Université Paul Sabatier – Toulouse III
IUT A - Toulouse Rangueil

Type abstrait de données

Semestre 2

Toulouse, le 17 décembre 2011

Chapitre 1

Les pointeurs

Les pointeurs sont un nouveau type en plus des tableaux et des enregistrements.
La gestion de la mémoire est dynamique est non statique comme les deux types précédents.

1.1 L'allocation statique

Définition L'espace mémoire est allouée une fois pour tout par le compilateur au début de l'excetion.
Les tableaux et les enregistrements sont statiques.

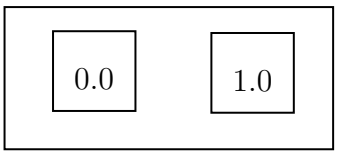
1.1.1 Exemple

Soit le type point :

```
1 type Point : enregistrement
2   abscisse <Reel>,
3   ordonnee <Reel>;
```

Listing 1.1 – Le type point

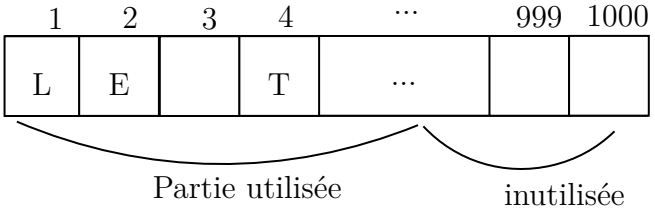
La variable est allouée statiquement en début d'execution.



Un texte de 1000 cractères.

```
1 type Texte : tableau [1 a 1000] de <caracteres>;
```

Listing 1.2 – Le type Texte



1.2 L'allocation dynamique

Définition Le programmeur peut dynamiquement soliciter de la mémoire en cours d'éxecution. Le mécanisme est celui des pointeurs, ou variable accès.

1.2.1 Notion de pointeur

Variable élémentaire (alloué statiquement) mais dont le contenu mémorise l'adresse d'une donnée en mémoire (allouée dynamiquement).

Exemple

Soit à définir un pointeur sur un entier

```
1 type PtrEntier : pointeur sur <Entier>
2 -- Pointeur sur est le constructeur de type
3 -- Entier est le type de donnees pointes
```

Listing 1.3 – Le type PtrEntier

Remarques

1 Une adresse mémorisée par un pointeur se materialise par une flèche : en programmation cette adresse est symbolique

2 L'accès a la donnée se fait toujours par le pointeur

1.2.2 Opération sur les pointeurs

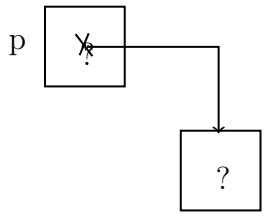
Opération	Syntaxe	Commentaire
Allocation d'espace mémoire	$allouer(p)$ où p est un pointeur	Réservation d'espace pour la donnée pointée par p et affectation de p par l'adresse de la zone allouée.
Libération d'espace mémoire	$liberer(p)$	Restitution de la zone pointée par p à la mémoire
Accès à la donnée pointée	$p\uparrow$	Accès à la zone pointée par p
Affectation	p1 <- p2	Affectation de deux pointeurs
Égalité	p1 = p2	Égalité de deux pointeurs
Différence	p1 /= p	Différence de deux pointeurs

1.2.3 L'allocation (allouer)

Définition Demande d'allocation mémoire conformément au type de la donnée pointée.

Syntaxe

```
1 allouer(p) -- ou p est un pointeur
```



Sémantique
- Allocation de mémoire
- Affectation en sortie de l'adresse de la zone allouée

Exemple

Soit p de type PtrEntier.

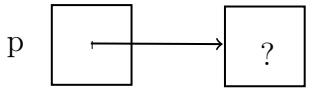
```
1 type PtrEntier : pointeur sur <Entier>;
2
3 allouer(p);
```

Remarque L'entier référencé par p n'est pas initialisé

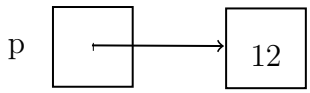
1.2.4 Accès à la donnée pointé (↑)

Définition Si p désigne l'adresse d'une donnée, $p\uparrow$ désigne la donnée pointé.

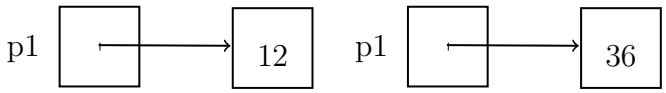
Exemple 1



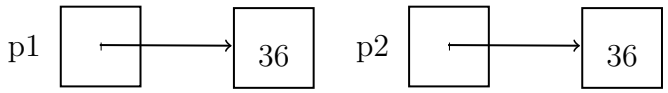

```
1 p↑<-12;
```



Exemple 2




```
1 p1↑<-p2↑
```



Remarque La donnée pointée n'a pas de nom (accès par le pointeur)
Les opérations sur la donnée pointée dépende du type de la donnée pointé.

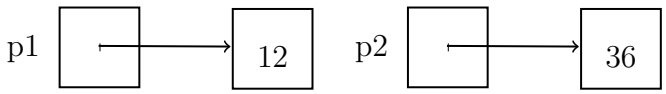
1.2.5 L'affectation (<-), l'égalité (=) et la différence (/=)

Définition Opération usuel de l'algorithmique mais un contenu égal à l'adresse !

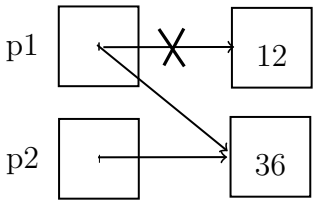
Syntaxe

```
p1 <- p2;
p1 = p2;
p1 /= p2;
```

Exemple




```
p1 <- p2;
```



L'entier 36 est référencé par p, et p2
L'entier 12 n'est plus atteignable

Remarque

1 Pointeur p peut référencer une zone dynamique sans préalablement réaliser l'opération allouer(p)

2 La constante NULL est une constante de type pointeur en général, utilisée pour indquer que le pointeur ne mémorise pas d'adresse.

3 La primitive allouer(p) affecte p par la constante NULL si on ne dispose plus de mémoire dynamique

4 La notation $p\uparrow$ n'a de sens que si $p \neq NULL$

1.2.6 La libération (liberer)

Définition Libération de la mémoire

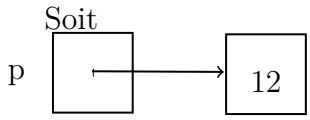
Syntaxe

```
liberer(p);
```

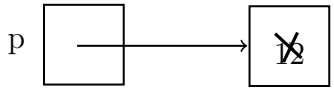
Sémantique

Libére la zone mémoire pointée par p

Exemple




```
1 liberer(p);
```



Remarques

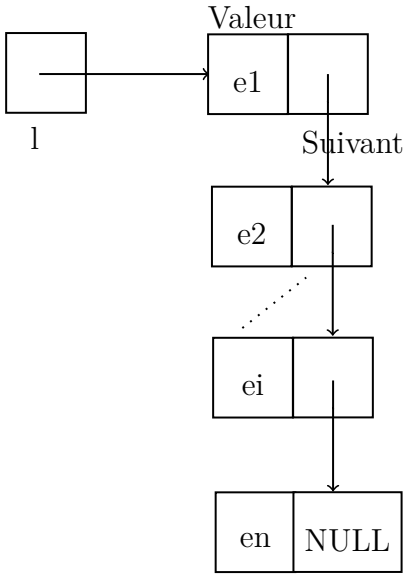
1 Le pointeur p n'est pas mis à jour par l'opération libérer.

2 Ne pas accéder à la donnée pointée via d'autre pointeur !

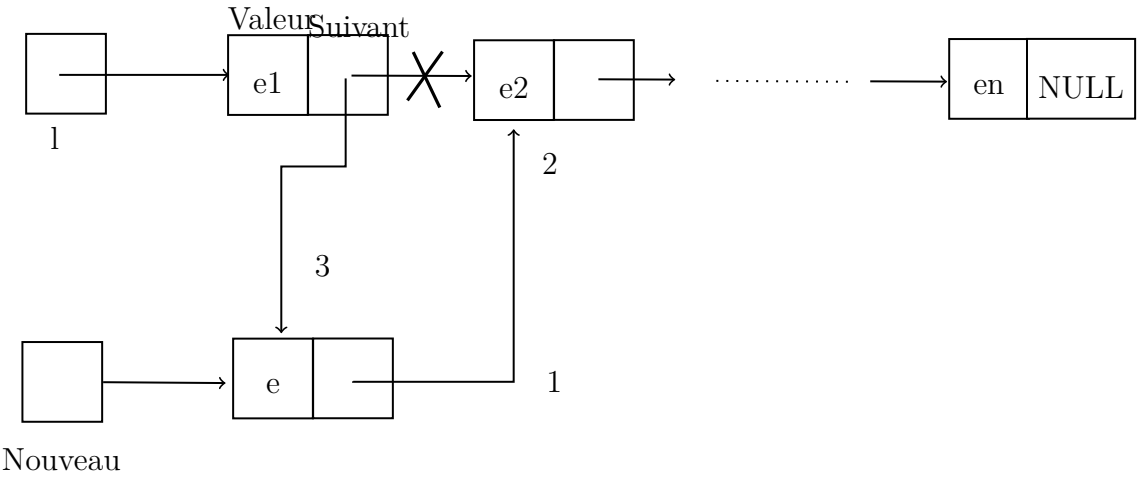
1.3 Un exemple : la liste simplement chainée

La donnée pointée réference une zone.

```
1 type PtrCellule : pointeur sur <Cellule>;
2 type Cellule : enregistrement
3   valeur <Entier>,
4   suivant <Entier>;-- L'adresse de la cellule suivante
5
6 glossaire
7   p <PtrCellule>;
```



Problème Ecriture de du code de l'insertion d'un élément l au rang 2 de la liste l




```
glossaire
  l <PtrCellule>;
  nouveau <PtrCellule>;

debut
  si l = NULL alors
    ecrire("La liste est vide!");
  sinon
    allouer(nouveau); --1
    si nouveau = NULL alors
      ecrire("Debordement memoire!");
    sinon
      nouveau↑.valeur <- e; --2
      -- chainer la cellule a e2
      nouveau↑.suivant<- nouveau; --3
    fin si;
  fin si;
fin
```

Remarques

1 Nouveau↑ : accès à l'enregistrement cellule pointé par nouveau nouveau↑.valeur : accès au champ valeur de l'enregistrement référencé par nouveau

2 SI l se réduit à un élément, dans ce cas on a p↑.suivant = NULL donc.

$$\begin{aligned} nouveau\uparrow.suivant &<- l\uparrow.suivant \\ \Leftrightarrow nouveau\uparrow.suivant &<- NULL \end{aligned}$$

Chapitre 2

Les types abstraits de données

Une structure de donnée se traduit dans un langage par un type. On distingue, du plus simple au plus complexe :

- Les types élémentaires (cf cours algo)
- Les types composés (cf cours algo)
- Les Types Abstraits de données (étudiés dans ce cours)

2.1 Notion de type abstrait

2.1.1 Définitions

Type Un type est un ensemble de valeurs et un ensemble d'opérations.

Type Abstrait Un type abstrait est un type où l'utilisateur de la donnée ignore la représentation de la donnée mémoire et le codage des opérations.

2.1.2 Différents point de vue d'un TAD

Concepteur

Il définit avec précision les opérations et les propriétés du type abstrait.
Ce point de vue est appelé **spécifications** c'est la définition du type (QUOI ?)

Programmeur

Il propose un codage des opérations du type
Ce point de vue est appelé **implémentation** (COMMENT ?)

Utilisateur

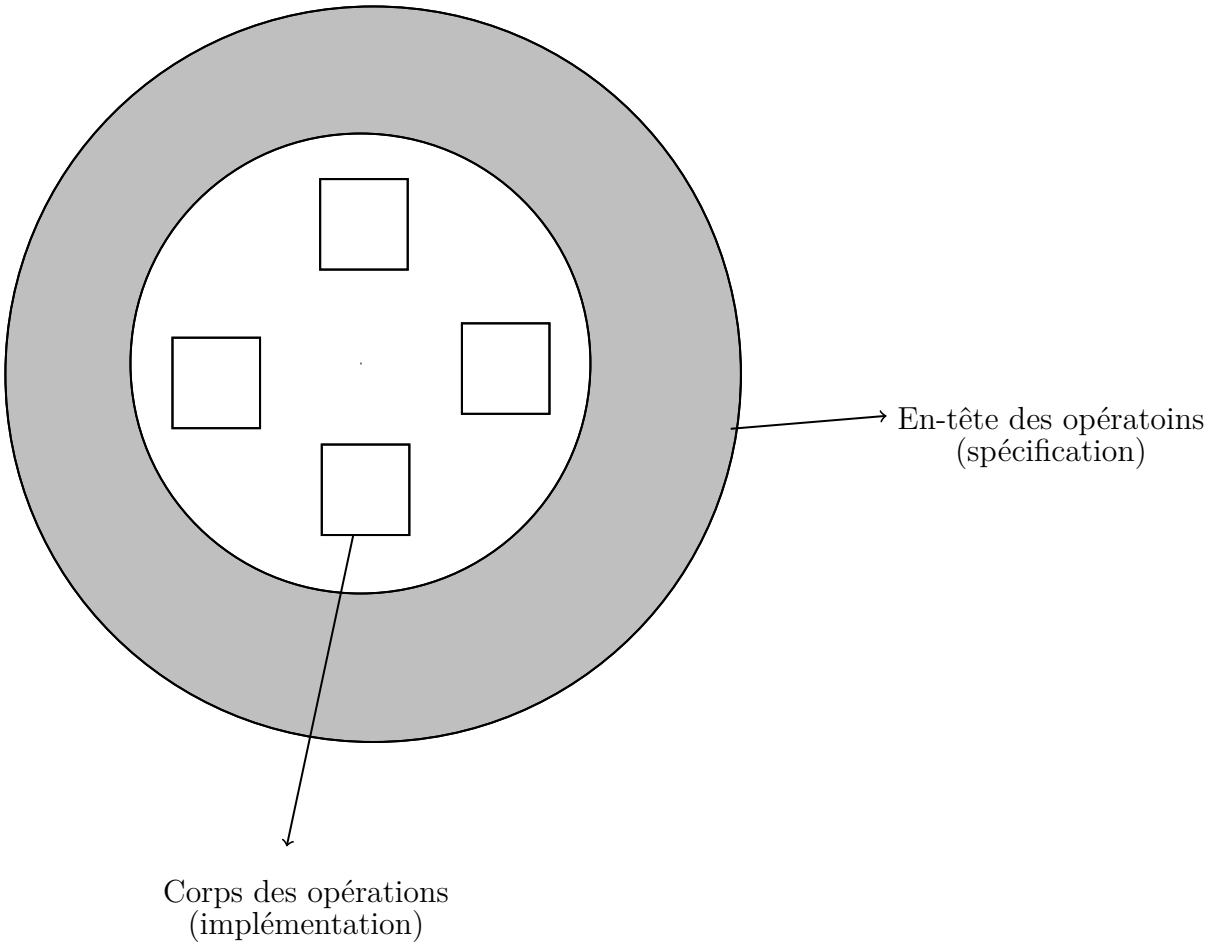
Il exploite les opérations du type pour son application.

2.1.3 Propriété des TADs

Encapsulation Les détails d'implémentations d'un type abstrait sont cachés (donc non accessible) au client.

Intérêt
- Le programmeur peut modifier son implémentation sans impacter le client
- La localisation, le code du type abstrait est enregistré au même endroit de l'application.
- Notion de module, par exemple paquetage ou classe.

2.1.4 En résumé



2.2 Spécification d'un TAD

Définition Ensemble des opérations et des propriétés du type.
Mode d'emploi de la structure de données

2.2.1 Opération

Énumération de l'ensemble des opérations selon la syntaxe pour une opération.

$$f : x_1 \times x_2 \times \ldots \times x_n \rightarrow y_1 \times y_2 \times \ldots \times y_n$$

f nom opération (symbole de fonction)

$x_1 \times x_2 \times \ldots \times x_n$ Domaines d'entrée de l'opération

$y_1 \times y_2 \times \ldots \times y_n$ Domaine de sortie de l'opération

Où les x_i et y_i sont des types dont l'un au moins est le type T étudié

Exemple

Soit à définir le TAD Point d'affichage d'un point à l'écran.

Opération
- Créer un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
- Connaître l'abscisse d'un point p
- Modifier la taille t d'un point p
- Translater un point p de tx et ty

Plus formellement Soit Point le type point.

$$\begin{aligned} unPoint : Reel \times Reel \times Couleur \times Reel &\rightarrow Point \\ PointOrigine &\rightarrow Point \\ taille : Point &\rightarrow Reel \\ modifierTaille : Point \times Reel &\rightarrow Point \\ translater : Point \times Reel \times Reel &\rightarrow Point \\ &\vdots \\ &\vdots \end{aligned}$$

Suite sur moodle clef = tads2

Remarque Une opération peut-être partielle
Par exemple la taille t du point doit être positive
On définit une pré condition : pour p de type Point x,yx et t de type Réel et c de type Couleur.
$unPoint(x, y, c, t)$ est défini par $t > 0$

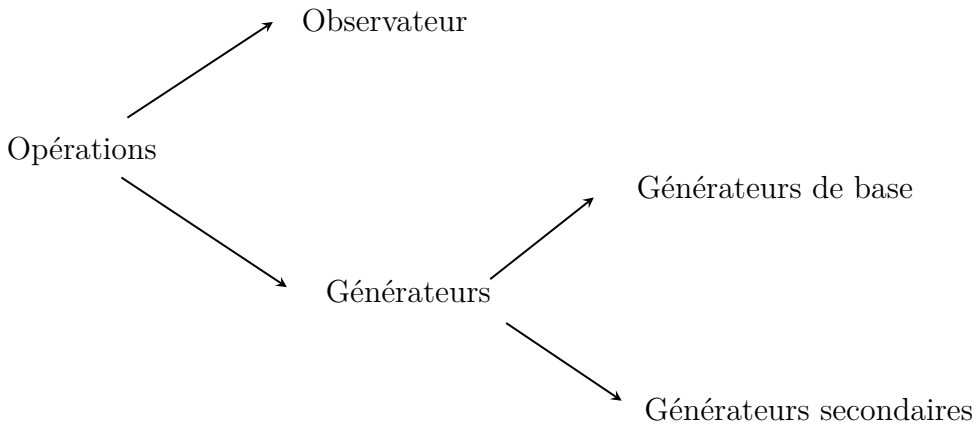
2.2.2 Propriétés

Opération Syntaxe du Type Abstrait de Données

Propriétés Sémantique du Type Abstrait de Données

Pour définir les propriétés On combine les opérations entre elles et on indique le résultat de ces combinaisons.

On définit pour les opérations



Observateur Opération qui fournit une caractéristique de la donnée (sans la modifier)

Opérateur Opération qui fournit une valeur du type abstrait étudié

Générateur de base Générateur qui permet de construire toutes les valeurs du type

Générateur secondaire Générateur autre qu'un générateur de base

Pour écrire les propriétés
- on fournit les valeurs des observateurs et des générateurs secondaires appliqués au générateurs de base
- On peut aussi procéder par équivalence avec le générateur de base

Pour le TAD point

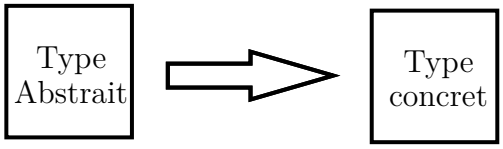
Générateur de base	unPoint
Générateur Secondaire	pointOrigine ; modifierTaille ; translater ;
Observateur	taille

Propriété du TAD Point

Pour x, y, tx, ty, de type Réel.
Pour c de type Couleur

$$pointOrigine = unPoint(0.0, 0.0, noir, 1.0) \quad (2.1)$$
$$taille(unPoint(x, y, c, t)) = t \quad (2.2)$$
$$modifierTaille(unPoint(x, y, c, tx), t2) = untPoint(x, y, c, t2) \quad (2.3)$$
$$translater(unPoint(x, y, c, t), tx, ty = unPoint(x \times ty, y \times ty, c, t) \quad (2.4)$$

2.2.3



- Opérations
- Propriétés

- Types
- Sous-programmes

Remarques

Propriété = Axiome définit par le constructeur
Par exemple on aurait pu imaginer :

$$modifierTaille(unPoint(x, y, c, t1), t2)$$
$$unPoint(x, y, c, t1 + t2)$$

Pré conditions valides lors de l'écriture des propriétés.

Les types élémentaires (entier, Réel, booléen ...) sont des types abstraits de données déjà définis dans le langage.
Pour Booléen : Vrai → Booléen
Faux → Booléen
Non : Booléen → Booléen Et : Booléen × Booléen → Booléen
...

Propriétés : Non(vrai) = Faux
Non(faux) = vrai
Et(vrai,vrai) = vrai
Et(vrai,faux) = faux.

Les types composés (Tableau, enregistrement) sont aussi des Types Abstrait de Données.
Pour tableau :
unTableau : Entier × Entier → Tableau[T]
ième : Tableau[T] × Entier → T
changeIème : Tableau[T] × Entier × T→ Tableau[T]

$ième(tab, i) \equiv tab[i]$ $changerIème(tab, i, el) equiv tab[i] < -e$
Propriétés :

```
1 ieme(changerIeme(tab,i,e),j)
2 si i = j alors
3   e
4 sinon
5   ieme(tab,j);
6 fin si;
```

Listing 2.1 – Propriété

Il existe des TADs fondamentaux les tables, listes, piles, fils, arbres, graphe
En général ces TADs sont génériques.
Par exemple pour le TAD liste on introduit le TAD $liste[T]$

2.3 Du type Abstrait au type concret

Définition Un type concret est la transition dans un langage d'un type abstrait.

Schéma de traduction

2.3.1 Étapes de traductions

Soit Ta un TAD et Tc le type concret correspondant.

Étape 1(Concepteur) Définition de Tc un entête de sous programme pour chaque opérations de Ta.

Étape 2 (Programmeur) Définir une représentation mémoire d'une valeur de Tc (Tableau, enregistrement, pointeur...)

Étape 3 (Programmeur) Codes les corps des sous programmes dans Tc (connaissant la représentation mémoire)

Les points 2 et 3 seront développés au chapitre suivant (implémentation d'un TAD)

2.3.2 En tête des sous programmes

Dans Ta, opération = fonctions (au sens mathématique)
Dans Tc, opération = procédure ou fonction
Dans Tc, on distingue :
- Les opérations de construction
- Les opérations de modification
- Les opérations d'évaluation

Opération de construction

Rôle Construire une valeur du type étudié, éventuellement à partir de valeur d'autres types. Opération appelée aussi **constructeur**

Caractéristique Le type Ta étudié n'apparait que dans le domaine de sortie de l'opération.

Règle Une opération de construction dans Ta se code par une fonction dans Tc(en général)

Exemple 1 Dans Ta : unPoint Réel × Réel × Couleur × Réel → Point
Dans Tc :

```
1 -- construit un point a partir de X et Y d'une couleur C et de
     taille T
2 -- Nacessite t > 0
3 fonction unPoint(entree x <Reel>, entree y <Reel>, entree t<Couleur
     >, entree t<Reel>)
4     retourne <Point>
5     declenche tailleInvalide;
```

Listing 2.2 – Tc

Exemple 2 Dans Ta : pointOrigine → Point
Dans Tc

```
1 --point a l'origine
2 fonction pontOrigine
3 retourne <Point>;
```

Listing 2.3 – Tc

Le non respect d'une pré condition se traduit par une levée d'exception (cf unPoint)

Un constructeur sans paramètre d'entrée est appelé constante du type abstrait pour un client

```
glossaire
  p <Point>;
debut
  p <- pointOrigine;
  si p = pointOrigine alors
    ⋮
  fin si;
fin
```

2.3.3 Opération de consultation

Rôle Fournit une caractéristique d'une valeur du type opération aussi appelé observateur .

Caractéristique Le type Ta étudié n'apparait que dans le domaine d'entrée de l'opération.

Règles Une opération de consultation dans Ta se traduit toujours par une fonction dans TC

Exemple Dans Ta
taille : Point → Réel

Dans Tc

```
-- retourne la taille d'un point
fonction taille (entree p <Point>)
  retourne <Reel>;
```

2.3.4 Opération de modification

Rôle Modifier une caractéristique d'une valeur d'un type.

Caractéristique Le type Ta apparait à la fin dans le domaine d'entrée et le domaine de sortie de l'opérateur.

Règle Une opération de modification dans Ta se code par une procédure dans Tc. (avec le mode mise à jour pour la valeur à modifier !)

Exemple Dans Ta : modifierTaille : Point × Réel → Point
Dans Tc

```
-- modifier la taille des points p avec la valeur t
-- necessite t >0
procedure modifierTaille(maj p <Point>, entree t <Reel>)
  declanche tailleInvalide;
```

2.3.5 Les opérations d'évaluation

Rôle Construit une nouvelle valeur du type abstrait à partir de la valeur existante du type. Opération avec ainsi similaire aux constructeur.

Caractéristique Le type Ta apparaît à la fin dans le domaine d'entrée et de sortie de l'opération. (comme une opération de modification).

Règle Une opération d'évaluation se traduit dans Ta par une fonction dans Tc.

Exemple Dans Ta : Point × Réel → Point
Dans Tc

```
-- construit une un nouveau point en considerant une nouvelle
   taille t
-- necessite t >0
fonction nouvelleTaille(entree p <Point>, entree t <Reel>)
  retourne <Point>
  declanche tailleInvalide;
```

2.3.6 Spécification d'un type concret

Définition Regroupement des en-têtes des sous programme de la spécification (cf 3.2), avec en commentaire les propriétés du type (cf 2).
Cette spécification est aussi appelée spécification algorithmique du type abstrait . (cf moodle pour Point et Tableau[T])

2.4 Utilisation d'un TAD

Un client d'un Type Abstrait de Données peut :
- Définir des variables de type T
- Définir des paramètres de sous-programme de type T
- Définir de nouveaux type en utilisant le type T
- Appeler des sous-programme définis par le type T

Remarque Le client n'a pas accès à la représentation mémoire et au codage des sous-programme définis dans T !

Exemple

Soit à calculer le point milieu d'un segment.
En tant que client :

```
type Segment enregistrement
  origine <Point>,
  extremite <Point>;
```

```
--determine le point milieu du segment du segment s

fonction pointMilieu (entree s <Segment>) retourne <Point>
glossaire
```

```
5   xMilieu <Reel>; -- Abscisse du point milieu
6   yMilieu <Reel>; -- ordonnee du point milieu
7 debut
8   xMilieu <- (abscisse(s.origine)+abscisse(s.extremite))/2;
9   yMilieu <- (ordonnee(s.origine)+ordonnee(s.extremite))/2;
10
11  retourner(unPoint(xMilieu, yMilieu, noir, 1.0);
12 fin
```

où abscisse, ordonnée et unPoint sont des opérations du type Point (cf Moodle)

2.5 Processus d'élaboration d'un TAD

2.5.1 Étape

1. Énumérer l'ensemble des opérations du type
2. Pour chaque opération, préciser son profil (nom de l'opération, domaine d'entrée et domaine de sortie)
3. lister l'ensemble des propriétés du type
4. Définir un entête de sous-programme pour chacune des opérations du type
5. Choisir une représentation mémoire pour coder les opérations et les valeurs du type
6. Coder avec la représentation mémoire choisie les corps des différents sous-programmes

Remarque

Les étapes 1 à 3 sont relatives au type abstrait. (formalisme)

Les étapes 4 à 6 concernent les types concrets (programmation)

L'étape 3 permet :
- De donner la sémantique des opérations du type
- D'aider au codage des opérations du type concret
- De définir des jeux de tests pour ces opérations

Chapitre 3

L'implémentation d'un Type Abstrait de Données

3.1 Implémentation d'un type concret

Définition Mise en œuvre informatique de la spécification algorithmique du type concret.
Tâche du ressort du programmeur du type.
L'implémentation doit respecter la spécification (spécification = cahier des charges pour le programmeur)

3.1.1 Tâche du programmeur

1. Choisir une représentation mémoire pour coder les opérations du type
2. Coder les corps des sous-programmes conformément à la spécification.
3. Regrouper au sein d'un module la représentation mémoire et le codage des opérations.

3.1.2 Conteneur d'un module d'implémentation

Un module peut contenir :
– des déclarations de constantes
– des déclarations de types dont l'un au moins correspond à la définition du type étudié.
– les corps des sous-programmes définies par la spécification.
– tout sous-programmes nécessaire à la mise en œuvre du type.

3.1.3 Exemple

1

Pour le TAD Point en représentation statique.

```
type Point : enregistrement
  abscisse <Reel>,
  ordonnee <Reel>,
  couleur <Couleur>,
  taille <Reel>;

-- construit un point d'abscisse x, d'ordonnee y, de couleur c et de taille t
-- necessite t > 0
fonction unPoint(entree x <Reel>, entree y <Reel>, entree c <Couleur>, entree t <Reel>)
      retourne <Point>
      declenche tailleInvalide

glossaire
  p <Point>; -- point retourne par la fonction

debut
  si t >= 0 alors
    declencher (tailleInvalide);
  fin si;
  p.abscisse <- x;
  p.ordonnee <- y;
  p.couleur <- c;
  p.taille <- t;
```

```
  retourner(p);
fin

fonction pointOrigine
  retourne <Point>
debut
  retourner(unPoint(0.0, 0.0, 1.0));
fin

-- fournit la taille d'un point p
fonction taille (entree p <Point>) retourne <Reel>
debut
  retourner(p.taille);
fin

--modifie la taille du point par la nouvelle taille t
procedure modifierTaille(maj p <Point>, entree t <Reel>)
  declenche tailleInvalide
debut
  si t <= 0 alors
    declencher (tailleInvalide);
  fin si;
  p.taille <- t;
fin
-- cf Moodle pour le module d'implementation complet
```

3.1.4 Implémentation du TAD Point en représentation dynamique

En représentation statique :
En représentation dynamique
(Un pointeur vers un enregistrement) Dans cette représentation dynamique

```
type Point : pointeur sur <EnrPoint>;
type EnrPoint enregistrement
  abscisse <Reel>,
  ordonnee <Reel>,
  couleur <Couleur>,
  taille <Reel>;

-- construire un point d'abscisse x, d'ordonnee y, de couleur c, de
    taille t
-- Necessite t > 0
fonction unPoint (entree x <Reel>, entree y <Reel>, entree c <
   Couleur>, entree t<Reel>)
      retourne <Point>
      declenche tailleInvalide

glossaire
  p <Point> -- point retourne par la fonction /!\ pointeur

debut
  si t <= 0 alors
```

```
    declencher (tailleInvalide);
  fin si;
  allouer(p);
  p↑.abscisse <- x;
  p↑.ordonne <- y;
  p↑.couleur <- c;
  p↑.taille <- t;
  retourner(p);
fin
-- fonction pointOrigine identique au code de pointOrigine en
   representation statique (cf exemple 1)

--fournit la taille d'un point p
fonction taille (entree p <Point>)
  retourne <Reel>
debut
  retourner(p↑.taille);
fin

-- modifier la taille d'un point p par la valeur t comme nouvelle
   taille
-- Necessite t > 0
procedure modifierTaille(maj p <Point>, entree t <Reel>)
  declenche tailleInvalide
debut
  si t <= 0 alors
    declencher(tailleInvalide);
  fin si;
  p↑.taille <- t
fin
-- ⋮
-- cf Moodle pour l'implementation complete en representation
   dynamique
```

3.2 Sémantique de valeur et sémantique de référence

Dans le type concret, deux nouvelles opération s'ajoutent : l'affectation(<-) et l'égalité(=).

Problème Quelle signification (sémantique) donner à ces opérations pour un client ?

Exemple Pour un client

```
glossaire
  p1 <Point>;
  p2 <Point>;
debut
  ⋮
  p1 <- unPoint(1.0,2.0,rouge,1.0); -- ???
  ⋮
```

```
  p2 <- p1; -- ???
fin
```

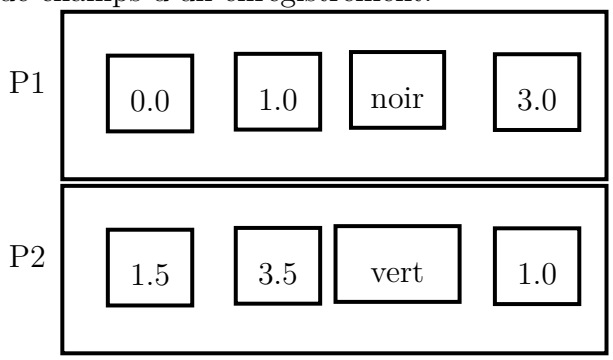
3.2.1 Sémantique de valeurs

Définition Une affectation x<-y à une sémantique de valeurs si le conteneur (la valeur) de y est recopiée dans x.

Remarque Avec une sémantique de valeur, toute opération de modification sur y après l'affectation y<-y n'affecte pas la valeur de x.

Exemple

Soit le TAD Point en représentation statique. (c'est-à-dire un enregistrement)
Cette représentation statique à une sémantique de valeur pour le TAD Point Car la copie (<-) et la comparaison (= et /=) de deux enregistrements travaillant sur les valeur est un enregistrement. de champs d'un enregistrement.




```
p1 <- p2; -- copie
```

```
glossaire
  pt <Point>;
  pr <Point>;

debut
  p2 <- unPoint(1.5, 3.5, vert, 1.0);
  p1 <- p2
  modifierTaille(p2, 3.5);
  ecrire(taille(p2));
  ecrire(taille(p1));
fin
```

On obtient l'affichage :
3.5
1.0

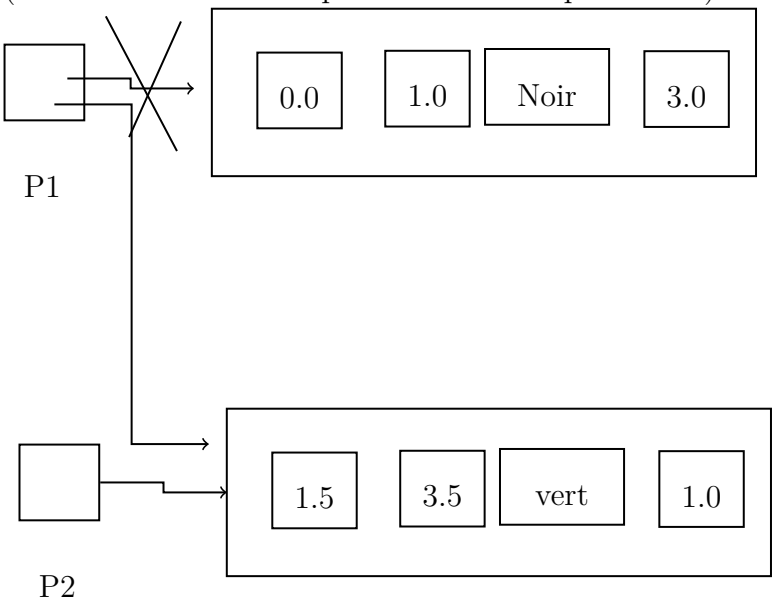
3.2.2 Sémantique de référence

Définition L'affectation x <- y désigne le même contenu mémoire pouvant être référencé a la fois par x et par y.

Remarque Après l'affectation x <- y avec un sémantique de référence toute modification sur y se répercute sur x !
On dit que x et y sont des alias pour désigner le même contenu.

Exemple

Soit le TAD Point en représentation dynamique (c'est-à-dire par un pointeur vers un enregistrement).
⇒ Sémantique de référence pour le TAD Point
(Car affectation et comparaison de deux pointeurs !)




```
p1 <- p2; -- copie
```

```
glossaire
  pt <Point>;
  pr <Point>;

debut
  p2 <- unPoint(1.5, 3.5, vert, 1.0);
  p1 <- p2
  modifierTaille(p2, 3.5);
  ecrire(taille(p2));
  ecrire(taille(p1));
fin
```

On obtient à l'affichage
3.5
3.5

Remarque

1- Avec une représentation par pointeur (et donc avec un sémantique de référence) on peut définir un type de sémantique de valeur
⇒ ajouter dans le TAD une opération de copie et une opération de comparaison.
Pour le TAD Point en représentation dynamique on définit

```
procedure copier(sortie p1 <Point>, entree p1 <Point>)
```

```
debut
  p1 ↑ <- p2 ↑; -- copie des enregistrement
fin

fonction egal(entier p2 <Point>, entree p1 <Point>)
    retourne <Booleen>
debut
  retourner (p1 ↑ = p2↑)
fin
```

2- En C, un tableau est représenté par un pointeur constant vers son premier élément.
⇒ Affectation de deux tableaux n'est pas autorisée !
Comparaison de deux tableaux sont autorisés : comparaisons de deux adresses (différent de la comparaison des élément des deux tableaux).

3.3 Exportation des opérateurs <-, = et /=

3.3.1 Implémentation sans exportation des opérateurs

La spécification du TAD limite les opérations en interdisant l'usage de l'affectation et des comparaisons.
⇒ la spécification n'inclut pas d'en tête pour <-, = et /=

Exemple

Pour le TAD Point en implémentation statique ou dynamique.
Pas d'indication de la spécification vis-à-vis des opérateurs implique l'interdiction au client d'utiliser les opérateurs. (voir spécification du TAD du chapitre 2)

```
glossaire
  p1 <Point>;
  p2 <Point>;
debut
  p1 <- unPoint(1.0, 0.0, noir, 3.0); -- Interdit
  p2 <- p1; -- Interdit
  --Entraine une erreur a la compilation, operation non definie.
fin
```

Listing 3.1 – Pour un client

Remarque Le statut d'exportation permet de protéger les données d'un type abstrait (renforce l'encapsulation)
Par exemple soit le TAD compteInformatique représente

```
type compteInformatique enregistrement
  motDePasse <Chaine>,
  fichier <Liste[Fichier]>;
```

3.3.2 Implémentation sans exportation

La spécification du type abstrait avec un en-tête par opérateur selon la syntaxe pour le type T.

```
-- t1 <- t2
procedure "<-" (sortie t1 <T>, entree t2 <T>);

-- t1 = t2
fonction "=" (entree <T>, entree t2 <T>)
  retourne <Booleen>;
```

Listing 3.2 – Spécification

```
glossaire
  p1 <Point>;
  p2 <Point>;
debut
  p1 <- p2; --ou "<-" (p1, p2) Autorise :-)
    ⋮
    si p1 = p2 alors -- :-)
      ⋮
    fin si;
fin
```

Listing 3.3 – Cotès client

Pour l'implémentation du type deux possibilités.

- Pas de corps pour l'opérateur si la sémantique donné par le langage correspond à celle du type.
- Écriture d'un corps pour l'opérateur à la sémantique du langage ne correspond plus à celle du type abstrait

3.3.3 Synthèse

	Implémentation sans exportations	Implémentation avec exportations
Utilisation (client)	opérateur non définie	Opérateur définie
Spécification (concepteur)	Pas d'entête pour l'opérateur	Entête pour l'opérateur
Implémentation (programmeur)	Pas de corps pour l'opérateur (il n'y a pas d'entête)	pas de corps opérateur supporté par le langage. Présence d'un corps, redéfinition de l'opérateur

Remarque

1- On peut accorder un statut d'exportation différent selon l'opérateur.

2- L'opérateur inégalité est toujours défini implicitement à partir de l'opérateur égalité.

3.4 Type fonctionnelle V.S type impératif

3.4.1 Type fonctionnelle

Type qui ne propose que des fonctions algorithmique. En général par d'affectation.

3.4.2 Type impératif

Type qui possède au moins une opération de modification avec une procédure et un mode mise à jour pour une variable du type. En général l'affectation est autorisée.