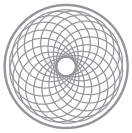


L3 info

Paradigmes de programmation

Eric Rudź

Table des matières

Programmation impérative

A la base, la programmation, c'est impératif.

« Machine » à transformation d'états

Machine de Von Neumann

1. UAL
2. Unité de séquençage
3. La mémoire
4. E/S

Classification

impératif	physique
déclaratif	mathématiques

Déclarative : fonctionnelle, logique, par contraintes, équationnelle, …

- Transformationnel (transforme les données en résultats)
- Interactif
- Réactif

Impératif

À la base : la notion d'état(state)

Écrire un programme impératif, c'est écrire un ensemble d'instructions de changements d'états

Exécuter un programme impératif c'est effectuer une suite de changement d'état

La notion d'environnement d'exécution c'est important

Cet environnement est constitué d'un certain nombre de choses (…) :

I. Séquentielle
   A. Zone mémoire
II. Parallèle
   A. Communication par mémoire partagée: zones mémoires partagées ou non
   B. Communication par message (distribuée):
      1. Zones mémoires
      2. Médium (canaux) de communication

~1960 :

- Fortran
- Lisp(ancêtre des langages fonctionnels mais n'est pas purement fonctionnel)

Paradigmes de programmation

~1970 :

- Algol
- PL1
- Pascal
- C

Autre façon de classer les types de programmation :

- Séquentielle
- Parallèle

A la base de la programmation impérative : l'affectation

Un changement de l'état (de l'environnement) d'exécution d'un programme ne peut provenir que de l'exécution :

- d'une instruction d'affectation (simple ou multiple) ;
- d'une instruction modifiant la valeur du compteur initial.

sémantique :

- Opérationnelle
- Dénotationnelle
- …
- Axiomatique

a,b : booléen
init : a=vrai
b=faux
i : a ← a et b ;
j : b ← non a ;
k : a ← a ou b ;

(i, vrai, faux) ->(j,faux,faux) ->(k, faux, vrai) -> (l, vrai, vrai)

((i, vrai, faux) -> (j, vrai, faux) -> (k, vrai, faux))∞

a, b : booléen

Initialement : a=vrai
b=faux

i : si a et b alors aller à l;
j : a ← a et b;
k : a← non a;
l : aller à m;
m :

Programmation fonctionnelle

Relationnel non fonctionnel : Prolog, solveurs de contraintes

Objetive Caml

- Lustre : langage réactif, fonctionnel flots de données
- Esterel : langage réactif, impératif

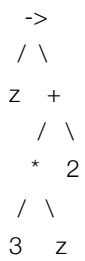
Langages fonctionnels

- Lisp ~1960
- Scheme ~1980
- Ordre supérieur
- Miranda
- Erlang
- Haskell purement fonctionnel

L'idée est de faire comme en maths, sur des choses simples. On utilise l'induction, la récursivité.

Exemple :

Soit la fonction z -> 3z+2 $\mathbb{Z} \to \mathbb{Z}$



en caml cette fonction s'écrit:

```
function z -> 3*z+2
```

Liaison en caml: binding

non lié : unbound

Caml

Phrase caml

- Expression
  - 3+2 par exemple
- Définition globale
- Définition de type
- Définition d'exception

```
# function(a) ->(function x->a*x) ;;
- : int -> int -> int = <fun>
```

jeudi 21 janvier 2010

Exemple

$ax^2+bx+c=0$

Δ>0 $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Δ=0 $\frac{-b}{2a}$

Δ<0 /

```
# let discriminant = function (a,b,c) -> b*.b -. 4. *. a *.c ;;  val discriminant : float * float * float -> float = <fun>
# discriminant(1.,-3.,2.);;
- : float = 1.
```

```
# let solutions(a,b,c)= let delta=discriminant(a,b,c) in ((-.b+.sqrt(delta))/.(2.*.a), (-.b -. sqrt(delta))/.(2.*.a));;
val solutions : float * float * float -> float * float = <fun>
```

```
# (if 3>2 then 5 else 3) * (if 2<5 then 4 else 7);;
- : int = 20
```

La récursivité

Fonction est_pair non récursive:

```
# let est_pair=function x->
if x mod 2=0
        then true
        else false;;
val est_pair : int -> bool = <fun>
# est_pair(2);;
```

Fonction est_pair récursive:

```
# let rec est_pair=function x->
if x=0
        then true
        else if x=1
                then false
                else est_pair(x-2);;
val est_pair : int -> bool = <fun>
# est_pair(3);;
- : bool = false
# est_pair(1000);;
- : bool = true
```

Fonction est_pair et est_impair récursive croisée:

```
# let rec est_pair= function x->
if x=0
        then true
        else est_impair(x-1)
 and
 let rec est_impair= function x->
if x=1
        then true
        else est_pair(x-1);;
val est_impair : int -> bool = <fun>
# est_pair(3);;
- : bool = false
# est_impair(2);;
```

type naturel=Zero | Successeur of naturel

lundi 25 janvier 2010

Récursivité terminale

```
let rec est_pair n= if n+0
        then true
        else not(est_pair(n-1));;
```

Le programme est_pair n'est pas en récursivité terminale.

vendredi 29 janvier 2010

Exos en vrac

définition inductive d'un ensemble:

- base : ensemble comportant au moins un élément
- constructeur(s) inductif(s) : au moins

- constructeurs:
  - constructeur(s) de base
  - constructeur(s) inductif(s)
- déconstructeur(s) inductif(s)
- prédicat(s) reconnaisseur(s) de genre.

```
(* version 1 *)
type naturel = Zero | Successeur of naturel

# let est_nul n = match n with
Zero -> true
  |
  _->false;;
val est_nul : naturel -> bool = <fun>

# let predescesseur n = match n with
  Successeur(m) ->m;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Zero
val predescesseur : naturel -> naturel = <fun>
```

```
# type naturel = Zero | Un | Successeur_de_successeur of naturel;;
type naturel = Zero | Un | Successeur_de_successeur of naturel

# let est_un n = match n with
  Un ->true
  |
  _ -> false;;
val est_un : naturel -> bool = <fun>

let predecesseur_de_predecesseur n = match n with
        Successeur_de_successeur(m)->m;;

let rec est_pair n = if est_nul(n)
        then true
        else if est_un(n)
                then false
                else est_pair(predecesseur_de_predecesseur(n))
```

```
let rec plus(n,m) = if n=0
        then m
        else plus(m,n-1)+1
```

$\forall m \in \bullet : m + 0 = m$

```
let rec  plus (m,n) = if n=0
        then m
        else plus(m+1,n-1)
```

```
(* supegal: int * int -> bool *)
(* les paramètres doivent être >=0 *)
(* supegal (m,n) vaut true quand m>=n, vaut false sinon *)

let rec supegal(m,n) =
if n=0
        then true
        else if m=0
                then false
                else supegal(m-1,n-1);;
```

$\forall m \in \bullet, \forall n \in \bullet \setminus \{0\} : m + n = (m + 1) + (n - 1)$

```
(* fois: int * int -> int *)
let rec fois(m,n)=
if m=0 || n=0
                then 0
                else if n=1
                        then m
                        else fois(n,m-1)+n;;
val fois : int * int -> int = <fun>
# fois(4,5);;
- : int = 20
```

```
(* quotient: int * int -> int *)
let rec quotient(m,n)=
if m<n
        then 0
        else quotient(m-n,n)+1;;

(* reste: int * int -> int *)
let reste(m,n)=
m-(quotient(m,n)*n);;

# quotient(13,4);;
- : int = 3
# reste(13,4);;
- : int = 1
```

```
(* division euclidienne: int * int -> int * int *)
let  rec division_euclidienne(m,n)=
if n<m
        then (0,n)
        else (fst(division_euclidienne(m-n,n))+1,snd(division_euclidienne(m-n,n)));;
```

open List;;

```
#   let rec positifs l = if l=[]
          then []
          else if hd(l)>0
                  then hd(l)::positifs(tl(l))
                       else positifs(tl(l));;
      val positifs : int list -> int list = <fun>

#  let rec positifs_aux (l,l2) = if l=[]
           then l2
          else if hd(l)>0
                  then positifs_aux(tl(l),hd(l)::l2)
                  else positifs_aux(tl(l),l2);;
val positifs_aux : int list * int list -> int list = <fun>

# let positifs l = positifs_aux(l,[]);;
val positifs : int list -> int list = <fun>
```

```
# let rec jusqua i = if i=0
           then []
           else i::jusqua(i-1);;
val jusqua : int -> int list = <fun>


# let rec colle(l1,l2)= if l1=[]
          then l2
           else hd(l1)::colle(tl(l1),    l2);;
val colle : 'a list * 'a list -> 'a list = <fun>
```

lundi 15 février 2010

```
# let rec colle_toutes ll= if ll=[]
          then []
          else colle(hd(ll),colle_toutes(tl(ll)));;
    val colle_toutes : 'a list list -> 'a list = <fun>
```

```
# let rec colle_toutes ll= if ll=[]
        then []
        else if hd(ll)=[]
                then colle_toutes(tl(ll))
                else hd(hd(ll))::colle_toutes(tl(hd(ll))::tl(ll));;
val colle_toutes : 'a list list -> 'a list = <fun>
# colle_toutes([[1;2;3];[4;5;6]]);;
- : int list = [1; 2; 3; 4; 5; 6]
```

vendredi 19 février 2010

```
Rassemble : ‘a list * ‘b list -> (‘a * ‘b) list
Rassemble([1;2;3],[‘a’;’b’;’c’]) = [(1,’a’);(2,’b’);(3,’c’)]

let rec rassemble (l1, l2)=
   if l1=[]
        then []
        else (hd(l1),hd(l2))::rassemble((tl(l1),tl(l2)));;
```

```
Separe   :(‘a * ‘b) list -> ‘a list * ‘b list
let rec separe (l)=
        if l=[]
                then [],[]
                else let (a,b)= hd(l) and (l1,l2)=separe(tl(l)) in (a::l1),(b::l2);;
```

Ordre supérieur

```
# let multiplie_par x y = x*y;;
val multiplie_par : int -> int -> int = <fun>
# multiplie_par 3;;
- : int -> int = <fun>
# multiplie_par 7 5;;
- : int = 35
# let ajoute x y = x+y;;
val ajoute : int -> int -> int = <fun>
# ajoute 7 8;;
- : int = 15
```

```
# let rec somme_list l = if l=[]
          then 0
          else hd(l)+somme_list(tl(l));;
val somme_list : int list -> int = <fun>
# somme_list([1;2;4]);;
```

vendredi 5 mars 2010

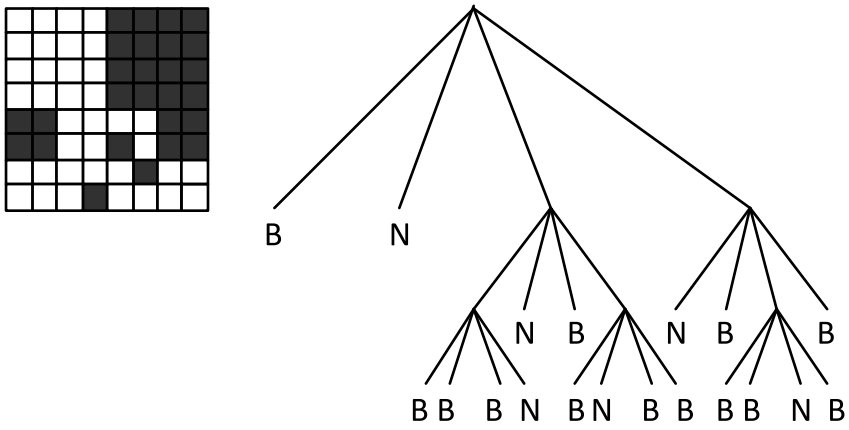
```
# let rec f l=
if l=[]
          then []
          else int_of_float(hd(l))::f(tl(l));;
val f : float list -> int list = <fun>
```

```
# let rec applique_a_tous g= function l -> if l=[]
  then []
  else g(hd(l))::((applique_a_tous(g))(tl(l)));;
val applique_a_tous : ('a -> 'b) -> 'a list -> 'b list = <fun>

# applique_a_tous(sqrt);;
- : float list -> float list = <fun>
# applique_a_tous(int_of_float);;
- : float list -> int list = <fun>
# (applique_a_tous(sqrt))([0.;1.;4.;9.]);;
- : float list = [0.; 1.; 2.; 3.]

# (map(sqrt))([0.;1.;4.;9.]);;
- : float list = [0.; 1.; 2.; 3.]
```

```
# let applique_a_tous3 g=let rec f l= if l=[]
            then []
           else g(hd(l))::f(tl(l))    in f;;
val applique_a_tous3 : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

lundi 8 mars 2010

Arbre



base : arbre_vide : unit->'a arbre

constructeurs

inductif embranche: 'a * 'a arbre * 'a arbre -> 'a arbre

prédicat : est_arbre_vide : 'a arbre -> bool

~~déconstructeur :~~

racine: 'a arbre -> 'a

fonctions d'accès: sag: 'a arbre -> 'a arbre

sad : 'a arbre -> 'a arbre

```
nb_noeuds : ‘a arbre -> int

let rec nb_noeufs a =

if est_arbre_vide(a)

        then 0

else 1+ nb_noeuds(sag(a))+nb_noeuds(sad(a))
```

```
embranche ( 1,
                embranche(4,arbre_vide(),arbre_vide()),
                embranche(7,
                                embranche(8,arbre_vide(),arbre_vide()),
                                arbre_vide()
                                )
                )
```

```
let rec map_arbre f = let rec
                        g a =
                        if est_arbre_vide(a)
                                then arbre_vide()
                                else embranche( f(racine(a)),
                                                g(sag(a)),
                                                g(sad(a))
                                              )
in
        g
```

```
# type 'a arbre=Av|E of 'a*'a arbre*'a arbre;;
type 'a arbre = Av | E of 'a * 'a arbre * 'a arbre
# let arbre_vide () = Av;;
val arbre_vide : unit -> 'a arbre = <fun>
# arbre_vide();;
- : 'a arbre = Av
# let embranche (r,g,d) = E (r, g, d);;
val embranche : 'a * 'a arbre * 'a arbre -> 'a arbre = <fun>
# let est_arbre_vide a = match a with
                                Av -> true
                                |
                                _->false      ;;
val est_arbre_vide : 'a arbre -> bool = <fun>
```

```
# # let racine a= match a with
        E(r, _, _) -> r;;
val racine : 'a arbre -> 'a = <fun>

# let sad a = match a with
        E(_, _, d) ->d;;
val sad : 'a arbre -> 'a arbre = <fun>

# # let sag a = match a with
        E(_, g, _) -> g;;
val sag : 'a arbre -> 'a arbre = <fun>
```

Deux nouveaux langages : Lustre et Esterel

Langages réactifs synchrones.

```
node CG1 (armer, desarmer, date_lim:bool)
retunrs (alarme : bool)
var etat_arme : bool
let
        alarme = date_lim and etat_arme
        etat_arme = false -> if armer then true
                                else if desarmer then false
                                        else pre(etat_arme)
        assert not(armer and desarmer)
tel
```

	0	1	2	3	4	5	6	7	8	9	10	11
armer	F	F	T	F	F	F	T	F	T	F		
désarmer	F	F	F	F	T	F	F	F	F	F		
date_lim	F	T	F	F	F	F	F	T	F	F		
alarme	F	F	F	F	F	F	F	T	F	F		
etat_arme	F	F	T	T	F	F	T	T	T	T		