

**ESPECIFICACIÓN ALGEBRAICA**  
**DE TIPOS DE DATOS ABSTRACTOS:**  
**EL LENGUAJE NEREUS**

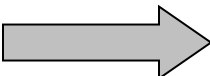

Se describen a continuación las construcciones del lenguaje algebraico *Nereus* utilizado en el curso para especificar tipos de datos abstractos.

## 1. ESPECIFICACIONES BÁSICAS

La especificación de un TDA se encapsula dentro de una clase en *Nereus*. A continuación se muestra en la sintaxis de una especificación básica en *Nereus*.

```
CLASS className [<parameterList>]
IMPORTS
  <importList>
BASIC CONSTRUCTORS
  <constructorList>
EFFECTIVE
TYPE (S)
  <typeList>
OPERATION(S)
  <operationList>
AXIOMS <varList>
  <axiomList>
END-CLASS
```

Podemos distinguir en una clase una especificación sintáctica, dada por la signature del tipo y una especificación semántica dada por la cláusula AXIOMS.

<b>CLASS</b> className [<parameterList>]		
<b>IMPORTS</b> <importList>		
<b>BASIC CONSTRUCTORS</b> <constructorList>		
<b>EFFECTIVE</b>		
<b>TYPE(S)</b> <sortList>		Signatura
<b>OPERATION(S)</b>		
<OperationList>		Especificación sintáctica
<hr/>		
<b>AXIOMS</b> <varList>		Axiomas
<axiomList>		Especificación semántica
<b>END-CLASS</b>		

El encabezamiento de la clase declara el nombre de la clase y una lista de parámetros *<parameterList>*. Cada clase puede tener cualquier cantidad de parámetros, dándose el caso particular de las clases no parametrizadas, cuando esa cantidad es igual a cero. *Nereus* provee la posibilidad de restringir cada uno de estos parámetros a un determinado tipo. La lista de parámetros *parameterList* está compuesta por pares de la forma *C1:C2* separados por comas, donde *C1* es un parámetro formal genérico restringido por una clase existente *C2*. En particular, *C1:ANY* expresa una parametrización sin restricción de tipos. Este tipo de parametrización sin restricción puede abreviarse simplemente como *C1*. En las secciones 2 y 3 se ampliará la descripción de clases parametrizadas.

La cláusula **IMPORTS** lista a las especificaciones importadas, es decir expresa relaciones cliente (“usa-a”). La especificación de la nueva clase está basada en las especificaciones importadas declaradas en *<importList>*.

La cláusula **BASIC CONSTRUCTORS** lista las operaciones constructoras básicas.

La cláusula **EFFECTIVE** declara nuevos tipos y operaciones definidos en forma completa. Se ampliará la descripción de esta cláusula en la sección 2.

Una declaración de tipos tiene la forma **TYPES**  $s_1, s_2, \dots, s_n$  o **TYPE**  $s$ , si se declara un único tipo.

En la cláusula **OPERATION (S)** se declaran las funcionalidades de las operaciones con la sintaxis **OPERATIONS**  $op_1; op_2; \dots, op_n$ ;. Si se declara una única operación la sintaxis es **OPERATION**  $op_1$ ;

Es posible definir operaciones en forma parcial. El dominio de definición de una función parcial puede hacerse explícito mediante el uso de aserciones, que deben suceder a la funcionalidad de la operación tras la palabra clave “**pre:**”.

Se muestra a continuación la signatura de la clase Boolean:

```
CLASS Boolean
BASIC CONSTRUCTORS True, False
EFFECTIVE
TYPE
Boolean
OPERATIONS
True: -> Boolean;
False: -> Boolean;
not_: Boolean -> Boolean;
_and_: Boolean * Boolean -> Boolean;
_or_: Boolean * Boolean -> Boolean;
_xor_: Boolean * Boolean -> Boolean;
_=>_: Boolean * Boolean -> Boolean;
_<=>_: Boolean * Boolean -> Boolean;
if_then_else_endif: Boolean * Boolean * Boolean -> Boolean;
```

Para facilitar el uso de formas condicionales la *CLASS Boolean* provee a la operación *if\_then\_else\_endif* y las ecuaciones asociadas son:

$$\begin{aligned} \text{if\_then\_else\_endif}(\text{true}, x, y) &= x \\ \text{if\_then\_else\_endif}(\text{false}, x, y) &= y \end{aligned}$$

Tras la palabra clave **AXIOMS** se declaran pares de la forma  $vI:CI$  donde  $vI$  es una variable universalmente cuantificada de tipo  $CI$ . Los axiomas incluidos a continuación de esta declaración expresan las propiedades requeridas por la especificación a partir de expresiones en lógica de primer orden construidas sobre términos y fórmulas. La lista de axiomas no puede ser vacía. Si no hubiera axiomas, porque las operaciones son totalmente diferidas, o porque no son necesarios, simplemente se omitirá la cláusula **AXIOMS** por completo.

Un término es una variable *tipada*, una constante o una aplicación de una operación en la que los argumentos satisfacen los tipos del dominio y el rango de la operación.

Las fórmulas son atómicas o compuestas. Una fórmula atómica es una ecuación entre dos términos del mismo tipo, separados por “=”.

Las ecuaciones de la forma *término*=*True* pueden ser abreviadas escribiendo simplemente el término. Por ejemplo, *vacíaLista (inicLista())=True* puede escribirse como *vacíaLista(inicLista())*.

Una fórmula compuesta (o predicado) puede ser construida a partir de los conectivos lógicos *not*, *and*, *or*,  $\Rightarrow$  y  $\Leftrightarrow$ .

Se muestran a continuación algunos axiomas de la clase *Boolean*:

```
AXIOMS x,y,z: Boolean;
not True = False;
not False = True;
False and x = False;
x and False = False;
True and x = x;
...
if_then_else_endif (true, x, y) = x ;
if_then_else_endif (false, x, y) = y;
END-CLASS
```

Todas las cláusulas son opcionales y no existe un orden entre ellas. La clase *Boolean* está implícitamente importada por todas las especificaciones y no es necesario incluirla en una cláusula **IMPORTS**.

Se presentan a continuación ejemplos de especificaciones básicas *Nereus*.

### **Ejemplo 1: La clase Pila**

```
CLASS Pila [elem: ANY]
BASIC CONSTRUCTORS inicPila, agregarPila
EFFECTIVE
TYPE Pila
OPERATIONS
inicPila: -> Pila;
agregarPila: Pila * elem -> Pila;
vacíaPila: Pila -> Boolean;
tope: Pila(p) -> elem
    pre: not vacíaPila(p);
eliminarPila: Pila (p) -> Pila
    pre: not vacíaPila(p);
AXIOMS p: Pila; e:elem;
tope(agregarPila(p,e))= e;
vacíaPila(inicPila()) = True;
vacíaPila (agregarPila(p,e)) = False;
eliminarPila(agregarPila(p,e))=p;
END-CLASS
```

## **Ejemplo 2: La clase Fila**

```
CLASS Fila [Elemento]
BASIC CONSTRUCTORS  inicFila, agregarFila
EFFECTIVE
TYPE Fila
OPERATIONS
  inicFila: -> Fila;
  agregarFila: Fila * Elemento -> Fila;
  vaciaFila: Fila -> Boolean;
  recuperarFila: Fila(f) -> Elemento
    pre: not vaciaFila (f);
  eliminarFila: Fila(f) -> Fila
    pre: not vaciaFila(f);
AXIOMS f: Fila; e: Elemento;
  vaciaFila (inicFila() ) =True;
  vaciaFila (agregarFila (f,e)) = False;
  eliminarFila (agregarFila(f, e)) = if vaciaFila(f)
                                     then inicFila()
                                     else agregarFila (eliminarFila (f), e)
                                     endif;
  recuperarFila (agregarFila ( f, e)) = if vaciaFila (f)
                                     then e
                                     else recuperarFila (f)
                                     endif;

END-CLASS
```

Otra especificación para la clase Fila:

```
CLASS Fila [Elemento]
BASIC CONSTRUCTORS  inicFila, agregarFila
EFFECTIVE
TYPE Fila
OPERATIONS
  inicFila: -> Fila;
  agregarFila: Fila * Elemento -> Fila;
  vaciaFila: Fila -> Boolean;
  recuperarFila: Fila(f) -> Elemento
    pre: not vaciaFila (f);
  eliminarFila: Fila(f) -> Fila
    pre: not vaciaFila(f);
AXIOMS f: Fila; e: Elemento;
  vaciaFila (inicFila() ) =True;
  vaciaFila (agregarFila (f,e)) = False;
  vaciaFila (f) => eliminarFila (agregarFila(f, e)) = inicFila();
  not vaciaFila (f) => eliminarFila(agregarFila (f,e)) = agregarFila (eliminarFila (f), e);
  vaciaFila (f) => recuperarFila (agregarFila ( f, e)) = e;
  not vaciaFila (f) => recuperarFila (agregarFila ( f, e)) = recuperarFila (f);

END-CLASS
```

### **Ejemplo 3: La clase Lista**

```
CLASS Lista [Elemento]
IMPORTS Nat
BASIC CONSTRUCTORS inicLista, agregarLista
EFFECTIVE
TYPE Lista
OPERATIONS
inicLista: -> Lista;
longLista: Lista-> Nat;
agregarLista: Lista (l) * Nat (i) * Elemento -> Lista
    pre: ( i >= 1) and ( i <= longLista(l)+ 1);
eliminarLista: Lista (l) * Nat (i) -> Lista
    pre: ( i >= 1) and ( i <= longLista(l));
recuperarLista: Lista (l) * Integer (i) -> Elemento
    pre: ( i >= 1) and ( i <= longLista(l));
AXIOMS l:Lista; e: Elemento; i,j:Nat;
longLista(inicLista())= 0;
longLista (agregarLista (l,i,e)) = 1 + longLista (l);
(i == j) => (eliminarLista(agregarLista(l,i,e), j) = l);
(i > j) => (eliminarLista (agregarLista(l, i,e), j) = agregarLista (eliminarLista(l,j), i-1, e));
(i < j) => (eliminarLista(agregarLista(l,e,i), j) = agregarLista(eliminarLista(l,j-1), e,i));
(i == j) => (recuperarLista (agregarLista(l,i, e), j) = e);
(i > j) => (recuperarLista(agregarLista(l, i, e),j) = recuperarLista (l, j));
(i < j) => (recuperarLista(agregarLista(l, i, e), j) = recuperarLista(l, j-1));
END-CLASS
```

### **Ejemplo 4: La clase árbol binario**

La clase *Arbin* define una especificación de árboles binarios. Sus operaciones constructoras básicas son las operaciones *inicArbin* y *crearArbin*. Las operaciones observadoras son *raiz*, *vacioArbin* y las transformadoras (modificadoras) *subIzquierdo* y *subDerecho*.

```
CLASS Arbin [Elemento]
BASIC CONSTRUCTORS inicArbin, crearArbin
EFFECTIVE
TYPE Arbin
OPERATIONS
inicArbin: -> Arbin;
crearArbin: Arbin * Arbin * Elemento -> Arbin;
vacioArbin: Arbin -> Boolean;
raiz: Arbin(t) -> Elemento
    pre: not vacioArbin(t);
subIzquierdo: Arbin (t)->Arbin
    pre: not vacioArbin(t);
```

```
subDerecho: Arbin (t) -> Arbin
  pre: not vacioArbin(t);
AXIOMS t1,t2: Arbin; e: Elemento;
vacioArbin (inicArbin ()) = True;
vacioArbin (crearArbin(t1,t2,e)) = False;
raiz(crearArbin(t1,t2,e)) = e;
sublzuierdo(crearArbin(t1,t2,e)) = t1;
subDerecho(crearArbin(t1,t2,e)) = t2;
END-CLASS
```

### **Ejemplo 5. La clase Heap de Nat**

```
CLASS Heap
IMPORTS Nat
BASIC CONSTRUCTORS inicHeap, agregarHeap
EFFECTIVE
TYPE Heap
OPERATIONS
inicHeap: -> Heap;
agregarHeap: Heap * Nat -> Heap;
vacioHeap: Heap -> Boolean;
raizHeap: Heap(h) -> Nat
  pre: not vacioHeap(h);
eliminarHeap: Heap (h) -> Heap
  pre: not vacioHeap(h);
AXIOMS h:Heap; e: Nat;
vacioHeap(inicHeap()) = True;
vacioHeap(agregarHeap(h,e)) = False;
vacioHeap(h) => (raizHeap(agregarHeap(h,e)) = e);
not vacioHeap(h) =>
  (raizHeap(agregarHeap(h,e)) =
    if (e < raizHeap(h))
      then e
      else raizHeap(h)
    endif);
vacioHeap(h) => (eliminarHeap(agregarHeap(h,e)) = inicHeap());
not vacioHeap(h) =>
  (eliminarHeap(agregarHeap(h,e)) =
    if (e < raizHeap(h))
      then h
      else agregarHeap(eliminarHeap(h),e)
    endif);
END-CLASS
```

## 2. ESPECIFICACIONES INCOMPLETAS

A continuación se muestra la sintaxis de una especificación incompleta en *Nereus*

```
CLASS className [<parameterList>]
IMPORTS
  <importList>
BASIC CONSTRUCTORS
  <constructorList>
DEFERRED
TYPE(S)
  <typeList>
OPERATION(S)
  <operationList>
EFFECTIVE
TYPE(S)
  <typeList>
OPERATION(S)
  <OperationList>
AXIOMS <varList>
  <axiomList>
END-CLASS
```

Las especificaciones incompletas agregan la cláusula **DEFERRED**. La misma declara tipos y operaciones que no están completamente definidos debido a que, o bien no hay suficientes ecuaciones para definir el comportamiento de las nuevas operaciones o, no hay suficientes operaciones para generar todos los valores de un tipo.

La cláusula **EFFECTIVE** agrega tipos y operaciones completamente definidos o completa la definición de algún tipo u operación definido en forma incompleta en alguna superclase. Una operación preexistente cuya definición se completa puede ser declarada en la cláusula **EFFECTIVE** dando sólo su nombre.

Se presentan a continuación ejemplos de especificaciones incompletas.

### Ejemplo 6: La clase ConOrden

La clase ConOrden especifica cualquier conjunto de valores ordenados.

```
CLASS ConOrden
IMPORTS Boolean
DEFERRED
TYPE ConOrden
OPERATIONS
  _==_: ConOrden * ConOrden -> Boolean;
  _<= _: ConOrden * ConOrden -> Boolean;
EFFECTIVE
```



#### OPERATIONS

```
_>=: ConOrden * ConOrden -> Boolean;  
_<=: ConOrden * ConOrden -> Boolean;  
_>_: ConOrden * ConOrden -> Boolean;
```

**AXIOMS** a,b,c: ConOrden;

```
a == a;  
a <= a;  
(a==b) or not (a==b);  
(a==b) => (b==a);  
(a <= b) or (b <= a);  
(a <= b) and (b <= c) => (a <= c);  
(a <= b) and (b <= a) = (a==b);  
(a >= b) = (b <= a),  
(a > b) = not(a <= b);  
(a < b) = (b > a);
```

**END-CLASS**

### Ejemplo 7: La clase Recorrible

La clase *Recorrible* especifica en forma incompleta el comportamiento de estructuras que pueden ser recorridas. Independientemente del tipo de estructura que se trate y cualquiera sea la forma de recorrerla, necesitaremos operaciones para acceder a un elemento (*primero*), a la estructura restante (*resto*), y reconocer el final (*fin*). Estas operaciones no pueden definirse en forma completa hasta no saber el tipo de estructura por ejemplo, un árbol o una lista.

**CLASS** Recorrible [Elemento]

**DEFERRED**

**TYPE** Recorrible

**OPERATIONS**

```
fin: Recorrible -> Boolean;  
primero: Recorrible (t) -> Elemento  
    pre: not fin (t);  
resto: Recorrible (t) -> Recorrible  
    pre: not fin(t);
```

**END-CLASS**

## 3. HERENCIA DE ESPECIFICACIONES

La cláusula **INHERITS** permite especificar relaciones de herencia entre especificaciones. Se muestra a continuación la sintaxis de una clase *Nereus* que incluye a la cláusula **INHERITS**.

**CLASS** className [<parameterList>]

**IMPORTS** <importList>

**INHERITS** <inheritList>

**BASIC CONSTRUCTORS**

<constructorList>

```
DEFERRED
TYPE (S) <typeList>
OPERATION(S)
<OperationList>
EFFECTIVE
TYPE(S)
<typeList>
OPERATION(S)
<OperationsList>
AXIOMS <varList> <axiomList>
END-CLASS
```

*Nereus* soporta herencia múltiple. La cláusula **INHERITS** expresa que la clase es construida a partir de la unión de las clases que aparecen en *<inheritList>*. Los componentes de cada una de ellas serán componentes de la nueva clase, y sus propios tipos y operaciones serán tipos y operaciones de la nueva clase.

*Nereus* permite definir instancias locales de una clase en las cláusulas **IMPORTS** y **INHERITS** mediante la siguiente sintaxis:

*ClassName* [*<parameterList>*] [*<bindingList>*]

donde los elementos de *<parameterList>* pueden ser pares de nombres de clases *C1:C2*, donde *C2* es una subespecificación importada de *ClassName*.

*<bindingList>* es una lista de renombres separados por comas, precedidos por la palabra clave **rename** de la forma siguiente: **rename** *nombreOrigen* **as** *nombreDestino*, que indica los renombres de miembros de *ClassName* para ser utilizados dentro de la clase donde se está instanciando *className*. Es decir, *nombreOrigen* corresponde a la parte propia de la clase *ClassName*, y será referenciado con *nombreDestino* dentro de la clase que se está definiendo.

El tipo de interés de la clase es también implícitamente renombrado cada vez que la clase es sustituida o renombrada.

Durante el proceso de construcción de una especificación de una clase puede suceder que dos o más tipos o funciones provenientes de especificaciones diferentes tengan el mismo nombre. En *Nereus*, dentro de una misma especificación, dos tipos (u operaciones con la misma aridad) se identifican. Si esto no es lo que se busca, *Nereus* ofrece el mecanismo de renombre de tipos y operaciones.

### **Ejemplo 8: La clase Nat**

```
CLASS ConOrdenNatural
INHERITS ConOrden
BASIC CONSTRUCTORS 0, suc;
EFFECTIVE
TYPE ConOrdenNatural
OPERATIONS
_==_, _<=_;
0: -> ConOrdenNatural;
suc: ConOrdenNatural -> ConOrdenNatural;
```

```
AXIOMS a,b: ConOrdenNatural;  
not (0 == suc(a));  
(suc(a) == suc(b)) = (a == b);  
0 <= suc(a);  
not (suc(a) <= 0);  
(suc (a) <= suc (b)) = (a <= b);  
END-CLASS
```

```
CLASS Nat  
INHERITS ConOrdenNatural  
EFFECTIVE  
TYPE Nat  
OPERATIONS  
_+_ : Nat * Nat -> Nat;;  
_*_ : Nat * Nat -> Nat;  
1: -> Nat;  
AXIOMS x,y, z: Nat;  
x + 0 = x;  
x + suc(y) = suc( x + y );  
x + y = y + x;  
(x + y) + z = x + (y + z);  
1 * x = x;  
x * 0 = 0;  
suc(x) * y = y + (x * y);  
x * y = y * x;  
(x * y) * z = x * (y * z);  
1 = suc (0)  
END-CLASS
```

### Ejemplo 9: La clase Heap[Elemento:ConOrden]

*Nereus* provee la posibilidad de restringir cada uno de los parámetros a un determinado tipo además instanciarse con cualquiera de sus subtipos.

El parámetro *Elemento* de la clase *Heap* restringe la instanciación a la clase *ConOrden* o sus subclases (por ejemplo Nat).

```
CLASS Heap [Elemento: ConOrden]  
BASIC CONSTRUCTORS inicHeap, agregarHeap  
EFFECTIVE  
TYPE Heap  
OPERATIONS  
inicHeap: -> Heap;  
agregarHeap: Heap * Elemento -> Heap;  
vacioHeap: Heap -> Boolean;  
raizHeap: Heap(h) -> Elemento  
    pre: not vacioHeap(h);
```

```
eliminarHeap: Heap (h) -> Heap
  pre: not vacioHeap(h);
AXIOMS h:Heap; e: Elemento;
vacioHeap(inicHeap()) = True;
vacioHeap(agregarHeap(h,e)) = False;
vacioHeap(h) => (raizHeap(agregarHeap(h,e)) = e);
not vacioHeap(h) => (raizHeap(agregarHeap(h,e)) =
  if (e < raizHeap(h))
  then e
  else raizHeap(h)
  endif);
vacioHeap(h) => (eliminarHeap(agregarHeap(h,e)) = inicHeap());
not vacioHeap(h) => (eliminarHeap(agregarHeap(h,e)) =
  if (e < raizHeap(h))
  then h
  else agregarHeap(eliminarHeap(h),e)
  endif);
END-CLASS
```

### **Ejemplo 10: La clase PostArbin**

La clase *PostArbin* refina *Arbin* (Ej. 4) especificando árboles binarios que se recorren en postorden. Nótese que hereda las operaciones y axiomas de *Arbin* y *Recorrible* (Ej. 7). Las operaciones *primero* y *resto* heredadas de *Recorrible* se definen ahora en forma completa a partir de un conjunto de axiomas. La operación *fin* se define también en forma completa renombrándola por *vacioArbin* que proviene de *Arbin*.

Los tipos *Arbin* y *Recorrible* se renombran por *PostArbin*. Las funciones *primero*, *resto* y *fin* se declaran efectivas como *PostArbin*. Nótese que sólo basta listar sus nombres y no repetir sus funcionalidades. *Arbin* se instancia con *Elem* y las operaciones *inicArbin* y *crearArbin* son renombradas.

```
CLASS PostArbin [Elem]
INHERITS Arbin [Elem] rename inicArbin as inicPostArbin, crearArbin as
  crearPostArbin, Recorrible [Elem] rename fin as vacioArbin
EFFECTIVE
TYPE PostArbin
OPERATIONS primero, resto, vacioArbin;
AXIOMS t1,t2: PostArbin; x : Elem;
vacioArbin(t1) and vacioArbin(t2) => primero (crearPostArbin (t1,t2,x)) = x;
vacioArbin(t1) and (not vacioArbin(t2)) =>
  primero (crearPostArbin (t1,t2,x)) = primero (t2);
not vacioArbin(t1) => primero (crearPostArbin(t1,t2,x)) = primero (t1);
vacioArbin(t1) and vacioArbin(t2) => resto(crearPostArbin(t1,t2,x)) = inicPostArbin();
vacioArbin(t1) and (not vacioArbin(t2)) =>
  resto (crearPostArbin (t1, t2, x)) = crearPostArbin(t1,resto(t2),x);
not vacioArbin(t1) => resto(crearPostArbin(t1,t2,x)) = crearPostArbin (resto(t1), t2,x);
END-CLASS
```

Se muestra a continuación la expansión de la clase **PostArbin**:

**CLASS** PostArbin [Elem]

**BASIC CONSTRUCTORS** inicPostArbin, crearPostArbin

**EFFECTIVE**

**TYPE** PostArbin

**OPERATIONS**

vacioArbin: PostArbin -> Boolean;

primero: PostArbin (t) -> Elem

**pre:** not vacioArbin(t);

resto: PostArbin (t) -> PostArbin

**pre:** not vacioArbin(t);

inicPostArbin: -> PostArbin;

crearPostArbin: PostArbin \* PostArbin \* Elem -> PostArbin;

raiz: PostArbin (t) -> Elem

**pre:** not vacioArbin(t);

sublzquierdo: PostArbin (t) -> PostArbin

**pre:** not vacioArbin(t);

subDerecho: PostArbin ( t) -> PostArbin

**pre:** not vacioArbin(t);

**AXIOMS** t1,t2: PostArbin ; e: Elem;

vacioArbin(inicPostArbin()) = True;

vacioArbin(crearPostArbin(t1,t2,e)) = False;

raiz(crearPostArbin(t1,t2,e)) = e;

sublzquierdo(crearPostArbin(t1,t2,e)) = t1;

subDerecho(crearPostArbin(t1,t2,e)) = t2;

vacioArbin(t1) and vacioArbin(t2) => primero (crearPostArbin(t1,t2,x)) = x;

vacioArbin(t1) and (not vacioArbin(t2)) => primero (crearPostArbin(t1,t2,x)) = primero (t2);

not vacioArbin(t1) => primero (crearPostArbin(t1,t2,x)) = primero (t1);

vacioArbin(t1) and vacioArbin(t2) => resto(crearPostArbin(t1,t2,x)) = inicPostArbin();

vacioArbin(t1) and (not vacioArbin(t2)) =>

resto (crearPostArbin (t1, t2, x)) = crearPostArbin(t1,resto(t2),x);

not vacioArbin(t1) => resto(crearPostArbin(t1,t2,x)) = crearPostArbin (resto(t1), t2,x);

**END-CLASS**

**Ejercicio propuesto:** Especificar la clase PreArbin