

## 15 Abstraction de données

### 15.1 Définitions

**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 des fonctions permettant de manipuler les éléments d'un type abstrait.

**Type Abstrait** : Type pour lequel,

- variante 1 : on peut ignorer
- variante 2 : on ne peut pas accéder à  
la **représentation interne** des données.

On peut éventuellement distinguer dans l'**interface** la partie *création*, permettant de créer des éléments du nouveau type, la partie *accès* pour accéder aux propriétés des éléments et la partie *utilisation* ou *fonctionnelle* ou encore *métier* permettant de les utiliser.

La programmation par objet a généralisé la création de nouveaux types de données. L'interface de création est constituée de la fonction **new** et d'un ensemble de fonctions d'initialisation (généralement nommées *constructeurs*). On y parle également d'*accesseurs* et de fonctions "métier".

### 15.2 Définition de nouveaux types abstraits par les utilisateurs

Pascal : *Enregistrement* - *Record*

C, Scheme : *Structure* - *Struct*

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

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

#### Interface

```
création : make-rat  
accès : denom numer  
métier : +rat -rat *rat /rat =rat
```

#### Implantation

```

1 (define (+rat x y)
2   (make-rat
3     (+ (* (numer x) (denom y))
4       (* (denom x) (numer y)))
5     (* (denom x) (denom y))))
6
7 (define (*rat x y)
8   (make-rat
9     (* (numer x) (numer y))
10    (* (denom x) (denom y))))

```

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

```

1 (define (make-rat n d) (cons n d))
2 (define (numer r) (car r))
3 (define (denom r) (cdr r))

```

## Les différents niveaux d'abstraction

```

1 utilisation des rationnels
2 ----- +rat -rat make-rat numer denom
3 représentation des rationnels, Utilisation des doublets
4 ----- doublets : cons, car, cdr
5 représentation des doublets
6 ----- ???

```

## 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.

```

1 (make-vector taille) : création vide
2 (vector e11 ... e1n) : définition par extension
3 (vector-ref v index) : accès indexé en lecture
4 (vector-set! v index valeur) : accès indexé en écriture

```

```

1 (define (make-rat n d)
2   (let ((pgcd (gcd n d)))
3     (vector (/ n pgcd) (/ d pgcd))))
4
5 (define (numer r) (vector-ref r 0))
6
7 (define (denom r) (vector-ref r 1))

```

## 15.4 Arbres binaires

**Grphe** : ensemble de sommets et d'arêtes

**Grphe orienté** : ensemble de sommets et d'arcs (arrête "partant" d'un noeud et "arrivant" à un noeud)

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

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

**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 associé à chaque noeud  $n$  et tel que si  $n1 \in fg(n)$  (resp.  $fd(n)$ ) alors  $v(n1) < v(n)$  (resp.  $v(n1) > v(n)$ )

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

## 15.5 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)
- 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.

### 15.5.1 Représentation interne - version 1

Consommation mémoire, feuille et noeud : 3 doublets.

Simple à gérer.

```
1 (define (make-arbre v fg fd)
2   (list v fg fd))
3
4 (define (arbre-vide? a) (null? a))
5
6 (define (val-arbre a) (car a)) ;;
7
8 (define (fg a) (cadr a))
9
10 (define (fd a) (caddr a))
```

### 15.5.2 Insertion sans duplication

```
1 (define (insere n a)
2   (if (arbre-vide? a)
3       (make-arbre n () ())
4       (let ((valeur (val-arbre a)))
5           (cond ((< n valeur) (make-arbre valeur (insere n (fg a)) (fd a)))
6                 ((> n valeur) (make-arbre valeur (fg a) (insere n (fd a))))
7                 ((= n valeur) a))))))
```

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

### 15.5.3 Recherche dans un arbre binaire

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

```
1 (define (recherche v a)
2   (and (not (arbre-vide? a))
3        (let ((valeur (val-arbre a)))
4            (cond ((< v valeur) (recherche v (fg a)))
5                  ((> v valeur) (recherche v (fd a)))
6                  ((= v valeur) #t))))))
```

### 15.5.4 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.

```
1 (define (make-arbre v fg fd)
2   (if (and (null? fg) (null? fd))
3       (list v)
4       (list v fg fd)))
5
6 (define (fg n) (if (null? (cdr n)) () (cadr n)))
7
8 (define (fd n) (if (null? (cddr n)) () (caddr n)))
```

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

## 15.6 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.

### 15.6.1 Interface de manipulation

```
1 >(define l1 '((a 1) (b 2)))
2 >(assq 'a l1)
3 (a 1)
4 >(assq 'c l1)
5 #f
6 >(assoc 'a l1)
7 (a 1)
8 >(define l2 '( ((a) (une liste de un élément)) ((a b) (deux éléments)) ))
9 >(assq '(a b) l2)
10 #f
11 >(assoc '(a b) l2)
12 ((a b) (deux éléments))
```

### 15.6.2 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

```
1 (define prem-alist car)
2 (define reste-alist cdr)
3 (define null-alist? null?)
4 (define make-alist list)
5 (define (add clé valeur l)
6   ;; rend une nouvelle aliste incluant une nouvelle liaison clé-valeur
7   ;; ne vérifie pas si la clé était déjà utilisée
8   (cons (list clé valeur) l))
```

```
1 (define (my-assoc clé al)
2   (let loop ((al al))
3     (cond ((null-alist? al) #f)
4           ((equal? (car (prem-couple al)) clé) (prem-couple al))
5           (else (loop (reste-alist al)))))
```