Chapitre 4

Seconds éléments de AsmL

Nous avons vu au chapitre trois les premiers éléments du langage AsmL, l'essentiel, à savoir les structures de contrôle. Comme pour tout langage impératif moderne, AsmL possède des structures de données prédéfinies et est un langage orienté objet permettant de définir des structures de données abstraites héritées, une fois de plus, de .NET. C'est cet aspect que nous allons étudier dans ce chapitre, bien qu'il n'interviendra pas pour la partie théorique.

4.1 Les structures de données prédéfinies

4.1.1 Les fichiers

Les méthodes d'accès aux fichiers.- Comme nous l'avons déjà dit, les ASM ne s'intéressent pas à l'interaction avec l'extérieur. Les méthodes d'accès aux fichiers sont dérivées de l'architecture .NET. Il y en a trois :

```
public ReadFile(fileName as String) as String
public WriteFile(fileName as String, content as String)
public AppendFile(fileName as String, content as String)
```

dont la sémantique est naturelle.

Exemple.- Écrivons un programme AsmL qui affiche à l'écran le contenu de la première ligne d'un fichier (texte) :

```
var F = "fichier.asml"
var s as String

Main()
  step
    s := ReadFile(F)
  step
    WriteLine(s)
```

4.1.2 Les types énumérés

Syntaxe.- Un type énuméré est défini de la façon suivante :

```
enum nom
 element1
 element2
 ...
 elementn
```

ou :

```
enum nom extends type
 element1
 element2
 ...
 elementn
```

si on veut préciser le type des items.

Exemple.- Utilisons un type énuméré pour réécrire un de nos exemples précédents :

```
// enum.asml
enum mode
  calcul
  affichage

Main()
  initially C as Integer = 19
  initially F as Integer = 0
  initially m = calcul
  step until fixpoint
    if m = calcul then
      F := (9*C)/5 + 32
      m := affichage
    if m = affichage
      WriteLine(F)
```

4.1.3 Les types structurés

Syntaxe.- Un type structuré est défini de la façon suivante :

```
structure nom
 champ1 as type1
 champ2 as type2
 ...
 champn as typen
```

Pour indiquer un champ, on utilise l'opération point comme en langage C.

Exemple.- Définissons un item d'un répertoire téléphonique comportant un nom et un numéro de téléphone :

```
// phone.asml

structure item
  nom as String
  phone as String

signal as item = item("Patrick", "123")
var I as item = item("","")

Main()
 step while I <> signal
  step
    Write("Nom : ")
  step
    I.nom := ReadLine()
  step
    Write("Num\u00E9ro de t\u00E9l\u00E9phone : ")
  step
    I.phone := ReadLine()
  step
    WriteLine(I.nom + " : " + I.phone + "\n")
```

4.1.4 Les ensembles

Un **ensemble** modélise un ensemble fini au sens mathématique, c'est-à-dire une collection dans laquelle l'ordre des éléments n'a pas d'importance.

Constructeur de types.- Le type composé ensemble est obtenu en utilisant le constructeur :

Set of *type*

4.1.4.1 Les ensembles définis en extension

Syntaxe.- Une constante ensemble défini en extension sera décrite en utilisant des accolades ouvrante et fermante et en séparant les éléments par des virgules, autrement dit comme en mathématiques.
Les opérations fondamentales sont :

add *expression* **to** *ensemble*
remove *expression* **from** *ensemble*

Exemple.- Déclarons un ensemble, initialisé comme ensemble vide, auquel on ajoute et on enlève des éléments :

```
// ensemble.asml

var Etudiants as Set of String = {}

Main()
  step
    WriteLine("Etudiants = " + Etudiants)
    add "Paul" to Etudiants
    add "Pierre" to Etudiants
  step
    WriteLine("Etudiants = " + Etudiants)
    remove "Paul" from Etudiants
    add "Arthur" to Etudiants
  step
    WriteLine("Etudiants = " + Etudiants)
```

4.1.4.2 Itération sur les collections

Exemple.- Avec les entités tels que les ensembles, on peut utiliser une nouvelle sorte d'itération (évidemment définissable) :

step foreach *variable* **in** *expression instruction*

avec un ordre non déterminé du parcours de l'ensemble.

Exemple.- Reprenons l'exemple précédent en affichant de façon différente :

```
// ensemble2.asml

var Etudiants as Set of String = {}

Main()
  step
    WriteLine("Etudiants = " + Etudiants)
    add "Paul" to Etudiants
    add "Pierre" to Etudiants
  step foreach s in Etudiants
     WriteLine(s)
```

4.1.4.3 Ensembles définis en compréhension

Syntaxe.- Le contenu d'un ensemble peut être caractérisé par un intervalle d'éléments d'un type ordinal (les entiers et les caractères), en spécifiant le premier élément et le dernier, ces deux éléments étant séparés par deux points (comme en langage Pascal).

Ils peuvent aussi être décrits par une propriété caractéristique en utilisant le symbole "|".

La relation binaire d'appartenance est `in`.

Exemple.- Définissons un ensemble caractérisé par un intervalle, puis un ensemble défini par une propriété caractéristique :

```
// ensemble3.asml

var A = {1..20}
var B = {i | i in A where i*i in A}

Main()
    WriteLine("B = " + B)
```

4.1.4.4 Autres opérations

Syntaxe.- On a les opérateurs `union` de réunion, `intersect` d'intersection et la méthode `Size()` qui renvoie le cardinal d'un ensemble.

Exemple.- Écrivons un programme AsmL qui utilise ces opérations et cette méthode :

```
// ensemble4.asml

A = {1, 5, 6, 9}
B = {1, 6, 7, 10}

Main()
  WriteLine("A U B = " + (A union B))
  WriteLine("A inter B = " + A intersect B)
  WriteLine("card(A) = " + Size(A))
```

4.1.5 Les listes

Notion.- Une **liste** (*sequence* en anglais) est une collection dans laquelle l'ordre des éléments est important.

Constructeur de types.- L'ensemble des listes dont les éléments sont du type *type* est dénoté par :

Seq of *type*

Listes définies en extension.- Une constante liste sera décrite en utilisant des crochets ouvrant et fermant et en séparant les éléments par des virgules, comme dans l'exemple suivant :

```
//liste1.asml

liste = [5, 5, 4, 3, 2, 1]

Main()
  WriteLine("liste = " + liste)
```

Itération sur les collections.- Dans le cas des listes, l'itération tient compte de l'ordre des éléments, comme le montre l'exécution du programme suivant :

```
//liste2.asml

liste = [5, 5, 4, 3, 2, 1]

Main()
  step foreach i in liste
    WriteLine(i)
```

Indexation d'un élément.- On peut accéder directement à un élément d'une liste, grâce à son index, le premier élément ayant 0 comme index, comme le montre l'exemple suivant :

```
//liste3.asml

liste = [5, 5, 4, 3, 2, 1]

Main()
  WriteLine("liste[2] = " + liste(2))
```

Autres opérations.- On se reportera à [GT02] pour les autres opérations.

4.1.6 Les tableaux

Ils sont émulés en considérant une liste et un ensemble d'index.

4.1.7 Les applications

Notion.- Une **application** (*map* en anglais) associe une valeur à une **clé** (*key* en anglais). L'ensemble des clés possibles est le **domaine** (*domain* en anglais). L'ensemble des valeurs possibles est l'**image** (*range* en anglais).

Constructeur de types.- L'ensemble des applications de A dans b est dénoté par :

Map of A **to** B

Applications définies en extension.- Une constante application sera décrite en utilisant des accolades ouvrante et fermante, en séparant les éléments par des virgules et en utilisant une flèche `->` pour associer une valeur à une clé, comme dans l'exemple suivant :

```
// map1.asml

var phone as Map of String to Integer = {"Bob" -> 100, "Carla" -> 101}

Main()
  WriteLine("Les entr\u00E9es sont " + phone)
  WriteLine("Le num\u00E9ro de Carla est " + phone("Carla"))
```

Applications définies en compréhension.- Une application peut également être définie par une propriété caractéristique, en utilisant la syntaxe suivante :

{ i -> f(i) | in S where *condition* }

comme le montre l'exemple suivant :

```
// map2.asml

var f as Map of Integer to Integer = {i -> i*i | i in {1..10}}

Main()
  WriteLine("f(2) = " + f(2))
```

4.2 Les classes

4.2.1 Classes et objets

Notion.- Nous supposons que le lecteur a déjà suivi une introduction à la programmation orientée objet. Une **classe** (*class* en anglais) est définie par des *champs* ou *membres* (*fields* et *members* en anglais) qui sont soit des **attributs** (*attributes* en anglais), soit des **méthodes**. Un élément de la classe est une **instance** (même mot en anglais) de celle-ci, qui est un **objet** (*object* en anglais).

Constructeur de types.- Une classe se déclare de la façon suivante :

```
class nom
  var champ1 as type1
  ...
  var champn as type n

  methode1
  ...
  methodep
```

Instantiation.- Il est inutile de déclarer un objet. On instantie une classe de la façon suivante :

```
o = new nom(c1, ..., cn)
```

en spécifiant une valeur pour chaque attribut (non déjà initialisé lors de la définition de la classe). On pourra changer la valeur de ces attributs ensuite, aucun n'étant privé.

Accès à un membre.- Comme en langage C, on accède à un membre en utilisant l'opérateur point :

```
nom.champ
```

comme le montre l'exemple suivant :

```
// point.asml

class Point
  var x as Integer
  var y as Integer

Main()
  step
    p = new Point(1,2)
  step
    WriteLine("p = " + (p.x, p.y))
```

Exemple avec méthodes.- L'exemple suivant nous montre comment utiliser les méthodes :

```
// point2.asml

class Point
  var x as Integer
  var y as Integer

  deplace(a as Integer, b as Integer)
    x := x + a
```

```
    y := y + b

  affiche()
    WriteLine("(" + x + ", " + y + ")")

Main()
  step
    p = new Point(1,2)
  step
    p.deplace(2,3)
  step
    Write("p = ")
  step
    p.affiche()
```

Passage des paramètres.- En AsmL, tout est objet. Un objet est passé par valeur mais on peut changer les valeurs de ses attributs (qui sont passés par référence) comme le montre l'exemple suivant :

```
// point3.asml

class Point
  var x as Integer
  var y as Integer

  affiche()
    WriteLine("(" + x + ", " + y + ")")

mult(a as Integer, p as Point)
  p.x := a*p.x
  p.y := a*p.y

Main()
  step
    p = new Point(1,2)
  step
    mult(2,p)
  step
    Write("p = ")
  step
    p.affiche()
```

Si on modifie la fonction `mult()` de la façon suivante :

```
mult(a as Integer, p as Point)
  p.x := a*p.x
  p.y := a*p.y
  a := a*a
```

on obtient une erreur dès la compilation :

```
error: cannot update: local cannot be updated
```

4.2.2 Héritage

Notion.- Une classe peut hériter d'une autre classe. La seconde est alors la **classe de base** et la première la **classe dérivée**. Comme en langage Java, AsmL ne permet que l'héritage simple.

Syntaxe.- La déclaration :

class A **extends** B

où B est une classe déjà définie, signifie que les attributs et les méthodes définies pour les instances de la classe B le sont aussi pour celles de la classe A, comme le montre l'exemple suivant :

```
// point4.asml

class Point
  var x as Integer
  var y as Integer

  deplace(a as Integer, b as Integer)
    x := x + a
    y := y + b

  affiche()
    WriteLine("(" + x + ", " + y + ")")

class PointColore extends Point
  var couleur as Integer

  affiche()
    WriteLine("(" + x + ", " + y + ") de couleur " + couleur)

Main()
  step
    p = new PointColore(1,2,3)
  step
    p.deplace(2,5)
  step
    Write("p = ")
  step
    p.affiche()
```

On remarquera au passage la surchage de la méthode `affiche()`.

4.2.3 Structures de données dynamiques

Notion.- Comme dans les langages C++ et Java, les structures de données dynamiques sont construites comme **structures auto-référentes**, c'est-à-dire dont un attribut est du type de la structure ou de la classe.

Syntaxe.- Pour un type `type` donné, le type :

`type?`

dénote les élément du type `type` plus la valeur `null`, qui veut dire indéfinie.

Exemple.- L'exemple suivant montre comment on définit l'une des structures de données dynamiques les plus simples, celle de liste chaînée :

```
// listeChainee.asml

class Sommet
  var n as Integer
  var suivant as Sommet?

class List
  var premier as Sommet?

  estVide() as Boolean
    if (premier = null) return true
    else return false

  insert(a as Integer)
    if estVide()
      s = new Sommet(a, null)
      premier := s
    else
      s = new Sommet(a, premier)
      premier := s
```

```
  affiche()
    var index = premier
    step
      Write("{ ")
    step until index = null
      Write(index.n + " ,")
      index := index.suivant
    step
      Write("}")

Main()
  var  L = new List(null)
  step
    L.insert(3)
  step
    L.insert(9)
  step
    L.insert(5)
  step
    Write("Liste = ")
  step
    L.affiche()
```

Exercices

Exercice 1.- *Écrire un programme AsmL qui demande le nom d'un fichier (texte), lit un texte au clavier et le place dans ce fichier. Le texte ne comportera pas le symbole '#' qui sera la valeur signal pour la fin du texte.*

Exercice 2.- (**Copie de fichier**)

Écrire un programme AsmL qui demande le nom d'un fichier source (existant), le nom d'un fichier but, puis qui copie le contenu du fichier source dans le fichier but.

Exercice 3.- (**Concaténation de fichiers**)

Écrire un programme AsmL qui copie dans un fichier le concaténaté de deux autres fichiers.

Exercice 4.- (**Arbre binaire**)

Un **arbre binaire** *est une structure de données dynamique dont chaque élément, appelé* **sommet**, *a une valeur et deux éléments du même type (appelés* **fils gauche** *et* **fils droit**). *L'élément de départ est la* **racine** *et un élément sans aucun fils (pointeur à nul) est une* **feuille**.

- 1^o) Définir, en langage AsmL, le type `sommet` pour un arbre binaire dont les valeurs sont des entiers.

- 2^o) Définir, en langage AsmL, le type `arbre`, comprenant une méthode qui renvoie la somme des valeurs des feuilles.

- 3^o) Écrire de programme de test.

Exercice 5.- (**Consommation**)

- 1^o) *Écrire une classe AsmL* `Car` *dont les attributs sont la comsommation (un réel spécifiant la consommation en litre par 100 kilomètres) et le niveau de carburant (en litre, réel également). Les méthodes sont* `addGas()` *(avec un argument entier spécifiant le nombre de litres de carburant ajouté),* `getGas()` *(sans argument, renvoyant un netier spécifiant le nombre de litres de carburant disponible) et* `drive()` *(avec un argument entier spécifiant le nombre de kilomètres à parcourir, ce qui a une conséquence sur le niveau de carburant disponible).*

- 2^o) *Écrire un programme AsmL complet permettant de tester la classe définie ci-dessus.*

Exercice 6.- (**Compte bancaire**)

- 1^o) *Définir une classe* **Compte** *(pour compte bancaire) dont les attributs sont un numéro de compte (un entier naturel) et le solde (un réel) et dont les méthodes sont une fonction de dépôt, une fonction de retrait (toutes les deux ont un argument réel), une fonction d'affichage du solde ainsi qu'une fonction de transfert (d'un compte à un autre d'une certaine somme).*

- 2^o) Écrire un programme AsmL complet permettant de tester la classe définie ci-dessus.

Exercice 7.- (**Arithmétique modulaire**)

- 1^o) *Définir une classe* `ModClass` *des entiers congrus modulo 15 (soit* $\mathbb{Z}/15.\mathbb{Z}$*). Le seul attribut* `val` *est un entier de 0 à 14. Les méthodes sont une méthode d'affichage et les opérateurs +, - et *.*

- 2^o) *Écrire un programme AsmL permettant de tester cette classe.*

Exercice 8.- (**File d'attente**)

Une **file d'attente** *est une structure de donnée dynamique qui stocke les données et les traite dans l'ordre d'arrivée (premier arrivé, premier servi). Seules deux opérations sont permises pour accéder à celles-ci :* `placer()` *ajoute un item en queue de la file d'attente et* `enlever()` *retire un item en tête de la file d'attente.*

- 1^o) Définir une classe `queue` de file d'attente d'entiers.

- 2^o) Écrire un programme AsmL qui permette de saisir un certain nombre d'entiers strictement positifs, la valeur sentinelle étant 0, puis un entier naturel n. Le programme affichera les entiers saisis depuis le n-ième jusqu'à la fin (le premier entier entré a pour numéro 0).

Les entiers saisis seront sauvegardés dans une file d'attente.

Exercice 9.- (**Deque**)

Une **deque** *(pour l'anglais Double-Ended QUEue) est une structure de données linéaire pour laquelle on peut ajouter et retirer un élément à l'un ou l'autre bout. Une deque comprend les méthodes* `push_t()` *(pour ajouter un élément la fin),* `pop_t()` *(pour enlever et récupérer un élément à la fin),* `push_b()` *(pour ajouter un élément au début),* `pop_b()` *(pour enlever et récupérer un élément au début),* `isEmpty()` *(pour savoir si la deque est vide, et donc qu'on ne peut pas récupérer d'élément) et* `isFull()` *(pour savoir si la deque est pleine, et donc qu'on ne peut plus ajouter d'élément).*

Concevoir une classe AsmL `Deque` dont les éléments sont des caractères.

Exercice 10.- (**Nombres rationnels**)

On veut implémenter une classe pour les nombres rationnels. Le numérateur et le dénominateur seront des entiers.

- 1^o) *Définir une classe* `Rat` *avec les méthodes d'affichage, d'addition, de soustraction et de multiplication.*

- 2^o) *Écrire un programme AsmL complet permettant de tester l'implémentation de cette classe.*

Exercice 11.- (**Temps**)

- 1^o) *Concevoir une classe* `Temps` *dont les attributs sont trois entiers représentant les heures (de 0 à 23), les minutes (de 0 à 59) et les secondes (également de 0 à 59) et possédant les méthodes de mise à l'heure globale (*`ajusterTemps()`*) à trois arguments, de l'heure uniquement (*`ajusterHeure()`*), de la minute uniquement (*`ajusterMinute()`*), de la seconde uniquement (de nom* `ajusterSeconde()`*), chacune à un argument, et les méthodes d'affichage globale (*`afficheTemps()`*), de l'heure uniquement, de la minute uniquement et de la seconde uniquement.*

On devra vérifier à chaque fois que les données sont valides et les rejeter si elles ne sont pas dans les intervalles voulus.

- 2^o) *Concevoir une classe dérivée* `Temps2` *possédant en plus la méthode* `tic()` *incrémentant d'une seconde le temps stocké dans un objet de la classe* `Temps`*. Bien entendu la seconde, la minute et l'heure doivent être ajustées en conséquence : après un tic de 10 : 59 : 59 on doit se retrouver avec 11 : 00 : 00.*

Exercice 12.- (**Date**)

- 1^o) *Concevoir une classe* `Date` *dont les attributs sont trois entiers représentant le jour, le mois et l'année et possédant une méthode d'affichage et une méthode* `jourSuivant()` *pour incrémenter le jour d'une unité.*

[Ne tenez pas compte du problème des années bissextiles.]

- 2^o) *Concevoir à nouveau les classes* `Temps` *et* `Date` *de façon à ce qu'elles possèdent en plus une méthode* `tac(Temps, Date)` *incrémentant d'une seconde le temps stocké dans l'objet de la classe* `Temps` *et éventuellement d'un jour l'objet de la classe* `Date`.

Exercice 13.- (**Pile**)

Une **pile** *est une structure de données abstraites contenant des entités d'un type donné mais on ne peut y accéder que par le sommet de la pile. Une pile s'initialise (nouvelle pile ne contenant rien), possède un test, pour savoir si elle est vide ou non, et deux méthodes* `empile()` *(on ajoute un élément au sommet de la pile) et* `depile()` *(on enlève un élément si la pile est non vide).*

- 1^o) Concevoir une classe `Pile` dont les éléments sont des caractères.

- 2^o) Utiliser la classe `Pile` précédemment définie pour l'implémentation d'une méthode `inverse()` d'une chaîne de caractères.

[*Pour inverser une chaîne de caractères, il suffit de placer ses caractères l'un après l'autre dans une pile, puis de les récupérer l'un après l'autre.*]

Exercice 14.- (**Vecteurs**)

Écrire une classe `Vecteur` *de l'espace comportant comme attributs trois coordonnées (réelles) et comme méthodes :* `add()` *pour additionner deux tels vecteurs,* `homothetie()` *pour multiplier les coordonnées par un réel fourni en argument,* `scal()` *pour obtenir le produit scalaire de deux vecteurs,* `vect()` *pour obtenir le produit vectoriel de deux vecteurs et* `affiche()` *pour afficher les coordonnées du vecteur sous la forme d'un triplet.*

Exercice 15.- (**Nombres complexes**)

Écrire une classe `Complex` *pour la manipulation des nombres complexes comportant comme attributs deux coordonnées réelles (la partie réelle et la partie imaginaire) et comme méthodes :* `affiche()` *sous la forme cartésienne* $a + i.b$, `trigo()` *pour afficher sous la forme trigonométrique* $\rho.exp(i.\theta)$, `add()` *pour additionner deux complexes,* `sub()` *pour soustraire deux complexes,* `mult()` *pour multiplier un nombre complexe par un nombre réel,* `prod()` *pour multiplier deux nombres complexes,* `div()` *pour diviser deux nombres complexes (le second étant non nul),* `module()` *pour calculer le module d'un nombre complexe et* `conj()` *pour calculer le conjugué d'un nombre complexe.*

Exercice 16.- (**Arbre binaire de tri**)

Un **arbre binaire** *est une structure de donnée dynamique définie récursivement : on part d'un nœud appelé* **racine**, *chaque nœud comprend une donnée, un* `fils gauche` *et un* **fils droit**, *chacun étant vide ou étant lui-même un nœud. Un nœud sans fils est appelé une* **feuille**.

- 1^o) Définir la classe `noeud` des éléments d'un arbre binaire d'entiers naturels en AsmL, qui comprendra trois attributs (un entier, qui est sa valeur, et deux nœuds : le fils gauche et le fils droit).

- 2^o) Définir la classe `arbre` en AsmL, qui comprendra comme attribut la racine qui est un nœud.

[On ne peut pas faire en faire grand chose pour l'instant, mais on va améliorer ces classes.]

Dans un **arbre de tri**, *on insère les éléments récursivement de la façon suivante : on a une liste d'entiers (non trié) sans doublon ; on va placer un nœud à*

l'arbre par entier ; le premier entier est placé dans la racine ; pour chaque entier suivant, si l'entier est strictement inférieur à la valeur de la racine, un nœud est ajouté à gauche à la bonne place, s'il est supérieur à la racine, il est ajouté à droite. Par exemple si on a l'arbre :

```
          5
     3          19

1       4    10    23
```

et que l'on veut ajouter 12, on obtient l'arbre :

```
          5
     3          19

1       4    10    23

               12
```

- 3^o) a) Ajouter la méthode `inserer()` (un entier) à la classe `noeud` ayant un argument entier d : si d est strictement inférieur à la valeur du nœud et si le fils gauche est nul, on remplace ce fils gauche par une feuille de valeur d, si le fils gauche est non nul, on insère (récursivement) la valeur d à ce fils gauche ; si d est strictement supérieur à la valeur du nœud, on a un traitement analogue pour le fils droit.

b) Ajouter la méthode `inserer()` (un nœud) à la classe `Arbre` dont l'argument est un entier : si la racine est nulle, la racine devient une feuille dont la valeur est cet entier ; sinon on insère cette valeur entière à la racine (en tant que nœud).

La **traversée en ordre** *d'un arbre binaire consiste à afficher les valeurs de ses nœuds récursivement de la façon suivante : si non vide traversée en ordre du fils gauche, racine, traversée en ordre du fils droit.*

On voit que, pour un arbre binaire de tri, on affiche ainsi les éléments dans l'ordre.

- 4^o) Ajouter une méthode `traversee()` à la classe `Arbre`, qui affiche les éléments sur une ligne. Cette méthode pourra faire appel à une méthode auxiliaire d'argument un nœud, qu'on appellera `assistant()`.

- 5^o) Écrire un test qui insère les valeurs 34, 23, 67, 12, 56, 3 et 5 à un arbre de tri et qui traverse cet arbre, c'est-à-dire qui trie la liste d'entiers ci-dessus.