Módulos

La **modularidad** es la capacidad que tiene un sistema de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de éstas partes, una tarea necesaria (e independiente) para conseguir dicho objetivo.

Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de módulo.

Idealmente un módulo debe poder cumplir las condiciones de caja negra, es decir, ser independiente del resto de los módulos y comunicarse con ellos (con todos o sólo con una parte) a través de unas entradas y salidas bien definidas

Cada uno de los módulo de un programa idealmente debería cumplir las siguientes características:

- **Tamaño pequeño**.- Facilita aislar el impacto que pueda tener la realización de un cambio en el programa, bien para corregir un error o por rediseño del algoritmo correspondiente.

- **Independencia modular**.- Cuanto más independientes son los módulos entre sí más fácilmente se trabajará con ellos, esto implica que para desarrollar un módulo no es necesario conocer detalles internos de otros módulos.

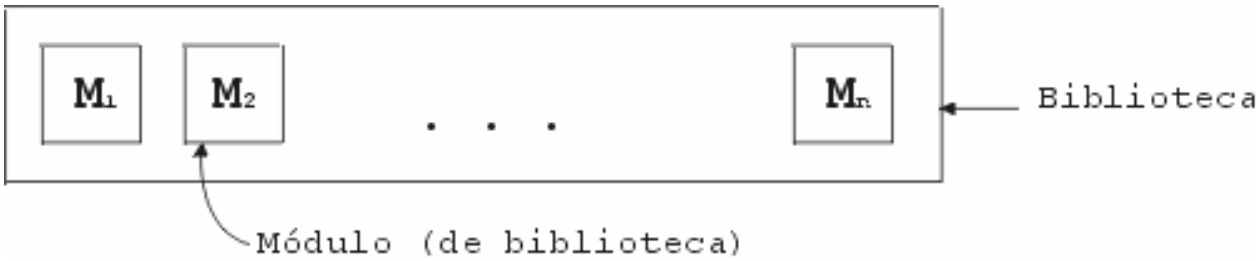
Cómo consecuencia de la independencia modular, un módulo cumplirá:

- Características de caja negra, es decir **abstracción.**
- Aislamiento de los detalles mediante **encapsulamiento**.

Se persigue crear programas modulares (que cumplen o tienen la característica de **modularidad**).Con esto se consigue desarrollar programas a partir de un conjunto de módulos, cada uno de los cuales desempeña una tarea necesaria para el correcto funcionamiento del programa global. En el caso de que el conjunto de módulos implicado esté sometido a una jerarquía, estaríamos hablando de un sistema de programación por capas, en la que cada "capa" representaría un nivel en la jerarquía de los módulos.

- Ej,: presentación, lógica.

Los programas de alto nivel usan en general funciones y procedimientos "de biblioteca“. Las bibliotecas se organizan en módulos (de biblioteca).



Un módulo contiene un número de entidades de exportación.

Entidades pueden ser:

- Tipos
- Constantes
- Variables
- Procedimientos y funciones

Exportación: disponibles para ser utilizados (importados) por otros módulos. Cada módulo tiene un nombre.

Los "programas principales“ (main) en C son llamados también módulos.

Se distinguen:

- Módulos de biblioteca
- Módulo principal

Las entidades declaradas son las exportadas por el módulo, y pueden ser:

- Tipos
- Constantes
- Variables
- Subprogramas (sólo se declara el cabezal y no el cuerpo)

Ejemplo:

- void readCaracter(char);

En general, basta declarar la mínima información necesaria para verificar el uso correcto de la entidad sólo desde el punto de vista sintáctico. Cada entidad declarada en el módulo de definición es implementada en el correspondiente módulo de implementación. En particular, en este último es donde se proveen los cuerpos de los subprogramas exportados. En el proceso de compilación de un módulo principal.

El compilador usa la información de los módulos de definición. El linker usa los módulos de implementación.

Técnica: Divide & Vencerás

- Se descompone el programa en módulos.
- Debe acordarse cuál es la comunicación entre módulos
  - Qué entidades importan/exportan.
  - En el caso de los procedimientos, especificar QUÉ deben hacer (y no es necesario conocer CÓMO).

Los módulos pueden desarrollarse y compilarse separadamente. Para obtener una versión ejecutable debe efectuarse el proceso de linking. Cambios en la implementación de entidades importadas no obligan a recompilar todo el sistema. Sólo se recompila el módulo de implementación y se reconecta todo el sistema. Naturalmente, los "códigos incompletos" generados por el compilador son también mantenidos en archivos.

Tipos Abstractos de Datos (TAD)

Abstracción

1. Ignorar detalles
2. Posponer la consideración de detalles

1. En ciencias experimentales

Se observan fenómenos y se crean modelos abstractos que explican aspectos de esos fenómenos.

Ejemplo: Leyes de Kepler sobre el movimiento planetario.

- Los planetas y el sol se reducen a puntos
- Se consideran sólo propiedades del movimiento de los planetas (modelo cinético)
- Se ignoran masa, composición química, etc.

2. En programación

Tenemos un lenguaje para expresar programas.

Si deseamos que los programas sean ejecutados por computadoras, entonces deberán finalmente ser expresados en lenguaje de máquina.

En estos lenguajes, las primitivas son elementales y por ello los programas deben ser especificados a un nivel de extremo detalle, lo cual los hace muy complejos.

Tratamos entonces de construir programas por refinamientos sucesivos.
Los programas se expresan en términos de componentes que no están expresados directamente en lenguaje de máquina. Luego se aplica el mismo procedimiento a cada componente, refinándolo, hasta alcanzar el nivel de detalle requerido.

- Un lenguaje de programación de "alto" nivel de abstracción es aquel cuyas primitivas pueden ser refinadas de modo mecánico a lenguaje de máquina. El programa que efectúa ese refinamiento es el compilador del lenguaje.
- Ejemplo: programa que lee coordenadas de vértices de un triángulo y calcula el área de éste.

-- Podemos expresarlo en un lenguaje de "alto nivel" como C de la siguiente forma:

```
#ifndef AreaTriang
#define AreaTriang

      < importaciones >
      < declaraciones (tipos, variables, constantes,
      prototipos)>

      < instrucciones (funciones, procedimientos) >
#endif
```

Esta es la estructura general de un programa. Contiene componentes abstractos, (escritos entre < ... >) a ser refinados.

Podemos comenzar introduciendo el tipo de datos que representará el concepto de punto (en coordenadas cartesianas):

```
< declaraciones >
      typedef struct Punto{
                  float x, y;

        }
      < ... >
```

y luego las variables del programa:

```
< declaraciones ... >
      Punto   p1, p2, p3;
< ... >
```

Luego refinamos la parte de instrucciones:

```
< instrucciones >
      < Leer p1, p2, p3 >;
      < Desplegar AreaTri (p1, p2, p3) >
< ... >
```

Donde estamos haciendo uso de "instrucciones abstractas"

Abstractas porque no están (todavía) dadas al nivel concreto de detalle del Lenguaje de programación. Pero podemos especificar su efecto con precisión.

Otra forma de decir lo mismo:

son instrucciones (se puede especificar su efecto) pero no son instrucciones concretas del lenguaje.

- El problema original (construir un programa) se resuelve por medio de una estructura cuyos componentes son o bien instrucciones concretas o bien otros programas a ser refinados separadamente.

  El método permite acotar el nivel de detalle a ser considerado cada vez.

- Otra forma de decir lo mismo: se usan abstracciones para particionar el problema dado. Luego se refinan las abstracciones siguiendo el mismo método.

  En el ejemplo introducimos una función (AreaTri) que hay que refinar:

```
< declaraciones ... >
float AreaTri (Punto p1, Punto p2, Punto p3)
{< declaraciones AreaTri ... >

      < Calcular base = distancia (p1, p2) >;
      < Calcular altura = distancia p3 a p1p2 >;
      return base * altura / 2.0;
}//END AreaTri;
```

- El uso de instrucciones abstractas se llama:
  - ABSTRACCION de PROCEDIMIENTO
  - SUBPROGRAMAS
  - PROCEDIMENTAL
  - (Inglés: PROCEDURAL ABSTRACTION)

  y es la forma más elemental de abstracción.

- Algunas instrucciones abstractas se refinan en subprogramas (procedimientos, funciones) en Pascal, Modula, C,…

  Otras simplemente en instrucciones que sustituyen textualmente a la instrucción abstracta.

Abstracción de datos

Nos interesa extender la idea de abstracción a los TIPOS de DATOS.

Es decir, usar TIPOS ABSTRACTOS de DATOS además de instrucciones abstractas, para diseñar programas.

¿Qué es un tipo abstracto de datos?

Es un tipo de datos, es decir, puede ser especificado como tal con precisión, pero no está dado como un tipo de datos concreto del lenguaje. Esto es: no está dado en términos de los constructores de tipo del lenguaje.

(En C:

tipos elementales: int, short, double, char , “POINTER TO T “(*)
tipos estructurales: Arreglo_Tipo[…], struct, tipos de estructuras dinámicas(uniones).)

¿Cómo se introduce un Tipo Abstracto de Datos?

Básicamente: dándole un nombre y asociando a él un número de operaciones aplicables a los objetos del tipo. En C, estas operaciones son subprogramas (procedimientos, funciones).

- Un TAD es un modelo matemático, junto con operaciones definidas sobre ese modelo.

- Encapsulan la definición y todas las operaciones con este tipo.

- Ej.: Conjuntos de números enteros con las operaciones de unión, intersección y diferencia.

- Se separa la especificación del tipo (que hace), de la implementación (como lo hace).

¿Cómo se refina un Tipo Abstracto de Datos?

Definiéndolo como un tipo concreto del lenguaje y refinando en forma acorde los subprogramas asociados.

Usualmente se habla de

ESPECIFICACION del T.A.D.
    DEFINICION
    correspondiendo a su introducción
y de
IMPLEMENTACION del T.A.D.
    correspondiendo a su refinamiento.

En este contexto, el tipo concreto provisto para definir el T.A.D. en una implementación dada se llama una REPRESENTACION del T.A.D.

Hay una analogía con las estructuras algebraicas:

Ej: monoide denomina una clase de estructuras, formadas por un conjunto, un elemento *x* del conjunto y una operación asociativa que tiene a *x* como neutro.

Podemos decir: un conjunto y unas operaciones que cumplen ciertas leyes.

Por otra parte, las secuencias finitas con la operación de concatenación y la secuencia vacía forman un monoide concreto.

La analogía es entre monoide concreto e implementación del concepto de monoide y entre éste y el "tipo abstracto" monoide.

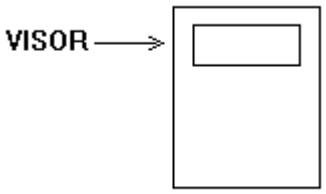
En general, tipo abstracto corresponde a clase de estructura algebraica, mientras implementación del tipo abstracto se corresponde con una estructura concreta que satisface la especificación de la clase.

Uso de Tipos Abstractos En C/C++

Veamos un caso de diseño de programa usando un tipo abstracto de datos.

PROBLEMA: Escribir un programa que simule una calculadora con la que se pueda operar con números racionales, dados en la forma $^{m}/_{n}$ con m, n enteros (n <> 0).

Pensamos en el modelo de una calculadora común, que consta de un visor donde se muestra el último resultado computado.



Con ese número se puede operar, sumando, restando, multiplicando o dividiendo el mismo con otro número, lo cual da lugar a un nuevo resultado que reemplaza al anterior.

También es posible borrar el último resultado o reemplazarlo por otro que se ingresa directamente.

El programa a construir aceptará ***comandos***, posiblemente con ***argumentos*** correspondiendo a cada una de las operaciones enumeradas.

Contestará a cada uno de esos comandos con el resultado, simulando de esta forma el visor de la calculadora.

He aquí un diseño del programa:

```
#ifndef CalcRacionales
#define CalcRacionales

< importaciones PP >
< declaraciones PP >

      < instrucciones PP >
#endif//END CalcRacionales.
```

Comenzamos por introducir las variables (ej. la estructura de datos del programa).

Necesitamos una variable para almacenar el último resultado computado (es decir, la variable que simula el visor de la calculadora).

¿Qué tipo tendrá esa variable?

Deberá almacenar números racionales.

Este tipo no es un tipo de datos primitivo del lenguaje. Ahora bien, queremos continuar el diseño del programa sin detenernos a pensar en este momento cómo representar números racionales.

El método será: usar el tipo Racional como un tipo abstracto, a ser refinado posteriormente.

Las operaciones asociadas a este tipo abstracto serán aquellas que vayamos detectando como necesarias en el transcurso del diseño del programa.

¿Cómo se especifica un T.A.D. en C/C++?

Se escribe un módulo de definición:

```
#ifndef Racionales
#define Racionales
typedef struct TipoRacional* Racional;
< declaraciones de operaciones sobre Racionales >
#endif Racionales.
```

La declaración typedef Racional introduce el nombre del T.A.D.

¡¡¡Luego iremos agregando especificaciones de las operaciones que sean necesarias. !!! Nótese que sólo introducimos el nombre del nuevo tipo y NO lo definimos en términos de constructores primitivos del lenguaje,se dice que el tipo Racional es opaco.

O sea: en el módulo de definición sólo especificamos el tipo, no lo implementamos.

Ventajas de este mecanismo:

Nos permite trabajar a un mayor nivel de abstracción, esto es: posponer el problema de implementar este tipo.

En general existirán varias implementaciones posibles y habrá que estudiarlas en detalle para optar por una de ellas. No queremos involucrarnos en ese proceso ahora.

El programa principal, como se verá, no dependerá de la implementación que oportunamente se escoja, sino sólo de la especificación del tipo.

Por lo tanto, modificar la implementación del tipo no implica tener que modificar el programa principal.

Ahora refinamos el programa principal, importando el (nombre de) tipo recién introducido.

```
< importaciones PP >
#include “Racionales.h”
 < operaciones necesarias >
< ... >
```

Ahora podemos declarar variables de tipo Racional en el programa principal. De hecho necesitamos dos de ellos, a saber:

una para el "visor"
otra para almacenar cada operando a ser combinado con el "visor" por medio de las operaciones aritméticas

```
< declaraciones PP >
< tipos, procedimientos, constantes PP >

//Variables

      Racional visor, opnd;


< ... >
```

Ahora necesitamos considerar los comandos aceptados por el programa. Introducimos el tipo de los comandos como una enumeración:

```
< tipos, procedimientos, constantes PP >
typedef enum {suma, resta, producto, división, borrar, ingreso,
fin} Comando;
< ... >
```

y declaramos una variable de este tipo para almacenar cada comando ingresado por el usuario.

```
< declaraciones PP .. >
 Comando c;
< ... >
```

Vamos ahora a refinar la parte de instrucciones del programa principal:

```
< instrucciones PP >
      < inicialización >;
      do{
            LeerComando (c);
            switch ( c ) {
                  case suma: < proceso suma >...break;
                  case resta: < proceso resta > ...break;
                  case prod: < proceso producto > ...break;
                  case divi: < proceso división > ...break;
                  case borrar: < proceso borrado > ...break;
                   case ingr: < proceso ingreso >...break;

            };
            < mostrar visor si corresponde >
      While(c ¡= fin);
      < terminación >
< ... >
```

Al refinar los varios procesos correspondientes a los comandos, obtendremos las operaciones requeridas para el tipo Racional.

```
< proceso suma >
      LeerRacional (opnd);
      SumaRacional (opnd, visor);
< ... >
```

Aquí LeerRacional lee un número racional en la variable dada como parámetro y Suma Racional suma el primer parámetro al segundo, modificando éste.

La resta, producto y división funcionan de manera similar. Veamos los otros casos:

```
< proceso borrado >
      BorrarRacional (visor)
< ... >

< proceso ingreso >
      LeerRacional (visor)
```

```
< ... >
```

Finalmente, mostramos el visor luego de procesado cada comando, excepto el de finalización:

```
< mostrar visor si corresponde >
      if comando ¡= fin
       EscribirRacional (visor)
< ... >
```

La inicialización no hace otra cosa que borrar el visor y desplegar mensajes apropiados. Esto último es lo único que hace por su parte la terminación.
Es decir, que hemos ya detectado todas las operaciones que el programa principal necesita usar en relación a los racionales.

Estas operaciones deberán ser importadas del módulo Racionales:

```
< operaciones necesarias >
SumaRacional, RestaRacional,
ProductoRacional, DivisionRacional,
BorrarRacional,
LeerRacional, EscribirRacional
< ... >
```

Todos estos procedimientos deben ser declarados en el módulo de especificación Racionales, y sus efectos deben ser especificados con precisión:

```
< declaraciones de operaciones sobre Racionales >
< operaciones básicas >
void SumaRacional (  Racional q, Racional& r);
/* precondición: q es un racional válido Q, r es un racional
válido R ó está borrado
   postcondición: r = Q + R ó r borrado si lo estaba
originalmente */
.
.
```

y similarmente se especifica el resto.

Las operaciones básicas mencionadas antes son:

la que permite formar racionales a partir de un numerador y un denominador enteros. (CONSTRUCTORA)
la que permite obtener el numerador y el denominador de un racional dado (DESTRUCTORAS/SELECTORAS).
la que permite detectar si una variable racional está o no "borrada" (definida o vacía) (PREDICADO)

Estas no son directamente referenciadas por nuestro programa principal, pero:

Serán útiles en general para otros programas que manipulen racionales.
Serán aquellas cuya implementación manipule directamente la representación que se elija para el tipo abstracto.
En particular, las operaciones ya declaradas arriba pueden definirse en términos de estas operaciones básicas.

Declaramos, por lo tanto:

```
< operaciones básicas >
Racional CrearRacional (int m, int n);
/* precondición: n <> 0. Retorna: m/n */

int Numerador (Racional q);
/* Precondición: q es un racional válido. Retorna: el numerador
de q */

int Denominador (Racional q);
/* Precondición: q es un racional válido. Retorna: el denomiador
de q */

bool EsVálido (Racional q);
< ... >
```

En este punto, el programa principal y el módulo Racionales pueden desarrollarse en forma separada. La comunicación entre ambos ha quedado precisamente establecida. Si se compila el módulo de especificación de racionales, entonces el programa principal puede ser compilado también pues sólo depende del módulo de especificación.

En forma independiente puede desarrollarse y compilarse el módulo de implementación de los racionales. Esto favorece el trabajo en grupo que puede operar en paralelo una vez definidos los módulos de especificación necesarios.

Por el lado del programa principal, resta solamente implementar el procedimiento de lectura de comandos. Veamos ahora cómo se puede refinar el TAD Racional. Debe escribirse el módulo de implementación Racionales:

```
//modulo de imlementación MODULE Racionales;
#include “Racionales.h”

struct TipoRacional {
  < representación elegida >;

}
< implementaciones de procedimientos >
```

Los tipos (abstractos) que se introducen sólo con su nombre en el módulo de definición se denominan **OPACOS**.

En la implementación deben ser definidos como punteros a variables que tendrán como tipo la presentación que se elija dar al tipo abstracto. Esto NO limita las posibilidades de representación: sólo agrega un nivel de "indirección" en el contexto del módulo de implementación -- Trabajamos con punteros a las representaciones en lugar de con las representaciones mismas.
(La imposición es necesaria para que el compilador pueda alojar las variables del tipo abstracto declaradas en el programa principal, manteniendo la independencia de éste de la representación que se elija).
En nuestro caso podemos tener:

```
< representación elegida >
struct TipoRacional{
      int num, denom;
}
< ... >
```

en cuyo caso el estado "borrado" puede ser representado como el puntero NULL ó bien un registro con variantes, una de las cuales corresponde al estado "borrado".

He allí pues dos posibles representaciones entre las cuales es posible optar.

Un ejemplo de implementación de la suma de racionales (como procedimiento):

```
void SumaRacional (Racional q,  Racional r){
      //declaración de variables
      unsigned int numq, denomq, numr, denomr;

      if (EsValido ( r )){
          numq = Numerador(q); denomq = Denominador(q);
          numr = Numerador(r); denomr = Denominador ( r );
          r->num = numq * denomr + denomq + numr;
          r->denom = denomq * denomr;
          Normalizar(r);
      }
}//END Suma Racional;
```

El procedimiento Normalizar calcularía la forma simplificada del racional dado como parámetro.
Puede ser local al módulo de implementación (aunque en este caso sería de considerar la idea de hacerlo exportable).
Notar que la implementación de SumaRacional podría desarrollarse sin acceder a la representación de Racional, haciendo uso de las operaciones selectoras (Numerador y Denominador) y la constructora (CrearRacional).

```
void SumaRacional (Racional q,  Racional r){
      //declaración de variables
      unsigned int numq, denomq, numr, denomr;

      if (r != NULL){
          numq = q->num; denomq = q->denom;
          numr = r->num; denomr = r->denom;
          AsignarNumerador (r , numq * denomr + denomq + numr);
          AsignarDenominador ( r , denomq * denomr);
          Normalizar(r);
      }
}//END Suma Racional;
```

¿Qué ventajas y desventajas tiene esta alternativa ?

Ventaja: entre otras no es necesario conocer la representación del TAD.

Conclusiones:

- El mecanismo de módulos permite especificar tipos abstractos durante el proceso de diseño de programas.
- Provee un método conveniente de abstracción y refinamiento.
- !!! Los programas que usan tipos abstractos permanecen independientes de la implementación de éstos. Las representaciones son opacas para los programas, lo cual garantiza:
- que los programas manipulen los objetos del tipo abstracto SOLO A TRAVES DE LAS OPERACIONES DEFINIDAS (consistencia en el uso del tipo).
- que los programas no deben modificarse si la implementación del tipo abstracto se modifica.
- Se tiene un mecanismo de módulos que favorece el trabajo en paralelo.

Conceptos Generales

Veremos algunos conceptos generales sobre TADs y, sobre todo, ciertos TADs particulares que son importantes en Programación

Repaso:

Un TAD se introduce, define o especifica dando su nombre y las operaciones que pueden aplicarse a sus objetos.

Esto se llama la ESPECIFICACION del TAD.

Un TAD se refina o implementa dando una representación y refinando correspondientemente las operaciones.

Esto se llama la IMPLEMENTACION del TAD.

En C/C++:

La especificación de un TAD se escribe en un módulo de definición. El nombre del tipo se declara sin que sea definido (tipo opaco). Las operaciones se especifican como cabezales de procedimientos (funciones o procedimientos que computan por efecto) indicando precondición y postcondición o valor retornado por medio de comentarios.

La implementación del TAD se escribe en el correspondiente módulo de implementación. La representación se da en la definición del tipo introducido en términos de un tipo concreto del lenguaje (de hecho, debe ser un tipo puntero) y luego se escriben las correspondientes implementaciones de los procedimientos.

Listas

Hemos visto la definición inductiva de listas.

Más precisamente, definimos inductivamente los tipos Alista, para cualquier tipo A (que es el tipo de los elementos de las listas en cuestión).

Los tipos Alista se definen inductivamente por medio de dos constructores:

[ ] : Alista

__ · __ : A * Alista -> Alista

En esta notación, indicamos los tipos de las operaciones: qué argumentos requieren y qué tipo de objeto producen. Se entiende entonces que, por definición, las Alistas son todos aquellos objetos que pueden ser formados combinando estas dos operaciones.

Ahora tratamos de presentar las listas como un TAD.

La primera observación es que la definición inductiva dada arriba no introduce un tipo, sino más bien un formador de tipos:

lista: Tipo -> Tipo

o, dicha de otra manera, todos los tipos: Alista para cualquier tipo A.

En C no podemos introducir formadores de tipo sino solamente tipos individuales. No tendremos un módulo para el formador de tipo lista sino módulos individuales para listas de cardinales, de enteros, de valles, de registros de diverso tipo, ... Todos similares entre sí, pero escritos separadamente y con diferentes nombres.

Consideremos entonces listas de cardinales. Vamos a introducirlo como un tipo abstracto. Al hacerlo, obtendremos un patrón para construir módulos que introduzcan los tipos abstractos de Listas de elementos de cualquier tipo. Estos módulos pueden obtenerse del patrón que veremos por medio de una edición simple del texto.

El módulo de definición lo escribimos como sigue:

```
#ifndef LNat
#define LNat;

……

typedef struct NodoLNat* LNat;

typedef unsigned int Natural;

….
#endif
```

¿Qué operaciones tendremos?

Un conjunto mínimo queda dado por el siguiente criterio:

CONSTRUCTORAS: si elegimos escribirlas directamente como funciones (especificación funcional) entonces tenemos una traducción simple de la definición inductiva.

```
LNat Vacía();
/* retorna la lista vacía */

LNat Cons (Natural n, LNat s);
/* retorna la lista n.s */
```

El uso de LNat como tipo opaco hace que las únicas operaciones en que pueden usarse variables o valores del tipo LNat sean las invocaciones a los procedimientos que se importen de módulos de biblioteca (como el que estamos definiendo).

En particular, para determinar si una lista es vacía o no, debemos aplicar a ella un procedimiento.

Este es un caso que se da con generalidad en tipos inductivos con más de un constructor: cada constructor corresponde a un caso o forma de elemento del tipo inductivo.

Para determinar si un elemento es de una cierta forma, usamos una función booleana.

Estas son además casos de PREDICADOS sobre los elementos del tipo abstracto (corresponden con
propiedades de los elementos). Todas tienen un único argumento que es el tipo abstracto.

En el caso de las listas, hace falta un predicado para decidir entre las dos formas posibles de listas: vacía o no vacía:

```
< ··· operaciones >
```

```
bool esVacía (LNat s);
/* retorna TRUE si s es vacía */

< ... >
```

Finalmente, debemos poder acceder a la información contenida en las listas. Sólo las listas no vacías
contienen información y ésta es, naturalmente, el valor de la cabeza y la cola de la lista.

Para descomponer listas no vacías usamos dos funciones.

Estos son dos casos de las usualmente llamadas (operaciones)

SELECTORAS:

```
< ... operaciones >

Natural Cabeza (LNat s);
/* precondición: s no vacía, retorna: la cabeza (primer
elemento) de s */

LNat Cola (LNat s);
/* precondición: s no vacía, retorna: la cola de s */

< ... >
```

Así concluye la especificación de las listas de cardinales como tipo abstracto. El método de selección de las operaciones: constructoras, predicados, selectoras puede aplicarse a todo tipo definido inductivamente.

El resultado es una especificación suficiente, en el sentido que toda otra operación sobre listas puede definirse en términos de las primitivas enumeradas arriba.

Ejemplos:

```
LNat Snoc (LNat s, Natural x){

      if esVacía (s)
       return Cons(x, s)
      else
            return Cons(Cabeza(s), Snoc(Cola(s), x) )

}
```

Funciones Recurrentes

IMPLEMENTACIONES de LISTAS:

Introducción:

Como se ha dicho, una implementación de un TAD consiste en un tipo de estructura de datos concreto que se elige como representación del TAD y las correspondientes implementaciones de los procedimientos.

En el caso de las listas especificadas arriba, la implementación natural se basa en la representación por medio de listas encadenadas.

Entonces escribimos el módulo de implementación (LNat.c, ó LNat.cpp):

```
//Implementación del modulo LNat (Lista de Naturales);
#include "LNat.h"

struct NodoLNat{

                Natural cabeza;
                LNat cola;
              }
< implementacion de operaciones >
```

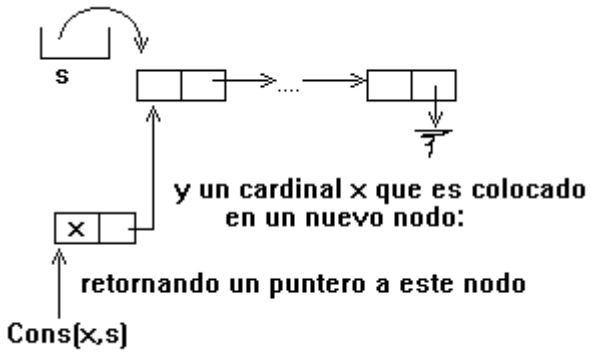
Veamos algunas ideas sobre las implementaciones.

Elegimos para representar la lista vacía la constante NULL (podriámos haber escogido otra), por lo tanto la constructora Vacía retorna NULL.

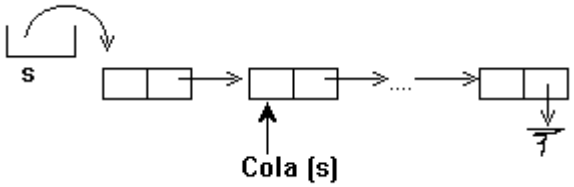
El predicado esVacía puede escribirse simplemente

```
bool esVacía (LNat s){
      return ( s == NULL )
}
```

La constructora Cons **recibe una lista s:**



La selectora Cola retorna un puntero a la cola:



Esta implementación es DINAMICA. El tamaño de la memoria ocupada crece con las listas. Para eliminar nodos hace falta en este caso un procedimiento adicional que borra enteramente una lista

```
void DisposeCardLista (LNat &s);
/* borra o elimina todos los nodos de s */
```

Notar que:

La función Cola no elimina el nodo cabeza, lo cual permite programar fácilmente procedimientos que recorren listas en forma secuencial sin perder la lista de entrada.

DisposeCardLista no es una función. Debe necesariamente operar por efecto.Observar que la lista s se pasa como referencia.

Otra implementación pensable para las listas con la especificación dada es por medio de vectores.

Una lista representada en un vector tiene necesariamente el tamaño acotado.

En rigor, se trataría de otro tipo abstracto: listas de tamaño acotado.

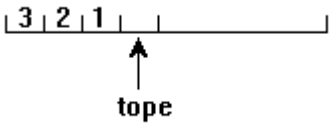
Esto se refleja en que el módulo correspondiente debería exportar una CONSTANTE igual al máximo tamaño posible de las listas y, deseablemente, también un predicado para verificar si una lista dada está o no llena.

Entonces:

```
#ifndef CardListasAcotados;
#define CardListasAcotados;


        typedef struct NodoCardListaAcot * CardListaAcot;
        const LARGO_MAX = 100;
        .
        .
#endif
```

La idea es representar las listas en vectores, con los elementos en orden inverso, lo cual facilita la implementación de Cons:

La lista [1, 2, 3] se representa:



y debe agregarse un índice al primer lugar disponible.

En el módulo de implementación escribimos:

```
struct NodoCardListaAcot{
            Natural elems[LargoMax];
            unsigned int tope;//indica el indice
            //que contiene el ultimo elemento
            //de la lista
};
```

Las implementaciones de los procedimientos pueden hacerse como ejercicio. Notar que Vacía, Cons y Cola deben crear nodos del tipo ReprVec.

La implementación por medio de vectores es ESTATICA.

No sirve, estrictamente hablando, para listas cualesquiera (de largo no acotado).

Y desperdicia espacio, en todos los casos de listas de largo menor que el máximo posible.

Otra especificación de listas:

La especificación anterior es mínima. Con esas operaciones como primitivas, se puede programar toda otra función sobre listas.

La especificación dada tiene también como ventaja que se formula directamente siguiendo un método generalizable a cualquier tipo inductivo.

Sin embargo, es conveniente disponer de más operaciones sobre listas. En particular, las listas se distinguen de otras estructuras lineales que veremos (pilas, colas) por admitir inserción y borrado de elementos en cualquier posición.

Con los módulos que hemos escrito, podemos agregar procedimientos de biblioteca sobre listas por la vía de escribir nuevos módulos. Por ejemplo:

//XLNAT.h Ó XLNAT.hpp

```
#ifndef XLNAT
#define XLNAT

#include “LNat.h”

LNat Snoc (LNat s, Natural x);
.
.
#endif XLNat;
```

y correspondientemente (módulo implementación XLNAT.c ó XLNAT.cpp):

```
#include “LNat.h”;

LNat Snoc (LNat s, Natural x)

{

…

…

}
```

En este módulo de implementación, todas las funciones están escritas, usando recurrencia, en términos de las primitivas del módulo LNat original.

Esto resulta en implementaciones no demasiado eficientes.

Por ejemplo, analícese cómo funciona snoc con las dos implementaciones vistas.
En ambos casos, crea una nueva lista repitiendo los elementos que son comunes con la original.
Otro ejemplo: escribir la función que calcula el Largo de una lista. Ver luego cómo computa cuando se usa la implementación estática vista arriba.
Puede comprobarse que, teniendo en cuenta la representación considerada, la implementación de esta función es muy ineficiente.
Llamamos a XLNat el módulo de listas extendido.
Para conseguir versiones más eficientes de los procedimientos de este módulo, debe admitirse que éstos utilicen la representación de listas que se elija, en lugar de ser definidos en términos de las funciones constructoras, predicados y selectoras primitivas.
También pueden usarse procedimientos que computan por efecto, en lugar de funciones.
Entonces, llegamos a otra versión del módulo de listas (Especificación Procedural) :

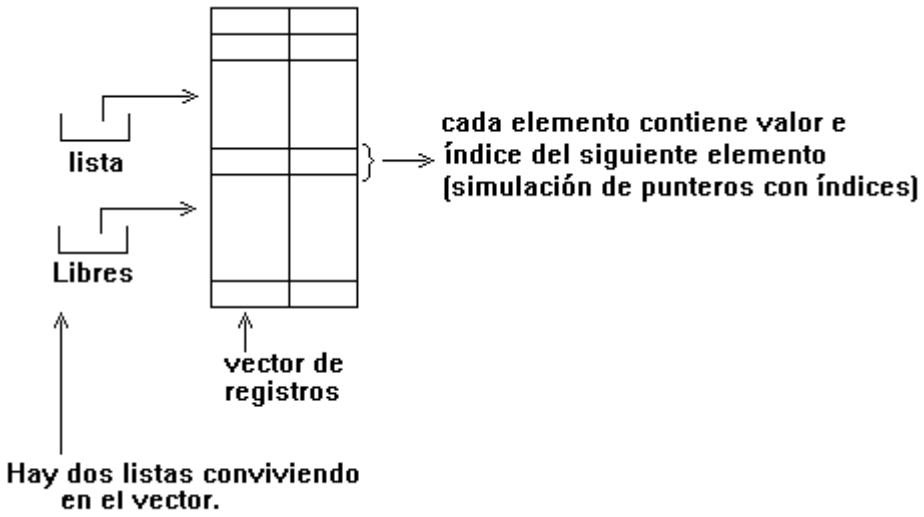
```
#ifndef  LNAT
#define  LNAT


typedef struct NodoLNat* LNat;

void Vacía (LNat s);
/* postcondición: s vacía */

void Cons (Natural x, LNat & s);
/* agrega x al frente de s */

void Snoc (LNat &s, Natural x);
/* agrega x al final de s */

void insSucck (Natural x, Natural K, struct NodoLNat
* & s);
/* precondición: 0 < k <= | s |, postcondición: x
insertado en la posición k + 1 de s */

void insDespués (Natural x,Natural n,LNat &s);
/* precondición: n está en s, postcondición>: x
insertado en s a continuación de la primera
ocurrencia de s */
```

```
void Borrar (Natural k, LNat &s);
/* precondición>: 0 < k <= | s |, postcondición: el
k-ésimo elemento de s es borrado */

bool esVacía (LNat s);

bool esMiembro (Natural n, LNat s);

Natural Largo(LNat s);

Natural Cabeza (LNat s);

LNat Cola (LNat s);

void avance (LNat &s );
/* equivalente a s := Cola (s) */

#endif
```

Si se implementa esta especificación usando la representación con listas encadenadas, podrán obtenerse versiones más eficientes de los procedimientos. Ver por ejemplo el caso de Snoc, ya discutido.

Para el caso de implementación estática, debería:

poderse utilizar eficientemente el espacio generado por operaciones de borrado.
poderse implementar las inserciones sin reubicar elementos.
Una idea es usar la siguiente representación:



Ejemplos:

LNat.h

```
#ifndef LNAT_H
#define LNAT_H
```

```
#include <stdlib.h>
typedef struct NodoLNat* LNat;
LNat Null();
bool Empty (LNat l);
int Head (LNat l );
LNat Tail (LNat l );
LNat Cons (int x,LNat l);

#endif
```

LNat.c

```
#include "LNat.h"

struct NodoLNat{
      int elem;
      struct NodoLNat* sig;
};

LNat Null(){
/* POST: devuelve la lista vacia */
return NULL;
}
bool Empty (LNat l){
/* POST: devuelve TRUE si la lista está vacia, FALSE en
otro caso */
return (l == NULL);
}

int Head (LNat l ){
/* PRE : l no es vacia */
/* POST: devuelve el primer elemento de l */
return (l->elem);
}


LNat Tail (LNat l ){
/* PRE : l no es vacia */
/* POST: devuelve l sin su primer elemento */
 return (l->sig);
}

LNat Cons (int x, LNat l ){
/* POST: inserta x al principio de la lista l y devuelve l*/
LNat nuevaLista;
nuevaLista=(LNat)malloc(sizeof(struct NodoLNat));
nuevaLista->elem = x;
nuevaLista->sig = l;
return nuevaLista;
}
```

LGNat.h

```
#ifndef LGNAT_H
#define LGNAT_H
#include <stdlib.h>
#include "LNat.h"

typedef struct NodoLGNat* LGNat;
LGNat NullLG();
// Crea la lista general vacía. Esto es [ ]

LGNat ConsLG(LNat l , LGNat lg);
/* Inserta un elemento al principio de la lista general. Esto es, dado el elemento
l y la lista general lg, retorna l.lg */

bool EmptyLG(LGNat lg);
/* Verifica si la lista general está vacía. Esto es, verifica si la lista general
es igual a [] */

LNat HeadLG(LGNat lg);
/* Retorna, si la lista general no es vacía, el primer elemento de la lista
general. Esto es, dada la lista general l.lg retorna l */

LGNat TailLG(LGNat lg);
/* Retorna, si la lista general no es vacía, la lista general sin su primer
elemento. Esto es, dada la lista general l.lg retorna lg */

#endif
```

LGNat.c

```
#include "LGNat.h"

struct NodoLGNat{
      LNat elem;
      struct NodoLGNat* sig;
};

LGNat NullLG(){
// Crea la lista general vacía. Esto es [ ]
return NULL;
}

LGNat ConsLG(LNat l , LGNat lg){
/* Inserta un elemento al principio de la lista general. Esto es, dado el elemento
l y la lista general lg, retorna l.lg */
LGNat nuevaLista;
nuevaLista=(LGNat)malloc(sizeof(struct NodoLGNat));
nuevaLista->elem = l;
```

```
nuevaLista->sig = lg;
return nuevaLista;
}

bool EmptyLG(LGNat lg){
/* Verifica si la lista general está vacía. Esto es, verifica si la lista general
es igual a [] */
 return lg==NULL;
}

LNat HeadLG(LGNat lg){
/* Retorna, si la lista general no es vacía, el primer elemento de la lista
general. Esto es, dada la lista general l.lg retorna l */
if(lg!=NULL)
 return lg->elem;
 else
 return NULL;
}

LGNat TailLG(LGNat lg){
/* Retorna, si la lista general no es vacía, la lista general sin su primer
elemento. Esto es, dada la lista general l.lg retorna lg */
 if(lg!=NULL)
 return lg->sig;
 else
 return NULL;
}
```