               c <- range b n;
               a*a + b*b = c*c ] *)

let triples n =
 range 1 n >>= fun a ->
 range a n >>= fun b ->
 range b n >>= fun c ->
 if a*a + b*b = c*c then return (a,b,c) else zero;;
|| val triples : int -> (int * int * int) list = <fun>
```


Revisitons nos exemples IV

```
(* [ (a,b) | a <- range 1 n;
              b <- range a n;
              a+b=n] *)

let partition n =
 range 1 n >>= fun a ->
 range a n >>= fun b ->
 if a+b=n then return (a,b) else zero;;
|| val partition : int -> (int * int) list = <fun>
```


On refait avec des tableaux

```
module ArrayMonad =
  struct
    type 'a t = 'a array
    let return x = Array.create 1 x
    let bind a f = Array.fold_right
        (fun x acc -> Array.concat [(f x); acc]) a [||]
    let zero = [| |]
    let ( >>= ) a f  = bind a f
  end
let range i j = Array.init (j-i+1) (fun x -> x+i);;
|| module ArrayMonad :
||   sig
||     type 'a t = 'a array
||     val return : 'a -> 'a array
||     val bind : 'a array -> ('a -> 'b array) -> 'b array
||     val zero : 'a array
||     val ( >>= ) : 'a array -> ('a -> 'b array) -> 'b array
||   end
|| val range : int -> int -> int array = <fun>
```


Revisitons nos exemples I

Encore une fois:

```
open ArrayMonad;;

(* echauffement *)

(* [ (x,y) | x <- range 1 3; y <- range 1 3 ] *)

[|1;2;3|] >>= fun x ->
[|1;2;3|] >>= fun y ->
return (x,y);;
|| - : (int * int) array =
|| [|(1, 1); (1, 2); (1, 3); (2, 1); (2, 2); (2, 3); (3, 1
||  |]
```

Revisitons nos exemples II

```
(* [ (x,y) | x <- range 1 3;
             y <- range 1 3;
             x+y > 3 ] *)
```

```
[|1;2;3|] >>= fun x ->
[|1;2;3|] >>= fun y ->
if x+y > 3 then return (x,y) else zero;;
|| - : (int * int) array = [|(1, 3); (2, 2); (2, 3); (3, 1); (3
|| |]
```


Revisitons nos exemples III

```
(* Pythagorean triples :
   [ (a,b,c) | a <- range 1 n;
               b <- range a n;
               c <- range b n;
               a*a + b*b = c*c ] *)
```

```
let triples n =
 range 1 n >>= fun a ->
 range a n >>= fun b ->
 range b n >>= fun c ->
 if a*a + b*b = c*c then return (a,b,c) else zero;;
|| val triples : int -> (int * int * int) array = <fun>
```


Revisitons nos exemples IV

```
(* [ (a,b) | a <- range 1 n;
             b <- range a n;
             a+b=n] *)
```

```
let partition n =
 range 1 n >>= fun a ->
 range a n >>= fun b ->
 if a+b=n then return (a,b) else zero;;
|| val partition : int -> (int * int) array = <fun>
```


Observation

Il semble qu'on puisse écrire facilement des programmes par comprehension si on dispose d'un type de donnée 'a t équipé avec des fonctions

```
bind : 'a t -> ('a -> 'b t) -> 'b t
return : 'a -> 'a t
zero : 'a t
```

Optimisations I

On peut facilement vérifier les équations suivantes, qu'on peut utiliser pour optimiser du code mal écrit:

```
(* bind m return = m *)
[|1;2;3|] >>= fun x -> return x;;
|| - : int array = [|1; 2; 3|]


(* bind (return e) f = f e *)

return 3 >>= fun x -> return (2*x);;
|| - : int array = [|6|]

(fun x -> return (2*x)) 3;;
|| - : int array = [|6|]
```


Optimisations II

```
(* bind e1 (fun x -> bind e2 (fun y -> e3))
   = bind (bind e1 (fun x -> e2)) (fun y -> e3)
   si x non libre dans e3
 *)
[|1;2;3|] >>= fun x ->
range x 3 >>= fun y -> return y;;
|| - : int array = [|1; 2; 3; 2; 3; 3|]


([|1;2;3|] >>= fun x -> range x 3)
>>= fun y -> return y;;
|| - : int array = [|1; 2; 3; 2; 3; 3|]
```


Optimisations III

```
(* bind zero f = zero *)

zero >>= (fun x -> return (x+1));;
|| - : int array = [||]
```


Conclusions

- la comprehension est un méchanisme puissant, qui permet d'écrire des programmes proches de la définition du problème ...
- ... mais il s'agit toujours de programmes, donc il faut faire attention à l'efficacité (revoir le cas de partition, par exemple)
- on n'a pas spécialement besoin d'une extension de syntaxe, tout peut se faire en utilisant bind, return et zero
- on peut donc utiliser de la comprehension sur *toute structure* qui dispose de ces opérations

Pour en savoir plus


J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky.
Programming with sets; an introduction to SETL.
Springer-Verlag New York, Inc., New York, NY, USA, 1986.


Simon L. Peyton Jones.
The Implementation of Functional Programming Languages.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
Voir le Chapitre 7


Simon Peyton Jones and Philip Wadler.
Comprehensive comprehensions.
In Proceedings of the ACM SIGPLAN workshop on Haskell workshop, Haskell '07, pages 61–72, New York, NY, USA, 2007. ACM.