Aspects impératifs

Introduction de traits “impurs”

- entrées/sorties
- exceptions
- opérations destructives (affectation), qui font référence à la *représentation des données en machine*

Aspects impératifs

Fonctionnel pur

L'évaluation d'une expression

- produit une valeur
- indépendante de la stratégie (si terminaison)

Introduction de traits “impurs”

L'évaluation d'une expression

- produit une valeur *(aspect fonctionnel pur)*
- change l'état de la machine: effet de bord *(aspect impur)*
- La stratégie influe sur l'effet des aspects impératifs

Aspects impératifs
Les exceptions

Le type *exn*

- est prédéfini
- c'est un type somme, avec constructeurs

```
type exn = ...
        | Division_by_zero
        | Failure of string
                    ...
```

```
# Division_by_zero;;
- : exn = Division_by_zero

# Failure "zut";;
- : exn = Failure "zut"
```

Aspects impératifs
Les exceptions

Génération d'une exception

```
#raise;;
- : exn -> 'a = <fun>
```

Aspects impératifs
Les exceptions

Génération d'une exception

```
#raise;;
- : exn -> 'a = <fun>
```

```
# let hd l = match l with
                [ ]      -> raise (Failure "hd")
              |h :: t   -> h ;;
val hd : 'a list -> 'a = <fun>
```

Aspects impératifs
Les exceptions

Génération d'une exception

```
#raise;;
- : exn -> 'a = <fun>
```

```
# let hd l = match l with
                [ ]      -> raise (Failure "hd")
              |h :: t   -> h ;;
val hd : 'a list -> 'a = <fun>
# hd [ ] ;;
Exception: Failure "hd"
```

Aspects impératifs
Les exceptions

Génération d'une exception

```
#raise;;
- : exn -> 'a = <fun>
```

```
# 1/0;;
Exception: Division_by_zero
```

Aspects impératifs
Les exceptions

Génération d'une exception

```
#raise;;
- : exn -> 'a = <fun>
```

- Non définissable dans le langage fonctionnel
- Effet de bord : interrompt tous les calculs en cours
- Déclenche la valeur exceptionnelle reçue en argument
- Polymorphe

Le type 'a du résultat est évalué pour être compatible avec le contexte, quand c'est possible

Aspects impératifs
Les exceptions

Génération d'une exception

```
#raise;;
- : exn -> 'a = <fun>
```

Polymorphisme

```
# 1 + (raise (Failure "zut"));;
Exception: Failure "zut"
# "hello"^ (raise (Failure "zut"));;
Exception: Failure "zut"
# 1 + raise (Failure "zut") ^ "hello";;
This expression has type int but is here used with type string
```

Aspects impératifs
Les exceptions

failwith

L'exception *(Failure s)* où *s* est un message d'erreur est très utilisée.

Il y a donc une fonction prédéfinie

```
# let failwith s = raise (Failure s);;
val failwith : string -> 'a = <fun>
```

Aspects impératifs
Les exceptions

Génération d'une exception

```
#raise;;
- : exn -> 'a = <fun>
```

```
# let hd l = match l with
                [ ]      -> Failwith "hd”
              | h :: t   -> h ;;
val hd : 'a list -> 'a = <fun>
# hd [ ] ;;
Exception: Failure "hd"
```

Aspects impératifs
Les exceptions

Définition de nouvelles exceptions

Extension du type exn

Possibilité de rajouter de nouveaux constructeurs

exception *Nom* of *type*

```
# exception Pred_exception of int;;
exception Pred_exception of int

#let pred n = if n<=0 then (raise (Pred_exception n)) else  n-1;;
val pred : int -> int = <fun>

# pred (-3);;
Exception: Pred_exception (-3).
```

Aspects impératifs
Les exceptions

Récupération d'exceptions

```
try e with    p1 -> expr1
            | p2 -> expr2
                ...
            | pn -> exprn ;;
```

Si l'exception *e* est filtrée par le motif p_i, elle est récupérée :

- remplacée par $expr_i$
- l'exécution se poursuit normalement

Aspects impératifs
Les exceptions

Récupération d'exceptions. *Exemple*.

```
# let f n =  try pred n with Pred_exception 0 -> 0;;
val f : int -> int = <fun>

# f 3;;
- : int = 2

# f 0;;
- : int = 0

# f (-1);;
Exception: Pred_exception (-1).
```

Aspects impératifs
Entrées/sorties

Fonctions de sorties

- Elles sont de type : *type* -> unit
- Elles prennent en argument une valeur *v*
- Elles renvoient () *(l'unique élément de type unit)*
- Elles ont pour effet de bord l'affichage à l'écran de *v*

Leur seul intérêt est l'effet de bord

La valeur retournée n'est là que pour l'expression fonctionnelle

A chaque type de base, sa fonction de sortie

Aspects impératifs
Entrées/sorties

Fonctions de sorties

```
# print_int;;
- : int -> unit = <fun>
# print_string;;
- : string -> unit = <fun>
# print_float;;
- : float -> unit = <fun>
# print_char;;
- : char -> unit = <fun>
```

Aspects impératifs
Entrées/sorties

Sorties: *print_int*

```
# print_int;;
- : int -> unit = <fun>

# print_int 5;;
5- : unit = ()

# print_int (5+6*2) ;;
17- : unit = ()

# print_int (2/0);;
Exception: Division_by_zero
```

Aspects impératifs
Entrées/sorties

Sorties: *print_float*, *print_char*

```
# print_float ((5.1+.8.2)/.3.);;
4.43333333333- : unit = ()

#print_char 'a';;
a- : unit = ()

#print_char '\n';;

- : unit = ()
```

Aspects impératifs
Entrées/sorties

Sorties: *print_string*

```
#print_string "hello";;
hello- : unit = ()
#print_string (string_of_float ((5.1+.8.2)/.3.));;
4.43333333333- : unit = ()
#print_string"\nIl y a dans les bois\nDes arbres fous d'oiseaux\n\n";;
Il y a dans les bois
Des arbres fous d'oiseaux

- : unit = ()
```

Aspects impératifs
Entrées/sorties

Sorties: *print_newline*, *print_endline*

print_newline = fun x ->match x with
( ) -> ... ;;

```
#print_newline;;
- : unit -> unit = <fun>
#print_newline ();;

- : unit = ()
# print_string "\n";;

- : unit = ()
```

Aspects impératifs
Entrées/sorties

Fonctions d'entrées: *read_int*

```
# read_int;;
- : unit -> int = <fun>
# let x = read_int();;
89
val x : int = 89
#  let x = fact (read_int());;
10
val x : int = 3628800
# read_int();;
5 6
Exception: Failure "int_of_string".
```

Aspects impératifs
Entrées/sorties

Fonctions d'entrées: *read_float*

```
# read_float;;
- : unit -> float = <fun>
# read_float();;
34.88
- : float = 34.88
# read_float();;
-34E-1
- : float = -3.4
```

Aspects impératifs
Entrées/sorties

Fonctions d'entrées: *read_line*

```
# read_line;;
- : unit -> string = <fun>

#  let s = read_line();;
Il y a dans les bois
val s : string = "Il y a dans les bois"
```

Aspects impératifs
Séquencement

$(e_1; \ ...; e_n)$

begin $e_1; \ ...; e_n$ *end*

Stratégie d'évaluation: évaluations successives de $e_1, \ ... \ ,e_n$

Type et valeur : ceux de e_n. Si $e_1, \ ... \ ,e_n$ s'évaluent normalement (sans exception)

Valeurs de $e_1, \ ... \ ,e_{n-1}$: oubliées, donc en général, toutes égales à (). Dans le cas contraire : warning

Intérêt de $e_1, \ ... \ ,e_{n-1}$: les effets de bords de leur évaluation.

Aspects impératifs
Séquencement

```
# 2; "Hello";;
Warning S: this expression should have type unit.
- : string = "Hello"

# 2; "Hello";  fact 3;;
Warning S: this expression should have type unit.
Warning S: this expression should have type unit.
- : int = 6
```

Aucun intérêt!

Aspects impératifs
Séquencement

Exemple: résultats d'examen

```
# let admission  f =
 print_string "\nnote d'ecrit: ";
 let e = read_float() in
        (print_string "\nnote d'oral: ";
         let o = read_float() in (f e o));;
val admission : (float -> float -> 'a) -> 'a = <fun>
```

La définition de la fonction admission *ne provoque aucun effet de bord (pas d'affichage à l'écran) car il s'agit d'une abstraction, donc le corps n'est pas évalué.*

Ca n'est plus le cas si on instancie le paramètre f.

Aspects impératifs
Séquencement

Exemple: résultats d'examen

```
# let admission  f =
 print_string "\nnote d'ecrit: ";
 let e = read_float() in
        (print_string "\nnote d'oral: ";
         let o = read_float() in (f e o));;

# let admission_M1 =
                    admission  (fun x y -> (3.*.x +. y)/.4.);;
```

Aspects impératifs
Séquencement

Exemple: résultats d'examen

```
# let admission  f =
 print_string "\nnote d'ecrit: ";
 let e = read_float() in
        (print_string "\nnote d'oral: ";
         let o = read_float() in (f e o));;

# let admission_M1 =
                    admission  (fun x y -> (3.*.x +. y)/.4.);;
```

provoque l'évaluation, donc des affichages

Aspects impératifs
Séquencement

Exemple: résultats d'examen

```
# let admission  f =
 print_string "\nnote d'ecrit: ";
 let e = read_float() in
        (print_string "\nnote d'oral: ";
         let o = read_float() in (f e o));;

# let admission_M1 =
                    admission  (fun x y -> (3.*.x +. y)/.4.);;
```

provoque l'évaluation, donc des affichages

```
note d'ecrit:
```

Aspects impératifs
Séquencement

Exemple: résultats d'examen

```
# let admission  f =
 print_string "\nnote d'ecrit: ";
 let e = read_float() in
        (print_string "\nnote d'oral: ";
         let o = read_float() in (f e o));;

# let admission_M1 =
                    admission  (fun x y -> (3.*.x +. y)/.4.);;
```

provoque l'évaluation, donc des affichages

note d'ecrit: 15.1

note d'oral:

Aspects impératifs
Séquencement

Exemple: résultats d'examen

```
# let admission  f =
 print_string "\nnote d'ecrit: ";
 let e = read_float() in
        (print_string "\nnote d'oral: ";
         let o = read_float() in (f e o));;

# let admission_M1 =
                    admission  (fun x y -> (3.*.x +. y)/.4.);
```

provoque l'évaluation, donc des affichages

```
note d'ecrit: 15.1

note d'oral: 9.5
- : float = 13.7
```

Ca n'est pas le but!

Aspects impératifs
Séquencement

Exemple: résultats d'examen

```
# let admission_M1 =
                admission  (fun x y -> (3.*.x +. y)/.4.);;
```

Provoque l'évaluation, donc des affichages

```
# let admission_M1 ( ) =
                admission  (fun x y -> (3.*.x +. y)/.4.);;
val admission_M1 : unit -> float = <fun>
```

On a retardé l'évaluation . Pour calculer chaque admission:

```
# admission_M1 ( ) ;;
```

note d'ecrit: ...

Aspects impératifs
Séquencement

Exemple: résultats d'examen
Version raffinée

On veut afficher non plus la moyenne obtenue avec la fonction de pondération du M1, mais le résultat *admis* ou *ajourné*

```
# let ponderation  x y =
     let r= (3.*.x+.y)/. 4. in
        print_string (if r<10. then "ajourne" else "admis");
        print_newline( );
        print_newline( )  ;;
val ponderation : float -> float -> unit = <fun>
```

Aspects impératifs
Séquencement

Exemple: résultats d'examen
Version raffinée

```
# let admission_M1 ( ) = admission ponderation;;
val admission_M1 : unit -> unit = <fun>

# admission_M1 ( ) ;;

note d'ecrit: 9.5

note d'oral: 12
admis

- : unit = ()
```

Aspects impératifs
Séquencement

La fonction *List.iter*

```
# List.iter;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

List.iter f [a_1; ... ; a_n] = (f a_1); ... ; (f a_n) ; ( ) ;;

```
# let print_list  l = List.iter print_int l ; print_newline();;
val print_list : int list -> unit = <fun>

# print_list ([4;6;8]@[1;2]);;
46812
- : unit = ()
```

Aspects impératifs
Séquencement

La fonction *List.iter*

```
# List.iter;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

List.iter f [a_1; ... ; a_n] = (f a_1); ...; (f a_n) ; ( ) ;;

```
# let print_list  l = List.iter (fun n -> print_int n; print_string " ") l;
print_newline();;
val print_list : int list -> unit = <fun>

#  print_list ([4;6;8]@[1;2]);;
4 6 8 1 2
- : unit = ()
```

Aspects impératifs
Séquencement

La fonction *List.iter*

Comparer et expliquer

```
#List.iter (fun n -> print_int n; print_string " ") [4;6;8;1;2];;
4 6 8 1 2 - : unit = ()

#List.map (fun n -> print_int n; print_string " ") [4;6;8;1;2];;
4 6 8 1 2 - : unit list = [(); (); (); (); ()]
```

Aspects impératifs
Séquencement

Effets et stratégies d'évaluation

```
# let map_left f  = List.fold_left  (fun l a -> l@[(f a)])  [ ];;
val map_left : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let map_right f l = List.fold_right (fun a ll ->f a::ll) l [ ] ;;
val map_right : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- L'une parcourt la liste [a_1; ... ; a_n] de *gauche à droite,* l'autre de *droite à gauche*
- Résultat: [f a_1; ... ; f a_n] , indépendant de la stratégie d'évaluation

Aspects impératifs
Séquencement

Effets et stratégies d'évaluation

On applique sur la liste une fonction qui présente des aspects impurs

```
# map_left (fun n -> print_int n; print_string " ") [4;6;8;1;2];;
4 6 8 1 2 - : unit list = [(); (); (); (); ()]

# map_right (fun n -> print_int n; print_string " ") [4;6;8;1;2];;
2 1 8 6 4 - : unit list = [(); (); (); (); ()]
```

La stratégie d'évaluation influe sur les effets de bords

Aspects impératifs
Les fichiers

Fichiers en écriture (sorties)

- Leur nom logique est du type *out_channel*
- Ouverts en ecriture comme suit

```
#let c = open_out "my_fic";;
val c : out_channel = <abstr>
```

- Dans le répertoire courant, crée le fichier physique *my_fic* s'il n'existait pas, l'écrase *(attention!)* sinon
- Fermeture comme suit

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

Aspects impératifs
Les fichiers

Fichiers en écriture (sorties)

```
#let c = open_out "my_fic";;
# output_string c "He bonjour,
";;
# output_string c "Monsieur du corbeau
Que vous etes joli";;
# output_string c "\nQue vous me semblez beau!";;
# close_out c;;
```

```
$ more my_fic
He bonjour,
Monsieur du corbeau
Que vous etes joli
Que vous me semblez beau!
$
```

Aspects impératifs
Les fichiers

Fichiers en lecture (entrées)

```
# let c = open_in "my_fic";;
val c : in_channel = <abstr>
# input_char c ;;
- : char = 'H'
# input_line c;;
- : string = "e bonjour,  "
# input_line c;;
- : string = "Monsieur du corbeau"
# input_line c;;
- : string = " Que vous etes joli"
# input_line c;;
- : string = "Que vous me semblez beau!"
# input_line c;;
Exception: End_of_file.
```

Aspects impératifs
Les fichiers

Copie d'un fichier dans un autre

```
let cp input output =

 let c_out = open_out output in
    let c_in = open_in     input in
       let rec copy ( ) =
         try   (output_string c_out (input_line c_in);
                output_char c_out '\n';
                copy( ) )
         with End_of_file -> close_out c_out

       in  copy( );;
val cp : string -> string -> unit = <fun>
```

Aspects impératifs
Les fichiers

Copie d'un fichier dans un autre

```
# cp "my_fic" "copie_my_fic";;
```

```
$ more copie_my_fic
He bonjour,
Monsieur du corbeau
Que vous etes joli
Que vous me semblez beau!
$
```

Aspects impératifs
Structures de données modifiables

Les références

- Ce sont des pointeurs
- La variable référencée est simultanément créée et initialisée à la déclaration de la référence. La valeur initiale en détermine le type.

Déclaration

```
# let p = ref 0;;
val p : int ref = {contents = 0}
```

Valeur de la variable pointée par p : !p

```
# !p;;
- : int = 0
```

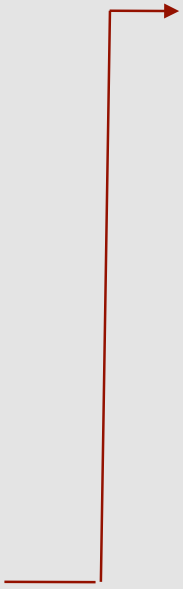
Aspects impératifs
Les références

Affectation

```
#  (:=)  ;;
-  : 'a ref -> 'a -> unit = <fun>
```

```
# p :=  !p+1;;
- : unit = ()
# !p;;
- : int = 1
# let incr ptr = ptr:= !ptr+1;;
val incr : int ref -> unit = <fun>
```

```
# incr p;;
- : unit = ()
# !p;;
- : int = 2
```



Aspects impératifs
Les références

Comparer

```
# let x =1;;
val x : int = 1
```

x *est une expression qui vaut 1*

```
# let f y = x + y;;
val f : int -> int = <fun>
```

f *est la fct successeur*

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

x *est une expression qui vaut 2*

```
# f 2;;
- : int = 3
```

f est inchangée

```
# let x = ref 1;;
val x : int ref = {contents = 1}
```

x *pointe vers 1*

```
# let f y = !x + y;;
val f : int -> int = <fun>
```

f *est la fct successeur*

```
# x := 2;;
- : unit = ()
```

x *pointe vers 2*

```
# f 2;;
- : int = 4
```

f *a changé : f y = y +2*

Aspects impératifs
Les références

 # let f p = incr p; fun x -> x+1;;

val f : int ref -> int -> int = <fun>

C'est une abstraction

 # let f = fun p -> (incr p; fun x -> x+1) ;;

Corps

 # let cpt= ref 0;;

 # let g= f cpt;;

Evaluation de (f cpt)

Valeur : la fonction successeur

Effets de bord: incrémente une variable pointée par cpt

Aspects impératifs
Les références

```
# let f p = incr p; fun x -> x+1;;
val f : int ref -> int -> int = <fun>
```

C'est une abstraction

```
# let f = fun p -> (incr p; fun x -> x+1) ;;
```

Corps

```
# let cpt= ref 0;;
# let g= f cpt;;
```

Effets de bord: incrémente une variable pointée par cpt

Quand cette évaluation se fait -elle?

A quel moment la variable pointée par cpt sera-t-elle incrémentée?

Aspects impératifs
Evaluation retardée

```
# let f p = incr p; fun x -> x+1;;
# let cpt = ref 0;;
#let g y = f cpt y;;
val g : int -> int = <fun>
```

let g = fun y -> f cpt y

ici, (f cpt y) non évaluée car corps de l'abstraction g

valeur de g = fun y -> f cpt y

Aspects impératifs
Evaluation retardée

```
# let f p = incr p; fun x -> x+1;;
# let cpt = ref 0;;
#let g x = f cpt x;;                                   let g = fun x -> f cpt x
val g : int -> int = <fun>
```

ici, (f cpt x) non évaluée car corps de l'abstraction g

```
# !cpt;;
- : int = 0
```

L'évaluation n'a pas eu lieu: cpt inchangé

```
# g 3;;
- : int = 4
```

Evaluation de (f cpt 3): cpt incrémenté

```
# g 10;;
- : int = 11
```

Evaluation de (f cpt 10): cpt incrémenté

```
# !cpt;;
- : int = 2
```

La preuve!

Aspects impératifs
η-réductions

```
# let f p = incr p; fun x -> x+1;;
# let cpt = ref 2;;
#let g  = f cpt ;;
val g : int -> int = <fun>
```

forme η-réduite de # let g x = f cpt x;;

Ici, g = (f cpt) évaluée car c'est une application

cpt est incrémenté, la valeur de g est désormais fun x->x+1

```
# !cpt;;
- : int = 3
```
La preuve!

```
# g 3;;
- : int = 4
```
Evaluation de (fun x -> x+1) 3 : cpt inchangé

```
# g 10;;
- : int = 11
```
Evaluation de (fun x->x+1) 10 : cpt inchangé

```
# !cpt;;
- : int = 3
```
La preuve!

Aspects impératifs et η-réductions: récapitulatif

```
# let f p = incr p; fun x-> x+1;;
val f : int ref -> int -> int = <fun>
# let cpt = ref 0;;
let g x = f cpt x;;
val g : int -> int = <fun>
# !cpt;;
- : int = 0
# g 3;;
- : int = 4
# g 10;;
- : int = 11
# !cpt;;
- : int = 2
```

```
let g = f cpt;;
val g : int -> int = <fun>
#!cpt;;
- : int = 3
# g 3;;
- : int = 4
# g 10;;
- : int = 11
#!cpt;;
- : int = 3
```

Aspects impératifs
Egalités logique et physique

Soient x et y deux variables

Egalité logique

$x = y$

Deux objets x et y de même valeur
Enregistrés (peut-être) à 2 adresses distinctes

Egalité physique

$x == y$

Deux noms distincts x et y du même objet
Même adresse dans la machine

L'égalité physique n'a aucun intérêt fonctionnel/théorique
Relatif à l'implémentation/l'informatique

Aspects impératifs
Egalités logique et physique

```
# let x = 1.1;;
val x : float = 1.1
```

```
# let y = 1.1;;
val y : float = 1.1
```

```
# x = y;;
- : bool = true
```

Même valeur

```
# x == y;;
- : bool = false
```

Adresses différentes

2 objets distincts de même valeur

```
# let z = x;;
val z : float = 1.1
```

```
# z == x;;
- : bool = true
```

Nouveau nom du même objet

let z = x : pas de création d' objet

Aspects impératifs
Egalités logique et physique

Soient p1 et p2 deux *références*

Egalité logique

p1 = p2

Teste l'égalité des valeurs des variables pointées par p1 et p2

Egalité physique

p1 == p2

Teste l'égalité des deux adresses p1 et p2

Aspects impératifs
Egalités logique et physique

```
# let p1= ref 1;;
val p1 : int ref = {contents = 1}

# let p2 = ref 1;;
val p2 : int ref = {contents = 1}

# p1 = p2;;
- : bool = true

# p1== p2;;
- : bool = false

# p1 := !p1 + 15;;
- : unit = ()

# p1;;
- : int ref = {contents = 16}
```

```
# p2;;
-   : int ref = {contents = 1}

# let p3 = p1;;
val p3 : int ref = {contents = 16}

# p3 == p1;;
- : bool = true

# p1 := !p1+3;;
- : unit = ()

# p1;;
- : int ref = {contents = 19}

# p3;;
- : int ref = {contents = 19}
```

Aspects impératifs
Egalités logique et physique

La construction *as*

Pour représenter l'adresse d'un sous-motif d'un motif

L'expression fun ((x, y), z) -> (x, y)
peut etre remplacée par fun ((x, y) as t, z) -> t

t = adresse de (x, y)

Résultat de la fonction : l'objet (x, y) figurant en argument
On n'a pas construit pas une nouvelle paire : gain de mémoire

Aspects impératifs
Egalités logique et physique

```
# let id = fun ((x,y) as t)  -> t;;
val id : 'a * 'b -> 'a * 'b = <fun>
# let x = (1,1);;
val x : int * int = (1, 1)
# x == id x;;
- : bool = true
# let id1 = fun (x,y) -> (x,y);;
val id1 : 'a * 'b -> 'a * 'b = <fun>
# x == id1 x;;
- : bool = false
```

Aspects impératifs
Les tableaux

Type *'a array*

```
# [|0; 1; 2; 3|]; ;
- : int array = [|0; 1; 2; 3|]

# [|'a'; 'b'; 'c'; 'd'|] ;;
- : char array = [|'a'; 'b'; 'c'; 'd'|]

# [|"vos"; "yeux"; "belle"; "marquise"|]; ;
- : string array = [|"vos"; "yeux"; "belle"; "marquise"|]

# [|1; 'a'|] ;;
This expression has type char but is here used with type int

# [||] ;;
- : 'a array = [||]
```

Aspects impératifs
Les tableaux

Accès à un élément

```
# let t = [|1; 2; 3; 4; 5|] ;;
val t : int array = [|1; 2; 3; 4; 5|]
# t.(0);;
- : int = 1
# t.(4);;
- : int = 5
# t.(10);;
Exception: Invalid_argument "index out of bounds".
```

Aspects impératifs
Les tableaux

Les éléments d'un tableau sont modifiables

```
# let t = [|1; 2; 3; 4; 5|] ;;
val t : int array = [|1; 2; 3; 4; 5|]

# t.(0)<- 10;;
- : unit = ( )

# t;;
- : int array = [|10; 2; 3; 4; 5|]
```

Aspects impératifs
Les tableaux

Quelques fonctions de la bibliothèque

```
# Array.length;;
- : 'a array -> int = <fun>

# let t = [|1; 2; 3; 4; 5|] ;;
val t : int array = [|1; 2; 3; 4; 5|]

# Array.length t;;
- : int = 5

# Array.map;;
- : ('a -> 'b) -> 'a array -> 'b array = <fun>

# Array.map (fun x -> x * x) t; ;
- : int array = [|1; 4; 9; 16; 25|]
```

Aspects impératifs
Les tableaux

Quelques fonctions de la bibliothèque

```
# Array.iter;;
- : ('a -> unit) -> 'a array -> unit = <fun>
# Array.iter (fun n -> (print_int n ; print_char ' ')) t;print_newline();;
1 2 3 4 5
- : unit  = ()
# Array.to_list t;;
- : int list = [1; 2; 3; 4; 5]
# Array.of_list  [1; 2; 3; 4; 5] ;;
- : int array = [|1; 2; 3; 4; 5|]
```

Aspects impératifs
Les tableaux

Créer un tableau

```
# Array.create;;
- : int -> 'a -> 'a array = <fun>

# let a = Array.create  8 2;;
val a : int array = [|2; 2; 2; 2; 2; 2; 2; 2|]

# Array.init;;
- : int -> (int -> 'a) -> 'a array = <fun>

# let a = Array.init 6  (fun n -> n mod 2 = 0) ;;
val a : bool array = [|true; false; true; false; true; false; true |]
```

Aspects impératifs
Les tableaux

Recherche dichotomique

```
# let dichotomie comp e t =
   let rec dicho d f =
       if f<d then false
       else
              let m=(d+f)/2 in
                     if e = t.(m) then true
                     else
                            if (comp e t.(m)) then (dicho  d (m-1))
                            else (dicho  (m+1) f)
   in dicho 0 (Array.length t -1) ;;

val dichotomie : ('a -> 'a -> bool) -> 'a -> 'a array -> bool = <fun>
```

Aspects impératifs
Les tableaux

Recherche dichotomique

```
# dichotomie (<=) "mes" [|"belle"; "d'amour"; "font"; "marquise";
    "me"; "mourir"; "vos"; "yeux"|];;

- : bool = false

# dichotomie (<=)  "yeux" [|"belle"; "d'amour"; "font"; "marquise";
    "me"; "mourir"; "vos"; "yeux"|];;

 - : bool = true
```

Aspects impératifs
Les tableaux

Echange de 2 éléments

```
# let swap t i j = let l= Array.length t -1 in
                   if i>l or j>l then failwith "swap"
                  else    let x =t.(i) in
                          (t.(i) <- t.(j); t.(j) <- x) ;;
val swap : 'a array -> int -> int -> unit = <fun>
# let t =  [|1; 2; 3; 4; 5; 6; 7; 8|] ;;
val t : int array = [|1; 2; 3; 4; 5; 6; 7; 8|]
# swap t 0 5;;
- : unit = ()
# t;;
- : int array = [|6; 2; 3; 4; 5; 1; 7; 8|]
```

Aspects impératifs
Les tableaux

Créer un tableau à deux dimensions

```
# Array.create_matrix;;
-   : int -> int -> 'a -> 'a array array = <fun>

# let a = Array.create_matrix 7 7 100;;
val a : int array array =
  [|[|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|]|]
```

Aspects impératifs
Les tableaux

Créer un tableau à deux dimensions

```
# a.(5).(5)<- 0;;
- : unit = ()
```

Aspects impératifs
Les tableaux

Créer un tableau à deux dimensions

```
# a.(5).(5)<- 0;;
-   : unit = ()
# a;;
- : int array array =
  [|[|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 0; 100|];
    [|100; 100; 100; 100; 100; 100; 100|]|]
```

Aspects impératifs
Les tableaux

Créer un tableau à deux dimensions

```
# let a = Array.create 7 (Array.create 7 100);;
val a : int array array =
  [|[|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|]|]
```

Aspects impératifs
Les tableaux

Créer un tableau à deux dimensions

```
# a.(5).(5)<-0;;
- : unit = ()
```

Aspects impératifs
Les tableaux

Créer un tableau à deux dimensions

```
# a.(5).(5)<-0;;
- : unit = ()
# a;;
- : int array array =
 [|[|100; 100; 100; 100; 100; 0; 100|];
  [|100; 100; 100; 100; 100; 0; 100|];
  [|100; 100; 100; 100; 100; 0; 100|];
  [|100; 100; 100; 100; 100; 0; 100|];
  [|100; 100; 100; 100; 100; 0; 100|];
  [|100; 100; 100; 100; 100; 0; 100|];
  [|100; 100; 100; 100; 100; 0; 100|]|
```

Pourquoi?

Aspects impératifs
Les tableaux

Réponse !

```
# let a = Array.create 7 (Array.create 7 100);;
# a.(2) == a.(5);;
- : bool = true
#  a.(2).(5) == a.(5).(5);;
- : bool = true
```

Tableau *a* :

- longueur 8
- les cases contiennent toutes l'adresse d'une même ligne
  - de longueur 8
  - dont chaque case contient la valeur 100.

Une seule ligne, pointée par chaque case de a

Aspects impératifs
Les tableaux

Réponse !

```
# let a = Array.create 7 (Array.create 7 100);;
```

A proscrire (en général) pour implanter une matrice!

Aspects impératifs
Les tableaux

Autre possibilité

```
# let a = Array.map (fun _ -> (Array.create 7 100))
                    (Array.create 7 ());;
val a : int array array =
  [|[|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
    [|100; 100; 100; 100; 100; 100; 100|];
   [|100; 100; 100; 100; 100; 100; 100|]|]

# a.(2) == a.(5);;
- : bool = false
```

7 créations

Aspects impératifs
Les enregistrements (records)

Enregistrements simples

Déclaration du type

$$\text{type nom} = \{\text{nom}_1 : t_1; \ldots ; \text{nom}_n : t_n \}$$

```
# type date = {mois : int ; annee : int};;
type date = { mois : int; annee : int;}
```

Création d'une valeur

$$\{\text{nom}_1 = \text{expr}_1; \ldots ; \text{nom}_n = \text{expr}_n \}$$

```
# let m={mois = 1; annee = 1985} ;;
val m : date = {mois = 1; annee = 1985}
```

Aspects impératifs
Les enregistrements (records)

Enregistrements simples

Accès à la valeur d'un champ

expr.nom

```
# m.mois;;
- : int = 1
# type individu= {nom : string; arrivee : date};;
type individu = { nom : string; arrivee : date; }
# let toto = {nom = "toto" ; arrivee = {mois=1; annee=1985}};;
val toto:individu={nom= "toto"; arrivee={mois=1; annee=1985}}
# toto.arrivee.annee;;
- : int = 1985
```

Aspects impératifs
Les enregistrements (records)

Accès aux champs par la notation *expr.nom*

```
# let bissextile a = a.annee mod 4 = 0 ;;
val bissextile : date -> bool = <fun>
```

Accès aux champs par filtrage

```
#let bissextile a = match a with
                    {mois =_; annee = x} -> x mod 4 = 0 ;;
val bissextile : date -> bool = <fun>

# bissextile {mois = 1; annee=1952};;
- : bool = true

# bissextile {mois = 1; annee = 1985};;
- : bool = false
```

Aspects impératifs
Les enregistrements (records)

Enregistrement à champs mutables

Les champs peuvent être des références:

```
# type coord = {abs : float ref ; ord : float ref};;
type coord = { abs : float ref; ord : float ref; }
```

Les champs mutables permettent une syntaxe simplifiée

```
# type coord = {mutable abs : float; mutable ord : float};;
type coord = { mutable abs : float; mutable ord : float; }

# let translation (a,b)  pt =
        pt.abs <- pt.abs+.a ;  pt.ord <- pt.ord+.b;;
val translation : float * float -> coord -> unit = <fun>
```

Aspects impératifs
Les enregistrements (records)

Enregistrement à champs mutables

```
# let translation (a,b) pt =
pt.abs <- pt.abs+.a; pt.ord <- pt.ord+.b;;

# let point={abs = 0. ; ord = 3.};;
val point : coord = {abs = 0.; ord = 3.}

# translation (1.,1.) point;;
- : unit = ()

# point;;
- : coord = {abs = 1.; ord = 4.}
```

Aspects impératifs
Les enregistrements (records)

Enregistrement à champs mutables

```
# let p1 = {abs = 2.; ord = 2.};;
val p1 : coord = {abs = 2.; ord = 2.}
# let p2 = {abs = 2.; ord = 2.};;
val p2 : coord = {abs = 2.; ord = 2.}
# let p3 = p1;;
val p3 : coord = {abs = 2.; ord = 2.}

# translation (1.,1.) p1;;
-   : unit = ()

# p2;;
- : coord = {abs = 2.; ord = 2.}

#p1;;
-: coord = {abs = 3.; ord = 3.}

#p3;;
- : coord = {abs = 3.; ord = 3.}
```

Aspects impératifs
Exemple: Les listes circulaires

Liste chaînée: suite de cellules

Cellule

1 champ portant une information

1 champ portant l'adresse de la cellule suivante, donc c'est un champ mutable.

```
# type 'a cellule={ info : 'a;  mutable suiv : 'a cellule};;
 type 'a cellule = { info : 'a; mutable suiv : 'a cellule; }
```

Liste circulaire: la dernière cellule contient l'adresse de la première

Aspects impératifs
Exemple: Les listes circulaires

```
#let rec x = {info = “bonjour” ; suiv = x};;
val x : string cellule =
 {info = “bonjour”;
  suiv =
   {info =“bonjour”;
    suiv =
     {info = “bonjour”;
      suiv =
      ....}}}}}}}}}}}}}}}
```

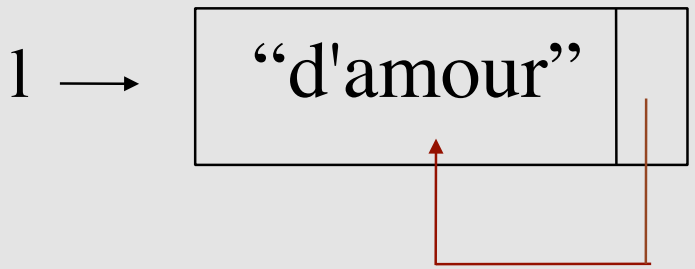
Aspects impératifs
Exemple: Les listes circulaires

```
#let make_list e = let rec x = {info = e ; suiv = x} in x;;
val make_list : 'a -> 'a cellule = <fun>
```

```
#let l = make_list "d'amour";;
val l : string cellule =
 {info = "d'amour";
  suiv =
   {info = "d'amour";
    suiv =
     {info = "d'amour";
          ....}}}}}}}}}}}}}}
```

Aspects impératifs
Exemple: Les listes circulaires

```
#let l = make_list "d'amour";;
l : string cellule =
 {info = "d'amour";
  suiv =
   {info = "d'amour";
    suiv =
     {info = "d'amour";
          ....}}}}}}}}}}}}}}
```

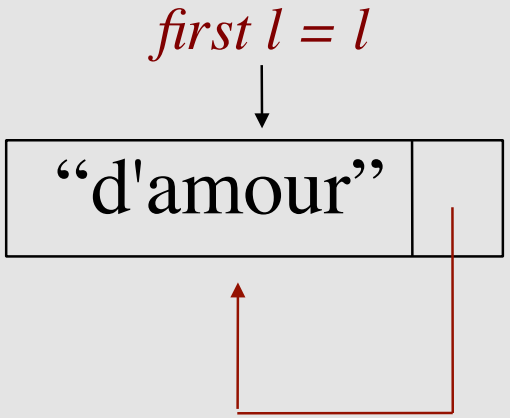


Aspects impératifs
Exemple: Les listes circulaires

Insertion en tête

On repère la liste par sa *dernière* cellule pour insérer juste après
Le dernier élément de la liste *l* est pointé par *l*
Le 1er élément est pointé par l.suiv.

```
# let first l = l.suiv;;
val first : 'a cellule -> 'a cellule = <fun>
```



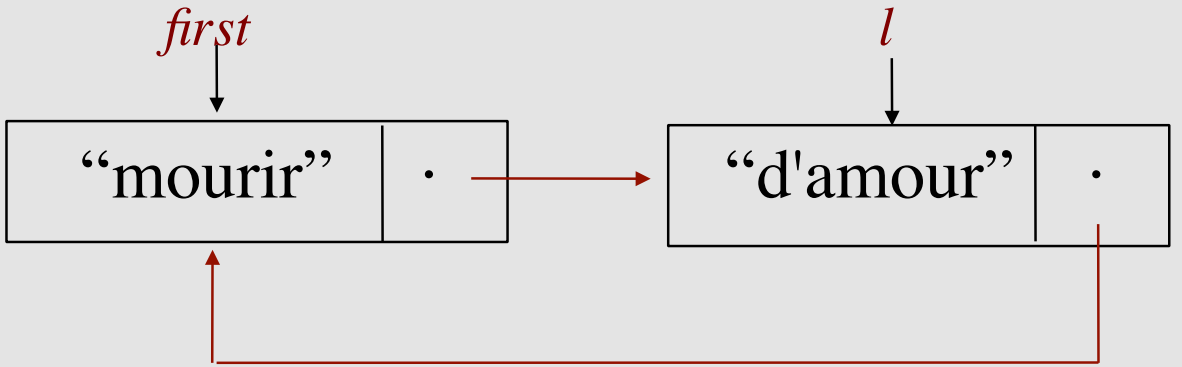
Aspects impératifs
Exemple: Les listes circulaires

Insertion en tête

```
# let insert  l e  =  l.suiv <- { info = e ; suiv = l.suiv } ;  l ;;
val insert : 'a cellule -> 'a -> 'a cellule = <fun>
```

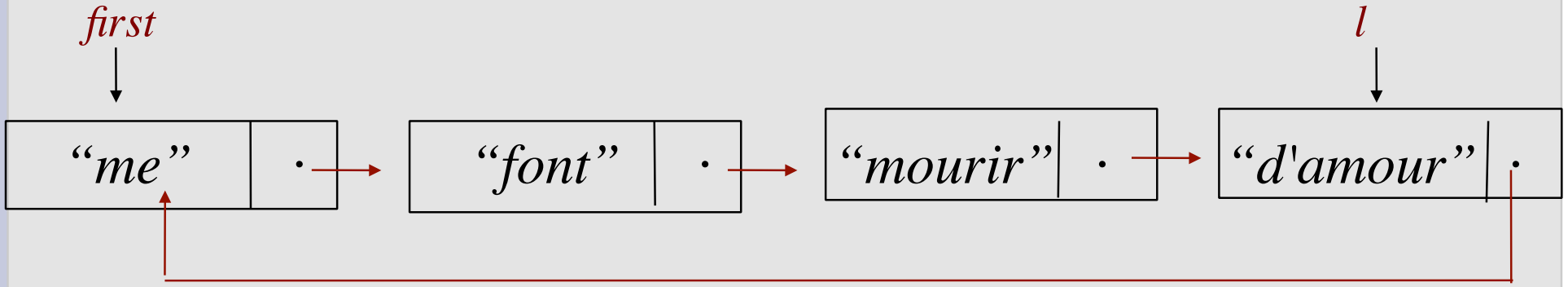
```
#insert l "mourir";;
val l : string cellule = {info = "d'amour";
                          suiv =
                                 {info = "mourir";
                                  suiv = ....}}}}}}}}}}}}}}
```

Pour que s'affiche le résultat

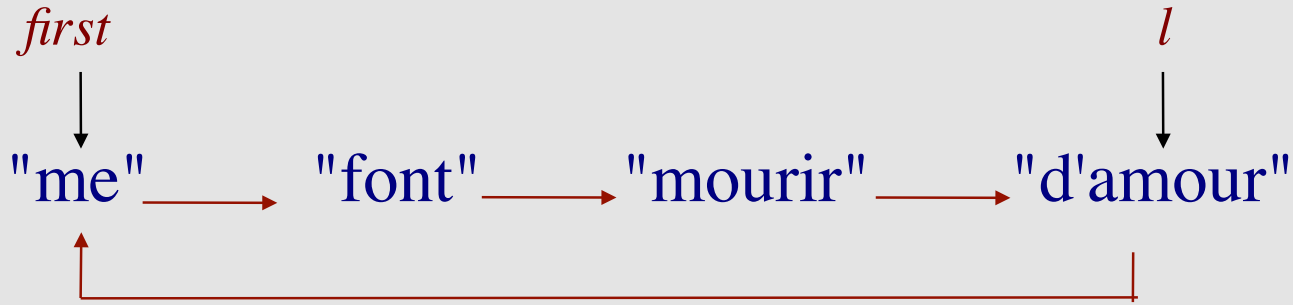


Aspects impératifs
Exemple: Les listes circulaires

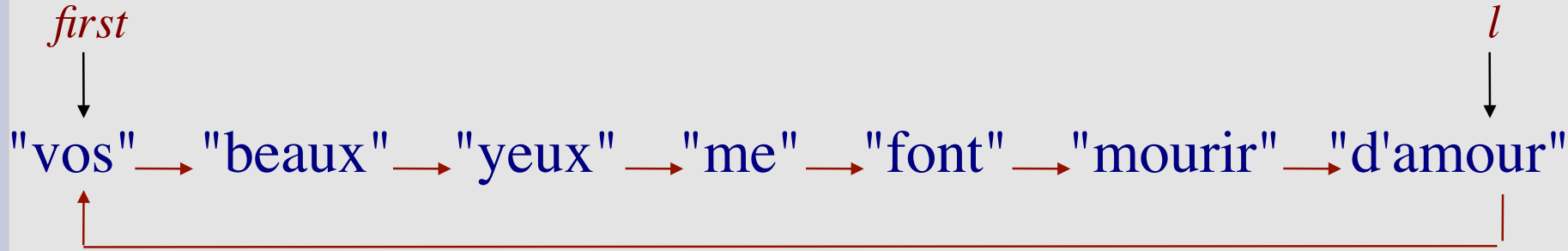
```
# insert l "font" ; insert l "me" ;;
l : string cellule = {info = "d'amour";
                      suiv =
                      {info = "me";
                       suiv =
                       {info = "font";
                        suiv =
                        {info = "mourir";
                         suiv =    ....}}}}
```



Aspects impératifs
Exemple: Les listes circulaires



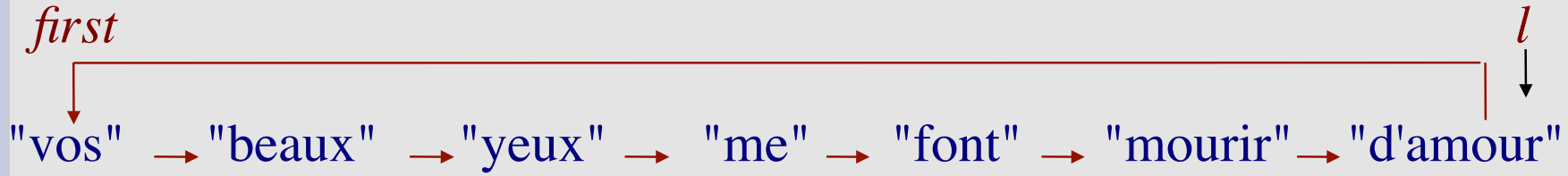

```
# insert l "yeux" ; insert l "beaux" ; insert l "vos" ;;
```




```
#(first l).info;;
- : string = "vos"
#l.info;;
- : string = "d'amour"
```

Aspects impératifs
Exemple: Les listes circulaires

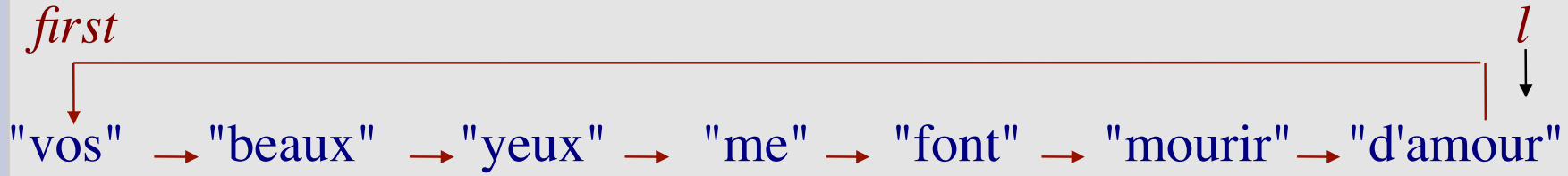
Insertion en queue




```
#let insert_tail l e =
```

Aspects impératifs
Exemple: Les listes circulaires

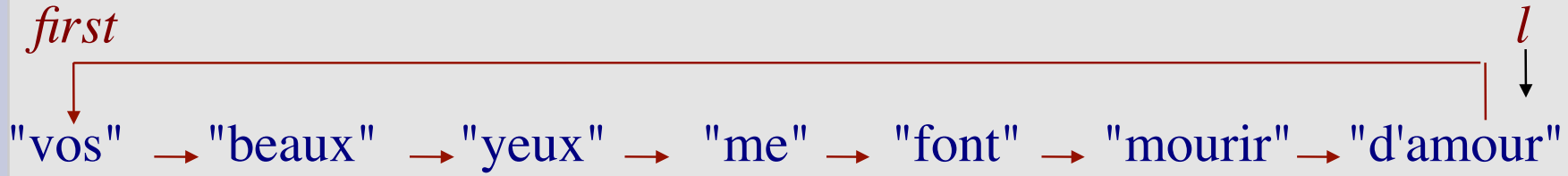
Insertion en queue




```
#let insert_tail l e = (insert l e).suiv;;
val insert_tail : 'a cellule -> 'a -> 'a cellule = <fun>
```

Aspects impératifs
Exemple: Les listes circulaires

Insertion en queue

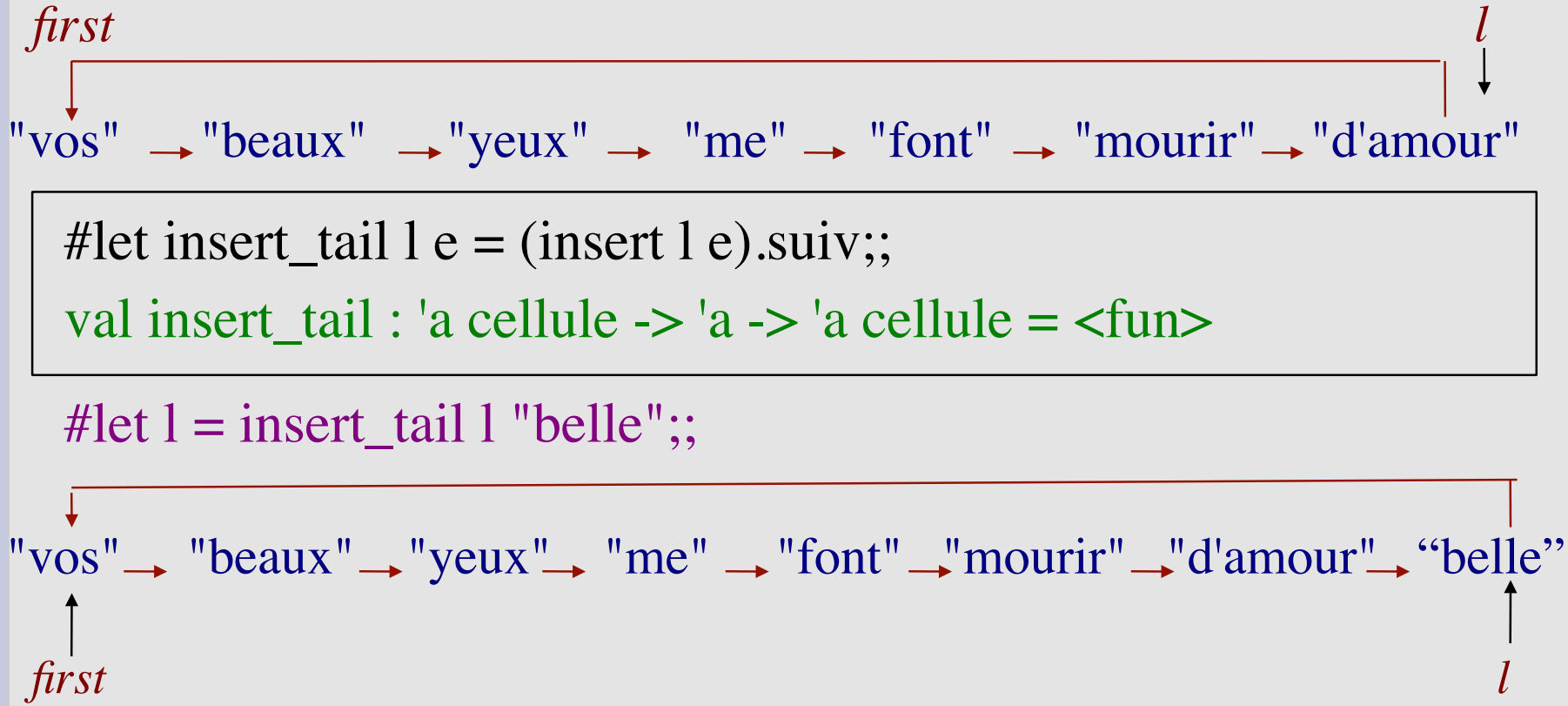



```
#let insert_tail l e = (insert l e).suiv;;
val insert_tail : 'a cellule -> 'a -> 'a cellule = <fun>
```

```
#let l = insert_tail l "belle";;
```

Aspects impératifs
Exemple: Les listes circulaires

Insertion en queue



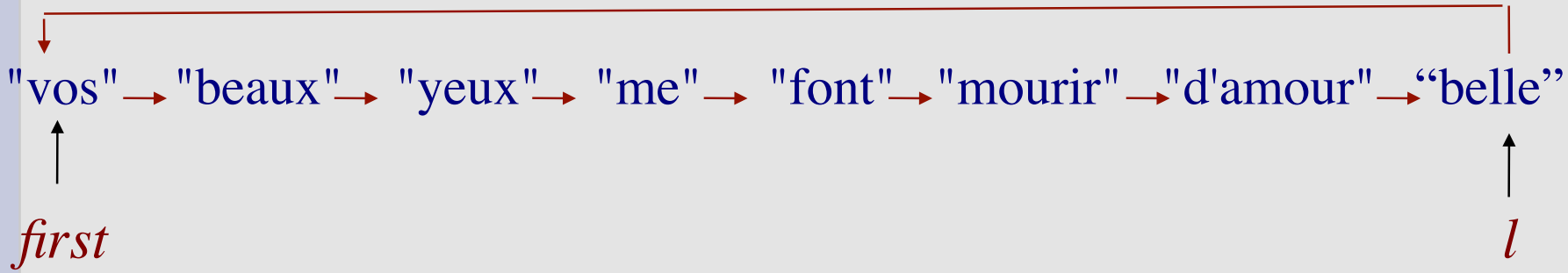

```
#let insert_tail l e = (insert l e).suiv;;
val insert_tail : 'a cellule -> 'a -> 'a cellule = <fun>
```

```
#let l = insert_tail l "belle";;
```

Aspects impératifs
Exemple: Les listes circulaires

Insertion en queue

```
#let insert_tail l e = (insert l e).suiv;;
val insert_tail : 'a cellule -> 'a -> 'a cellule = <fun>
```

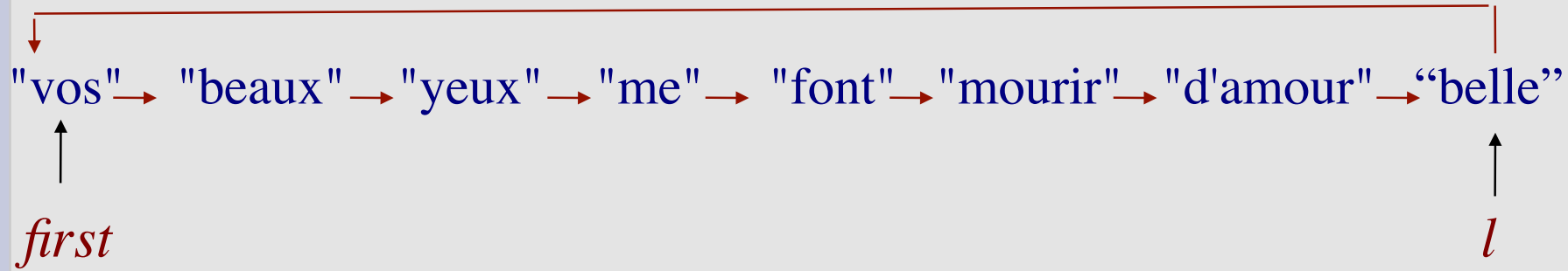



```
#let l = insert_tail l ”marquise";;
```

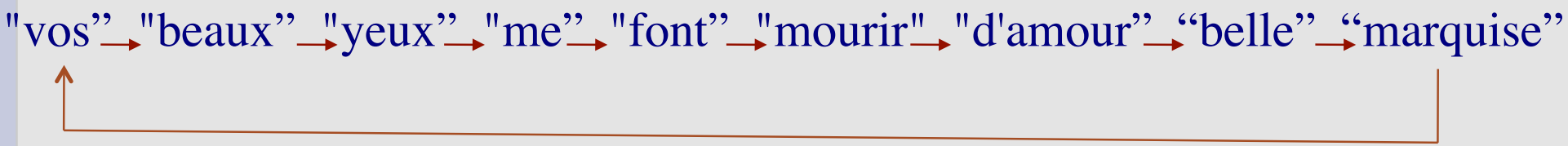
Aspects impératifs
Exemple: Les listes circulaires

Insertion en queue

```
#let insert_tail l e = (insert l e).suiv;;
val insert_tail : 'a cellule -> 'a -> 'a cellule = <fun>
```




```
#let l = insert_tail l ”marquise";;
```



Aspects impératifs
Exemple: Les listes circulaires

Suppression de la tête

```
# let elim_head l = l.suiv <- l.suiv.suiv ; l ;;
val elim_head : 'a cellule -> 'a cellule = <fun>
```

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste en liste circulaire

Reconstituer cette liste chainée à partir de la liste :

["vos"; "beaux"; "yeux" ; "me" ; "font"; "mourir"; "d'amour"; "belle"; "marquise"]

On crée une liste chaînée *r* de premier élément *"vos"*. Puis on parcourt de gauche à droite la liste restante en insérant en queue l'élément courant dans r.

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste en liste circulaire

Reconstituer cette liste chainée à partir de la liste :

["vos"; "beaux"; "yeux" ; "me" ; "font"; "mourir"; "d'amour"; "belle"; "marquise"]

On crée une liste chaînée *r* de premier élément *"vos"*. Puis on parcourt de gauche à droite la liste restante en insérant en queue l'élément courant dans r.

```
# let circ_list_of_list  l  =  match l with
                                  [ ]      -> failwith “circ_list_of_list”
                              | hd :: tl -> let r = make_list hd  in
```

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste en liste circulaire

Reconstituer cette liste chainée à partir de la liste :

["vos"; "beaux"; "yeux" ; "me" ; "font"; "mourir"; "d'amour"; "belle";
"marquise"]

On crée une liste chaînée *r* de premier élément *"vos"*. Puis on parcourt de gauche à droite la liste restante en insèrant en queue l'élément courant dans r.

```
# let circ_list_of_list  l  =  match l with
                                  [ ]      -> failwith “circ_list_of_list”
                              l hd :: tl -> let r = make_list hd  in
                                                List.fold_left insert_tail  r tl ;;
val circ_list_of_list : 'a list -> 'a cellule = <fun>
```

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste en liste circulaire

```
# circ_list_of_list  ["vos"; "beaux"; "yeux" ; "me" ; "font"; "mourir"] ;;
- : string cellule = {info = "mourir";
                      suiv = {info = "vos";
                              suiv = {info = "beaux";
                                      suiv ={info = "yeux";
                                             suiv = {info = "me";
                                                     suiv = {info = "font";
                                                             suiv =
                                                                ...}}}}}}
```

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste en liste circulaire

```
#elim_head (circ_list_of_list ["vos"; "beaux"; "yeux" ;"me" ;"font";
    "mourir"]);;
- : string cellule ={info = "mourir";
                     suiv = {info = "beaux";
                             suiv = {info = "yeux";
                                     suiv = {info = "me";
                                             suiv = {info = "font";
                                                     suiv = {info = "mourir";
                                                             suiv =
                                                                 ...}}}}}}
```

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste circulaire en liste

```
# let list_of_circ_list  l  =


                                (first l).info  ::  list_of_circ_list    l.suiv;;
```

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste circulaire en liste

```
# let list_of_circ_list  l  =
```

```
(first l).info  ::  list_of_circ_list    l.suiv;;
```

Comment s'arrête-t-on?

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste circulaire en liste

```
# let list_of_circ_list  l  =
    let rec  parcours  p =




    in
          (first l).info  ::  parcours  l.suiv;;
```

parcours *doit parcourir l à partir du suivant*

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste circulaire en liste

```
# let list_of_circ_list  l  =
    let rec  parcours  p =


                    (first p).info :: parcours p.suiv
    in
          (first l).info  ::  parcours  l.suiv;;
```

Comment parcours *s'arrête-t-il ?*

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste circulaire en liste

```
# let list_of_circ_list  l  =
    let rec  parcours  p =
                if p == l  then
                   [ ]
                else
                   (first p).info :: parcours p.suiv
    in
           (first l).info  ::  parcours  l.suiv;;
```

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste circulaire en liste

```
# let list_of_circ_list  l  =
    let rec  parcours  p =
                if p == l  then
                   [ ]
                else
                   (first p).info :: parcours p.suiv
    in
           (first l).info  ::  parcours  l.suiv;;
```

```
val list_of_circ_list : 'a cellule -> 'a list = <fun>
```

Aspects impératifs
Exemple: Les listes circulaires

Transformation d'une liste circulaire en liste

```
#  list_of_circ_list (circ_list_of_list
       ["vos"; "beaux"; "yeux" ;"me" ;"font"; "mourir"]);;

- : string list = ["vos"; "beaux"; "yeux"; "me"; "font"; "mourir"]
```