Programmation impérative en Caml

C. Charignon

Il est temps de voir comment on programme de manière impérative (c'est-à-dire avec des boucles et des variables, comme dans le tronc commun avec Python) avec Caml.

Table des matières

Première partie

Cours

1 Introduction

Tout d'abord, il est prouvé que les styles de programmation impératifs et récursifs sont équivalent : tout programme écrit dans un style peut aussi être écrit dans l'autre. Cependant, certain programmes sont plus facilement codés dans un style.

Voici les éléments de programmation utilisés en priorité en programmation récursive :

- Nombreuses fonctions (récursives).
- Les informations sont transmises via des arguments supplémentaires.
- Types persistants.

Et en programmation impérative :

- Boucles.
- Procédures. En Caml, une procédure renvoie `()`, de type `unit` (c'est le `None` de Python).
- Les informations sont conservées dans des « variables ».
- Types modifiables (mutables).

Remarque : J'ai déjà eu l'occasion de le dire mais je le redis : Le mot « variable » a un sens assez flou en informatique et en science de manière générale... Je rechigne presonnellement à l'utiliser pour un objet persistant qui ne peut donc pas varier. Vous avez eu l'occasion de vous en rendre compte : Caml utilise le terme « identifiant » et Python opte pour le terme « nom ».

2 Le type « unit » et la séquence

Le type `unit` de Caml est l'analogue du `NoneType` de Python. Son unique élément est `()` (tout comme l'unique élément dans le type `NoneType` est `None` dans Python). C'est le type renvoyé par toute commande qui ne renvoie rien, mais qui éventuellement a un effet de bord (modifier la mémoire, produire un affichage...), autrement dit une procédure. Par exemple `print_int` est une procédure de type `int -> unit`.

En programmation récursive, on ne l'utilise quasiment pas, ce qui fait que nous ne l'avons pour ainsi dire jamais rencontré jusqu'ici.

Lorsqu'on tape deux expressions dans un programme, elle doivent toujours être reliées par un opérateur. Lorsque les expressions sont de type `int`, cela peut être $+$ ou $*$, avec des listes cela peut être `: :` ou `@` ...

Pour relier deux expressions e_1 et e_2 lorsque e_1 est de type unit (c'est-à-dire e_1 est une *instruction* si on emploie ce vocabulaire), l'opérateur utilisé s'appelle la « séquence ». Il signifie juste : « effectue l'action décrite par e_1 puis évalue e_2 ».

Dans un programme Python, la séquence est en général représentée par un retour à la ligne, autrement dit l'opérateur de séquence est `\n`. En Caml c'est le point-virgule `;`.

Remarque : En Python aussi on peut utiliser le point virgule. Par exemple `x = 1 ; print(x)`. Pour des question de lisibilité, on réserve toutefois le point virgule aux toutes petites expressions.

3 Références

Une référence est la manière la plus simple de créer une variable modifiable. Voici la syntaxe sur un exemple :

```
let ma_variable = ref 0 in (* Je crée une référence à un entier. Type int ref. *)
    ma_variable := 3;  (* Je modifie la variable. C'est une instruction (type unit). Notez
        ↪ bien le :=, ainsi que le ; pour séparer cette instruction de la suite.  *)
    print_int !ma_variable (* J'affiche le « contenu » de la variable. Notez le point
        ↪ d'exclamation pour l'obtenir. *)
;;
```

Pensez qu'une référence est un tableau à une case.

4 Tableaux

Les tableaux fonctionnent comme en Python, à ceci près qu'ils sont de longueur fixe, et qu'ils ne peuvent contenir que des éléments du même type. Il s'agit donc en fait plutôt de tableaux à la numpy.

Voici un petit exemple, voir la fiche de syntaxe donnée en début de semestre pour les autres fonctions utiles.

```
let exemple = [|1;2;3|] ;;
exemple.(0);; (* Accéder à un élément *)
exemple.(0) <- 2;; (* modifier un élément *)
```

5 Boucles

Voici la syntaxe Caml des boucles :

```
for i= 0 to n-1 do
    (* Faire des trucs de type unit... *)
done
```

```
while condition do
    (* Faire des trucs de type unit... *)
done
```

Quelques importants :

- On ne peut couper une boucle « for » au milieu de son exécution. (Pas de return, donc on ne peut pas écrire return au milieu d'une boucle.) Si vous ne savez pas à l'avance combien d'itérations seront nécessaires, utilisez une boucle conditionnelle, ou une fonction récursive.
- Le contenu d'une boucle doit être de type unit, puisqu'il n'est pas possible de renvoyer une valeur au milieu d'une boucle. En particulier, le **done** sera suivi d'un `;`, sauf si c'est la fin du programme ou du bloc logique.

  Soit `f :int -> unit` la *procédure* utilisée dans la boucle. Alors le code :

  ```
  for i= 0 to n-1 do
      f i
  done;
  ```

  équivaut à `f 0 ; f1; …; f (n-1);`

  Et donc la boucle entière est donc un élément de type `unit`.
- Pour l'instant, les seules instructions que nous connaissons sont l'affichage (`print_int` et variantes). C'est assez anecdotique : en général, nous utiliserons des instructions qui permettent de modifier une variable mutable.

6 Copie superficielle (rappel)

Comme en Python, les objets de type mutable ne sont pas copiés par défaut, et peuvent être modifiés par une procédure.

Exemple :

```
let augmente x=
  x:= x+1
;;

let x=ref 2 in
    augmente x;
    x;;
```

Remarque : Le fonction `augmente` existe déjà dans Caml : c'est `incr`.

Exemple 2

```
let x=[|1;2;3|] in
    let y=x in
        y.(1)<- 0;
        x;;
```

:

7 Portée d'une variable

Après un `let maVariable = ... in` , l'identifiant `x` reste défini dans l'expression suivant le **`in`**.

Il y a cependant une subtilité : c'est que la séquence (le point-virgule en Caml) est prioritaire sur le **`in`**. De même que `2+ 3* 2` est automatiquement compris comme `2+ (3*2)`. Par exemple, si `t` est un tableau, la formule **`let`** `x = 2` **`in`** `t.(0)<- x ; x+1` est automatiquement comprise comme :
`let` `x = 2` **`in`** `(t.(0)<- x ; x+1)`.
En revanche dans l'exemple :

```
for i=1 to 12 do
    let x=i in
        blabla
done;
x
```

Le x final n'est pas défini. La définition du x s'arrête au « done ». En Python, le comportement est contraire, ce qui peut être perturbant.

8 Création d'un type enregistrement

Il s'agit de la deuxième méthode pour créer un type en Caml (la première était le type « somme » vu au chapitre sur les arbres). On crée différents champs, et on associe à chacun un type.

```
type contact = { nom: string ; adresse: string ; telephone: int};;
```

Pour créer un élément de ce type, on choisit la valeur de chaque champ :

```
let MX = {nom = "monsieur X"; adresse = "rue Z"; telephone=1234567890};;
```

Pour récupérer la valeur d'un champ, on utilise un point, ce qui rappelle Python : `MX.nom; ;`

A priori les types ainsi créés sont persistants. On peut rajouter «**`mutable`**» au moment de définir un champ.

```
type contact = { nom: string ; mutable adresse: string ; mutable telephone: int};;
```

Ici on considère que le nom n'est pas susceptible de changer : seules l'adresse et le téléphone sont mutables.
Alors on peut modifier un champ par la commande `<-` (même syntaxe que pour un tableau, mais pas pour une référence).

```
MX.telephone <- 0123456789;;
```

Remarque : Le type référence est défini ainsi :

```
type 'a ref = {mutable content : 'a };;
```

9 Création de piles et files mutables

Les files mutables sont disponibles dans le module `Queue`, les piles dans le module `Stack`. Toutefois à titre d'exercice, voyons comment les reprogrammer nous même.

9.1 Piles

On peut juste se baser sur les listes. Par exemple une référence à une liste fournit une pile mutable.

9.2 Files d'attente

Il existe principalement deux manières de programmer des files mutables :

- dans un tableau ;
- à l'aide de deux piles, comme nous avons fait pour les files persistantes.

10 Exemple d'utilisation de piles : expression algébrique postfixée

10.1 Présentation du principe

Les premières calculatrices utilisaient souvent des piles (les HP ont continué très longtemps). En effet, il y a un moyen très pratique de noter des calculs qui ne nécessite pas de parenthèses, et qui pourra être traduit très facilement pour un ordinateur, c'est la notation polonaise inversée. Elle consiste à noter un opérateur après ses arguments. Par exemple $1+2$ sera noté $12+$. On dit que l'opérateur est alors "postfixe" (vous vous souvenez des opérateur "infixe" ou "préfixe" ?).

Plus compliqué : que signifient les calculs suivants ?

- $123\times+$
- $12+3\times$
- $123--+$
- $1234---+\times$

Remarque : Le symbole $-$ peut avoir plusieurs sens : il importe de se mettre d'accord. Voulons-nous d'un opérateur binaire ou unaire ?

Une fois qu'on a convaincu l'utilisateur d'utiliser la notation polonaise inversée, il devient facile [1] de programmer la calculette.

On part d'une pile vide, et on lit les instructions tapées par l'utilisateur dans l'ordre. Quand on lit un nombre, on l'empile. Quand on lit un opérateur unaire (qui prend un seul argument : par exemple une fonction cos sin, etc.. ou le signe $-$) on dépile le nombre au sommet de la pile, on lui applique l'opérateur, et on le rempile. Enfin, si on rencontre un opérateur binaire ($+,\times$ etc...) on dépile les deux nombres au sommet de la pile, on leur applique l'opérateur, on rempile le résultat.

Si l'utilisateur ne s'est pas trompé, à la fin la pile contiendra un seul élément : le résultat voulu.

Notons que ceci ne demande qu'une seule lecture de la suite d'instruction : le temps d'exécution est linéaire (à condition que les opérateurs utilisées s'exécutent tous en temps borné bien sûr).

Programmons ceci. Pour commencer, on définit un type lexème :

```
type 'a lexeme = OpBin of ('a -> 'a-> 'a) (* opérateur binaire *)
               | Nb of 'a (* nombre *)
;;
```

Remarque : Si vous voulez prendre en compte les opérateurs unaire, rajoutez le constructeur `OpUn of ('a -> 'a)`.

10.2 Version persistante

Plaçons-nous en mode récursif.

- Une formule sera une liste de lexèmes. On la parcourra au moyen d'une fonction récursive.
- La pile sera une simple liste.

```

let applique o pile =
  (* Entrée : un opérateur binaire o
              une pile pile (en fait de type list)
     Sortie  : la pile obtenue en remplaçant les deux éléments x,y au sommet de la pile par
         ↪ (o x y)
   *)
  match pile with
  | x::y::en_dessous -> (o x y) :: en_dessous
  | _ -> failwith "erreur syntaxe"
;;


let evalue_rec f =
  (* Entrée : f, formule, de type lexeme list
```

1. tout est relatif

```
15      Sortie : le résultat de l'évaluation de f *)
16
17
18    let rec aux pile a_lire =
19      (* pile : la pile des calculs intermédiaires (en l'occurence, une 'a list
20         a_lire : liste des lexèmes restant à lire *)
21
22      match a_lire with
23      | [] -> (* On a fini la lecture. Normalement, la pile ne contient qu'un élément : le
            ↪ résultat. *)
24         begin
25           match pile with
26           |[res] -> res
27           |_ -> failwith "Erreur syntaxe"
28         end
29
30
31      | Nb n :: suite_f -> (* On lit un nombre : on le met dans la pile *)
32         aux (n::pile) suite_f
33
34
35      | OpBin o :: suite_f -> (* On lit un opérateur binaire : on l'applique aux deux éléments
            ↪ au sommet de la pile. *)
36         aux ( applique o pile  )   suite_f
37    in
38
39    aux [] f
40  ;;
```

10.3 Version impérative

À présent, voyons une version impérative.

- Une formule sera un tableau de lexèmes, qu'on parcourra par une boucle for.
- La pile sera mutable, du module `Stack` de OCaml.

```
1   let evalue_imp formule =
2     (* Entrée : formule, de type lexeme array
3        Sortie : le résultat de cette formule. *)
4     let pile = Stack.create () in
5
6     (* On parcours la formule, de gauche à droite *)
7     for i =0 to Array.length formule -1 do
8       (* On lit formule.(i), qui est de type lexeme.  *)
9       match formule.(i) with
10
11      | OpBin o -> (* On applique o aux deux éléments au sommet de la pile *)
12         let x = Stack.pop pile in
13         let y = Stack.pop pile in
14         Stack.push (o y x) pile
15
16      | Nb n -> Stack.push n pile
17    done;
18
19    let res = Stack.pop pile in
20    if not (Stack.is_empty pile) then
21      failwith "erreur syntaxe"
22    else
23      res
24  ;;
```

11 Exemple : tableaux redimensionnables

Exercice 1. ** Tableaux redimensionnables (à la Python)

Le type « `List` » de Python est un tableau que l'on peut agrandir si besoin. Une telle structure est appelée «tableau redimensionnable», ou «tableau dynamique».

On propose de l'implémenter par le type enregistrement suivant :

```
type 'a tabRedim ={mutable longueur : int; mutable donnees : 'a vect};;
```

Ainsi, nos tableaux redimensionnables seront constitués d'un entier appelé «longueur», et d'un vecteur appelé «donnees». Le principe est que le tableau «donnees» sera en général plus grand que nécessaire : les cases inemployées à la fin pourront servir lorsqu'on voudra agrandir le tableau.

1. Écrire une fonction pour créer un tableau vide.
2. Écrire une fonction `creeTabRedim` prenant en entrée un entier n et un élément x et renvoyant un tableau redimensionnable de n cases contenant des x.

   Le tableau `donnees` créé sera créé deux fois plus grand que nécessaire pour laisser de la place aux agrandissements ultérieurs.
3. Écrire les fonctions de lecture et écriture.
4. Écrire une fonction pour rajouter un élément à la fin d'un tableau. La plupart du temps, il suffira d'incrémenter la longueur, et d'écrire l'élément à rajouter dans la case vide suivante de `donnees`.

   Cependant, si `donnees` est déjà plein, on créera un nouveau tableau, deux fois plus grand, et on y recopiera les données précédentes.
5. M. X décide de créer une variable globale `vide` pour représenter le tableau vide. Il exécute ensuite le code suivant :

```
let vide = { longueur=0, donnees=[||]};;
let t = vide in
ajoute 1 t;
ajoute 2 t;;
let s = vide in
ajoute 3 s;;
```

   Que vaut `s` ? Commenter la pertinence du choix de M. X.
6. Créer une fonction `nouveau_tab` de type `unit -> 'a tabRedim` qui renvoie un nouveau tableau redimensionable. Le tableau `donnees` sous-jacent sera pris de longueur 1.
7. Quelle est la complexité de l'ajout d'un élément à la fin du tableau au pire, et au mieux ?
8. Soit $n \in \mathbb{N}$. On part d'un tableau `t` initialement vide. On rajoute un nombre d'éléments qui a nécessité n agrandissements de `t`. Donner un encadrement du nombre d'éléments rajoutés et la comblexité de ces ajouts. Calculer l'ordre de gradeur du nombre d'opérations par élément ajouté.
   On dit que l'ajout d'un élément dans un tableau a un coût *amorti* en $O(1)$.
   Solution : Après n agrandissement, le tableau sous-jacent a pour longueur 2^n. Et le nombre d'éléments est entre $2^{n-1}+1$ et 2^n.

   Comptons le nombre d'écritures dans le tableau sous-jacent. Pour tout $i \in [\![1, n]\!]$, entre $i-1$-ième agrandissement (non inclu) et le i-ième (inclu) il y a eu 2^{i-1} écritures pour remplir les cases vides, puis 2^i écritures pour tout recopier dans le nouveau tableau de taille 2^i.

   Ainsi le nombre total d'écriture pour les n agrandissements est $\sum_{i=1}^{n}(2^{i-1}+2^i)$, qui vaut $\frac{3}{2}\sum_{i=1}^{n}2^i$ soit $\frac{3}{2}(2^{n+1}-1)$.

   Pour la complexité moyenne par élément rajouté, le pire des cas est celui où on s'arrête juste après un agrandissenent. De sorte que le nombre d'éléments dans `t` est $2^{n-1}+1$ et le nombre d'écritures dans le tableau sous-jacent est $\frac{3}{2}(2^{n+1}-1)+1$. Alors

$$\begin{aligned}\text{coût moyen par insertion } &= \frac{\frac{3}{2}(2^{n+1}-1)+1}{2^{n-1}+1}\\ &\underset{n\to\infty}{\sim} 6\\ &\underset{n\to\infty}{=} O(1)\end{aligned}$$

   Remarque : Dans le meilleur des cas, si on s'arrête juste avant l'agrandissement, on trouve un coût équivalents à 3 écritures par insertion.

Deuxième partie

Exercices

Exercice : programmation impérative en Caml

Exercice 1. ** Jouons avec le type unit

La fonction `List.iter` permet d'appliquer une *procédure* à chaque élément d'une liste.

1. Utilisez-la pour créer une procédure `affiche_liste` qui affiche le contenu d'une liste d'entiers.
2. Reprogrammez-la.
3. Reprogrammez-la en vous basant sur un `List.fold`.

1 Version impérative d'algos déjà vus

Exercice 2. * Parcours d'arbres

Programmer un parcours en largeur puis en profondeur d'un arbre de manière impérative.

Exercice 3. ** Tours de Hanoï

Programmer la résolution du jeu des tours de Hanoï de manière impérative.

Exercice 4. ** Dichotomie

Programmer la recherche dichotomique dans un tableau trié.

2 Type enregistrement

Exercice 5. ** Calculs physiques et homogénéité

On propose de calculer des grandeurs physiques : il s'agira de flottants munis d'une unité. L'unité sera représentée par un tableau de 7 éléments, dans l'ordre suivant `[|m;kg;s;A;K;cd;mol|]`. Par exemple, une accélération est en $m.s^{-2}$: son unité sera représentée par le tableau `[|1 ;0 ;-2 ;0 ;0 ;0 ;0|]`.

Une grandeur physique est alors formée d'un flottant et d'une unité. On utilise un enregistrement :

```
type grandeur = { valeur: float; unite : int array};;
```

Par exemple la constante g pourra être définie par :

```
let g ={ valeur = 9.81; unite = [|1;0;-2;0;0;0;0|] };;
```

1. Écrire des fonctions d'addition, opposé, inverse et produit pour ce type. On renverra des messages d'erreur lorsque les unités sont incompatibles.
2. *Exemple :* Un objet de masse 2 kg et de volume 0.1 m^3 est plongé dans l'eau. Calculer son accélération.
   Écrire un prédicat `flotte` prenant en entrée la masse et le volume d'un objet et qui détermine si il flotte ou pas.
3. Écrire enfin les fonction exponentielle et cosinus.

3 Piles et files

Exercice 6. * Correction de copies

Un professeur corrige une pile de copies exercice par exercice. Ainsi il corrige d'abord le premier exercice de chaque copie de la pile, la reposant ensuite sur une deuxième pile. Une fois la pile finie, il prend la deuxième pile et recommence avec l'exercice 2. Ainsi de suite jusqu'à épuisement des exercices.

1. Écrire l'algorithme en pseudo-code.
2. Programmer cette méthode en Python. On prendra en entrée une pile contenant le nom des élèves, et le nombre d'exercices. Pour simuler la correction d'une copie on écrira un message «Correction de l'exercice k» où k est le numéro de l'exercice en cours.

Exercice 7. * Renverser une pile

Écrire une procédure, prenant une pile et en retournant le contenu avec une complexité linéaire en la longueur de la pile.

Exercice 8. ** Passage à l'écriture normale

Le but est de transformer une expression écrite en notation polonaise inversée en une écriture classique. Par exemple Le principe est simplement de procéder comme lorsqu'on effectue le calcul, sauf qu'au lieu d'effectuer les opérations, on créera la chaîne de caractères correspondante.

Exercice 9. **! Des chiffres et pas de lettres

Le but est de prendre une liste l de lexèmes et un nombre m et de trouver un calcul utilisant ces lexèmes pour obtenir ce nombre. Par exemple pour obtenir 2 à l'aide de +, 1 et 1, on peut faire 1 + 1 (c'est-à-dire 1 1 + en notation postfixée).

Le but est d'obtenir la liste de toutes les possibilités. Chaque possibilité étant elle-même une liste de lexèmes, représentant le calcul en notation polonaise inversée.

La stratégie sera d'essayer toutes les permutations de l et de garder celles qui correspondent à une expression postfixée valide, et dont le résultat est m.

1. Récupérer une fonction `calcule` d'évaluation d'expression postfixée. Il est ici indispensable que cette fonction lève une exception en cas de formule mal construite.
2. Écrire une fonction `essai` prenant en entrée une expression postfixée et un entier et regardant si le calcul de cet expression donne cet entier.

   Si c'est le cas, on renverra la liste contenant uniquement cette pile.

   Si ce n'est pas le cas, ou si l'expression est mal construite (c'est-à-dire si `calcule` lève une exception), on renverra une liste vide.

   On donne la syntaxe pour gérer les exceptions :

```
try
    à renvoyer si tout va bien
with
    |_ -> à renvoyer en cas d'erreur
```

   Remarque : En fait, dans le « with », on peut filtrer selon le type d'erreur. D'où le « `|_ ->` ».
3. Écrire ou récupérer du chapitre sur les arbres une fonction prenant une liste `l` et renvoyant la liste des permutations de `l`. On pourra utiliser deux fonctions mutuellement récursives. (En fait on est en train de parcourir l'arbre des possibilités...) À chaque appel la fonction principale prendra en arguments la liste des éléments de `l` déjà pris et la liste des éléments restant. Quant à la fonction auxiliaire, elle sera chargée de parcourir la liste des éléments restant et de rappeler à chaque fois la fonction principale.
4. Finalement écrire une fonction pour résoudre le problème posé.
5. *bonus* Dans la vraie version du jeu on ne donne qu'une liste d'entier, et on autorise autant d'opérations qu'on veut. Créer une fonction pour résoudre cette version.

Quelques indications

2 Pour un parcours en largeur, utiliser une file mutable du module `Queue`. Pour un parcours en profondeur, une pile mutable du module `Stack`.

3 Utiliser une pile contenant les prochains mouvements à faire. On pourra convenir que cette pile contient des quadruplets, un quadruplet (n, d, a, i) signifiant « déplacer n anneaux de la tiqe d vers la tige a, en utilisant i comme intermédiaire ».

5
1. Écrire une fonction pour additionner deux tableaux termes à termes.
2. La poussée d'Archimède exerce une force « vers le haut » de norme égale au poids d'eau déplacé par l'objet.
3. Les fonctions exponentielle et cosinus doivent prendre des nombres sans unité en argument. Renvoyer une erreur si ce n'est pas le cas.

7 Utiliser une file !

Quelques solutions

1

```
let affiche_int_liste l=
  (* Entrée : l, liste d'entiers
     Sortie : rien
     Effet de bord : affiche les éléments de l *)
  List.iter
    (fun x ->print_int x; print_char ' ')
    l;;

let res = affiche_int_liste [9;5;6;5;4];;

let rec mon_itere p l =
  (* Entrée : p, procédure
              l, liste
     Sortie : rien
     Effet de bord : applique p à chaque élément de l
   *)
  match l with
  |[] -> ()
  |t::q -> p t; mon_itere p q
;;
```

5

```
type grandeur = { valeur : float; unite : int array };;


let plus x y =
  (* Vérifier que x et y ont la même unité *)
  if x.unite <>  y.unite then failwith "Pas homogène"
  else
    (* renvoyer le rés *)
    {valeur = x.valeur +. y.valeur ;
     unite = x.unite }
;;


let sommeTab x y =
    let z = Array.make (Array.length x) 0  in
    for i=0 to Array.length x -1 do
        z.(i) <- x.(i) + y.(i)
    done;
    z;;



let produit x y=
  (* Le produit de deux grandeur est toujours défini. Par contre il va falloir calculer
     ↪ l'unité du résultat.*)
  { valeur = x.valeur *. y.valeur ; unite = sommeTab x.unite y.unite }
;;


let produit2 x y =
  { valeur = x.valeur *. y.valeur ;
    unite = Array.init 7 (fun i-> x.unite.(i) + y.unite.(i))
  }
;;

let g={valeur =9.81; unite =[|1;0;-2;0;0;0;0|] };;
produit2 g g;;


```

```
let oppose x =
(* Renvoie -x *)
  { valeur = -. x.valeur;
    unite = x.unite}
;;

oppose g;;

let inverse x =
  (* Renvoie 1/x *)

  (* Création du tableau des unités *)
  let n_unite = Array.make 7 0 in
  for i =0 to 6 do
    n_unite.(i) <- - x.unite.(i)
  done;

  {valeur = 1. /. x.valeur;
   unite= n_unite}
;;

let inverse x =
  (* Renvoie 1/x *)

  {
    valeur = 1. /. x.valeur;
    unite= Array.map (fun z -> -z) x.unite
  }
;;

(* q 2) *)

let m ={valeur = 2.; unite=[| 0;1;0;0;0;0;0|]}
and v = {valeur = 0.1; unite=[| 3;0;0;0;0;0;0|]}
and rho = {valeur= 1000.; unite=[|-3;1;0;0;0;0;0|]}
and moinsUn= {valeur= -1.; unite=[|0;0;0;0;0;0;0|]}
           ;;

 produit
      g
      (
        plus (produit rho ( produit v (inverse m)) )
             moinsUn

      )
;;
```

8

9