3 Les enregistrements

En informatique, un *type abstrait* (en anglais, *abstract data type* ou ADT) est une spécification théorique d'un ensemble de données et de l'ensemble des opérations qu'on peut effectuer sur elles. On qualifie d'abstrait ce type de données car il correspond à un cahier des charges qu'une structure de données doit ensuite mettre en œuvre.

3.1 Exemple : le type abstrait `complexe`

Nous souhaitons représenter un nombre complexe $z \in \mathbb{C}$ par deux réels a (la partie – champ `re`) et b (la partie imaginaire – champ `im`) : $z = a + ib$:

1. **Type abstrait** : `complexe`
2. **Type abstrait importé** : `réel`
3. **Opérations primitives** :
   - Constructeurs :
     - `zero` : $\longrightarrow$ `complexe`
     - `ecrire_re` : `réel` $\times$ `complexe` $\longrightarrow$ `complexe`
     - `ecrire_im` : `réel` $\times$ `complexe` $\longrightarrow$ `complexe`
   - Accès
     - `lire_re` : `complexe` $\longrightarrow$ `réel`
     - `lire_im` : `complexe` $\longrightarrow$ `réel`
   - Axiomes
     [1] $\texttt{lire\_re}(\texttt{zero}) = 0$
     [2] $\texttt{lire\_re}(\texttt{ecrire\_re}(\texttt{a}, \texttt{z})) = \texttt{a}$
     [3] $\texttt{lire\_re}(\texttt{ecrire\_im}(\texttt{b}, \texttt{z})) = \texttt{lire\_re}(\texttt{z})$
     [4] $\texttt{lire\_im}(\texttt{zero}) = 0$
     [5] $\texttt{lire\_im}(\texttt{ecrire\_re}(\texttt{a}, \texttt{z})) = \texttt{lire\_im}(\texttt{z})$
     [6] $\texttt{lire\_im}(\texttt{ecrire\_im}(\texttt{b}, \texttt{z})) = \texttt{b}$

Une fois ce type abstrait défini, nous pouvons définir des opérations non primitives (et suivre les étapes (1) à (3) de la démarche algorithmique décrite dans le premier chapitre). Par exemple, considérons l'opération calculant le conjugué d'un nombre complexe (on rappelle que si $z = a + ib \in \mathbb{C}$ avec $a, b \in \mathbb{R}$, alors le conjugué de z est noté $\overline{z}$ et est défini par : $\overline{z} = a - ib$).

(1) Son profil est `conjugue` : `complexe` $\longrightarrow$ `complexe`

(2) À titre d'exemple, si $\texttt{z} = \texttt{a} + \texttt{ib}$, avec $\texttt{a} = \texttt{lire\_re}(\texttt{z})$ et $\texttt{b} = \texttt{lire\_im}(\texttt{z})$, alors $\texttt{conjugue}(\texttt{z}) = \texttt{a} - \texttt{ib} = \texttt{ecrire\_im}(-\texttt{lire\_im}(\texttt{z}), \texttt{z})$.

(3) Cette opération ne nécessite qu'un seul axiome :

[1] $\texttt{conjugue}(\texttt{z}) = \texttt{ecrire\_im}(-\texttt{lire\_im}(\texttt{z}), \texttt{z})$.

Exercice 1

Pour chacune des opérations suivantes, donnez son profil, un exemple (si cela vous semble utile) et un jeu d'axiome(s) :

1. L'opération calculant le module d'une nombre complexe : $|a + ib| = \sqrt{a^2 + b^2}$.
2. L'opération calculant la somme de deux nombres complexes : $(a_1 + ib_1) + (a_2 + ib_2) = (a_1 + a_2) + i(b_1 + b_2)$.
3. L'opération calculant le produit de deux nombres complexes : $(a_1 + ib_1) \times (a_2 + ib_2) = (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + a_2 b_1)$.
4. L'opération donnant l'inverse d'un nombre complexe : si $z \neq 0$, $\dfrac{1}{z} = \dfrac{\overline{z}}{|z|^2}$.
5. L'opération calculant le rapport de deux nombre complexes : si $z_2 \neq 0$, alors $\dfrac{z_1}{z_2} = z_1 \times \dfrac{1}{z_2}$.
6. L'opération qui au réel positif ρ et au réel θ associe le nombre complexe de module ρ et d'argument θ (rappel : sa partie réelle est $\rho \cos\theta$, sa partie imaginaire est $\rho \sin\theta$).

3.2 Manipulation algorithmique d'enregistrements

La manipulation d'un champ `c` d'un enregistrement `x` peut se faire via la notation `x.c`. Dans l'exemple précédent de du type `complexe`, nous pourrions, par exemple, manipuler la partie réelle directement à partir de la syntaxe `z.re`.

Dans ce cours, nous tâcherons à utiliser le moins possible cette notation. Nous ne l'utiliserons que pour définir (algorithmiquement ou en C), les opérations de lecture et d'écriture. -

```
1  fonction ecrire_re(a : réel, z : complexe) : complexe
2  début
3  |  z.re ← a
4  |  retourner z
5  fin
6  fonction ecrire_im(a : réel, z : complexe) : complexe
7  début
8  |  z.im ← a
9  |  retourner z
10 fin
11 fonction lire_re(a : réel, z : complexe) : complexe
12 début
13 |  retourner z.re
14 fin
15 fonction lire_im(a : réel, z : complexe) : complexe
16 début
17 |  retourner z.im
18 fin
```

Une fois ces opérations primitives définies, la notation avec un point sera à éviter le plus possible (application du principe d'encapsulation qui sera décrit et justifié à la section 3.4). Ainsi, si on veut décrire un algorithme de la fonction `conjugué`, on peut écrire :

```
1 fonction conjugue(z : complexe) : complexe
2 Variable :
3 c : complexe début
4 |  c ← zero /* initialisation */
5 |  c ← ecrire_re(lire_re(z), c) /* La partie réelle de c est la partie réelle de z */
6 |  c ← ecrire_im(−lire_im(z), c) /* La partie imaginaire de c est l'opposée de la partie imaginaire de z */
7 |  retourner c
8 fin
```

Exercice 2

Donnez un algorithme pour chacune des opérations non primitives introduites dans l'exercice précédent.

3.3 Implantation des enregistrements en C

```c
/*************************/
/* complexe.c.           */
/* auteur : Jean Lieber. */
/* date : 07/02/14       */
/*************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

```

```c
#define PARTIE_REELLE_PAR_DEFAUT 0.
#define PARTIE_IMAGINAIRE_PAR_DEFAUT 0.

struct Complexe
{
  double re ;
  double im ;
} ; // Ne pas oublier ce point-virgule !

typedef struct Complexe complexe ;

/* SIGNATURES DES OPERATIONS PRIMITIVES */
/* Constructeurs et */
complexe zero () ;
complexe ecrire_re (double a, complexe z) ;
complexe ecrire_im (double b, complexe z) ;
/* Acces */
double lire_re (complexe z) ;
double lire_im (complexe z) ;

/* IMPLANTATION DES OPERATIONS PRIMITIVES */
/* Constructeurs */
complexe ecrire_re (double a, complexe z)
{
  z.re = a ;
  return z ;
}
complexe ecrire_im (double b, complexe z)
{
  z.im = b ;
  return z;
}

complexe zero ()
{
  complexe z ;
  z = ecrire_re (PARTIE_REELLE_PAR_DEFAUT, z) ;
  z = ecrire_im (PARTIE_IMAGINAIRE_PAR_DEFAUT, z) ;
  return z ;
}

/* Accès */
double lire_re (complexe z)
  {
  return z.re ;
}

double lire_im (complexe z)
{
  return z.im ;
}

/* OPERATIONS NON PRIMITIVES */

complexe conjugue (complexe z)
{
  complexe c ;
  c = zero() ;
  c = ecrire_re (lire_re (z), c) ;
  c = ecrire_im (-lire_im(z), c) ;
  return c ;
```

```
}

void afficher_complexe (complexe z)
{
  double a, b ;
  a = lire_re(z) ;
  b = lire_im (z) ;
  printf ("%f %s %fi",
    a, (b < 0. ? "-" : "+"), fabs(b)) ;
  // fabs : valeur absolue
}
```

Prog. 3.1 – Une implémentation du type abstrait `complexe`

3.4 Le principe d'encapsulation appliqué aux enregistrements

L'encapsulation est un principe de programmation issu de la programmation par objets mais qui peut s'appliquer à des langages de programmation qui ne sont pas conçus à l'origine pour ce type de programmation, tel que `C`. Appliqué à un type enregistrement, il consiste à masquer, pour l'*utilisateur* de ce type, la façon dont le développeur de ce type l'a encodé. Si on suit ce principe, cela permet de modifier la façon dont un type est représenté sans que son utilisation, dans un autre programme, par exemple, soit modifiée.

Voyons comment ce principe s'applique au type `complexe`. Le développeur du type a décrit dans un premier temps ce type comme dans les sections ci-dessus : un `complexe`. est donné par un enregistrement à deux champs représentant les parties réelle et imaginaire. Un utilisateur de ce type respectant le principe d'encapsulation n'accèdera à ce type qu'à travers les fonctions d'écriture et de lecture (correspondant, dans le type abstrait, aux constructeurs et aux accès). Il pourra, par exemple, effectuer ainsi une implantation de l'opération `conjugué` (comme présenté ci-dessus) et plein d'autres opérations. Supposons à présent que le développeur du type décide de changer l'implantation. Par exemple, il décidera d'encoder désormais un nombre complexe par un enregistrement à deux champs : le module et l'argument.

Une fois cette modification faite, il redéfinira les opérations `zéro`, `lire_re`, `écrire_re`, `lire_im` et `écrire_im` (qui ne seront plus des opérations primitives) sur la base des nouvelles opérations primitives, par exemple :

```
double lire_re (complexe z)
{
  return lire_module (z) * cos(lire_argument (z)) ;
}
```

Prog. 3.2 – Une implémentation du type abstrait `complexe`

Si le principe d'encapsulation a été respecté, il n'y aura pas de changement pour l'utilisateur du type. En revanche, si à un moment, l'utilisateur n'a pas respecté le principe d'encapsulation et a écrit `z.re`, alors son code ne fonctionnera plus.

Exercice 3

On veut utiliser les enregistrements pour représenter les types suivants :

- Type `point2D` des points du plan, un point étant donné par une abscisse (un réel), une ordonnée (un réel) et le « point par défaut » étant l'origine (le point d'abscisse et d'ordonnée nulles).
- Le type `triangle` des triangles du plan, un triangle étant donné par trois sommets, de type `point2D`. Les sommets du triangle par défaut sont tous l'origine (on a donc un triangle réduit à un point).
- Le type `appartement` des appartements, un appartement étant donné par son aire en mètres carrés (un réel), un nombre de pièces (un entier naturel), un étage (un entier relatif – pour considérer le cas des appartements en sous-sol) et le fait que l'appartement est actuellement occupé ou non (un booléen). L'appartement par défaut fait 20 m^2, contient 2 pièces, est au 8^e étage et est occupé.

Pour chacun de ces types, suivez la démarche du cours et donnez :

- Une description du type abstrait (nom, type(s) abstrait(s) importé(s), profils et axiomes des opérations primitives) ;
- Un algorithme pour chacune des opérations primitives ;

- Une implantation en `C` du type, y compris la traduction des opérations primitives et une procédure pour afficher une valuer du type.

Exercice 4

On considère les types définis dans l'exercice précédent et on demande de définir les opérations non primitives suivantes (jeux d'axiomes, algorithmes, traduction en `C`) :

- `distance` qui à deux points associe leur distance (on considèrera la distance euclidienne canonique, cette distance est donc $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ où x_1 et x_2 sont les abscisses des deux points et où y_1 et y_2 sont leurs ordonnées) ;
- `égaux_point2D` qui teste si deux points sont égaux ;
- `périmètre` qui donne le périmètre d'un triangle ;
- `égaux_triangle` qui teste si deux triangles sont égaux (i.e., leurs ensembles de sommets sont les mêmes, même s'ils ne sont pas représentés dans le même ordre ; par exemple, si $A = Z$, $B = X$ et $C = Y$ alors les triangles ABC et XYZ sont égaux).