Introduction à CAML

Solange Coupet Grimal

Cours Logique, Déduction, Programmation 2002–2004[*]

1 Définition de types

1.1 Produits et enregistrements

1.1.1 Le type `unit`

Il existe un type prédéfini appelé `unit` avec une seule valeur notée `()`.

```
# ();;
- : unit = ()
```

Ce type corresponde à l'ensemble avec un seul élément, qui est unité a gauche et à droite pour le produit Cartésien. On peut s'en servir pour retarder l'évaluation :

```
# let expr = fun () -> 3 + 5;;
val expr : unit -> int = <fun>
# let expr () = 3 + 5;;
val expr : unit -> int = <fun>
# expr ();;
- : int = 8
```

En mathématiques on peut définir le produit Cartésien de deux ensembles A et B comme l'ensemble des fonctions $f : \{1,2\} \longrightarrow A + B$ tels que $f(1) \in A$ et $f(2) \in B$. On peut mettre en correspondance f au couple $(f(1), f(2))$, et cette correspondance est une bijection. En Caml ce codage des produits Cartésiens est fait par les records.

```
type nom_du_type  = { nom1 : t1; ... ; nomn : tn }
```

On peut alors définir des objets ayant ce type par

```
let obj = { nom1=expr1; ... ; nomn :exprn }
```

et accéder aux composantes (projections du produit Cartésiens) par

[*]Texte révisé par Luigi Santocanale le 15 octobre 2005, et adapté au langage Objective Caml.


```
let projdeobjsurnomk = obj.nomk
```

Par exemple :

```
# type nocomplexe = { re: float; im :float};;
type nocomplexe = { re : float; im : float; }
# let zero = {re=0.;im=0.};;
val zero : nocomplexe = {re = 0.; im = 0.}
# let multcomplexe  x y =
  let
    partiereelle = x.re *. y.re -. x.im *.y.im
  and
    partieimaginaire = x.re *. y.im +. x.im *. y.re
  in
    { re = partiereelle ; im = partieimaginaire };;
            val multcomplexe : nocomplexe -> nocomplexe -> nocomplex
```

Ce type de traitement des produits Cartésiens est certainement plus aisé pour le programmeur : il ne faut pas se rappeler de l'ordre avec lequel les champs sont organisé dans une structure.

1.2 Sommes avec constructeurs

1.2.1 Constructeurs constants

On peut définir des types énumérés de la façon suivante :

```
type T = <ident> | ... | <ident>
```

Les identificateurs doivent être tous distincts et le type T est alors l'ensemble des valeurs énumérées dans le type et qui ont pour nom ces identificateurs. Deux valeurs de noms différents sont distinctes.

```
# type traffic_light = Red | Green | Orange;;
type traffic_light = Red | Green | Orange
# type jour = Lu | Ma | Me | Je | Ve | Sa | Di;;
type jour = Lu | Ma | Me | Je | Ve | Sa | Di
# Lu;;
- : jour = Lu
```

Le type prédéfini `bool` peut-être redéfini de la façon suivante :

```
# type bool = Vrai | Faux;;
type bool = Vrai | Faux
# let ou t1 = match t1  with
    Vrai    -> (fun x -> Vrai)
  | Faux   -> (fun x -> x);;
    val ou : bool -> bool -> bool = <fun>
```

1.2.2 Constructeurs avec arguments

Syntaxe :

```
type <ident> = <ident> of <type>|  ... |<ident> of <type>;;
```

```
type T = C1 of T1| ... | Cn of Tn;;
```

signifie que T est exactement l'ensemble des éléments de la forme

```
Ci x
```

ou x est de type Ti, 1<=i<=n. Les Ci sont supposes tous distincts. Le type T représente l'union disjointe des types.

Exemple :

```
# type num = Int of int | Fl of float;;
type num = Int of int | Fl of float
# Int 3;;
- : num = Int 3
# Fl 3.;;
- : num = Fl 3.
# let add_num = function
    (Int x1 , Int x2) -> Int (x1+x2)
  |  (Fl x1 , Fl  x2)  -> Fl  (x1+.x2)
  |  (Int x1 ,Fl  x2)  -> Fl (float_of_int x1+. x2)
  |  (Fl x1 , Int x2)  -> Fl (x1 +. (float_of_int x2));;
        val add_num : num * num -> num = <fun>
```

Dans la déclaration de type, le type suivant un constructeur peut être omis. L'identificateur correspondant représente alors une constante du type T. Cette définition généralise donc celle du paragraphe précédent.

Exemple. On se propose d'écrire une fonction qui renvoie la plus grande des 2 racines de l'équation :

$$ax^2 + bx + c = 0$$

dans le cas ou elle a 2 racines distinctes.

```
# type extend_float = None | Some of float;;
type extend_float = None | Some of float
# let max_root a b c =
  if a = 0. then None
  else
    let
      delta = b*.b -. 4.*.a *.c
    in
      if delta <= 0. then None
```



```
      else
        if a>0. then
          Some ((-.b +. sqrt delta)/.2.*.a)
        else
          Some ((-.b -. sqrt delta)/.2.*.a);;
                      val max_root : float -> float -> float -> exte
# max_root (-.1.) 2. 3.;;
- : extend_float = Some 3.
# max_root 1. 1. 1.;;
- : extend_float = None
```

Exemple d'utilisation :

```
# let f a b c =
  match max_root a b c with
      (Some x) ->  abs_float x
    |  None    ->  -1.;;
      val f : float -> float -> float -> float = <fun>
# f 1. 1. 1.;;
- : float = -1.
# f (-1.) 2. 3.;;
- : float = 3.
```

La définition du type `extend_float` a donc permis de se passer de `failwith` et de rester en purement fonctionnel.

Autre exemple, extension des entiers avec plus et moins l'infini :

```
# type extend_int = Min | Some of int | Max;;
type extend_int = Min | Some of int | Max
```

Exercice. Définir les fonctions somme et multiplication sur le type `extend_int`.

Nous observons que

```
 type 'a option = None | Some of 'a;;
```

est prédéfini en Ocaml. On peut utiliser ce type pour définir des fonction partielles. Par exemple, la fonction `index`, retournant l'indice d'un élément dans une liste, est définie de la façon suivante :

```
# let index el liste =
    let rec index_acc el acc = function
         [] -> None
       | t::q -> if t = el then Some(acc)
                 else index_acc el (acc + 1) q
    in
    index_acc el 0 liste ;;
val index : 'a -> 'a list -> int option = <fun>
```

1.2.3 Types à un seul constructeur

Dans ce cas le nouveau type est isomorphe au type de base.

Exemple.

```
# type age = Old of int;;
type age = Old of int
# type date = Year of int;;
type date = Year of int
# Old 40;;
- : age = Old 40
# Year 1996;;
- : date = Year 1996
```

Il s'agit de 2 nouveaux types, isomorphes au type int. Ils permettent de faire la différence entre un entier désignant un age et un entier désignant une date.

1.3 Types récursifs

Un type est dit récursif quand dans une définition

```
type T = C1 of T1| ... | Cn of Tn;;
```

`T` apparait dans au moins l'un des `Ti`.

Exemple.

```
# type liste = Nil | Cons of int*liste;;
type liste = Nil | Cons of int * liste
# let rec length = function Nil -> 0
  | (Cons(_,l)) -> length l+1;;
  val length : liste -> int = <fun>
# type int_tree = Leaf of int | Node of int_tree * int_tree;;
type int_tree = Leaf of int | Node of int_tree * int_tree
```

C'est le type des arbres binaires avec noeuds sans étiquette et feuilles étiquetées par des entiers.

```
# Node(Node(Leaf 1, Leaf 2), (Leaf 3));;
- : int_tree = Node (Node (Leaf 1, Leaf 2), Leaf 3)
# let rec nb_node = function (Leaf x)    -> 1
  | (Node(x,y)) -> (nb_node x)+(nb_node y)+1;;
  val nb_node : int_tree -> int = <fun>
# nb_node (Node(Node(Leaf 1, Leaf 2), (Leaf 3)));;
- : int = 5
```


1.3.1 Types polymorphes

```
# type 'a tree = Leaf of 'a | Node of 'a tree *'a tree;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
# type int_tree = int tree;;
type int_tree = int tree
# type 'a liste = Nil | Cons of 'a * 'a liste;;
type 'a liste = Nil | Cons of 'a * 'a liste
```

Dans les listes usuelles, `Cons` est noté de manière infixe par `::`et `Nil` est note `[]`.

```
# type ('a,'b) term =
    Term of 'a * ('a,'b) term list
  | Var of 'b;;
    type ('a, 'b) term = Term of 'a * ('a, 'b) term list | Var of 'b
```

Lorsque le type est polymorphe, devant l'identificateur de type on met la liste des variables de types dont il dépend. Cette liste est parenthésée et les éléments sont séparés par des virgules.

1.4 Retour sur le filtrage

Un motif est une construction syntaxique de la forme :

```
Motifs ::= ident
        | _
        | (motif)
        | constant
        | constucteur motif
        | (motif, ...   ,motif)
        | []
        | (motif :: motif)
        | [motif; ... ;motif]
```

Problème : étant donnes une valeur `v` et un motif `M(x1, ...,   xn)` dont les variables sont `x1, ..., xn` et chacune ne figurent qu'une seule fois dans ce motif, trouver une substitution `s` de `x1, ..., xn` telle que

```
v = M( s(x1), ..., s(xn) );;
```

autrement dit, soit

```
s(x1) = a1
s(x2) = a2
...
s(xn) = an

v = M(a1, ... , an)
```

Exemples :

```
("hello", (3,4))        et (x, y)       | {x = "hello", y = (3,4)}
                                        |
[1;2;3]                 et (a::l)       | {a = 1 ,  l = [2;3]}
                                        |
5                       et n            | {n = 5}
                                        |
()                      et  ()          | {}
                                        |
true                    et  _           | _ est remplace par x et {x = true}
```

Plus généralement, Caml filtre une liste de valeurs avec une liste de motifs ne partageant pas les mêmes variables.

```
(true,false)    et      (x,y)   |       {x = true , y = false}
                                |
(true,false)    et      (_,_)   |       La 1ere occurrence de _ est remplacee
                                |       par une variable "fraiche" (jamais
                                |       utilisee jusque-la), soit  x. De meme
                                |       l'autre est remplacee par y. La
                                |       substitution est alors :
                                |       {x = true, y = false}
                                |
(true,true)      et     (x,x)   |       La liste de motifs ne convient pas car
                                |       la variable x figure dans les 2 motifs
                                |
```

Enfin l'expression :

```
                        match <expr> with
                                p1 -> <expr_1> |
                                   ...
                                pn -> <expr_n>
```

peut être remplacée dans le cas où `n=1` par

```
        let p1 = <expr> in <expr_1>
```

Exemple.

```
# let centre_de_gravite  = fun
    (a1,a2) (b1,b2) (c1,c2) -> ((a1+.b1+.c1)/.3. ,(a2+.b2+.c2)/.3.);;
  val centre_de_gravite :
  float * float -> float * float -> float * float -> float * float = <f
# centre_de_gravite  (1.,1.) (2.,2.) (3.,3.);;
```



```
- : float * float = (2., 2.)
# centre_de_gravite (0.,0.) (1.,1.) (-1., -1.);;
- : float * float = (0., 0.)
# let iso_barycentre a b c a' b' c'=
  match centre_de_gravite  a b c with
      (x, y) -> match centre_de_gravite a' b' c' with
          (x', y') -> ((x+.x')/.2. , (y +. y')/.2.);;
      val iso_barycentre :
  float * float ->
  float * float ->
  float * float ->
  float * float -> float * float -> float * float -> float * float =
# let  iso_barycentre a b c a' b' c'=
  let (x,y)= centre_de_gravite a b c in
  let (x',y') = centre_de_gravite a' b' c' in
    ((x+.x')/.2. , (y +. y')/.2.);;
      val iso_barycentre :
  float * float ->
  float * float ->
  float * float ->
  float * float -> float * float -> float * float -> float * float =
# iso_barycentre (1.,1.) (2.,2.) (3.,3.) (0.,0.) (1.,1.) (-1., -1.);
- : float * float = (1., 1.)
```

1.5 Synonymes

```
# type pair_liste = (int list)*(int list);;
type pair_liste = int list * int list
# type 'a pair_liste = ('a list)*('a list);;
type 'a pair_liste = 'a list * 'a list
```

En aucun cas il ne s'agit d'un nouveau type mais d'une abbreviation du nom du type.

1.6 Exemples

Le type des arbres quelconques dont les noeuds portent des étiquettes d'un type donne `'a` peut être défini de la façon suivante :

```
# type 'a tree = Tr of 'a*'a tree list;;
type 'a tree = Tr of 'a * 'a tree list
```

On peut alors définir un arbre de la façon suivante :

```
# let my_tree = (Tr(7,
                  [Tr (2, [Tr (3, []); Tr (4, []); Tr (5, [])]);
                   Tr (6, [Tr (7, []); Tr (8, [])]);
                   Tr (8, [])]));;
```

```
    val my_tree : int tree =
  Tr (7,
   [Tr (2, [Tr (3, []); Tr (4, []); Tr (5, [])]);
    Tr (6, [Tr (7, []); Tr (8, [])]); Tr (8, [])])
```

Sachant qu'il existe un itérateur `fold_left` récursif terminal prédéfini sur les listes tel que

```
    fold_left f a [b1; ... ;bn] = (f ... (f a b1) ... bn)
```

et en supposant définies les fonctions `max` et `union`, par exemple par :

```
# let max x y = if x>=y then x else y;;
val max : 'a -> 'a -> 'a = <fun>
# let rec ajouter x = function
    [] -> [x]
  | (y::oths) ->
      if x = y then x::oths else
        y::(ajouter x oths);;
        val ajouter : 'a -> 'a list -> 'a list = <fun>
# let rec union ens1 ens2 = match ens1 with
    [] -> ens2
  | (x::autres) -> ajouter x (union autres ens2);;
    val union : 'a list -> 'a list -> 'a list = <fun>
```

la taille et la hauteur d'un arbre quelconque ainsi que la liste des valeurs portées par ses noeuds et l'ensemble de ces valeurs (liste sans répétition) peuvent être définies par :

```
# open List;;
# let rec size   = fun (Tr(r,l)) ->
  1 + fold_left (fun s t -> s + (size t)) 0 l;;
  val size : 'a tree -> int = <fun>
# let rec height = fun (Tr(r,l)) ->
  1+ fold_left (fun h t -> max h (height t)) 0 l;;
  val height : 'a tree -> int = <fun>
# let rec list_of_tree = fun (Tr(r,l)) ->
  r::fold_left(fun l t ->(append l (list_of_tree t))) [] l;;
  val list_of_tree : 'a tree -> 'a list = <fun>
# let rec set_of_tree = fun (Tr(r,l)) ->
  union [r]  (fold_left (fun l t ->  union l (set_of_tree t)) [] l);;
  val set_of_tree : 'a tree -> 'a list = <fun>
```

On obtient les résultats suivants :

```
# size my_tree;;
- : int = 9
# height my_tree;;
- : int = 3
# list_of_tree my_tree;;
```



```
- : int list = [7; 2; 3; 4; 5; 6; 7; 8; 8]
# set_of_tree my_tree;;
- : int list = [8; 5; 3; 4; 2; 6; 7]
```

On s'aperçoit que les 4 fonctions sont de la forme :

```
let rec fonction = fun (Tr(r,l)) ->
   f r (fold_left (fun x t -> g x (fonction t)) a l);;
```

On peut donc définir l'itérateur `fold_tree` sur les arbres de la facon suivante :

```
# let rec fold_tree  f g a =
  fun (Tr(r,l)) ->
    let recur y = fold_tree f g a y in
      f r (fold_left (fun x y -> g x (recur y)) a l);;
      val fold_tree : ('a -> 'b -> 'c) -> ('b -> 'c -> 'b) -> 'b ->
  <fun>
```

Ainsi les 4 fonctions précédentes sont définies par :

```
# let size = fold_tree (fun r s -> 1+s) (fun x y -> x + y) 0;;
val size : '_a tree -> int = <fun>
# let height = fold_tree (fun r h -> 1+h) max 0;;
val height : '_a tree -> int = <fun>
# let list_of_tree = fold_tree (fun r l ->  r::l) append [];;
val list_of_tree : '_a tree -> '_a list = <fun>
# let set_of_tree = fold_tree (fun r l -> union [r] l) union [];;
val set_of_tree : '_a tree -> '_a list = <fun>
# height my_tree;;
- : int = 3
# size my_tree;;
- : int = 9
# list_of_tree my_tree;;
- : int list = [7; 2; 3; 4; 5; 6; 7; 8; 8]
# set_of_tree my_tree;;
- : int list = [8; 5; 3; 4; 2; 6; 7]
#
```

2 Sémantique opérationnelle

Nous avons vu que toute expression Caml correcte a un type et une valeur. Lee mécanisme d'évaluation d'un *terme fermé* a été étudié sur des exemples détaillés. En revanche la notion d'évaluation dans un environnement a été présentée de façon plus informelle et va être précisée dans ce qui suit.

Liaison. C'est un couple `(i, v)` ou `i` est un identificateur et `v` une valeur. On dit que `i` est liée a `v`.

Environnement. C'est une liste de liaisons.

Environnement initial. C'est l'environnement chargé lorsqu'on lance OCaml. Il contient toutes les liaisons correspondant aux objets prédéfinis, c.-à-d. aux objets du module `Pervasives`. Notons le `E_init`.

Définition globale. Elle a pour objet de rajouter un couple à l'environnement.

Exemple. Dans ce qui suit `E` est l'environnement courant après chaque commande.

```
E = E_init
```

```
#let pi = 3.14;;
val pi : float = 3.14
```

```
E = (pi,3.14) :: E_init
```

```
#let r = 2.1;;
val r : float = 2.1
```

```
E = (r,2.1)::(pi,3.14)::E_init
```

```
#let s = pi*.r*.r;;
val s : float = 13.8474
```

L'expression `Pi*.r*.r` contient 2 variables libres, `pi` et `r`. Elle n'est évaluable que si `pi` et `r` sont liées, c'est-a-dire si l'environnement contient des liaisons correspondant à `pi` et `r`, ce qui est le cas.

```
E = (s, 13.8474)::(r, 2.1)::(pi, 3.14)::E_init
```

```
#2. *. s;;
- : float = 27.6948
```

```
E = (s,13.8474)::(r,2.1)::(pi,3.14)::E_init
```

L'expression est évaluée dans `E`, mais `E` n'est pas modifié, car on ne définit aucune nouvelle liaison.

```
#a;;
>a;;
>^
Unbound value a.
```

En effet `a` n'est lié à aucune valeur dans `E`.

```
#let pi = 3.1416;;
```



```
 pi : float = 3.1416
```

```
E = (pi, 3.1416)::(s, 13.8474)::(r, 2.1)::(pi, 3.14)::E_init
```

L'ancienne valeur de `pi` n'est plus accessible.

```
#pi;;
 - : float = 3.1416
```

```
E = (pi, 3.1416)::(s, 13.8474)::(r, 2.1)::(pi, 3.14)::E_init
```

Définitions en parallèle. Elles font toutes leur évaluation dans le même environnement courant, qui est ensuite modifié.

```
#let x = 0;;
 x : int = 0
```

```
E = (x, 0)::(pi, 3.1416)::(s, 13.8474)::(r, 2.1)::(pi, 3.14)::E_init
```

```
#let x = 3 and y = x+1;;
x : int = 3
y : int = 1
```

`y` a été évalué dans l'environnement dans lequel `x` vaut `0`. Puis l'environnement courant devient :

```
E = (y, 1)::(x, 3)::(x, 0)::(pi, 3.1416)::(s, 13.8474)::
    (r, 2.1)::(pi, 3.14)::E_init
```

Définitions locales. Elles modifient momentanément l'environnement. C'est dans cet environnement modifié que se fait l'évaluation de l'expression sur laquelle elle porte.

```
#let x = 10 in x+1;;
- : int = 11
```

L'expression `x+1` est évaluée dans `(x,10)::E` mais `E` n'est pas modifié.

```
#let x=x+1 and y=x+2 in x+y;;
- : int = 9
```

`x` et `y` sont évalués dans le même environnement `E` donc `x` prend momentanément la valeur `4` et `y` prend momentanément la valeur `5` puisque `x` vaut `3` dans `E`.

`x+y` est évalué dans `(y,5)::(x,4)::E` donc `x+y` vaut `9`.

```
#x;;
- : int = 3
#y;;
- : int = 1
```

Ceci montre que `E` n'a pas été affecté par ces définitions locales.

```
#let x = x+1 in let y = x+2 in x+y;;
- : int = 10
```

`x` est évalué dans `E` donc `x` donc prend momentanément la valeur `4`. `y` est évalue dans `(x,4)::E` et donc prend momentanément la valeur `6` puis `x+y` est évalue dans `(y,6)::(x,4)` et donc prend la valeur `10`. Comme précédemment, `E` est inchangé.

2.1 Évaluation des fonctions. Fermetures. Liaison statique

Considérons une définition de fonction non récursive de la forme

```
let f  = fun x -> corps_f
```

Nous avons dit que toute expression de cette forme est déjà réduite et que le résultat de l'évaluation de f est :

```
fun x -> corps_f
```

Ceci est (à peu près) vrai si `corps_f` ne contient pas de variables libres. Pour prendre en compte le cas plus général où cette expression contient des variables libres (précédemment définies) comme par exemple dans cette session :

```
#let v = 2;;
        val v : int = 2

#let f = fun x -> if v > 3*2 then x+3 else x*x*3 + v;;
val f : int -> int = <fun>
```

la notion de « valeur de `f` »doit être un peu affinée.

En fait, la valeur d'une fonction est un couple `<fun x   ->simpl(corps_f), E>` où `E` désigne l'environnement courant au moment où la fonction est définie et `simpl(corps_f)` est l'expression obtenu en simplifiant `corps_f` en vertu de certaines règles propres à l'implémentation du langage dans un but d'optimisation.

Le nouvel environnement est alors :

```
(f, <fun x->simpl(corps_f), E>) :: E
```

et la valeur `<fun x->simpl(corps_f), E>` de la fonction est appelée *fermeture*. En effet, toutes les variables libres apparaissant dans `simpl(corps_f)` doivent être liées dans l'environnement `E` et donc ce couple ne contient plus aucune variable libre, i.e. dont la valeur n'est pas définie. En ce sens, il est « fermé », d'où le nom de fermeture.

Exemple.


```
E = E_init;;

        #let v = 2;;
        val v : int = 2

E = (v,2) :: E_init

        #let f = fun x -> if v > 3*2 then x+3 else x*x*3 + v;;
        val f : int -> int = <fun>

E = (f, <fun x -> x*x*3+v, (v,2) :: E_init>) :: (v,2) :: E_init

        #let v = 100;;
        val v : int = 100

E= (v,100) :: (f, <fun x -> x*x*3+v, (v,2):: E_init>) :: (v,2):: E_init
```

Cette session montre que l'évaluation de `f` est faite de telle sorte que ses variables libres de `corps_f` (ici `v`) seront toujours évaluées dans l'environnement tel qu'il était au moment où la définition de `f` a été donnée. La redéfinition ultérieure de `v` n'affecte en rien la valeur de `f` dans l'environnement. On dit que la liaison est *statique* ou *lexicale*. Si l'évaluation de f était faite avec, pour la variable v, la dernière liaison en date dans l'environnement au moment de cette évaluation, la liaison serait dite *dynamique* ou *fluide* (implantée dans d'anciennes versions de Lisp).

2.2 Évaluation des applications : appel par valeur.

L'évaluation de `(e1 e2)` dans l'environnement `E` se fait de la façon suivante :
– évaluation de `e2` dans `E`, soit `v2` la valeur obtenue ;
– évaluation de `e1` dans `E` : on obtient nécessairement une fermeture de la forme `<fun x -> exp, E`
– évaluation dans `(x, v2) : :E'` de `exp` ;
– `E` ne change pas.
Exemple.

```
E= (v,100) :: (f, <fun x -> x*x*3+v, (v,2):: E_init>) :: (v,2):: E_init

        #f (2*v);;
        - : int =  120002
```

On évalue `2*v` dans `E` : on obtient la valeur `200`.
On évalue `f` dans `E` : on obtient
`<fun x -> x*x*3+v, (v,2) : : E_init>`.
On évalue `x * x * 3 + v` dans `(x,200) : :(v,2) : : E_init` et on obtient `200*200*3 + 2 = 1`

Remarques :

1. Comme on l'a vu dans le chapitre 1, paragraphe 7, on évalue `2*v` *avant* de le substituer à x dans `x*x*3+v`. Il s'agit d'une évaluation par valeur. C'est évidemment plus efficace puisque le calcul `2*v` ne s'effectue qu'une fois au lieu de 2 dans l'appel par nom.
2. L'évaluation par nom aurait consisté à remplacer `x` par `2*v` puis à évaluer `(2*v)*(2*v)+v`. Mais il faut alors noter quelque part que dans cette dernière expression, les 2 premières occurrences de `v` sont à évaluer dans `E` et la dernière dans `E'= (v,2):: E_init`. Pour ce faire il faut étendre la notion de fermeture à des variables quelconques (et pas uniquement fonctionnelles). En fait on évalue `x*x*3+v` dans
`(x, <2*v, E>) :: (v,2):: E_init`.
`x` est alors remplacé par `2*v` que l'on évalue ensuite dans `E` donc `x` est remplace par `200` et `v` est remplace par `2`.

Plus généralement :

2.3 Évaluation des applications : appel par nom.

L'évaluation de `(e1 e2)` dans l'environnement `E` se fait de la façon suivante :
- évaluation de `e1` dans `E` : on obtient nécessairement une fermeture de la forme `<fun x -> exp, E'>` ;
- évaluation dans `(x, <e2, E>)::E'` de `exp` ;
- `E` ne change pas.

2.4 Évaluation des fonctions récursives

Étude d'un exemple

```
let rec fact = fun n -> if n=0 then 1 else n*fact(n-1);;
```

Dans l'environnement `E'`, au moment de sa définition, la valeur de `fact` est une fermeture définie par :

```
@fact = <fun n -> if n=0 then 1 else n*fact(n-1), (fact, @fact)::E'>
```

et donc le nouvel environnement est

```
E = (fact,@fact)::E'
  = (fact, <fun n -> if n=0 then 1 else n*fact(n-1), E>)::E'
```

On remarque que la définition de `@fact` présente une circularité puisque `@fact` est définie en fonction de `@fact` lui-même. De même le nouvel environnement `E` est défini en fonction de `E` lui-même.

Évaluation de `(fact 2)` dans `E` :
- Évaluation de `fact` dans `E` :
  `@fact = <fun n -> if n=0 then 1 else n*fact(n-1), E>` .

- Évaluation de `if n=0 then 1 else n*fact(n-1)` dans `E2 = (n,2)::E`;
- Évaluation de `(fact (n-1))` dans `E2`.
- Évaluation de `n-1` dans `E2` : `1`.
- Évaluation de `fact` dans `E2` : `@fact`.
- Évaluation de `if n=0 then 1 else n*fact(n-1)` dans `E1 = (n,1)::E`. `E1` est empilé sur `E2`.
- Évaluation de `fact (n-1)` dans `E1`.
- Évaluation de `(n-1)` dans `E1` : `0`.
- Évaluation de `fact` dans `E1` : `@fact`.
- Évaluation de `if n=0 then 1 else n*fact(n-1)` dans `E0 = (n,0)::E` `E0` est empilé sur `E1`, puis dépilé : `1`.
- Évaluation de `n` dans `E1` : `1`
- Évaluation de `1*1` : `1`. `E1` est dépilé.
- Évaluation de `n` dans `E2` : `2`.
- Évaluation de `2*1 = 2`.

Dans tout appel récursif sur `v`, `fact` est ainsi évaluée dans l'environnement `Ev = (n,v)::E` où `E` est le second membre de la fermeture de `fact`. Les environnements `Ev` sont empilés lors des différents appels récursifs et dépilés à la fin de l'évaluation du corps de la fonction dans `Ev`.

De façon générale, la fermeture associée à une fonction définie par

```
let rec f = fun x -> corps_f
```

dans un environnement E est :

```
@f = <fun x->corps_f, (f, @f)::E>
```

`(f,@f)` est donc rajouté dans l'environnement dans lequel se feront les évaluations de façon à pouvoir toujours évaluer `f` lors des rappels récursifs.

Expression	Pile des environnements
`fact 2`	`E`
`fact 2`	
`fact 2`	
`@fact = <fun ...,E> 2`	
`if n = 0 then 1 else n*fact(n-1)`	`(n,2)::E` `E`
`n*fact(n-1)`	
`n*fact(1)`	
`n*@fact (1)`	

`n* if n = 0 then 1 else n*fact(n-1)`	`(n,1)::E`
	`(n,2)::E`
	`E`

`n* n*fact(n-1)`	
`n* n*fact (0)`	
`n* n*@fact (0)`	

`n* n* if n = 0 then 1 else fact(n-1)`	`(n,0)::E`
	`(n,1)::E`
	`(n,2)::E`
	`E`

`n* n* 1`	

`n* n * 1`	`(n,1)::E`
	`(n,2)::E`
	`E`

`n* 1 * 1`	
`n* 1`	
`n  * 1`	`(n,2)::E` `E`
`2 * 1`	
`2`	
`2`	`E`