Itérateurs

INF 441 : Programmation Avancée

Xavier Rival

15 mai 2017

Quelques éléments administratifs

Tutorat :

mardi 16 mai, 18h30-20h, salle 30

sous réserve de disponibilité des enseigants

mardi 30 mai, 17h45-19h15, salle 30, à confirmer

- syntaxe OCaml, utilisation des outils
- apporter votre machine si nécessaire
  (installation OCaml sur votre environnement personnel)
- merci de me dire si vous êtes intéressés

Café-DIX : réunion d'information sur la 3A et les PA d'info

prévu mercredi 17 mai et mercredi 24 mai, de 13h00 à 14h00

Itération sur une structure

Cours précédents : structure des données, et structure du code

- **types fondamentaux** : structuration des données
- **modules** et **foncteurs** : structuration des programmes
- **classes** : structuration des programmes et des données

Ce cours et le suivant : **techniques d'itération**

- **importance fondamentale en programmation**
  boucles et récursions sont indispensables pour des programmes expressifs (Turing-complètude)
- une grande part du **coût** en temps / mémoire en découle
- techniques **non triviales**, qui nécessitent de connaître les détails des structures. . .

Les structures itératives sont elles compatibles avec la notion d'abstraction qui nous intéresse ? (paramétrage, masquage de détails locaux)

Exemple : somme des éléments d'un tableau

Très naturel en **style itératif** :

```
let sum_arr a =
  let r = ref 0 in
  for i = 0 to Array.length a - 1 do
    r := !r + a.(i)
  done;
  !r
```

Aussi possible en **style récursif** :

```
let sum_arr a =
  let rec aux acc i =
    if i >= 0 then aux (acc + a.(i)) (i - 1)
    else acc in
  aux 0 (Array.length a - 1)
```

Peut on rendre **l'écriture de telles fonctions plus systématique ?**

Exemple : somme des éléments d'une liste

Très naturel en **style récursif** :

```
let sum_lst l =
  let rec aux acc l =
    match l with
    | [ ] -> acc
    | x :: m -> aux (acc + x) m in
  aux 0 l
```

Aussi possible en **style itératif** :

```
let sum_lst l =
  let r = ref 0
  and c = ref l in
  while !c != [ ] do
    r := List.hd !c + !r;
    c := List.tl !c
  done;
  !r
```

Peut on rendre **l'écriture de telles fonctions plus systématique ?**

Itération et coût

Un problème général :

- on se donne une **structure** $\mathcal{S}$ collectant des éléments
  (e.g., liste, arbre, graphe, tableau, table de hachage. . .)
- on fixe une **opération** f sur les éléments ainsi stockés
- on souhaite effectuer l'opération f sur tous les éléments de $\mathcal{S}$

Coût : évidemment **temps** + **espace**

Par exemple, pour le **coût en espace** :

- espace nécessaire pour **chaque exécution de** f
- espace utilisé **en pile** ? (appels récursifs terminaux ?)
- espace utilisé **sur le tas** ? (autres éléments à conserver, e.g., pour un parcours de graphe)

Les meilleurs bornes ne sont pas les mêmes selon la structure :
exemple : listes vs arbres

Itération et généralité du code

Quelques opérations sur **les arbres AVL** :

- test à l'appartenance, recherche, ajout, suppression
- affichages préfixe, infixe, suffixe
- filtrage (garder les éléments tels que...)
- existence (recherche d'un élément tel que...)
- etc

Les **invariants structurels internes** sont **complexes** (relations numériques d'équilibrage, préservation de l'ordre)...

Lorsque l'on veut ajouter une nouvelle opération sur les arbres AVL :

- quand **partager du code d'itération** ?
  (le plus souvent possible, lorsque ça ne casse ni efficacité ni correction)
- quand faut il tout de même **réécrire les boucles / récursions** ?

Itération et abstraction par une interface

Quelques opérations sur **les arbres AVL** :

- test à l'appartenance, recherche, ajout, suppression
- affichages préfixe, infixe, suffixe
- filtrage (garder les éléments tels que. . .)
- existence (recherche d'un élément tel que. . .)
- etc

On souhaite pouvoir **substituer une autre structure**, pour accomplir **le même rôle**, par exemple des **arbres rouges et noirs** :

- invariant **AVL** : chaque noeud stocke une valeur numérique liée à la différence de hauteur de ses fils
- invariant **rouges et noirs** : chaque noeud stocke un booléen lié à la différence de hauteur de ses fils

Il faut donc **cacher l'itération derrière une couche d'abstraction**

Itération infinie / non bornée

Certaines programmes **s'exécutent pour une durée non bornée** :

- un **système d'exploitation** (votre OS favori)
- une **interface graphique** (e.g., le bureau Linux, MacOS, Windows...)
- un **serveur** (site web ou base de données)

Exemple d'une **interface graphique** :

À chaque action de l'utilisateur (clic, déplacement de souris, entrée au clavier), mettre à jour l'affichage... et attendre l'action suivante.

On parle de **systèmes réactifs**.

Comment construire de tels programmes ?
si possible de manière claire et efficace

Structures itératives

Un programme sur **deux séances**.

Cette séance : **itérateurs abstraits**

permettent la séparation entre itération et opérations locales

La prochaine séance : **itération non bornée**

structures de flux (“streams”) et threads

Dans tous les cas, on construit des programmes autorisant un bon niveau d'abstraction du code, soit en utilisant des primitives du langage, soit en construisant / utilisant des librairies adaptées

Outline

Division du code

Considérons tout d'abord le **masquage des structures d'itération** derrière une **couche d'abstraction** :

- **modules** avec **interfaces** masquant les structures
- **classes** avec types abstraits

Considérons une **structure de "container"** (implémentée à l'aide de listes, d'arbres équilibrés, de tables de hachage, de tableaux, etc) :

Code "Fournisseur"

structure de données
fonctions spécifiques
add, mem, remove...

Code "Client"

fonctions génériques
aucune hypothèse
sur le code fournisseur

Interface abstraite

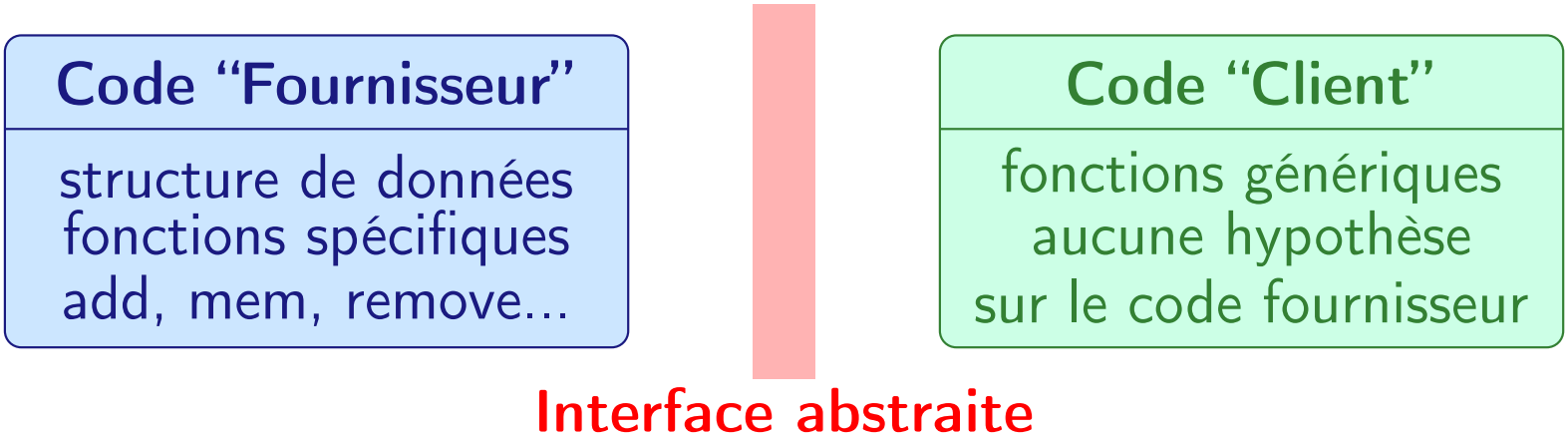
```
type t
val add: t -> elt -> t
val mem: t -> elt -> bool
```

Division du code

Considérons tout d'abord le **masquage des structures d'itération** derrière une **couche d'abstraction** :

- **modules** avec **interfaces** masquant les structures
- **classes** avec types abstraits

Considérons une **structure de "container"** (implémentée à l'aide de listes, d'arbres équilibrés, de tables de hachage, de tableaux, etc) :



Lorsque le client doit **itérer sur la structure**, de quel côté placer le code itératif (**boucle**, **fonction récursive**) ?

Un exemple

- Une valeur de type `t` décrit un **intervalle borné d'entiers**, comme par exemple $[12, 35]$
  bien sûr, la représentation est **abstraite**
- On souhaite effectuer une opération `f: int -> unit` sur chaque valeur comprise dans un tel intervalle.
  on pourrait aussi considérer d'autres opérations (somme, etc).

Sans type abstrait et en utilisant des paires, on obtiendrait :

```
type t = int * int

let iter f (a, b) =
  for i = a to b do
    f i
  end
```

Considérons à présent le cas où on utilise une interface abstraite...

Contrôle de l'itération : côté fournisseur

Première solution :

- placer le code itératif **côté fournisseur**
- le **client** doit passer l'opération à effectuer
  e.g., via une fonction d'ordre supérieur
  et l'itération échappe ensuite à son contrôle

Code "Fournisseur"	Code "Client"
structure de données fonctions spécifiques add, mem, remove...	fonctions génériques aucune hypothèse sur le code fournisseur

On obtient la **signature** :

```
type t
val add: t -> elt -> t
val mem: t -> elt -> bool
val iter: (elt -> unit) -> t -> unit
val fold: (elt -> 'a -> 'a) -> t -> 'a -> 'a
```

Contrôle de l'itération : côté fournisseur, exemple

Implémentation côté "fournisseur" :

```
module Intv =
  (struct
    type t = int * int
    let rec iter (f: int -> unit) (a, b) =
      if a > b then ( )
      else
        begin
          f a;
          iter f (a + 1, b)
        end
  end: ABSF)
```

Code "client" pour un affichage des éléments du tableau :

```
Intv.iter (Printf.printf "%d;") intv
```

- ne maîtrise **rien** de l'itération (à part son lancement...)

Contrôle de l'itération : côté fournisseur, interruption

Autre problème : **trouver la première solution** d'une équation dans un intervalle donné

- solution caractérisée par la fonction `f: int -> bool` renvoie `true` lorsque l'argument est solution
- on souhaite utiliser notre **type abstrait** et notre **itérateur**
- il faut toutefois éviter de prolonger l'itération après avoir trouvé une première solution
- nous avons vu que c'est le **fournisseur** qui contrôle l'itération

Solution : **une exception**

```
exception Stop of int
try
  Intv.iter (fun i -> if f i then raise (Stop i)) intv
with Stop i ->
  Printf.printf "solution found: %d\n" i
```

Contrôle de l'itération : côté client

Seconde solution :

- placer le code itératif **côté client**
- le **fournisseur** contient un type et des primitives pour **initialiser** l'itération, et pour **solliciter le passage** à l'élément suivant

Code "Fournisseur"	Code "Client"
structure de données fonctions spécifiques add, mem, remove...	fonctions génériques aucune hypothèse sur le code fournisseur

On obtient la **signature** :

```
type t
val add: t -> elt -> t
val mem: t -> elt -> bool
type it
val iter_new: t -> it
val iter_next: it -> elt option
```

Contrôle de l'itération : côté client

Implémentation côté "fournisseur" :

```
module Intv =
  (struct
    type t = int * int
    type it = { mutable next: int;
                interv:       t }
    let iter_new ((a, b) as intv) =
      { next   = a;
        interv = intv }
    let iter_next it =
      if it.next <= snd it.interv then
        let res = it.next in
        it.next <- it.next + 1;
        Some res
      else
        None
  end: ABSTRACT_INTERV)
```

L'**itérateur** contient :

- **l'intervalle**
- un champ mutable décrivant l'**état de l'itération** (valeur atteinte)

Itération suivante :

- l'état est mis à jour
- la valeur suivante est renvoyée

Contrôle de l'itération : côté client, exemple

Voyons maintenant le code côté **client**, pour la même fonction d'affichage :

```
let it = Intv.iter_new intv in
let rec aux ( ) =
  match Intv.iter_next it with
  | None -> ( )
  | Some v ->
      Printf.printf "%d; " v;
      aux ( ) in
aux ( )
```

Éléments de comparaison :

- le code client se charge non seulement de **l'opération d'affichage**, mais aussi de **demander l'élément suivant** et de la **condition d'arrêt**
- le module `Intv` **sélectionne l'élément suivant**

Contrôle de l'itération : côté client, interruption

Retour sur le **second problème** :
trouver la première solution d'une équation dans un intervalle donné

```
let it = Intv.iter_new intv in
let rec aux f =
  match Intv.iter_next it with
  | None -> None
  | Some v ->
      if f v then Some v
      else aux f in
aux ( )
```

Éléments de comparaison :

- cette fois-ci le contrôle **d'arrêt de l'itération** est direct
  puisque la structure itérative (et sa condition de sortie) sont contrôlées dans le code client
- donc nous n'avons plus besoin d'une exception...

Outline

Définition et utilisation d'un réducteur

Étudions tout d'abord le premier mode d'itération :

Définition : réducteur

On appelle **réducteur** une structure itérative abstraite implémentée **entièrement côté fournisseur**, et qui localise le **contrôle de l'itération** de ce même côté.

- aucune hypothèse sur **le style** :
  impératif et à l'aide de mutables ou **fonctionnel pur**
- aucune hypothèse sur **la valeur de retour** :
  peut être `unit` (effets de bord : affichage...)
  ou bien un autre type (avec ou sans effets de bord
- nous allons considérer plusieurs exemples

Définition et utilisation d'un réducteur

Étudions tout d'abord le premier mode d'itération :

Définition : réducteur

On appelle **réducteur** une structure itérative abstraite implémentée **entièrement côté fournisseur**, et qui localise le **contrôle de l'itération** de ce même côté.

- existe aussi en **Java** :

```
List<String> ctr = new /* ... */ ;
/* ... */
for( String s : ctr ){
    /* ... */
}
```

- nous allons voir quelques exemples en **OCaml**

Réducteurs de type `unit` : listes (1)

Premier cas : la valeur de retour est de type `unit`
autrement dit, l'opération itérée est un **effet de bord**

Réducteur sur les listes, opérant **de gauche à droite** :

```
let rec iter f l =
  match l with
  | [ ] -> ( )
  | x :: m -> f x ; iter f m
```

On peut utiliser une **déclaration locale** :

```
let iter f =
  let rec aux l =
    match l with
    | [ ] -> ( )
    | x :: m -> f x ; aux m in
  aux
```

Réducteurs de type `unit` : listes (2)

Remarque : l'**ordre d'itération** est fixé dans la fonction `iter`

- ce choix est donc **abstrait** du point de vue du **client**
- idéalement, la documentation de `iter` explicite ce choix
  sinon, le client ne peut faire aucune hypothèse. . .

Itération sur les listes en ordre inverse :

```
let iter_revord f =
  let rec aux l =
    match l with
    | [ ] -> ( )
    | x :: m -> aux m ; f x in
  aux
```

On observe que ce code **n'est pas récursif terminal** :
l'appel récursif à `aux` n'est pas la dernière opération du second cas.

Solution récursive terminale :

```
let iter_revord f l = iter f (List.rev l)
```

Réducteurs de type `unit` : arbres (1)

Considérons un type **arbre** :

```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree
```

et un **réducteur** associé :

```
let iter f =
  let rec aux = function
    | Leaf -> ( )
    | Node (t, x, u) -> aux t; f x; aux u in
  aux
```

- ce code n'est pas récursif terminal
  on peut le transformer, mais c'est un peu plus complexe
- l'ordre d'itération est plus complexe :
  l'opération `f` est **infixe** (entre les deux sous arbres)
  on traite **le sous arbre gauche, puis le sous arbre droit**

Réducteurs de type `unit` : arbres (2)

Transformation de notre réducteur en **style itératif** :

- il faut utiliser une **pile** pour stocker les calculs restants (exercice)

Transformation de notre réducteur en **style récursif terminal** :

```
let iter_tr f =
  let rec aux k = function
    | Leaf -> k ( )
    | Node (t, x, u) ->
        let k0 ( ) =
          f x;
          aux k u in
        aux k0 t in
  aux (fun () -> ())
```

- `k` et `k0` sont des **continuations** :
  décrivent la suite du calcul à effectuer
- on appelle ce code en **style par passage de continuations**

Réducteur de type non `unit`

Premier cas : le réducteur calcule une valeur de retour :
le code client doit passer une fonction à plusieurs arguments
(l'un des arguments sert d'**accumulateur**)

Réducteur sur les listes, opérant **de gauche à droite** :

```
let rec fold f l a =
  match l with
  | [ ] -> a
  | x :: m -> fold f (f a x) m
```

On peut utiliser une **déclaration locale** :

```
let fold f l a =
  let rec aux a = function
    | [ ] -> a
    | x :: m -> aux (f x a) m in
  aux a l
```

`(fold f [x0;x1;x2] a)` calcule `f (f (f a x0) x1) x2`

Exemple : réducteurs sur listes et arbres

Définition similaire sur un **type arborescent** :

- mêmes considérations sur l'ordre des paramètres
  on peut appliquer une opération de gauche à droite ou de droite à gauche
- comme pour les précédents réducteurs, cet ordre est **abstrait**

Exemple :

```
let rec fold f a t =
  match t with
  | Leaf -> a
  | Node (l, x, r) -> fold f (f x (fold f a r)) l
```

De même que pour `iter`, cette fonction n'est pas récursive terminale, mais on peut la transformer en style par passage de continuations

Quelques remarques

Style de programmation :

- impératif ou fonctionnel : pas observable par le code client
- par contre, le code client suppose généralement la structure **préservée**

Ordre de traitement des éléments de la collection :

- **entièrement contrôlé par le code fournisseur**
- **important** sauf si l'opération que l'on répète est **commutative** (si type de retour `unit`) ou **commutative** / **associative** (cas général)

Considérations liées au coût de l'itération :

- consommation de mémoire en **pile** : récursion terminale ?
- transformation en récursion terminale :
  coût pouvant être favorable ou non :
  calcul d'une autre structure, calcul de clôtures...

Outline

Définition et utilisation d'un itérateur

Nous considérons maintenant le second mode d'itération :

Définition : itérateur abstrait

On appelle **itérateur abstrait** une structure itérative abstraite avec deux primitives permettant :

- de **créer** un nouvel itérateur
- d'**effectuer un pas d'itération**

On peut définir des itérateurs **mutables** ou **immutables** :

- un itérateur **mutable** stocke un **état interne**
- dans le cas d'un itérateur **immutable**, un pas d'itération doit **produire un nouvel itérateur**

Nous allons maintenant voir quelques exemples...

Itérateur mutable : listes

On considère un itérateur permettant de visiter les éléments d'une liste dans l'ordre partant du premier élément jusqu'au dernier.

Définition du type (qui pourra être **abstrait derrière une interface**) :

```
type 'a iter = { mutable cur: 'a list }
```

Primitives sur le type `iter` :

```
let iter_new l = { cur = l }
let iter_next it =
  match it.cur with
  | [ ] -> None
  | a :: b ->
      it.cur <- b;
      Some a
```

Interface abstraite :

```
type 'a iter
val iter_new : 'a list -> 'a iter
val iter_next : 'a iter -> 'a option
```

Itérateur mutable : listes

Voyons maintenant le **code client**...

Affichage des éléments d'une liste de flottants :

```
let display l =
  let it = iter_new l in
  let rec aux ( ) =
    match iter_next it with
    | None -> ( )
    | Some f -> Printf.printf "%f\n" f; aux ( ) in
  aux ( )
```

Calcul de la somme des éléments d'une liste d'entiers :

```
let sum l =
  let it = iter_new l in
  let rec aux acc =
    match iter_next it with
    | None -> acc
    | Some i -> aux (acc + i) in aux 0
```

Itérateur immutable : listes

Lorsque l'on rend l'itérateur immutable, la fonction `iter_next` doit également **renvoyer un nouvel état de l'itérateur**.

Définition du type (qui pourra être **abstrait derrière une interface**) :

```
type 'a iter = 'a list
```

Primitives sur le type `iter` :

```
let iter_new l = l
let iter_next it =
  match it with
  | [ ] -> None
  | a :: b ->
      Some (a, b)
```

Interface abstraite :

```
type 'a iter
val iter_new : 'a list -> 'a iter
val iter_next : 'a iter -> ('a * 'a iter) option
```

Itérateur immutable : listes

Le **code client** propage la **valeur courante de l'itérateur** à **chaque itération** :

```
let display l =
  let it = iter_new l in
  let rec aux it =
    match iter_next it with
    | None -> ( )
    | Some (f, it) ->
        Printf.printf "%f\n" f;
        aux it in
  aux it
```

Le code est **similaire à la version mutable** :

- même structure, interface abstraite très similaire
- code fonctionnel pur

Itérateur sur une structure d'arbres

Pour construire un itérateur **sur une structure arborescente**, il faut stocker une **pile** dans l'état interne :

```
type 'a iter = { mutable cur: 'a tree list }
let iter_new t = { cur = [ t ] }
let rec iter_next it =
  match it.cur with
  | [ ] -> None
  | Leaf :: a ->
      it.cur <- a;
      iter_next it
  | Node (x, a, y) :: l ->
      it.cur <- x :: y :: l;
      Some a
```

Ordre d'itération :

- dans quel ordre les éléments sont ils extraits ?
- comment obtenir un autre ordre ?

Quelques remarques

Style de programmation :

- impératif ou fonctionnel : modifie à la fois client et fournisseur
- le code client suppose généralement la structure **préservée**

Ordre de traitement des éléments de la collection :

- **élément suivant** : contrôlé par le **code fournisseur**
- **condition d'arrêt** : explicitement contrôlée par le **code client**

Considérations liées au coût de l'itération :

- en **espace** : taille de l'état interne
- en **temps** : principalement lié au coût de `iter_next`

Outline

Transformation de structure

Un **problème légèrement différent** : **transformation de structure**

Code "Fournisseur"	**Code "Client"**
structure de données fonctions spécifiques add, mem, remove...	fonctions génériques aucune hypothèse sur le code fournisseur

Interface abstraite

- le code client fournit une **opération locale de transformation** des éléments de la structure
- le code fournisseur **effectue la transformation** en tous points
  - soit par effet de bord, via un itérateur classique
  - soit en produisant une nouvelle structure, via un transformateur

Une **variante** : **changements locaux à la structure abstraite**

Transformateur

Un type spécifique de **réducteurs** :

```
let rec map f l =
  match l with
  | [ ] -> [ ]
  | x :: m -> (f x) :: (map f m)
```

Exercice :

- transformation en fonction **récursive terminale** : utilisation d'un accumulateur, et inversion de la liste
- définition d'une fonction locale

Fonction similaire pour un **type arborescent** :

```
let rec tree_map f l =
  match l with
  | Leaf -> Leaf
  | Node (t, x, u) ->
      Node (tree_map f t, f x, tree_map f u)
```

Visiteurs

Limitations :

- les fonctions `_map` **préservent** la forme de la structure à l'identique
- cela ne permet pas de faire une **modification partielle**
  e.g., simplification d'expressions, évaluation, remplacement de certains noeuds...

Pour cela, nous allons voir un autre **motif de programmation** :

Définition : visiteur

Un **visiteur** est un type de donnée décrivant une opération de transformation générique **sur les noeuds d'un type somme récursif**.

- le concept est assez abstrait, nous allons donc mieux le comprendre sur un exemple
- on écrit souvent un visiteur à l'aide de **classes** et d'**héritage**, mais ce n'est pas une obligation...

Visiteurs : expressions arithmétiques

On considère les expressions arithmétiques suivantes :

```
type expr =
  | Cst of int
  | Var of string
  | Add of expr * expr
  | Mul of expr * expr;;
```

Opérations : évaluation, simplification, fixation de la valeur d'une variable

Principe :

1. un type enregistrement `visitor`, comprenant **une fonction de transformation pour chaque noeud dans l'AST**
2. une **fonction de transformation générique**
   i.e., une fonction, qui est paramétrée par la transformation définie noeud par noeud
3. une **instance** du visiteur est obtenue **en redéfinissant un ou plusieurs des champs** de `visitor`

Définition du type des visiteurs

On commence **par le type des visiteurs** :

```
type visitor =
    { e_cst: (int -> expr);
      e_var: (string -> expr);
      e_add: (expr -> expr -> expr);
      e_mul: (expr -> expr -> expr) };;
```

```
type expr =
  | Cst of int
  | Var of string
  | Add of expr *
      expr
  | Mul of expr *
      expr;;
```

Le visiteur par défaut se limite à **des fonctions qui appliquent les constructeurs** (autrement dit on se contente de construire la structure) :

```
let default =
  { e_cst = (fun i -> Cst i);
    e_var = (fun s -> Var s);
    e_add = (fun e0 e1 -> Add (e0, e1));
    e_mul = (fun e0 e1 -> Mul (e0, e1)); };;
```

Visiteurs : exemple

Ensuite, on définit la **fonction d'application** :

```
let transf vis =
  let rec aux e =
    match e with
    | Cst i -> vis.e_cst i
    | Var s -> vis.e_var s
    | Add (e0, e1) -> vis.e_add (aux e0) (aux e1)
    | Mul (e0, e1) -> vis.e_mul (aux e0) (aux e1) in
  aux
```

Rappel : le **type du visiteur** passé en argument

```
type visitor =
    { e_cst: (int -> expr);
      e_var: (string -> expr);
      e_add: (expr -> expr -> expr);
      e_mul: (expr -> expr -> expr) };;
```

L'effet du **visiteur par défaut** est la **fonction identité**.

Visiteur et utilisation (1)

Simplification des expressions de la forme $0 \times e$ ou $e \times 0$:

```
let vis_simpl =
  let f_mul e0 e1 =
    match e0, e1 with
    | Cst 0, _ | _, Cst 0 -> Cst 0
    | _, _ -> Mul (e0, e1) in
  { default with e_mul = f_mul }
let simpl = transf vis_simpl
```

On note que :

- les champs du visiteur par défaut sont **réutilisés**
  donc, dans ces cas, `transf vis_simpl` coïncide avec la fonction identité
- la définition de `f_mul` se concentre sur le **cas particulier** de la transformation que l'on définit...

Visiteur et utilisation (2)

On peut définir **une autre opération très facilement** :
remplacement de chaque variable par une constante définie par une fonction

```
let vis_var_def var_to_val =
  let f_var s =
    Cst (var_to_val 1) in
  { default with
    e_var = f_var }
let vis_var_def var_to_val =
  transf (vis_var_def var_to_val)
```

C'est l'intérêt de la construction à base de visiteur :
chaque opération se définit sans refaire tous les cas

Visiteur et utilisation (3)

Pour finir, on peut aussi définir une **fonction de calcul et simplification des expressions** (qui calcule complètement toute expression sans variable) :

```
let vis_eval =
  let f_add e0 e1 =
    match e0, e1 with
    | Cst i0, Cst i1 -> Cst (i0 + i1)
    | _, _ -> Add (e0, e1) in
  let f_mul e0 e1 =
    match e0, e1 with
    | Cst i0, Cst i1 -> Cst (i0 * i1)
    | Cst 0, _ | _, Cst 0 -> Cst 0
    | _, _ -> Mul (e0, e1) in
  { default with
    e_add = f_add ;
    e_mul = f_mul }
let eval = transf vis_eval
```

Outline

Principaux éléments à retenir

Nous avons vu plusieurs **motifs** de programmation, certains **plus spécifiques au langages fonctionnels**, d'autres **plus généraux**...

Principe général :

rendre la notion d'itération plus générale / abstraite

Principales constructions :

- **réducteurs** : `iter`, `fold`
  fonction d'itération décrite par le fournisseur d'une structure
- **itérateur** : `iter_next`
  interface permettant d'accéder aux éléments successeurs ;
  itération côté code client
- fonctions de **transformation** : `map`
- **visiteurs** :
  description abstraite de transformations sur des types arborescents

Travaux personnels

1. **Implémenter** et **tester** les fonctions de réducteurs et d'itérateurs sur **les listes**

2. En s'insipirant des exemples sur les intervalles, implémenter des **réducteurs** et taitérateurs sur les tableaux

3. Appliquer ces fonctions à **la recherche de l'ensemble des solutions d'une équation parmi les éléments d'un tableau**
   (l'équation sera donnée par une fonction abstraite sur les éléments)