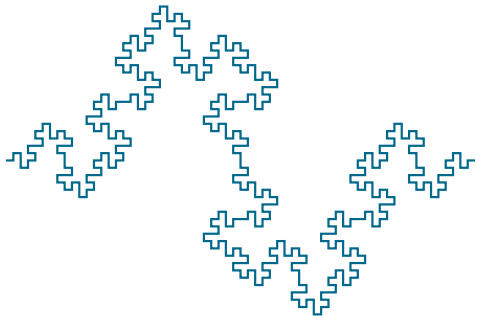
HLIN403 – Programmation Applicative

Abstraction de données - Arbres binaires

Christophe Dony – Annie Chateau
Université Montpellier – Faculté des Sciences



Introduction

Abstraction de données

Abstraction de données

Donnée : entité manipulée dans un programme, définie par un type.
ex : Les entiers 1, 3 et 5 sont des données d'un programme réalisant des calculs.

Abstraction de données : représentation interne de données d'un certain type accessible via un ensemble de fonctions constituant son interface.

Interface : ensemble de fonction de manipulation de données d'un certain type.

Type Abstrait : Type pour lequel on ne peut accéder à la représentation interne des données.

Définition de nouveaux types abstraits

Pascal : *Enregistrements* - *Records*

C : *Structures* - *Structs*

JAVA : *Classes* - *Class* - Réalisation de l'encapsulation "données - fonctions".

Un exemple : les nombres rationnels (d'après Abelson Fano Sussman

Interface :

On peut distinguer dans l'interface la partie *créationnelle*, permettant de créer des données, de la partie *fonctionnelle* ou *métier* permettant de les utiliser.

La programmation par objet a généralisé la création de nouveaux types de données. L'interface créationnelle est constituée de la fonction `new` et d'un ensemble de fonctions d'initialisation (généralement nommées *constructeurs*).

Un exemple : les nombres rationnels (d'après Abelson Fano Sussman

Exemple, interface pour les nombres rationnels :

```
make-rat    ; createur
denom       ; accesseur
numer       ; accesseur
+rat        ; interface fonctionnelle
-rat
*rat
/rat
=rat
```

Un exemple : les nombres rationnels (d'après Abelson Fano Sussman

Implantation

```
(define +rat (lambda (x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (denom x) (numer y)))
              (* (denom x) (denom y)))))

(define *rat (lambda (x y)
    (make-rat (* (numer x) (numer y))
              (* (denom x) (denom y)))))
```

Interface de création, Représentation No 1 : doublets, sans réduction

```
(define make-rat (lambda (n d) (cons n d)))
(define numer (lambda (r) (car r)))
(define denom (lambda (r) (cdr r)))
```

Les différents niveaux d'abstraction

```
utilisation des rationnels
------- +rat -rat make-rat numer denom

représentation des rationnels
------- doublets : cons, car, cdr

représentation des doublets
------- ???
```

Interface de création, Représentation no 2, vecteurs et réduction

Vecteur : Collection dont les éléments peuvent être rangés et retrouvés via un index, qui occupe moins d'espace qu'une liste contenant le même nombre d'éléments.

Manipulation de vecteurs.

```
(make-vector taille) : création vide
(vector el1 ... eln) : définition par extension
(vector-ref v index) : accès indexé en lecture
(vector-set! v index valeur) : accès indexé en écriture
```

Interface de création, Représentation no 2, vecteurs et réduction

```
(define make-rat (lambda (n d)
  (let ((pgcd (gcd n d)))
    (vector (/ n pgcd) (/ d pgcd)))))

(define numer (lambda (r) (vector-ref r 0)))

(define denom (lambda (r) (vector-ref r 1)))
```

Arbres binaires

Graphe : ensemble de sommets et d'arêtes

Graphe orienté : ensemble de sommets et d'arcs (arête "partant" d'un noeud et "arrivant" à un noeud)

Arbre : graphe connexe sans circuit tel que si on lui ajoute une arête quelconque, un circuit apparaît. Les sommets sont appelés **noeuds**.

Exemple d'utilisation : Toute expression scheme peut être représentée par un arbre.

Arbres binaires

Arbre binaire :

- soit un arbre vide

- soit un noeud ayant deux descendants (fg, fd) qui sont des arbres binaires

Arbre binaire de recherche : arbre dans lesquels une valeur v est associée à chaque noeud n et tel que si $n_1 \in fg(n)(resp.fd(n))$ alors $v(n_1) < v(n)$ (resp. $v(n_1) > v(n)$)

Arbre partiellement équilibré : la hauteur du SAG et celle du SAD diffèrent au plus de 1.

Arbres binaires de recherche en Scheme

- Création :

  `(make-arbre v fg fd)`

- Accès aux éléments d'un noeud n :

  `(val-arbre n)`, `(fg n)`, `(fd n)`

- Test

  `(arbre-vide? n)`

Arbres binaires de recherche en Scheme

- Interface métier :
  - `(inserer valeur a)` : rend un nouvel arbre résultat de l'insertion de la valeur n dans l'arbre a. L'insertion est faite sans équilibrage de l'arbre.
  - `(recherche v a)` : recherche dichotomique d'un élément dans un arbre
  - `(afficher a)` : affichage des éléments contenus dans les noeuds de l'arbre, par défaut affichage en profondeur d'abord.

Représentation interne - version 1

Consommation mémoire, feuille et noeud : 3 doublets.

Simple à gérer.

```
(define make-arbre (lambda (v fg fd)
  (list v fg fd)))

(define arbre-vide? (lambda (a) (null? a)))

(define val-arbre (lambda (a) (car a))) ;;

(define fg (lambda (a) (cadr a)))

(define fd (lambda (a) (caddr a)))
```

Insertion sans duplication

```
(define insere (lambda (n a)
  (if (arbre-vide? a)
      (make-arbre n () ())
      (let ((valeur (val-arbre a)))
        (cond ((< n valeur) (make-arbre valeur
                                        (insere n (fg a)) (fd a)))
              ((> n valeur) (make-arbre valeur
                                        (fg a) (insere n (fd a))))
              ((= n valeur) a))))))
```

```
(define a (make-arbre 6 () ()))
(insere 2 (insere 5 (insere 8 (insere 9 (insere 4 a)))))
```

Recherche dans un arbre binaire

Fonction booléenne de recherche dans un arbre binaire. La recherche est dichotomique.

```
(define recherche (lambda (v a)
  (and (not (arbre-vide? a))
       (let ((valeur (val-arbre a)))
         (cond ((< v valeur) (recherche v (fg a)))
               ((> v valeur) (recherche v (fd a)))
               ((= v valeur) #t))))))
```

Seconde représentation interne

Coût mémoire, noeud (3 doublets), feuilles (1 doublet).

Nécessite un test supplémentaire à chaque fois que l'on accède à un fils.

```
(define make-arbre (lambda (v fg fd)
  (if (and (null? fg) (null? fd))
      (list v)
      (list v fg fd))))

(define fg (lambda (n) (if (null? (cdr n)) () (cadr n))))

(define fd (lambda (n) (if (null? (cdr n)) () (caddr n))))
```

Toutes les autres fonctions de manipulation des arbres binaires (interface fonctionnelle) sont inchangées.

L'exemple des dictionnaires

Le type dictionnaire est un type récurrent en programmation, on le trouve dans la plupart des langages de programmation (java.util.Dictionary).

En scheme, il a été historiquement proposé sous le nom de listes d'association. Une liste d'associations est représentée comme une liste de doublets ou comme une liste de liste à deux éléments.

Interface de manipulation

```
>(define l1 '((a 1) (b 2)))
>(assq 'a l1)
(a 1)
>(assq 'c l1)
#f
>(assoc 'a l1)
(a 1)
>(define l2 '( ((a) (une liste de un élément))
               ((a b) (deux éléments)) ))
>(assq '(a b) l2)
#f
>(assoc '(a b) l2)
((a b) (deux éléments))
```

Interface : construction et manipulations

Scheme n'a pas prévu d'interface de construction, on construit directement les listes.

Ceci implique que l'on ne peut pas changer la représentation interne des dictionnaires.

Imaginons une interface de construction

```
(define prem-alist car)
(define reste-alist cdr)
(define null-alist? null?)
(define make-alist list)
(define add (lambda (clé valeur l)
                (cons (list clé valeur) l)))
```

Interface : construction et manipulations

```
(define my-assoc (lambda (clé al)
 (letrec ((loop (lambda (alis)
    (cond ((null-alist? alis) #f)
          ((equal? (car (prem-couple alis)) clé) )
                   (prem-couple alis))
          (else (loop (reste-alist alis)))))))
    (loop al))))
```